EUROPEAN CYBER SECURITY ORGANISATION

August 2024 v1.0

TECHNICAL PAPER

SOFTWARE SUPPLY CHAIN SECURITY

WORKING GROUP 6
Technologies & Innovation
and Defence & Space

ecs-org.eu

ABOUT

The **European Cyber Security Organisation (ECSO)** is a not-for-profit membership-based organisation established in 2016. Uniting more than 320 stakeholders, ECSO develops a competitive European cybersecurity ecosystem that provides trusted cybersecurity solutions, advances Europe’s technological independence, and unifies its cybersecurity posture. ECSO also leads the European project ECCO, supporting activities needed to develop, promote, coordinate and organise the European-level Cybersecurity Competence Community.

EMPOWERING EUROPEAN CYBERSECURITY COMMUNITIES

EXECUTIVE SUMMARY

This paper stems from the consideration that software is highly spread in nowadays societies. Despite this significant level of adoption, how software is made remains a less known aspect. In particular, the complexity of software development has only increased in the recent years, with an evolution in methodologies and processes used, the tools, the languages, and the speed. Not to mention very recent potential developments brought by using large language models in writing code (however such implications will not be the focus of this work).

Even though developers can strive to **build secure code** and follow best practices, it is evident from this analysis that the way software is built today implies a heavy reliance on third parties' contributions, as well as the usage of externally provided tools and components. If this allows modern software to exist, such **complex ramifications of dependencies creates some cybersecurity challenges.**

This paper discusses the **lifecycle development of software** and provides an analysis of how software is commonly developed today, what tools and technologies are typically used and what risks exist. The main objective is to identify the most relevant cybersecurity challenges, and what are the implications of how the software is developed and consumed. A specific focus is on the **software supply chain** due to the increase of the attack surface. In essence, compromising upstream components of the software supply chain will mean that malicious activity can happen in environments that are distant from the those who develop the software and even more distant from those who sell the final product or service. One key element stemming from the analysis of the software lifecycle is the importance of software supply chain as a critical aspect of **European sovereignty and autonomy.**

The ultimate goal is to provide **recommendations** with clear references to **frameworks on software supply chain**, and good practices for development, maintainance and risk exposure reduction.

This paper also propose some areas where **further innovation** is needed in order to increase the overall security posture of the current software development ecosystem, keeping an eye on automation, open-source software development and techniques to reduce the available attack surface.

CONTENTS

INTRODUCTION

1.

1.1. Problem Statement

The world of software development has changed extremely in the past few years. In a short time, software is included in pretty much any product or service of our daily lives. In addition, the landscape of software development has changed dramatically, in terms of methodologies, processes, platforms, tools, languages, speed, complexity, to name a few. Thus, the notion of software supply chain, i.e., how software is embedded into these products and services and how 3rd party software can be integrated in (software) vendor's offers, has become a major challenge for companies and administrations.

A major change that could be observed over the past years is that developers now rely on externally provided tools and components. This dependence is extreme, up to a point where software developed by an organisation (and the associated control of source code) can be around 10%[1] of the full software product. This includes development platforms and tools (gcc, eclipse), configuration and build tools, libraries and associated packages, underlying operating systems, and more. The trend of open-source, while generally benefitting organisations, has accelerated this dimension as it allows developers to avoid rewriting existing code and to focus on the core functionality of their software.

This softwarisation trend is also opening a whole new attack surface. In this model, the attacker does not interact with the product or service directly to carry out its attack. Rather, s/he inserts itself in the software supply chain, compromising some component in an environment distant from the product or service developer, and even further away from the final product owner or service operator. The threat is real, and we have already seen instances of it with extreme prejudice for the victims.

This document responds to the fact that the software supply chain is a critical aspect of European sovereignty and autonomy, and a crucial support for cyber-secure and cyber-safe products and services. The objective is to analyse multiple aspects of software supply chain security. The scope of this document includes:

1. An introduction to modern software development and deployment processes, tools and techniques, setting the scene for attack surface analysis and risk identification.

2. A threat model related to the essential aspects of such software development and its related supply chain, including agile development environments (which heavily rely on the availability of tools and libraries). It also considers the complexity of software building and deployment toolchains (including reliance on third-party software components and frameworks).

[1] The Linux Foundation indicates that "it has been estimated that Free and Open Source Software (FOSS) constitutes of any given piece of modern software solutions." Source: https://www.linuxfoundation.org/blog/blog/a-summary-of-census-ii-open-source-software-application-libraries-the-world-depends-on#:~:text=It%20has%20been%20estimated%20that,and%20non%2Dtech%20companies%20alike. Last Accessed on 3rd April 2024

3 The methods and tools related to risk assessment in this domain, also addressing the risk related to embed third party components, and to understand quantitatively this risk and the possibilities to remediate it.

4 Recommendations on approaches, best practices, and innovation needs that help to address and mitigate supply chain related risks.

Many of today's software systems and services expose a highly dynamic behaviour. They integrate dynamic libraries at runtime, which provenance and semantics cannot be anticipated until they are actually executed. Since our investigation focuses on development environments, processes, and tools, this paper does not discuss dynamic libraries unless they contribute to the dynamic deployment of the software component.

Furthermore, the risks that result from the specific functionalities of the software consumed are not in the scope, e.g. consumption of AI models. This topic is addressed in the ECSO technical papers focused on the specific technologies, such as the one on Artificial Intelligence.

The target audience of this document includes both software development organisations consuming 3rd party and open-source software throughout the development lifecycles of their products, as well as end-users who consume via its user interface a software product and service built on 3rd party and open-source software and services. This clarification is needed to compile the overview about possible controls/safeguards. However, the recommendations for these two target groups differ following their different levels of control; this pwill be indicated in the recommendations section.

1.2. Related work

Following a number of supply-chain related incidents in the past two years, the topic of software supply chain threats and risks has raised increased attention. An ENISA study provides an overview and an analysis of 24 supply chain related attacks that occurred between January 2020 and July 2021[2]. However, this report strictly focuses on software components provided by 3rd party commercial vendors, excluding open-source. While there are good arguments for their scoping, we think that open-source software should be considered because of their relevance for today's software development.

One of the results of the SPARTA project is a taxonomy of open-source supply chain attack vectors, which supports developers in becoming aware of open-source risks and work towards mitigation strategies[3]. The taxonomy is available as an open-source service[4].

Open-source and supply chain risks have also been considered when designing criteria for the cybersecurity certification of cloud services. The German C5 criteria catalogue[5] as well as the upcoming European Cloud Security certification scheme include related requirements. In its special publication 800-160[6], NIST has collected security principles and best practices that can also help to identify and mitigate supply chain risks. NIST SP 800-160 has been complemented by the NIST SP 800-161[7] , which define dedicated "Cybersecurity Supply Chain Risk Management Practices for Systems and Organizations".

Open-source and other organisations have recognized the importance of securing open-source software and open-source based supply chains in the development of secure software and services. They came up with efforts to provide practices for a secure software lifecycle, as well as frameworks focusing on the secure consumption of open-source components within the software supply chain.

The Enduring Security Framework (a public-private cross-sectional working group) released "Securing the Software Supply Chain for Developers[8] to provide actionable guidance to developers to secure the software supply chain. It describes the security criteria, how to manage security during the software development process, what actions to take in writing secure code, how to verify third-party components, how to harden the build environment, and finally how suppliers should securely deliver the software produced to customers.

[2] ENISA. Threat Landscape for Supply Chain Attacks. July 2021

[3] P. Ladisa, H. Plate, M. Martinez and O. Barais, "SoK: Taxonomy of Attacks on Open-Source Software Supply Chains," in 2023 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 2023 pp. 1509-1526. doi: 10.1109/SP46215.2023.10179304

[4] Risk Explorer for Software Supply Chains. Available at https://github.com/SAP/risk-explorer-for-software-supply-chains [Last access on 29 January 2024]

[5] BSI - Federal Office for Information Security. Cloud Computing Compliance Criteria Catalogue – C5:2020

[6] NIST. Engineering Trustworthy Secure Systems. SP 800-160 Vol. 1 Rev. 1. November 2022.

[7] NIST. Cybersecurity Supply Chain Risk Management Practices for Systems and Organizations. SP 800-161 Rev. 1. May 2022

[8] The Enduring Security Framework. Securing the Software Supply Chain: Recommended Practices for Managing Open-Source Software and Software Bill of Materials. December 2023

The Open Source Security Foundation (OpenSSF) has published a number of guides on Open-Source software and its evaluation, e.g. the Concise Guide for Evaluating Open Source Software[9]. In particular, In February 2022, the OSSF has launched the Alpha-Omega project[10], where both developer support for critical open-source projects will be offered and at least 10.000 widely deployed OSS projects will be analysed. They also introduced SLSA (Supply Chain Levels for Software Artefacts)[11], a checklist of standards and controls to prevent tampering, improve integrity, and secure packages and infrastructure in projects, businesses or enterprises. It defines four levels of assurance, from simple provenance information via a documented, automated build process, to high confidence and trust via peer-review of source code changes with hermetic, reproducible builds.

The OWASP Software Component Verification Standard[12] aims at establishing a framework for identifying activities, controls, and best practices, which can help in identifying and reducing risk in a software supply chain. Such controls are grouped in six control families: inventory, Software Bill Of Materials (SBOM), build environment, package management, SCA, and software pedigree and provenance.

The Microsoft Open Source Software (OSS) Secure Supply Chain (SSC) Framework combines requirements and tools to reduce risks associated with the consumption of open-source software. It is based on three core concepts: control all consumed open-source software, use of maturity model to help in prioritising the requirements to implement and secure the software supply chain at scale.

Another relevant publication is the ENISA guideline for securing the Internet of Things[13], which puts forward useful recommendations related to 3rd party software including open-source. This ECSO study complement it with a set of aligned recommendations.

[9] OpenSSF. Concise Guide for Evaluating Open Source Software. https://best.openssf.org/Concise-Guide-for-Evaluating-Open-Source-Software [Last access on 29 January 2024]

[10] OpenSSF. The Alpha-Omega project. https://openssf.org/community/alpha-omega/ [Last access on 29 January 2024]

[11] SLSA (Supply-chain Levels for Software Artifacts). https://slsa.dev/ [Last access on 29 January 2024]

[12] OWASP Software Component Verification Standard. https://owasp.org/www-project-software-component-verification-standard/ [Last access on 29 January 2024]

[13] ENISA. Guidelines for Securing the Internet of Things. November 2020

1.3. EU Policy considerations related to software supply chain

The EU has not overlooked that the most sophisticated cyber-attacks in recent years target software and hardware components used by many companies downstream in the supply chain. One of the most relevant pieces of legislation with policy implication for the Software supply chain is the Cyber Resilience Act. This bill is reaching its finish line in 2024, after being proposed by the European Commission at the end of 2022.

The objective of the CRA is to establish a minimum level of cybersecurity for all digital devices (both software and hardware) sold in the EU internal market. One of the aims of the CRA is indeed strengthening the security of the whole supply chain by facilitating the secure development of products with digital elements and their components. More precisely, it does so by defining cybersecurity rules for placing products on the market; requirements for the design, development, and production of products; requirements for the vulnerability handling process; and rules on market surveillance and enforcement.

The CRA will complement other European cybersecurity regulations, by providing a horizontal level playing field of cybersecurity criteria and interplaying other relevant legislations.

It is worth highlighting the critical requirements concerning the software supply chain mentioned in the regulation: organisations will have to draft a Software Bill of Material (SBOM), open-source software dependencies will have to be tracked, and consequently vulnerabilities throughout the supply chain will have to be addressed.

HOW WE DO DEVELOP SOFTWARE TODAY

2.

Having set the context for the current analysis, this section focuses on how software is commonly developed today. This presentation is crucial in order to grasp the potential risks concerning the software supply chain. This section concludes by discussing some frameworks that could be useful to mitigate such risks.

2.1. Today's Software and Software Development

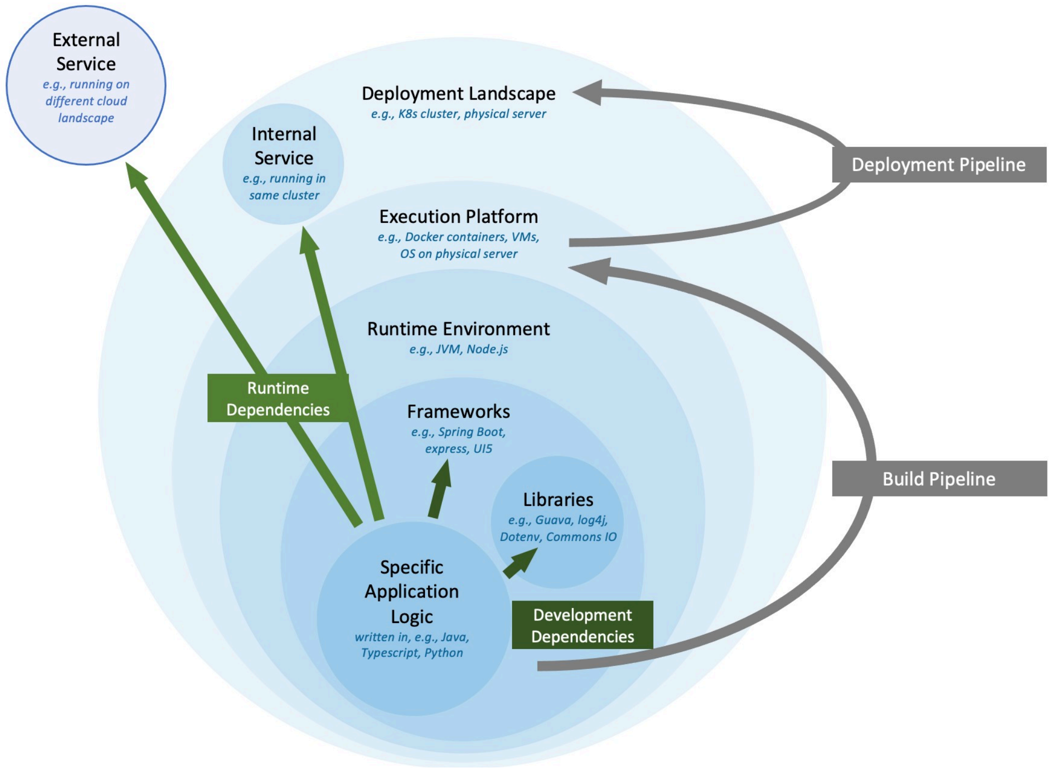

External Service
e.g., running on different cloud landscape
Internal Service
e.g., running in same cluster
Deployment Landscape
e.g., K8s cluster, physical server
Deployment Pipeline
Execution Platform
e.g., Docker containers, VMs, OS on physical server
Runtime Environment
e.g., JVM, Node.js
Runtime Dependencies
Frameworks
e.g., Spring Boot, express, UI5
Libraries
e.g., Guava, log4j, Dotenv, Commons IO
Build Pipeline
Specific Application Logic
written in, e.g., Java, Typescript, Python
Development Dependencies

Figure 1 *High-level Technology Stack*

Today, providers of software products and services heavily reuse 3rd party and open-source solutions to develop their offering. Often, solution-specific application logic only amounts to 10% or less of an overall application code base, while the greater share is comprised of 3rd party and open-source software like libraries and frameworks. Such dependencies can exist both at development time, in which case they are bundled and distributed together with the application code, as well as at runtime, in which case an application interacts with internal or external services during its execution.

A typical technology stack also comprises runtime environments, which are executed – depending on the deployment model – with the help of cloud technologies or directly on physical systems like end-user computers or smart phones. Again, especially in case of cloud solutions, those environments heavily rely on 3rd party and open-source.

At high-level, all those components are developed in a comparable fashion (see Figure 1). The source code is kept and managed in versioning control systems and pulled by (automated) build pipelines to perform functional and non-functional tests. Provided all quality assurance tests pass, the software is packaged for easy consumption, and made available on distribution platforms like software marketplaces and package repositories or directly deployed into production environments. Compared to traditional development processes, the time needed to integrate, test, and even deploy code changes into production environments, has been significantly decreased through Continuous Integration (CI) and Continuous Delivery (CD) processes.

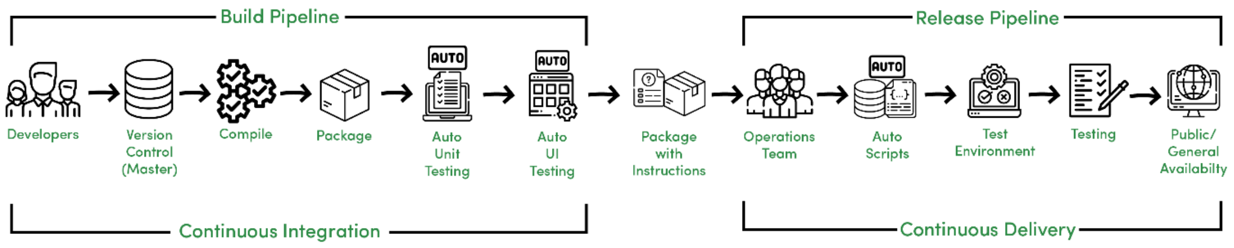


Figure 2 *Continuous Integration and Continuous Delivery Process*

Figure 2 illustrates common stakeholders and systems involved in the development, CI/CD processes and distribution. The members of development projects, e.g. DevOps engineers or software architects, have privileged access to project resources like the source code management or build system.

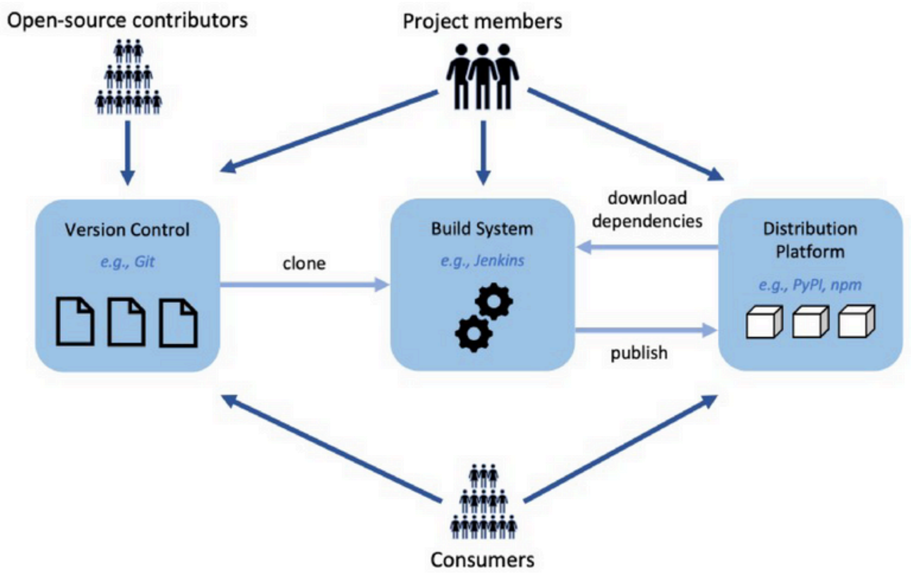


Figure 3 *Continuous Integration and Continuous Delivery Process*

Figure 3 illustrates the actors and the management of an open-source project. The open-source projects can additionally receive contributions from potentially unknown project-external contributors. The downstream consumers typically download pre-built packages from software marketplaces or package repositories, but they can also consume and inspect the source code in case of open-source.

Build systems, due to the CI/CD processes running thereon, have an important role regarding security. The leaked credentials for a SolarWinds build system, for example, were at the origin of the SolarWinds attack, and permitted the attacker to plant backdoors in the Orion product. Moreover, build systems commonly depend on and execute numerous 3rd party and open-source components, especially runtime dependencies of the software being built. The so called package managers automate the download and installation of such dependencies, which greatly facilitated the adoption of open source during software development. But 3rd party and open source is also used on other systems depicted in Figure 1, be it integrated development environments (IDE) used by developers or the source code management system.

In summary, the software supply chain of a given technology stack comprises all the systems, technologies, people, and processes involved in creating, building, and distributing its constituting elements. Those are inherently distributed and managed by different and partly unknown parties, including commercial and non-commercial organisations as well as individuals.

In this context, the secure software lifecycle processes are proactive approaches employed by one party to build security into its respective software product, treating the ’disease’ of poorly designed, insecure software at the source, rather than ’applying a band aid’ to stop the symptoms through a reactive penetrate and patch approach. These processes work software security deeply into the full product development process and incorporate people and technology to tackle and prevent software security issues.

There are prescriptive secure software lifecycle processes that explicitly recommend software practices such as Microsoft Security Development Lifecycle (SDL)[14], Touchpoints[15] or SAFECode[16]. The practices of these processes are integrated and cover a broad spectrum of the lifecycle phases, from software requirements to release/deployment and software maintenance.

These secure software lifecycle models can be integrated with any software development model and are domain agnostic such as Agile Software Development, DevOps, Mobile, Cloud Computing, Internet of Things, Road Vehicles and eCommerce.

[14] Microsoft. The Security Development Lifecycle (SDL) https://www.microsoft.com/en-us/securityengineering/sdl [Last access on 29 January 2024]

[15] G. McGraw, Software Security: Building Security In. Addison-Wesley Professional, 2006.

[16] The Software Assurance Forum for Excellence in Code (SAFECode). https://safecode.org/ [Last access on 29 January 2024]

On the other hand, organisations may wish to or be required to assess the maturity of their secure development lifecycle. Some assessment approaches are Software Assurance Maturity Model (SAMM)[17], Building Security In Maturity Model (BSIMM)[18] and Common Criteria (CC)[19].

The successful adoption of these best practices involves organisational and cultural changes in software development companies. The organisation, starting from the CEO, must support the extra training, resources, and steps needed to use a secure software development lifecycle. Additionally, every software developer must uphold his or her responsibility to take part in such improvement process.

2.2. Supply Chain Risks

Because of this growing concern for supply chain risk, institutions such as the National Institute of Standards and Technology has published management practices specifically focused on this area of concern and continues to refine these practices as experience in handling supply chain risk increases.

As mentioned, the final software product or service is an integration of various elements and each element carries the marks of the people, processes, and technologies used in its creation, and in many cases ongoing refinement and support. The term "ICT supply chain" or “Software Supply Chain” is used to describe those acquisition and outsourcing linkages.

Software products thereby inherit cybersecurity risks from upstream 3d party and open-source components, and the criticality of supply chain risk management is increasing because the volume of such upstream components is growing exponentially. Altogether, the significant number of upstream components part of a given technology stack and used for its development, considering all stakeholders and infrastructure, results in a significant attack surface. Additional challenges stand from the fact that organisation’s supply chains often span multiple countries and are dynamic multi-tiered and complex, making it difficult for an organisation to view all layers of its supply chain.

It is important to recognise that vulnerabilities can come through the supplier, through the product, through the software that is used to develop a product, such as language libraries, and integrated development environments, as well as through the mechanisms used to transfer the product from one organization to another. Supply chain vulnerabilities may be found in the systems/components within the software development lifecycle (i.e., being developed and integrated), the development and operational environment directly impacting the software development lifecycle and even the delivery environment that transports ICT systems and components (logically and physically).

[17] OWASP. Software Assurance Maturity Model (SAMM). https://owasp.org/www-project-samm/ [Last access on 29 January 2024]

[18] Building Security In Maturity Model (BSIMM). https://www.synopsys.com/software-integrity/software-security-services/bsimm-maturity-model.html [Last access on 29 January 2024]

[19] The Common Criteria. https://commoncriteriaportal.org/index.cfm

The Supply Chain Risk Management (SCRM) is a term to describe the implementation of the processes and practices needed to address this growing organizational concern. In order to be successful, an organization must integrate these practices into all of its ongoing acquisition activities.

The SCRM depends on the acquisition strategy, which essentially drives the structure of the supply chain. Acquisition strategy defines such supply-chain-related actions as which capabilities will be built internally or externally, how many vendors will be involved in the program, how the work will be allocated among those vendors and other related tasks. All of this impact who will be expected to address the various activities related to the software across the lifecycle, and its software assurance.

Therefore, it would be nice to consider different acquisition strategy that impacts in the practices, processes, and tools that the acquisition organisation should follow. For example, when talking about vendor supply software components we distinguish the following cases:

1 The contractor defines the requirements. Then, the contractor will also be determining where components will be sourced, what will be the acquisition strategy, and the actual acquirer will have only limited ability to influence these decisions based on the specific contract restrictions that they incorporate.

2 The acquirer defines the requirements and establishes criteria for the selection and management of suppliers. Specific clauses in the contract that restrict the kinds of suppliers and define how trust relationships can be set up.

3 The acquirer actually looks at fit-for-use solution, because the product already exists (COTS) and it is evaluated based on how the software component appropriately addresses a specific need. The acquirer will have really limited insight into the acquisition and the relationships with the subcontractors who will be building pieces of those products.

Downstream consumers have limited control about and limited visibility into the security posture of upstream components. For instance, whether and which security best-practices are applied during the software development lifecycle, and whether this is documented. In some cases this depends on the discretion of the respective development organisation or developer.

However, some good practices exist to help vendors meet and maintain rigorous security level, e.g. the Network Equipment Security Assurance Scheme (NESAS) - Security from GSMA. Still, some industries are more mature than other in applying those. Downstream consumers have limited control about and limited visibility into the security posture of upstream components. For instance, whether and which security best-practices are applied during the software development lifecycle, and whether this is documented. In some cases this depends on the discretion of the respective development organisation or developer.

This lack of visibility and traceability of ICT supply chains can lead to security risks impacting the confidentiality, integrity, or availability of information or information systems and reflect the potential adverse impacts to organisational operations (including mission, functions, image, or reputation), organisational assets, individuals, other organisations, and countries.

Those risks apply to all downstream consumers, direct or indirect, including the providers of commercial software and services as well as their customers. This is further aggravated by the high degree of automation with which upstream components are updated, downloaded, and installed, with little or no human intervention.

Some examples of software supply chain risks are insertion of counterfeits, unauthorised production, tampering, theft of software, insertion of malicious software and hardware (e.g., GPS tracking devices), the presence of known vulnerabilities and poor software development practices in the supply chain.

While consumers could theoretically vet each and every component by themselves, this is virtually impossible due to the large number of dependencies and versions used throughout the development. Furthermore, existing standards and metrics used for assessing the security maturity of open-source projects are not yet widely adopted.

After the assessment is completed, even if a consumer concluded to reject a given component due to security issues, this may be difficult for transitive dependencies, i.e., dependencies of dependencies, because this could require modifying the code of the direct dependant.

Finally, the security risk is also aggravated by the fact that many open-source projects only receive little funding and contributions, which makes it difficult to securely run their projects and renders them potentially more susceptible to social engineering attacks.

2.2.1. Existing Frameworks for a Secure SW Supply Chain

Enduring Security Framework “Securing the Software Supply Chain for Developers”	The framework “Securing the Software Supply Chain for Developers” targets developers and lists principles to secure the entire software development lifecycle (SDLC). It considers threats to the development of secure code, to the verification of third-party components, to the hardening of the build system, and to the delivering of code and provides recommended mitigations.
Microsoft Open Source Software (OSS) Secure Supply Chain (SSC)	Compared to the framework “Securing the Software Supply Chain for Developers”, the Microsoft Open Source Software (OSS) Secure Supply Chain (SSC) Framework only focuses on the secure consumption of open-source software, but provides a more detailed guidance in this context. Microsoft OSS SSC defines 8 practices (e.g., scan and update third-party components) and a list of associated operational requirements (e.g., scan OSS for malware, perform security reviews of OSS). It also comes with a maturity model that organizes the requirements into 4 different levels and lists tools that can support fulfilling the requirements.
OWASP Software Component Verification Standard (SCV)	Similarly to OSS SSC, the OWASP Software Component Verification Standard (SCV) provides 6 families of controls (e.g., inventory, pedigree and provenance) and 3 levels of verification requiring an increasing number of requirements for higher assurance. However, it applies to software components in general and thus the requirements are not specific for OSS consumed within the supply chain like in the case of the OSS SSC framework.
OpenSSF “Supply chain Levels for Software Artifacts (SLSA)”	Defining maturity levels is the main goal of the “Supply chain Levels for Software Artifacts (SLSA)” framework. Compared to Microsoft OSS SSC and OWASP SCV, SLSA has a narrower scope as it focuses on integrity, thus ensuring that the consumed code has not been tampered.
Taxonomy for open-source software supply chain attacks	The “taxonomy for open-source software supply chain attacks” created within the SPARTA project complements such efforts as it provides the most complete taxonomy of attacks whereas the frameworks above focus on safeguards by providing recommendations. The taxonomy takes the perspective of the attacker. Similar to what the MITRE ATT&CK framework does for attacks to infrastructures, it helps in describing how attackers can exploit the software supply chain to spread malwares. Having a complete taxonomy is key for establishing whether the identified recommendations cover the known attack vectors. By providing a base of knowledge related to attacks, it helps in designing and researching novel countermeasures.

TOOLS AND TECHNOLOGIES WE USE

3.

This section deeps dive on the common tools and technologies that developers deal with on a daily basis. In particular, the analysis focuses on the role and the importance of languages and of production platforms, trying to demystify their complexity as well as explaining their relevance in secure software environments.

Furthermore, the section discusses the main components of a computing infrastructure, linking those with their related libraries and applications. Finally, the popular software platforms, namely servers and databases, are reviewed in order to close the loop of this overview and providing a comprehensive understanding of how a modern software development ecosystem looks like in the real world.

3.1. Languages & Software Production Platforms

3.1.1. Compilers

Ken Thompson published a paper in 1984 called "Reflections on Trusting Trust"[20]. The ultimate point of the demonstration was that if a component of a system is implicitly trusted (compiler in his example), then compromising it would mean that the entire system can no longer be trusted. Is the vulnerability described in the paper relevant for today's software compilers? Trying to answer the question is just missing Ken Thompson's message: we cannot ignore the eventuality of compromised trusted system. He reminded that such an attack was practical to implement instead of just theoretical, which finds particular echo and resonates with more recent state-of-the-art cyber strategies like Zero-Trust Approach.

Furthermore, we should never lose sight of the fact that software flaws are rather originated by unintentional coding mistakes than malicious acts. Recently, four researchers in MIT's Computer Science and Artificial Intelligence Laboratory[21] studied a dozen common C/C++ compilers to see how they dealt with undefined code (such as dividing by zero, null pointer dereferencing or buffer overflows). Unlike other code, compiler writers are free to deal with undefined behaviour however they wish. Researchers found out that, over time, compilers are becoming more aggressive in how they deal with such code, more often simply removing it, even at default or low levels of optimization. Since C/C++ is fairly liberal about allowing undefined behaviour, it is more susceptible to subtle bugs and security threats as a result of unstable code.

[20] K. Thompson. "Reflections on Trusting Trust". Communications of the ACM – August 1984 – Volume 27 – Number 8 – Turing Award Lecture. doi: 10.1145/358198.358210

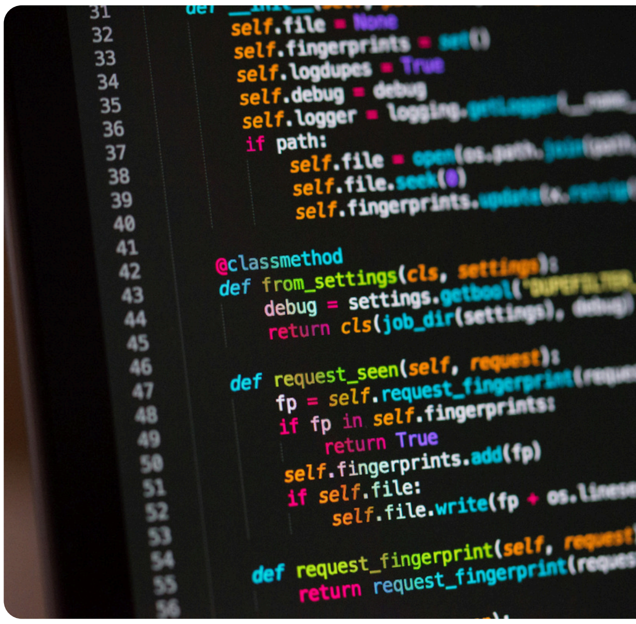
[21] X. Wang, N. Zeldovich, M. F. Kaashoek, and A. Solar-Lezama. "Towards optimization-safe systems: analyzing the impact of undefined behavior". In the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13). Pag. 260–275. doi: 10.1145/2517349.2522728

But let’s take a step back and remind what a compiler actually does and why it is an important piece in the context of secure software supply chain. Most people involved in the tech industry have sadly only a vague idea of how compilers transform human-readable code into the machine language actually used by computers. Here are some enlightenments:

1

The compiler sequentially goes through each source code file in the program and does two important tasks: first, it checks the lines of co-de to make sure they follow the rules of the language.

If they don’t, the compiler will give an error (and the corres-ponding line number) to help pinpoint what needs fixing. The compilation process will also be aborted until the error is fixed. Second, it translates the source code into a machine language file called an object file.



```
self.file = None
self.fingerprints = set()
self.logdupes = True
self.debug = debug
self.logger = logging.getLogger(__name
if path:
    self.file = open(os.path.join(path,
    self.file.seek(0)
    self.fingerprints.update(x.rstrip

@classmethod
def from_settings(cls, settings):
    debug = settings.getbool('DUPEFILTER_DEB
    return cls(job_dir(settings), debug)

def request_seen(self, request):
    fp = self.request_fingerprint(reque
    if fp in self.fingerprints:
        return True
    self.fingerprints.add(fp)
    if self.file:
        self.file.write(fp + os.linese

def request_fingerprint(self, request
    return request_fingerprint(reque
```

2

After the compiler creates one or more object files, then another program called the linker kicks in. The job of the linker is to take all the object files generated by the compiler and combine them into a single executable program, linking library files (a library file is a collection of precompiled code that has been “packaged up” for reuse in other programs.

And if needed, recent examples of vulnerabilities[22] emphasised how central was the role of libraries management in the secure software supply chain). Finally, the linker makes sure all cross-file dependencies are resolved properly. For example, if something is defined in one file, and then used in another file, the linker connects the two together. If the linker is unable to connect a reference to something with its definition, a linker error will be generated, and the linking process will abort.

[22] Early 2022, an open-source developer deliberately altered computer libraries, "faker.js" and "colors.js", on which he was working. Beyond a joke or a malicious act, this strike 2.0 points out to the precariousness of the open-source world. Another example is the critical Log4shell vulnerability affecting a broadly used library developed in open source by very few volunteers.

There is no need to further detail compiler and linker's activities to understand their primary role in the security chain of creating a software. The bright side of it is that, properly configured, compiler allows the detection of programming errors or dangerous use of the development language. Most of the compilers embed hardening features capable of significantly improving the security of the software developed. They offer different levels of warnings to inform the developer of the use of risky components or the presence of programming errors.

On the other hand, it is true that the level enabled by default should usually be increased, which requires to understand the compilation options used. For the same version of language, some default behaviours may vary from one compiler to another, or some warnings issued during compilation are different depending on compiler's version.

It is therefore essential to know exactly the compiler used, its version, but also all the options activated and why. Compilers should not be considered as a black-box bullet-proof trusted component but rather turned into a valuable tool and ally for better software supply chain security.

3.1.2. Software Development Platforms

If compiler is one of the core instruments used during software development lifecycle, the complete toolbox employed nowadays by developing teams is much broader.

This section addresses the tools entailed in code creation interface, namely the Integrated Development Environment (IDE) and also the platforms used for code storage (repository and version control). The tools supporting other activities such as build, and testing phases are covered in next sections.

Nowadays, developers are responsible for more than working on software features, they have to focus in the meantime on maintainability, scalability, reliability, and security. And that is why security functionalities (like static code analysis) are more and more inserted in developer's "to-do-almost everything-tool", the IDE. Most of IDEs today alert developers about potential issues such as a section of code not being reachable, a method never being called or SQL injection vulnerability. It enables to test code as early as possible in the chain, even before the build, based on concepts such as continuous integration (see Section 2.1).

Then, apart from IDEs, another crucial aspect when considering the software development, is ensure proper version and configuration management with an associated repository. Dedicated platforms and tools integrate components to enable track software changes during development and are designed for coordinating work among programmers. But as developers increasingly rely on these solutions to manage and store their source code, it is essential also to consider the importance of their security.

For instance, in the case of Web development, there are publicly available tools that enable complete downloads of the repository content if the repository directory is accessible. The repository metadata and content can give an attacker helpful information for further attacks. The repository not only potentially reveals the web page source code, but passwords, secret tokens or confidential customer data could also be exposed. Besides these, the attacker may also use this information to discover even more vulnerabilities which may escalate to more dangerous attacks, like database takeovers (using the hardcoded credentials, Time of Check and Time of Use (TOCTOU), and even Remote Code Execution, etc.

3.1.3. Software Validation Platforms

“Verification” and “Validation” are two widely and commonly used terms. According to the Capability Maturity Model Integration (CMMI)[23] for software engineering both terms are defined as follow: verification confirms that work products properly reflect the requirements specified for them. In other words, verification ensures that “you built it right.”; validation confirms that the product, as provided, will fulfill its intended use. In other words, validation ensures that “you built the right thing.” Even if these activities have obvious differences, in the context of this document, we will refer to them as one single activity since many tools can be used in both activities. Both activities will be called “software validation”.

Two major families exist when we want to test software:

Dynamic analysis

Refers to the analysis of running code. This is commonly referred as testing.

Static analysis

Refers to the analysis of code at fixed points durting its development.

On one hand we have the dynamic test which evaluates applications from the outside, as they execute. It finds accidental vulnerabilities looking at all the actual behaviours of a system, not just the expected behaviours. On the other hand, there is the static analysis, with its whitebox visibility, which is certainly the more thorough approach and may also prove to be more cost-efficient with the ability to detect bugs at an early phase of the software development life cycle. Performing code review such as static code analysis also provides the opportunity during software development to realize other important activities when it comes to software evaluation.

[23] CMMI Product Team, The. CMMI for Software Engineering, Version 1.1, Continuous Representation (CMMI-SW, V1.1, CMU/SEI-2002-TR-028. https://doi.org/10.1184/R1/6572396.v1

Software security analysis (validation) is a broader activity than only finding vulnerabilities. It plays an essential role in controlling Clean Code principles (Security, Maintainable, Readable and Reliable, Testable, Efficient and Portable). Moreover, one of the functions of a platform facilitating the development of combined software is the verification of the traceability of the code, which is a crucial component in the context of Secure Software Supply Chain. Traceability tends to answer the following questions: Who developed these specific lines of code? For which functionality are these lines developed? Who tested them?

A software validation platform also plays a role when we consider external partnership (commercial or open source) helping to identify outdated external component and thus avoiding their usage. Even if it is not the role of software validation platforms to create formal, machine-readable inventory of software components and dependencies, they can well contribute to the verification of SBOM (Software Bill of Material) accuracy.

3.1.4. Security Analysis Platforms

After having discussed Software validation tools, this section focuses more on the specific activities of finding vulnerabilities, or security analysis, in particular static and dynamic analysis are discussed and the available tools.

First, source code analysis tools or Static Application Security Testing (SAST) Tools are used to find security flaws, for instance buffer overflows or SQL injection flaws. Such tools are useful to highlight problematic code and to help fix it easily. However, efficient tools that target security flaws based on secure coding rules (CERT, CWE) are rare. Indeed, a lot of static tools are only linter or extended-grepper to find common vulnerabilities and based only on syntactic search on code.

Moreover, static analysis tools are in general not efficient in finding certain flaws based on weak cryptography and authentication or access control issues. Also, these tools are not easy to use, and they pose some challenges when it comes to dealing with a high number of false positives at the end of the analysis. As previously mentioned, another challenge of SASTs consists in the possibility of accessing in full all the necessary code to compile (libraries, compilation options...). Therefore, especially when open-source libraries are embedded in the development process, SCA (software compositional analysis) tools are a necessary step. SCA are tools which allows the identification of open-source libraries used in a software and to list, for each identified library, the set of known vulnerabilities for the associated version.

On the other hand, Dynamic code analysis or dynamic application security testing (DAST) are tools for identifying both compile-time and run-time vulnerabilities, such as configuration errors that appear only in a realistic execution environment. DAST tools use known vulnerabilities or malicious data to test software, like a fuzzer would do. Typical malicious inputs are, for example, long input strings and unexpected input data. The idea behind a fuzzer is to “bombard” the software with this kind of malicious data to cause a crash or obtain useful information to exploit a vulnerability.

Overall, DAST analysis are easy to automate. They operate on the running software and can detect a wide range of vulnerabilities with a low effort, but they require full availability of the application. DAST tools are typically used during the testing phase of software development and fit well with continuous integration and delivery workflows.

KEY TAKEAWAY /ASSOCIATED RISK

Nowadays developers use a varied and complex mix of tools to perform their work. Among those, compilers, IDE platform, DAST and SAST tools are the most used ones. Overall, such tools shouldn't be seen as black boxes as their correct usage and configuration can make the difference in the security posture of a software system.

Failure to know the functioning of these tools, their purpose, their strength and weaknesses, will prevent organisation from choosing the right ones, from configuring them correctly and it will be making these technologies as an extension of the attack surface rather than an allied in defence.

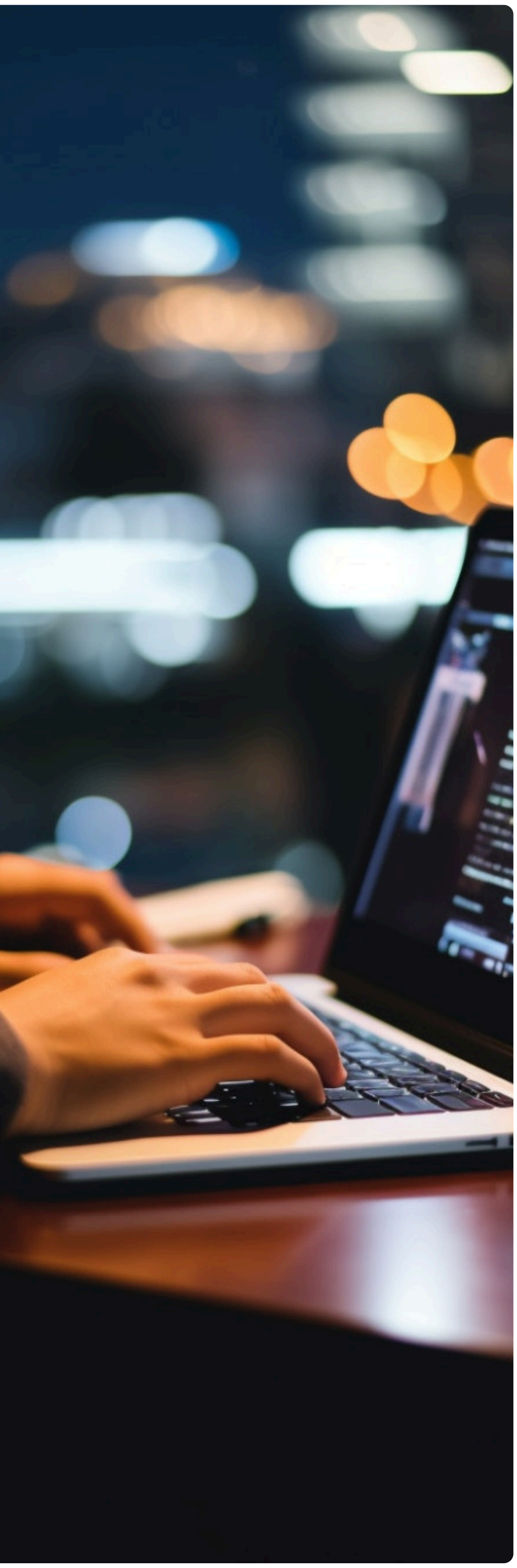
3.2 Computing Infrastructures

3.2.1. Operating Systems

Operating systems are the cornerstone on which each software runs. Due to the complexity of hardware architectures and interfaces, operating systems have become increasingly complex over the years. It is an extremely hard market, both commercially and technically, where only very few mainstream operating systems exist (Windows, Linux, MacOS) but where operating system-like platforms in embedded systems are flourishing. Software systems cannot run without an operating system.

As a result, the technical skills to develop and maintain operating systems are also in short supply, and they tend to be attracted to major companies. The first risk is thus to lose the capability to use and maintain these operating systems.

An associated risk is that there are very few operating system developers in Europe. Major operating systems are owned by US-based companies (Microsoft for Windows, Apple for IOS and MacOS, Google for Android) or supported by US-based organisations (The Linux Foundation for Linux). The only potential alternative is the China-manufactured smartphone ecosystem, where the US ban on technologies has forced Chinese companies to develop alternative operating systems for their platforms.



Europe has not had this necessity to develop its own operating system technologies, apart possibly for niche markets (e.g. RIOT), and as a result is at risk of losing access to operating systems or to the scientific and technical capabilities to develop such technologies. As a companion to the Chips act, Europe should ensure a continuous capability to source operating systems, either by ensuring a reliable open source supply chain, or by developing the next generation of these technologies in Europe

3.2.2. Servers and Cloud Environments

One of the key aspects of server and cloud environments are all the management and orchestration software that is required for them to work properly. Network operating systems such as ONOS, Ryu, OpenDayLight or Floodlight are one of the key technologies to ensure the proper management of software-defined networks. Orchestrators and cloud management software such as Kubernetes, Proxmox or Openstack are critical to the operation of datacentres. Function as a Service (FaaS) runtimes such as OpenWhisk or OpenFaaS are critical to the proper delivery of computing functions to customers.

These software-based technologies are crucial to the development of future networks, clouds, and servers. They are often delivered under open-source licences but are strongly supported by private entities that are both technology users, technology providers and developers, and are acting as standard-setting organisations in an informal manner. They are also highly attractive organisations for talented individuals and are creating a brain drain that reduces access to these environments.

The key risks are thus to either loose access to the technology (if the source code is removed, the repository destroyed or altered, etc), or loose the capability to understand, deploy and efficiently operate these technologies independently of technology providers.

3.2.3. Edge and IoT

On the other side, edge computing is becoming highly relevant for modern environments, including sensing capabilities, remote control and actuating, and local computing and storage. These environments are extremely diverse in terms of form factor, capabilities, energy. The Edge and IoT ecosystem is thus extremely diverse in terms of technologies, manufacturers, OEMs, vendors, resellers and integrators. Europe is a player in IoT, for example with the RIOT[24] operating system in open source, and with technology providers such as Bosch, Schneider, etc. It needs to preserve access and ownership of these technologies, by promoting them and the associated standards, and ensuring that they remain on one hand anchored in Europe, on the other hand widely open to contributions from outside.

KEY TAKEAWAY /ASSOCIATED RISK

Access to technologies: Europe needs to ensure that it has access to required technologies, either by developing commercial alternatives or supporting open-source alternatives to technologies not directly available in the EU. Such technologies include operating systems, virtualization and containerization platforms, and management platforms for future clouds and networks.

For the specific case of open source, it needs to ensure access in the long term, for example by hosting copies of open-source repositories. The Software Heritage project[25] is an example of such repositories that might help in ensuring access to the code. One of the downsides of this is the delay that software updates may suffer.

Mastering technologies and talent retention: in addition to code, Europe must preserve the human resources capable of mastering these technologies to deliver the services it requires.

Trend setting and R&D: Europe must regain the capability to deliver innovative software solutions in these technological domains, stop being a follower and develop advanced methods and techniques that comply with its values.

[24] RIOT - The friendly Operating System for the Internet of Things. https://www.riot-os.org/ [Last access on 30 January 2024]
[25] The Software Heritage project. https://www.softwareheritage.org/ [Last access on 30 January 2024]

3.3 Libraries / Applications

3.3.1. Major Software Libraries

Many applications rely on libraries for specific functions. This is not a new trend, as for example C code has included the C standard library for decades, and the C compiler without the C library is close to useless.

This trend has exacerbated in recent world. Major examples include web development frameworks such as Apache Cordova, Django or Spark, web content management systems such as Drupal, Wordpress or Joomla, scientific libraries such as TensorFlow or Scikit Learn, graphical development tools such as React, JQuery, NodeJS or AngulaJS, open-source application frameworks such as Spring, data stores such as Hadoop. These few examples demonstrate the diversity of use, multiplicity of languages and necessity of use of software libraries. Most programming environments (Python, Perl, Java, Javascript, Rust, …) have extremely large and organized software library repositories on which developers heavily rely.

The key risks are thus to either loose access to the technology (if the source code is removed, the repository destroyed or altered, etc), or loose the capability to understand, deploy and efficiently operate these technologies independently of library owners.

3.3.2. Middleware

In addition to these libraries, running code relies on many pieces of middleware to execute, ensuring functions such as routing, naming, message passing, publish and subscribe. There exist several platforms, protocols and brokers that are widely used in industry to build complex applications and information systems. Examples include Apache Kafka, MQTT, IBM Websphere, JBoss or Oracle Fusion. As one can see from these examples, several of these middleware platforms are commercial, and only a few are freely available.

KEY TAKEAWAY /ASSOCIATED RISK

Access to libraries: similarly to operating systems and virtualisation platforms, Europe needs to maintain access to library or middleware code.

Understanding of code: in addition to code, Europe must preserve the human resources capable of mastering these libraries and middleware platforms to deliver the services it requires.

3.4. Associated Software Platforms

3.4.1. Databases

Over the past decade, the dynamic evolution of database technologies, with a keen focus on MySQL and Elasticsearch, has significantly benefited software developers. MySQL, a reference in the domain of relational databases, underwent transformative updates over the last decade. The last releases not only bolstered performance but also introduced vital security enhancements, providing developers with a more robust and feature-rich platform.

The concerns surrounding MySQL's acquisition by Oracle Corporation spurred the emergence of MariaDB, a fork that has ensured the continuous development of MySQL as an open-source database. This development has empowered developers with the freedom to choose a database solution aligned with their preferences and open-source principles.

Elasticsearch, a powerful distributed search and analytics engine, has also become integral to developers' toolkits. Its evolution beyond simple text search to encompass diverse use cases, such as log and event data analysis, has empowered developers to create sophisticated and efficient applications. The seamless integration of Elasticsearch into the ELK (Elasticsearch, Logstash, Kibana) stack has simplified log processing and visualization, offering developers a cohesive and user-friendly environment.

Moreover, the raise of cloud-native technologies has brought further convenience to developers working with MySQL and Elasticsearch. Managed services provided by Cloud players have streamlined deployment and scalability, allowing developers to focus more on application logic and less on infrastructure management. Containerization technologies, particularly Docker, have facilitated smoother deployment and management of database instances.

Another interesting element to notice when considering associated platforms is the great usage made by developers of real-time analytics and data processing. They have become paramount in modern applications, and both MySQL and Elasticsearch have responded adeptly. More functionalities dedicated to availability and low latency, have empowered developers to build responsive and scalable applications.

Eventually security, a major concern, has seen significant improvements in both MySQL and Elasticsearch. Commitment of both technologies to security is reflected by the implementation of features like role-based access control, providing developers with the confidence to manage sensitive data securely.

In summary, the past decade's evolution in database technologies, particularly within MySQL and Elasticsearch, has significantly empowered software developers. From enhanced performance and security measures to simplified deployment through cloud-native solutions, developers now enjoy a more positive and productive environment for their daily work, enabling them to focus on creating innovative and efficient applications.

3.4.2. Web servers

Web server technologies, exemplified by Apache, continue to play a pivotal role in software development mainly due to their fundamental function in serving web content and applications.

Even though they suffered from their potential lack of adaptability, modular architecture, and extensive community support, developers still rely on these established servers, including Nginx and Microsoft's IIS, for their stability and versatile capabilities. One major advantage is still that they allow seamless deployment across various operating systems.

The integration of web servers with cloud technologies has even further solidified their relevance. Apache, for instance, is facilitated by cloud technologies due to its compatibility with containerization, like Docker, and orchestration tools such as Kubernetes. Many cloud providers today provide the necessary tools and features to ease their usage.

This strategy from cloud technologies providers to propose web servers streamlined deployment is beneficial for many software developers. It enables consistent and scalable application hosting across diverse cloud environments. In a sense, cloud services provide the infrastructure agility that complements web servers, enhancing their efficiency and enabling developers to leverage the benefits of both established server technologies and modern cloud ecosystems.

In this symbiotic relationship, web servers remain a major element in software development, bridging the gap between traditional and cloud-native application architectures.

KEY TAKEAWAY /ASSOCIATED RISK

Database and web server technologies have evolved in recent years and have significantly facilitated the work of developers. Such development, coupled with the usage of cloud-native technologies, has enhanced the creation of modern and scalable applications. Security remains a focus and it is assured by specific features tasked at providing the necessary confidence to the software development ecosystem.

RECOMMENDATIONS

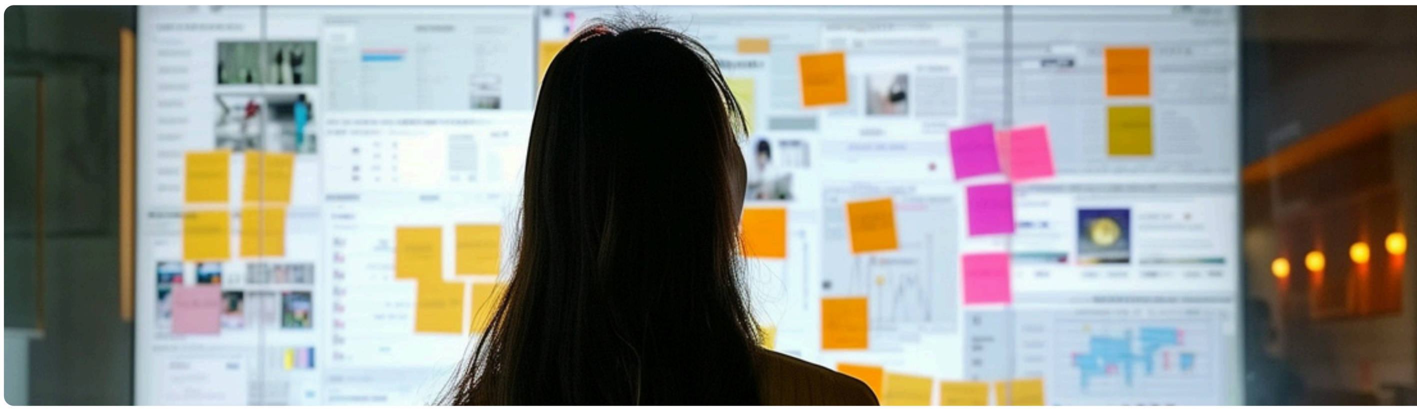
4.

Building on the previous analysis, this section presents a set of recommendations to help organisation secure their software supply chain. As described in Section 2, the development of modern software involves a wide variety of environments, tools, services, and components. To prevent software security vulnerabilities, defend against cyberattacks, and allow for regulatory compliance, security need to be considered in every step and aspect of the software supply chain, and every stakeholder must contribute.

Most of the codebase of an application is composed by 3rd party and open-source software. This means that the development of secure software does not only require to implement security measures and follow secure coding best practices for the newly developed code but also to securely handle the acquisition of the consumed components.

At the same time, the security of each system implementing the supply chain (cf. Figure 1) has to be handled following best practices (e.g. isolating build environments) and through hardening. In fact, compromising a vulnerable system involved in the development, build, and delivery could allow the injection of malicious code into the built software and affect all downstream users.

In order to be successful, an organization must integrate secure software development practices into all of its ongoing procurement activities. The following paragraphs contain some recommendations on how to do so, including both practices that are available and ready to use as well as research and innovation needs.



4.1. Frameworks and Development practices for a Secure SW Supply Chain

RECOMMENDATION 1

Clarify and consolidate frameworks for a secure software supply chain.

The OSS SSC framework includes a mapping of requirements to other relevant specifications including OWASP SCV and SLSA. However, a more in-depth comparison is needed especially with respect to the maturity levels defined. Due to the overlap among the different frameworks, a consolidation effort is needed to provide a unified guidance clearly defining cross framework requirement mappings and how SLSA levels, OSS SCC maturity levels, and OWASP SCV level should be used.

RECOMMENDATION 2

Applying secure software development practices (short-term view).

It is important to keep working toward strengthening the existing guidelines for developers on the correct way to write/consume code; such guidelines should be strong and actionable. Moreover, it would be beneficial to make freely accessible tools for checking compliance with the right coding guidelines and ensure that they are largely deployed in all CI/CD pipelines (i.e., make it impossible for any code going through CI/CD not to be verified).

Another short-term recommendation would be to promote the use of security requirements specifications, as well as the risk analysis on these requirements (a.k.a. manifests), in order to make security visible to the end users, and clearly explain the needs of each software.

Finally, it is always important to leverage the knowledge of past code mistakes while building code checking tools. In the same way, existing standards (e.g. TCP/IP, X.509, PDF, …) should be reviewed and rewritten when weaknesses are found.

RECOMMENDATION 3

Applying secure software development practices (long-term view).

Modern computer languages offer enhanced security functions, such as out-of-bounds checking, memory safety, dynamic and static type safety, more secure concurrency models, immutability, sandboxing, and isolation. These features make such languages ideal for new security-critical developments, while languages that allow unrestricted memory access and lack compile-time and runtime type checking should be used with caution. Notable examples of modern languages with strong security features include Rust, Go, Java, C# and Swift. In contrast, languages such as C/C++ and, to some extent, dynamic languages like PHP and JavaScript pose greater security challenges due to issues like memory corruption, buffer overflows, race conditions, unsafe input handling, dynamic code evaluation, and limited static analysis capabilities.

RECOMMENDATION 4

Maintain and update software securely.

There should be some form of automatic trigger notification after some time that update verification hasn’t been active in a software. Such trigger should block the usage of the software in question. Also, further development should be made in the use of code signature, updates and development of a general architecture/service to maintain and update software securely. Quite importantly, the update processes should include vetting management mechanisms with the intent to address attackers exploiting the common belief of always and quickly updating to latest versions to then ship malicious versions of open-source artifacts.

Finally, attention should be paid on hight test coverage of implemented functions, to limit regressions in future development and maintenance while improving efficiency. Failing to do so won't guarantee that the code will still be running in the intended way after an update.

4.2. How to reduce the risk exposure

Section 2.2 highlighted a new thread of risks to the software supply chain originating from the reliance on 3rd party and open-source components. The lack of transparency and traceability was identified as a major obstacle.

RECOMMENDATION 5

Create and maintain a Software Bill of Materials (SBOM) in a machinereadable format, and use Software Composition Analysis tools.

An effort to increase transparency is the software bill of materials (SBOM). An SBOM is a list of components related to a given software artifact, comparable to the list of ingredients on food packaging.

Maintaining an inventory of components used in the developed software is key for its maintenance and auditability. Moreover, the availability of SBOM data for components to be consumed allows for verifications that can be exercised by downstream users, depending on the content and properties of SBOMs. For instance, some specifications for SBOMs suitable for automated analysis requires a machine-readable format, a timestamp, a signature, and individual components to be identifiable using consistent naming schemes and digests (to describe their provenance and pedigree).

The creation of SBOMs can be automated. SPDX, CycloneDX, and SWID are prominent machine-readable standards and formats for SBOMs. A detailed software bill of materials must be produced and maintained by the project members (cf. Figure 2), ideally using automated Software Composition Analysis (SCA) tools. To avoid tampering, the SBOM must be securely hosted and distributed by package repositories, and carefully checked by downstream users regarding their security, quality, and licence requirements.

On top of producing SBOMs, Software Composition Analysis tools can automate the inspection of the components of a software product. Hence, they are an important step in CI/CD pipelines to match desired security requirements (e.g., integrity checks, quality and security metrics verification, provenance etc.).

RECOMMENDATION 6

Design the build process to be reproducible.

In the context of ensuring the integrity of built software, the build process should be designed to be reproducible. This means that all the results for every build starting from a given source code are identical. Who is producing the software should describe all the requirements and steps to reproduce the build, while who delivers the software should securely provide such information. This allows consumers to verify that there are no discrepancies, and that no malicious code was injected during the build process.

RECOMMENDATION 7

Maintain an inventory of all systems and software used throughout the development lifecycle.

Not only it is important to have an inventory of the developed software, including both proprietary code as well as its 3rd party and open-source dependencies, but also all the systems and software used throughout the development lifecycle should be reflected in an accurate inventory. Patch management is required to make sure that those are not only up-to-date but also free of malicious and vulnerable code that could be exploited to compromise project resources, e.g., source code, build configuration or binary artifacts.

It depends on the respective component or ecosystem whether automated update and vetting mechanisms are available, and individual risk analyses can be employed to decide whether such mechanisms or manual updates are preferred. Blindly updating in an automated fashion (e.g., specifying version ranges) should be avoided since, some attack examples misused this feature to deliver malicious versions to downstream users. While exact version specifications prevent upgrade or downgrade attacks, they cannot prevent a compromised index from serving a malicious package having the same version identifier.

RECOMMENDATION 8

Create and maintain a European trusted repository of vetted open-source components.

Maintaining an up-to-date and vetted library of open-source components in Europe as a distributed and secure repository, would allow for better control over the consumed components and may prevent attacks that tamper packages in public hosting systems. Having such library in the form of a trusted, verified repository of source code, both package repository administrators and consumers can opt for building packages directly from such source code, rather than accepting pre-built artifacts.

Maintainer of package repositories would reduce the risk of hosting components originated by subverted project builds, and consumers would eliminate all risks related to the compromise of 3rd-party build services and package repositories. Alternatively, having a European library of vetted components, may reduce the risks of consuming subverted legitimate packages while sharing the effort of establishment of internal repository.

RECOMMENDATION 9

Define and use metrics to assess the security posture of open-source software components across different dimensions.

Security, quality, and health metrics can be used to assess the security posture of open-source software components and the security risks resulting from their use. Such metrics may consider, for instance, information about the lifelines of a project or whether given security best-practices are applied. They help end-users to decide which components to consume by making them aware of the security implications and can be computed prior to the first use but also on a continuous basis for used dependencies.

The OpenSSF Scorecards and the Fosstar rating core are examples of frameworks to compute a security rating for open-source components. Next to supporting downstream users in selecting components, they also aim at improving the security best practices of open-source projects. As the rating comes from the evaluation of different metrics, they can be used to highlight specific areas to be improved and provide greater visibility on the security posture of projects. The SLSA levels of assurance are another means to measure the security posture of an open-source project and provide visibility about the best-practices and security guidelines adopted.

RECOMMENDATION 10

Take responsibility for your contribution to supply chain security, acknowledging your role in the software ecosystem.

Reducing the security risks in the software supply chain also requires organizational commitment and investments. All involved stakeholders must cooperate as the successful implementation of some recommendation requires contributions by different actors.

As an example, code signing enforces binary and application integrity, hence ensures that a program comes from a valid source (authenticity) and that has not been modified since it was signed (integrity). To make it effective, project members should digitally sign the packages when uploading them to a package repository, administrators of package repositories should enforce code signing either providing signing functionalities or verifying the signatures, and consumers should assess the integrity of the downloaded packages and that the provenance is a trusted source by checking the signatures.

Organisations should thus support and provide trainings and resources to develop the required skills. Moreover, consumers of open-source components should also consider investing in their maintenance via funding and contributions to ensure projects can be managed securely both in terms of applied best practices and processes as well as increasing robustness against attack (e.g., social-engineering attacks).

4.3. Where the innovations are needed (concrete innovation)

Though several efforts are being made to provide structured guidance on how to secure the software supply chain, still the complexity of today's supply chains (and open-source supply chains in particular) results in a significant attack surface that requires to be further investigated.

RECOMMENDATION 11

Automate the vetting process for open-source artefacts.

Attackers have numerous opportunities for injecting malicious code into open-source artifacts, and mandated countermeasures usually address the known attack vectors. Vetting components in an automated and efficient way is thus key for the early identification of malicious code and for being able to anticipate attacks exploiting new vectors not yet covered by commonly used safeguards. Existing works for the detection of malware need to be extended and adjusted to the characteristics of malicious packages used in open-source supply chain attacks as they are usually characterized by a small fraction of harmful code hidden in a bigger corpus of legitimate code.

RECOMMENDATION 12

Advance software composition analysis to include an indication of the level of assurance.

The dependency of software products on open-source components that may be subject to known vulnerabilities also need additional support. The existing software composition analysis (SCA) tools represent an important first step, however they lack information about the level of assurance they provide in terms of precision and recall. In fact, a secure software supply chain should not only prescribe the usage of SCA tools but should require a level of assurance from the tool being used.

RECOMMENDATION 13

Support community efforts to provide code-level data sets to facilitate AIbased approaches.

Research in the area of vulnerability and malicious code would greatly benefit of a community effort for the creation and maintenance of code-level datasets that would enable AI-based approaches and the comparison of different approaches.

The use of Artificial Intelligence (AI) is being explored for the development of cybersecure products during the lifecycle of software systems engineering. AI can be used for the collection of cybersecurity requirements, automatic generation of cybersecure software, testing and verification phase as well as for the estimation of the efforts necessary to build cybersecure components.

RECOMMENDATION 14

Prioritise investigations into attack surface reduction.

Another promising direction to be investigated is the idea of reducing the attack surface of software products by removing all dependencies that are not actually needed. Software developers declare direct dependencies on open-source components whose functionality they want to use in the software under development or during development, e.g. compile dependencies or build plugins.

Those dependencies have their own dependencies, so-called indirect or "transitive dependencies", all of which are automatically resolved and downloaded by package managers. However, some of the transitive dependencies may not be needed in the specific development context, e.g. because a certain functionality of a direct dependency is not used. While the removal of such "bloated dependencies" can reduce the supply chain's attack surface, their identification is not straight-forward, e.g. due to dynamic programming features.

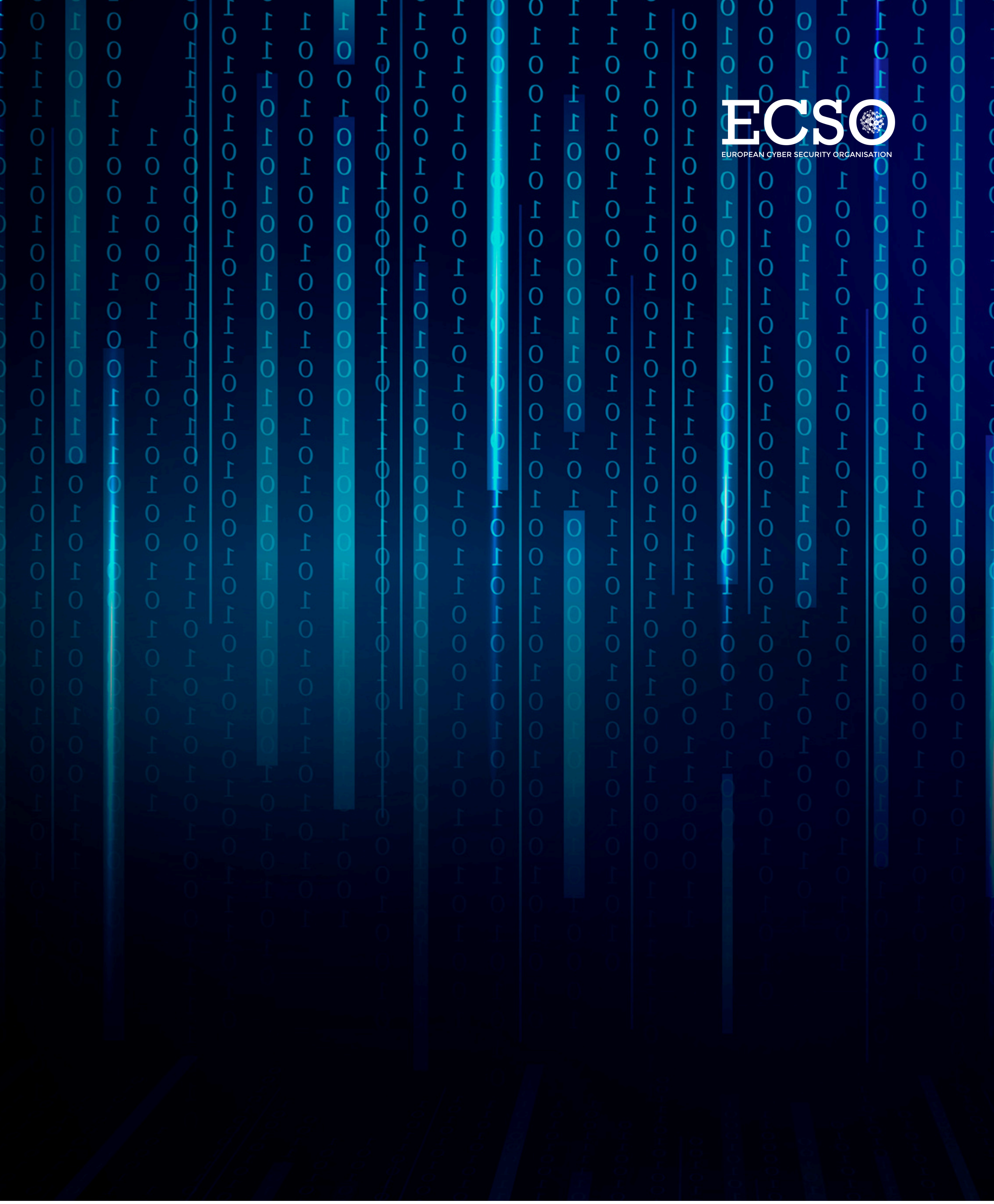
AKNOWLEDGEMENTS

5.

The European Cybersecurity Organisation’s (ECSO) WG6 aims to contribute to define the cyber security EU R&I roadmap and vision to strengthen and build a resilient EU ecosystem. From the analysis of the challenges of digitalisation of the society and industrial sectors this WG identifies what are the capacities and capabilities to sustain EU digital autonomy by developing and fostering trusted technologies.

The following is a special acknowledgement of the active contributions in various capacities from ECSO WG6 members.

EXPERT CONTRIBUTIONS: Ana Ayerbe (Tecnalia), Anett Madi-Nator (Cyber Services), Costanza Pestarino (ECSO), Henrik Plate (SAP), Herve Debar (IMT), Matteo Mole (ECSO), Patricia Mouy (CEA), Roberto Cascella (ECSO), Serena Ponta (SAP), Yoann Klein (Huawei), Volkmar Lotz (SAP)



ECSO
EUROPEAN CYBER SECURITY ORGANISATION

Avenue des Arts 46, 1000, Brussels (Belgium)
wg6_secretariat@ecs-org.eu
www.ecs-org.eu