ICITPM: Integrity validation of software in iterative Continuous Integration through the use of Trusted Platform Module (TPM)

Antonio Muñoz[1], Aristeidis Farao[2],
Jordy Ryan Casas Correia[1], and Christos Xenakis[3]

[1] Computer Science Department, University of Malaga Campus de Teatinos s/n, 29071,Malaga, Spain
amunoz@lcc.uma.es ryan@uma.es
[2] Neurosoft S.A., Athens, Greece
a.farao@neurosoft.gr
[3] Department of Digital Systems, University of Piraeus, Greece
xenakis@unipi.gr

Abstract. Software development has passed from being rigid and not very flexible, to be automated with constant changes. This happens due to the creation of continuous integration and delivery environments. Nevertheless, developers often rely on such environments due to the large number of amenities they offer. They focus on authentication only, without taking into consideration other aspects of security such as the integrity of the source code and of the compiled binaries. The source code of a software project must not be maliciously modified. Notwithstanding, there is no safe method to verify that its integrity has not been violated. Trusted computing technology, in particular, the Trusted Platform Module (TPM) can be used to implement that secure method.

Keywords: CI/CD pipeline · code integrity · Trusted Computing · TPM

1 Introduction

We are witness to the increasing adoption of development tools. These building tools are the Agile, the Development and Operations (DevOps) and the Continuous integration/continuous delivery (CI/CD). Automation is a key aspect in both of them to build, deliver, and test high-frequent increments of features [11],[20],[35].

We share the perspective of DevOps practices intended for optimizing times between change commitment to the system and change implemented in normal production code. While Agile practice focuses on eliminating processes. Thus, today CI/CD pipeline is considered among the best practices for delivering code changes more frequently and reliably in code implementation. Continuous integration (CI) and continuous delivery (CD) embody a culture, set of operating principles, and collection of practices that enable application development teams to deliver code changes more frequently and reliably.

CI/CD is one of the best practices for DevOps teams due to CI/CD automated enabling software developers to dedicate themselves to collection accurate requirements, security, and improved codes. The major challenges, which specifically concern security in a CI/CD pipeline are the following:

- *Automation* is frequently used, withal the most times is not integrated with security tools; lack of these tools is an event that leads to malicious behaviors requiring alerts.
- *Accountability* in CI/CD pipelines is a vital dependency that ensures transparency among developers' actions, e.g., Man-In-The-Middle, [21].
- *Low performance of security tools* leads to delay the process of automation; developer's daily commitments demand real-time interactions and results.
- *D*eploying virtual machines leads to a single-point-of-failure, enabling adversaries to execute attacks without any trace if they get control over the Hypervisor, e.g., "hyperjacking" [32].

Notwithstanding, the CI/CD technique entails the next future consequences resulting in information leakage in a big corporation DataBase. Based on the above challenges, this paper presents a solution for an identified gap to grant integrity in the CI/CD process. We proposed a mechanism for integrity validation of software in iterative continuous integration based on secure hardware elements. Our solution uses the Trusted Computing technology, in particular, we have designed the proposed solution using the Infineon Trusted Platform Module (TPM) 2.0 version [9]. A gap in the whole process to grant software integrity in CI/CD process has identified. Particularly between Assembly and Testing Server communications as we further describe.

It emphasizes in on-demand deployments cases on developer premises. We have developed a tool that from TPM functionalities assists in the CI/CD process to bridge the identified gap granting integrity in the whole process. This tool enables alerts about any change in the software from the final development to deployment versions. For this purpose, we provide a review of agile methodologies and the importance of security in automation. Then, a description of security risks in assembly and test server is provided, including current security threats identified, implementation of a scenario as a proof of concept with some threats, and our approach as a solution to the considered problem. TPM functionalities are used to grant software and platform integrity as seal-bind key, remote attestation, and signing functions.

The rest of the paper is organized as follows. In Section 2, we present the background and give a high-level description of the utilized agile methodologies and their security role in automation. Next, Section 3 presents the related work. Next, Section 4 describes the security role in automation. Section 5 presents a proof-of-concept on a vulnerable Jenkins server. Section 6 is dedicated to describe the code integrity challenge in the CI/CD pipeline we have afforded. Section 7 presents our proposal the Trusted Integrity Platform (TIP), while Section 8 demonstrates the implementation setup, the tested scenario, and the results. Finally, Section 9 summarizes the paper and presents future work.

2 Background

CI can be understood as those guided practices that enable continuous surveillance in code repositories allowing development teams to implement changes in code and their check in. While they require mechanisms for the integration and validation of code changes derived from multi-platform features from contemporary applications. Technically, we can define CI's main goal as a set of tools to build, package and test applications in an automated and consistent way. This consistency implies teams increase the frequency in code changes commitments triggering improved team collaborations and increased quality in software.

CD technique binds at the end of CI performing an automation in application delivery to particular infrastructures. It is widely extended the use of different environments (i.e. production, development and testing) with code changes submission between them. CD provides an automated way to actually perform those changes, keeping stored packaging parameters bound to every delivery. We aimed to be an automated process since any service calls to databases, web servers, and any other services to be resumed, restarted, stopped or followed to deploy the application must be automatically performed.

We have previously mentioned that the objective is delivering quality code and applications to users, for this reason CI/CD demands constant testing, which is generally offered as performance, regression and other set of tests done within CI/CD pipeline. Developers submit their code to commit into the version control repository. Also it is usual to establish a minimal rate of daily committing code per team to facilitate tasks in identifying defects and bugs on smaller delta pieces of code rather than large developments. Also, working on smaller commit cycles reduces parallel working on the same code in multiple developer teams. Many teams that implement continuous integration often start with version control configuration and practice definitions. Even though checking in code is done frequently, features and fixes are implemented on both short and longer time frames.

Different techniques are used to control and filter code for production in CI. Among the most applied practices is to require developers run regression tests in their own environments, which implies that only code that passed regression tests were committed. We notice that it is common that development teams have at least one development and testing environment. This environment allows reviewing and testing application changes. A CI/CD tool such as Jenkins[1], CircleCI[2], AWS CodeBuild[3], Azure DevOps[4], Atlassian Bamboo[5], or Travis CI[6] is used to automate the steps and provide reporting.

A typical CD pipeline [20] includes the following stages: (i) built; (ii) test and (iii) deploy. Nonetheless, improved pipelines include also the following stages:

- Picking code from version control and executing a build.
- Allowing any automated action such as restarting or shutting down both cloud infrastructure, services or service endpoints.
- Moving code to the target computing environment.
- Setting up and managing environment variables.
- Enabling services as API services, database services or web servers to be pushed to application components.
- Allowing rollback environments and the execution of continuous tests.
- Alerting on delivery state and data log are provided.

[1] https://jenkins.io/
[2] https://circleci.com/
[3] https://aws.amazon.com/es/codebuild/
[4] https://docs.microsoft.com/en-us/azure/devops/?view=azure-devops
[5] https://www.atlassian.com/software/bamboo
[6] https://travis-ci.com/

Jenkins provides files to manage records about building, testing and deploying stages that describe their pipelines. Those files define keys, certificates, environment variables, and other parameters. The CI/CD technique provides mechanisms to the misalignment between developers with operations. We notice that developers push frequent changes in their codes and operations while they expect stable applications after changes. However, they can push these changes with automated procedures to achieve stability in their operations. With standardized environment configurations, rollback procedures are automated and provide continuous testing in delivery, and separation of environment variables from applications.

Additionally, there are mechanisms to measure the impact of implementing the CI/CD pipeline, like as a key performance indicator (KPI). Nevertheless, we have to consider that KPIs can change the lead time, and the mean time to recovery (MTTR) from an incident when CI/CD with the implementation of continuous testing [22]. We have identified a gap of security in the CI/CD pipeline. In particular, as it is next briefly described, a threat in the code integrity. Also we have proposed an approach for that based on Trusted Computing advanced security features.

3 Related Work

Most of tools used in CD pipeline are web-based applications as Jenkins. Several approaches have focused on detecting vulnerabilities. Deepa et al. [15] provide approaches for Securing web applications from injection and logic vulnerabilities. Other solutions [19] follow approaches based on static analysis and runtime Protection and mitigation of vulnerability impact based on security testing techniques [24].

OWASP [29]proposes threat modeling focused on detecting threats and vulnerabilities as early as possible. The most widely used threat modeling method is known as STRIDE [18]. Secure DevOps [**?**,28]is a set of tools designed to help organizations implant secure coding in the CD process. Lipke [25]studied threats in CD pipeline using STRIDE methodology. This work implements a proof of concept based on Docker. Some approaches follow detection of vulnerabilities in CD pipeline applications as Bird [13]. Schneider [33] proposes a four-staged dynamic security scanning methodology (pre-authentication scanning, post-authentication scanning, backend scanning and scanning workflows specific to the targeted application). Also, this author introduces the SecDevOps Maturity Model(SDOMM). This can be considered as instrunctions for automatically achieve particular security aspects in CI pipeline. Kuusela proposes different testing techniques [23]. There are techniques based on improve the security of CD pipelines as Bass et al. [10] approach that proposes an engineering process within trusted components embedded in parts of the pipeline, which is intimately related to our approach although no trusted hardware is mentioned. In [36] have applied different tactics of security between CD components communications with encouraging results. Rimba et al. [30] present an approach based on the use of composing patterns to address security issues in CD pipeline.

Regarding the security tools the OWASP Zed Attack Proxy (ZAP) [12] is an open source security test initially designed as a security application and as a professional tool for penetration testing. Behaviour Driven Development Security [37] is a security framework for self-verifying testing by your own security specifications using a natural language in terms of “given that”, “when”, “then”, etc. to describe security requirements in “stories” (specifications) also considered as executable tests. JFrog[3] is a set of DevOps tools following the approach to accelerate the delivery of binaries, securely through delivery pipeline, enabling end-to-end DevOps automation pipeline. Security

Monkey [5] is an OpenSource project from Netflix that enables monitor Amazon EC2 (AWS), OpenStack, Google Cloud Platform (GCP) and GitHub instances changes for assets. Black Duck software toolbox [1] provides automatic means to track your code, providing solutions to mitigate security risks. Finally, Snyk [7] provides tools for monitoring vulnerabilities and fix them for npm, Maven, NuGet, RubyGems, PyPI among others. Notwithstanding, all of these methodologies and tools do not provide a solution to grant source code integrity avoiding a malicious modification of the code in CI/CD pipeline.

4 Security Role in Automation

A CI/CD environment consists of (i) the *Source Code Control Server* which is responsible to manage changes to project's documents (ii) the *Assembly Server* which receives the changes and assembles them; (iii) the *Testing Server and Deployment Server* that validates that the project work and then publishes the latest version. Conceptually every server is located on different premises. Many developers integrate CI/CD procedures only to Assembly and Testing Servers. This fact is a consequence of the drastic change in today's software delivery way, Source Code Control Server repositories and product development (mostly manually) are not innovations of CI/CD. Henceforth, our solution uses CI/CD for Assembly and Testing Servers.

Prior development environments and project deployments were performed only on trust premises without presuming the security of the platform but trusting on software robustness. This case used to occur because security measures implemented on software to operate against a various number of adversary models. Nowadays, dockerization and virtualization are getting used to protect against unexpected events. Nevertheless, currently deployed software is not considered trustworthy. It occurs because, on most occasions, software security measures are not carefully considered. It tends to isolate deployment in host machines, restricting privileges and hardware access at maximum, but only these measures are questionable. The underlying software that controls these virtual machines acting as an intermediary layer among every virtual machine and hardware is the *Hypervisor*. Being dedicated to handling virtual machines, this leads it to a single-point-of-failure. If an attacker gets control over it, then she will be able to handle every virtual machine without tracking of the source of this attack. This technique is known as "hyperjacking" [32], and its most common implementation is to insert a malicious *Hypervisor* to forge the original one.

Fig. 1 depicts how this attack can be implemented in four steps:

1. Developer implements a new feature and this is uploaded to Source Code Control Server (Git based server in most cases).
2. Changes finished in Source Code Control Server are sent to Assembly and Tests Server.
3. Assembly and Tests Server assembles a new software version and conducts unitary test and linkage prepared for this software.
4. Once recommended tests are passed, a new version of the software is made public (deployment).

Under the assumption that every communication between every point described is secure, we have identified several weak points. For this reason, we have considered the next cases: If developers' computers are infected, then it is plausible malicious code modifies the Source Code Control Server could be infected, but it is even harder to implement uncontrolled changes due to the incremental

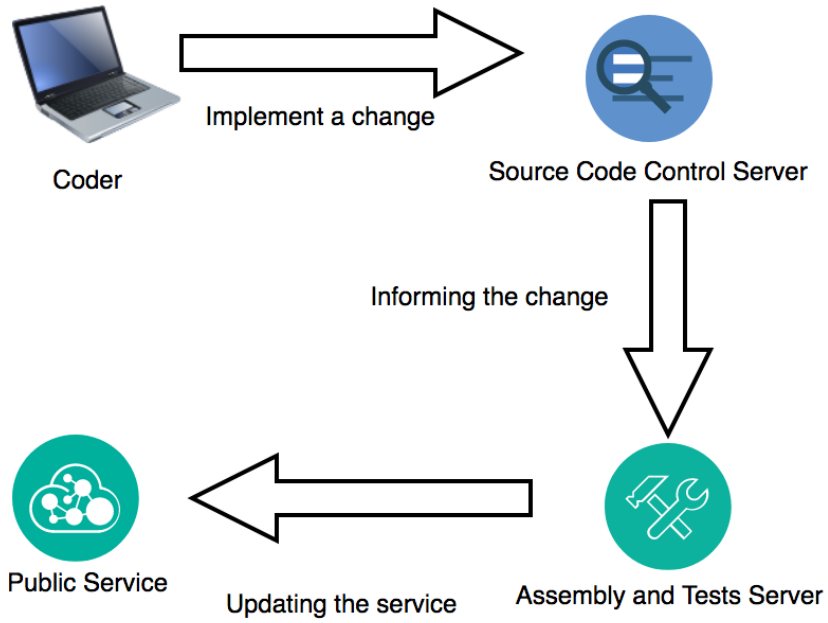

Fig. 1: Identified Risk in Continuous Integration Process

control version at the file level, which identifies quickly an undesired change. We consider that the most important vulnerability is identified in Assembly and Testing Server. By default this is considered as trusted because its interaction is restricted to insert source code. Notwithstanding, we have identified a gap in this process described in next example.

We assume that a malicious agent granted access to *Assembly and Testing Server*. This pretends to insert a piece of code for detecting every time source code is generated and files modified. Then that routing replaces some key source code file opening a backdoor. This case is not trivial to detect as previously described since source code is assumed that has been checked and then it is valid.

4.1 Security Risks in Assembly and Testing Servers

DigitalOcean [21] published that the proper way to ensure a CI/CD environment for a company devoted to virtual server deployment under premises is the isolation from external access. Since the CI/CD system has granted access to your repository and credentials to deploy in different environments, it is essential to keep in a safeguard your credentials to guarantee the final product integrity. However, a CI/CD server protection is not trivial, several alternatives exist as secure shell (SSH), private key APIs connecting through services as GitHub[7] or GitLab[8] to our CI/CD environment. It is recommended to have a proper password and implement 2-factor authentication. Despite this fact

[7] `https://github.com/`
[8] `https://about.gitlab.com/`

Milka [27] revealed that less than 10% of Google users make use of 2-factor authentication. A fail in securing those keys could lead into source code filtering or code modifications by impersonation attacks. Using an intermediate interface, some CI/CD solutions provide an interface to manage Assembly and testing server (i.e. Jenkins or GitLab) through a web interface. In the case of Jenkins, it is enabled as credential based access. Thus, the security of access interface is another issue to consider. We notice that many providers ignore recommendations about CI/CD server isolation. Indeed, in "Who is Using Jenkins" [9] there are projects as KDE[10], Apache[11], AngularJS[12] and Ubuntu[13] that are publicly accessible. We have used Apache Software Foundations from Jenkins public access.

A huge number of software projects are based on open source tools to streamline its development. Among others, the capability to observe, modify and publicly discuss any part of source code provide considerable advantages. Whichever from all identified risks could take place, especially avoiding recommendations. An example of this is found in 2003, attacking Linux kernel [17] inserting a backdoor in a CVS repository as mirror of the main repository. A digital interruption was performed in the server and then the change was inserted. It is detailed "that change was never approved and it is not present in the main repository". Detected change was the next in wait function 1.1. Although this threat could seem harmless, it assigns users under execution identifier **0**. This fact grants all privileges over the computer.

```
    if ((options == (__WCLONE|__WALL)) && (current->uid = 0))
retval = -EINVAL;
```

Listing 1.1: wait function

In [21] authors present "Failures in a CI/CD pipeline are immediately visible and halt the advancement of the affected release to later stages of the cycle. This is a gatekeeping mechanism that safeguards the more important environments from untrusted code". They physically separate Testing Server and Assembly Server producing a fake perception of security since the integrity of source code is not granted. The mere fact of using a CI/CD environment does not prevent severe security breaches.

We have to consider the potential facts of an attacker with administrative access granted to the server. The computer hosting Assembly and Testing Server is a precious target for external attackers. In some cases, the use of underlying hardware for particular benefits in these servers is powerful (able to compile and execute several proofs quickly to reduce development and deployment possible delays). In most cases, CI/CD solutions offer the possibility to multimode execution simultaneously. For instance, for bitcoin mining, in [8] is described a campaign of crypto-jacking in China with more that 50000 infected servers. Other possibilities as malicious emails, executing DDoS coordinated using botnets as occurred in 2017 with Mirai [6], or economic data rescue with a ransomware, are lower probable since data used in this kind of server should be sited in a different computer just to manage source code. There are cases of ethical hacking and vandalism, harms to development computer and intellectual property stored and damages to final user data.

[9] https://wiki.jenkins.io/pages/viewpage.action?pageId=58001258
[10] https://kde.org/
[11] https://www.apache.org/
[12] https://angularjs.org/
[13] https://ubuntu.com/

5 A proof-of-concept: Vulnerable Server launching Jenkins

Let us introduce an example to show the identified breach in the security of the CI/CD process. We make use of a simple application with a form for initializing sessions. This application provides service to dog owners, a gadget bound to the collar enabling owners to know the precise location of their pets in an emergency. Such as a case is when users login to the portal to see these data.

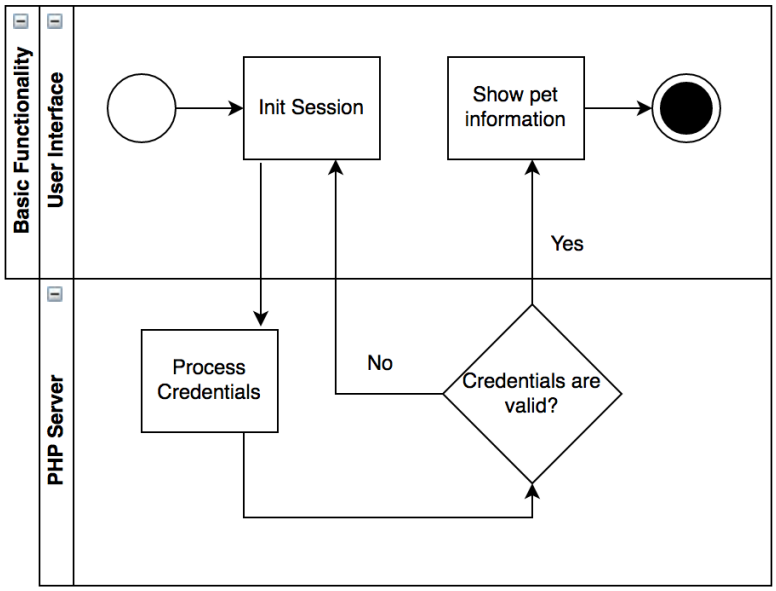

Fig. 2: UML Flow diagram describing basic functionality of our code

The application that evidences the identified weakness is developed using *Nuxt.js* and *Vue.js* for the front-ent and *PHP* for the back-end. Its functionality is presented in Fig. 2 through a UML flow diagram. Deployment is done through a *PHP internal server*, and *Nuxt.js* interface is compiled using Web Package.

On the one hand, GitLab was chosen as the *Source Code Control Server* for the project and is categorized as private. GitLab allows login sessions to its platform username/password and supports single sign on through social networks. Also, 2-factor authentication mechanisms are available. On the other hand, Jenkins is chosen as the *Assembly and Testing server*. This is an open source automation server enabling developers to reliably build, test, and deploy their software. The connection between Jenkins and GitLab is done with a mere configuration, to automate the generation of a new version to test for each new change in the repository. We can consider this is a comprehensive and mature CI/CD solution.

Let us assume that the *Assembly Server* violates the administrator's privileges after the access granted. The code has been developed in C# simulating a malware to install in a violated server.

This code detects when project files are written when cloning from GitLab and replaces their contents and is executed in the background to be unnoticed.

To perpetrate the attack, the *dog pictures* utilized by the application are replaced by cat pictures and form text is manipulated during the login session. Moreover, changes produced by replying to the repository are not undone.

The workflow is presented below:

- GitLab is properly contacted using the SSH key that is provided
- Folder is updated with changes uploaded to GitLab as initially configured. Dependencies are downloaded then it audits its correctness.
- Deploy the package.
- A "success message" indicates that CI/CD pipeline was properly completed.

Internal php server is used to deploy the project, we pay attention to everything while assembly is correct, but indeed deployment was not as initially expected.

This case as a real use-case of CI/CD environment presenting that only by passing Jenkins tests the project is ready to be published, keeping malicious modifications without any additional control measures. Inspecting the remote repository in GitLab, it is not possible to realize any change. It happens due to the fact that all files are just as uploaded files, even though the change was done by an authorized developer and it is empty. We facilitate to inject malware code to open backdoors or tasks. While we fill in the login session form, we perpetrate a man-in-the-middle attack exposing final client credentials hardly perceptible in CI/CD pipeline. This process includes source code inspection just after uploading changes to the Source Code Control Server repository, but the result after the package assembling is not carefully inspected since it is produced by a "trustworthy server" *Assembly and Testing server*.

6 Code integrity in the CI/DP pipeline

Our challenge is to establish a secure and trustworthy integrity code control when an assembly code computer is not trusted. This solution is based on TPM. We have decided to include it, since it provides a set of functionalities related to software integrity with trust guarantees provided by device design.

TPM is an international standard for a secure crypto-processor (ISO/IEC 11889). TPM is a secure microcontroller designed to issue integrated cryptographic keys. Among its features it is affordable (around €20[14]), indeed it is included in a large number of current platforms (not adding additional costs). It is widely documented allowing its usage for particular users and work purposes. Also, it is accepted as a trust solution as a cornerstone element for secure booting in modern computers, file encryption (BitLocker [26]). TPM enables device and user trust identification, key and certificate issuing, secure key storage, detecting not authorized modification, secure encryption with several algorithms and producing cryptographic hash values among others.

[14] `https://www.amazon.com/914-4136-105-Module-Infineon-Chip-9665/dp/B075FBGTG9/ref=sr_1_2?dchild=1&keywords=TPM+Chip+infineon&qid=1592740968&sr=8-2`

Our target is to ensure software integrity deployed on premises, such as developed code and final code remains unchanged. For this purpose, we propose TPM's utilization to provide a secure platform.

We have named the Trusted Integrity Platform (TIP) to our solution as is a new component in CI/CD pipeline. This is a TPM provided server with a trust software stack for testing software project integrity. TPM provides guarantees to build a TIP server with a controlled software stack so we can assure that no malicious code can alter the project code. This is the anchor for integrity validation comparisons. TPM public key is used to control TIP server integrity because TIP server trusted boot is bound to TPM sealed key.

Git provides data integrity in the sense that those repositories that store source code have integrity measures, in [14] "this functionality is implemented in lower stages in Git as part of its philosophy". A list with all hash values from every project file is created and an ensemble hash for every commit record change (using git commit).

Each commit is bound to an associated hash issued from file tree data. When a commit is requested, as a file change, this change is propagated by hashes in the tree and commit. This enables Git repository to detect any change produced. We assume that information from a Git server is trusted and therefore data integrity is achieved. Next step describes security measures for a TIP server as an element dedicated to integrity verification. TIP containing the TPM conducts integrity verification.

Given that software under execution in a TIP server is predictable, this allows implementing secure booting protocol as a TPM functionality to check and verify that software stack remains unchanged. This is useful for securing TIP server using TPM functionalities. We consider that 3-factor integrity proof [16], [31] is needed in the whole CI/CD pipeline. This is the cornerstone contribution of this approach, it is based on checking source code integrity from three different sources for detecting a possible code manipulation~~.~~:

- The *first integrity proof measure*: is taken before installing all dependencies required for the project; this guarantees that source code from the assembly and testing server is identical to Source Code Control Server.
- The *second integrity proof measure*: it guarantees that source code under assembly remains unchanged from external agents in assembly and testing servers.
- The *third integrity proof measure*: it guarantees that the whole process was successfully completed without undesired modifications after the project was assembled.

Every integrity proof measure is taken following particular steps we have categorized in three phases:

- **Suspicious code reception**: Assembly and testing server forwards to TIP server a compress file with source code, this is considered as "suspicious.zip". If the *uncompressing* phase is not successful, this file is discarded and integrity proof given as invalid.
- **Trust code reception**: TIP server retrieves source code from Git repository considered as trusted.
- **BigHashes proofs**: TIP server verifies, using TPM functionalities, that "suspicious.zip" content and source code from repository are identical. This is conducted consulting every hash file from Git server. These Git registered metadata are linked as a unique chain named bigHash and TPM hash functionalities are used to verify bigHash values. Therefore, when both bigHash

values (project bigHash and suspicious.zip bigHash) are identical, integrity proof is considered successful.

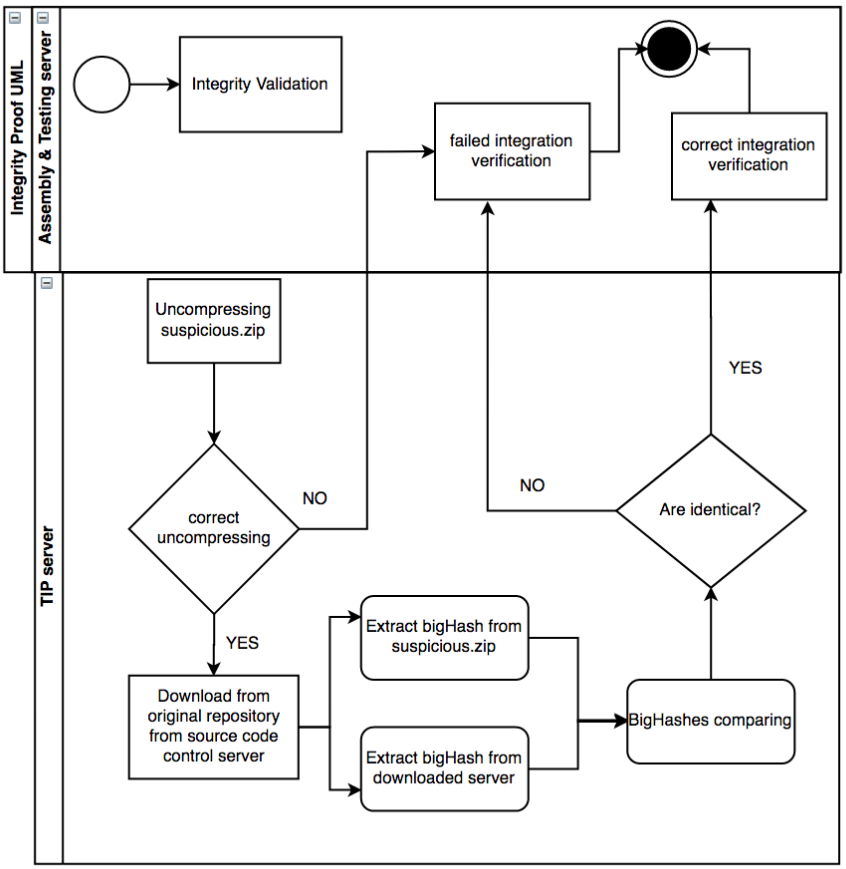

Fig. 3: Integrity Proof Verification UML Flow Diagram

Fig. 4 shows a sequence diagram with TIP process communications in CI/CD pipeline. This shows the 3-factor verification described as well as the point of check of every integrity proof checking. Retrieving source code by TIP server is conducted in every integrity check, but it has been simplified in the diagram (likewise assembly and testing commands from every project).

7 Our proposal: Trusted Integrity Platform

We have presented a solution based on three integrity proofs. The first one is taken before installing and guarantees integrity between code instances from the assembly and testing server and Source

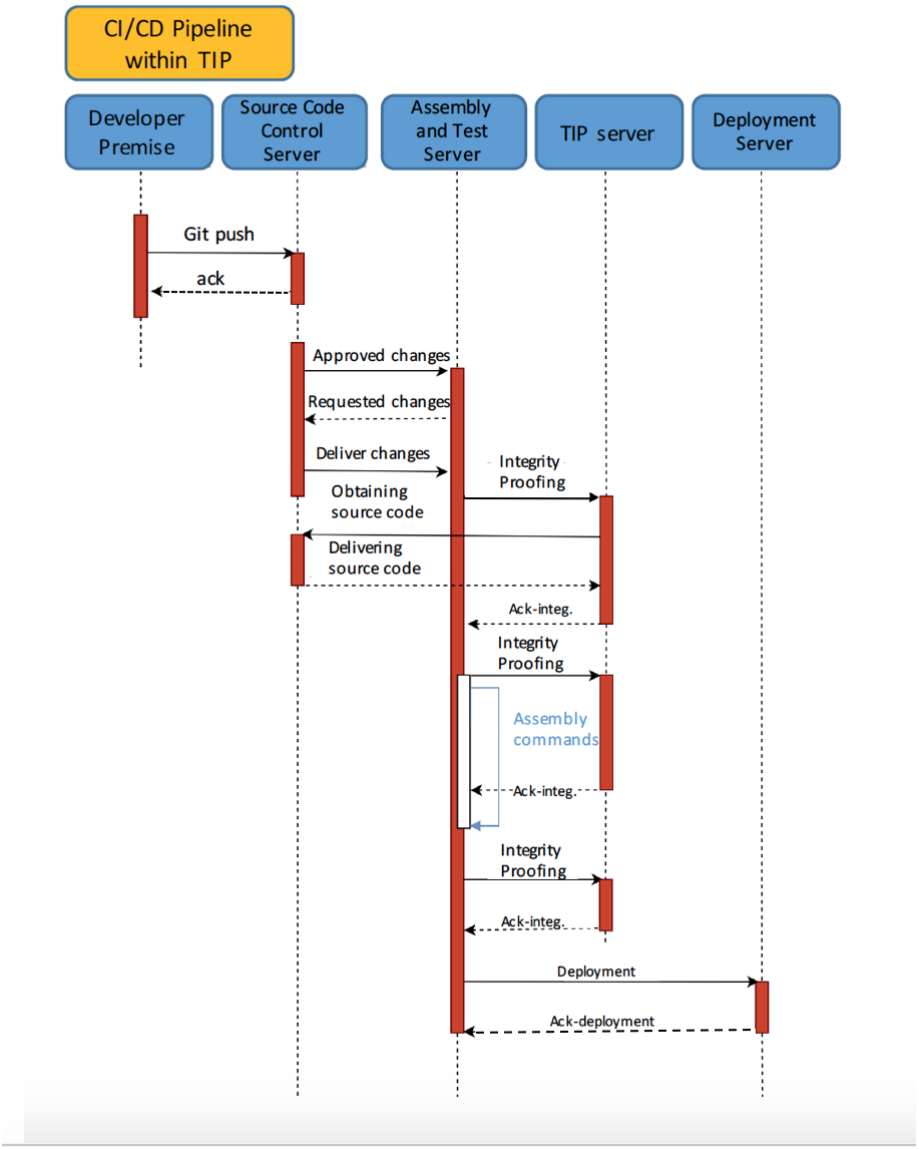

Fig. 4: Sequence diagram CI/CD pipeline within TIP server

Code Control Server. The second integrity validation compares code from an external agent and assembly and testing server remain unchanged. And the third validation checks that the whole process was successfully completed and code in unchanged after the assembling.

This section describes a TIP server implementation for testing its feasibility and as a proof of concept. We have implement a solution based on Windows 10 1809. We have implemented a component

written in C# that relies on TPM functionality to access code. The *TIP* server has been implemented with PowerShell scripts and receives PHP script as input. The internal PHP server may attend requests from both *Assembly and Testing Servers* for this implementation. The algorithm that enables communication with the TIP server actually implements project integrity validation. This script is based on PowerShell and it is tested on Jenkins. The procedure script is included as part of CI/CD pipeline testing batches. Also, we have included the TIP server script communication from Jenkins in PowerShell. A comprehensive description of the full procedure is following described.

Suspicious code reception: Firstly, setting uploading parameters are required. Hence, a temporal folder within the TIP Server is created to contain every Jenkins work-space file. Next, all files from Jenkins work-space are compressed into a file. Once the compression step is completed files from selected folder are taken and filtered. These are filtered and those included in label *ToExclude* list are removed, preserving work-space.

Once file is sent to the TIP server we have a variable $tipServer as a script input parameter. TIP server is implemented in PHP, then a gateway.php file is the main file in charge of TIP server with some configurable variables. Most of those variables are HTTP control headers, to allow remotely TIP server deployment. Once all settings are done the ZIP file is uncompressed if it has been successfully uploaded.

At this point the file is uncompressed in a *suspect*'s labeled folder. Communications between the *Source Code Control Server* repository and the *TIP* server are performed through *POST* requests. The first factor of the code integrity is fulfilled.

Trust code reception and BigHashes proofs: the trustworthy repository cloning takes place. Once it is successfully cloned the BigHash values are computed with PowerShell script and then retrieves the trusted repository's TPM hashes. Suspicious integrity repository hash value is retrieved to validate its integrity checking the matching. Finally, the whole integrity validation procedure bigHash values are compared and result of validity is obtained.This process fulfills the second and the third factor of the code integrity validation.

Our solution implements TPM functionalities to compute the hash values. We notice that to compute a complete Git folder hash, every file has to be accessed to link every hash values to file. For better performance results, we have decided to reduce hash computations and instead of computing every hash (for each file) to reuse Git hash values. Once, folder integrity validation is done whether content is suspicious, we make use of git hash-object. This command gets dynamically the object hash value since Git cache content could not be object hash value. Once linked hash values are ready, TPM hash functionalities are used to compute values. In particular, we have used SHA-256 (SHA-1 security vulnerabilities [34]). We pay attention to the hash of a variable size not exceeding TPM input buffer limitation. It is important to mention as we aimed TPM buffer is limited to 32 bytes [2], therefore addition manual control is required.

7.1 Utilization of TPM public keys in TIP server

We have proposed an additional security measure on the TIP server that represented the third integrity code validation. TPM equipped computers can use TPM functionalities for key generation and encryption with the particularity that decryption can only be performed by the TPM (binding key) protecting created key from disclosure. TPM works with a particular key hierarchy that starts

with endorsement root key that is unique for each TPM chipset and is assigned while manufacturing. We highlight that endorsement key private part will not be exposed as we have used in TIP server.

This step consists of each change submitted to Git server carrying a complete copy of the project is signed using the Developer private key considered as trusted. TIP server stores the project copy accessed when an integrity proof is required, that is decrypted using Developer public key (previously loaded). Therefore, three copies are taken as input (Developer version, Git stored version and suspicious from Assembly and testing server version) integration proofs are computed, then three versions should be identical.

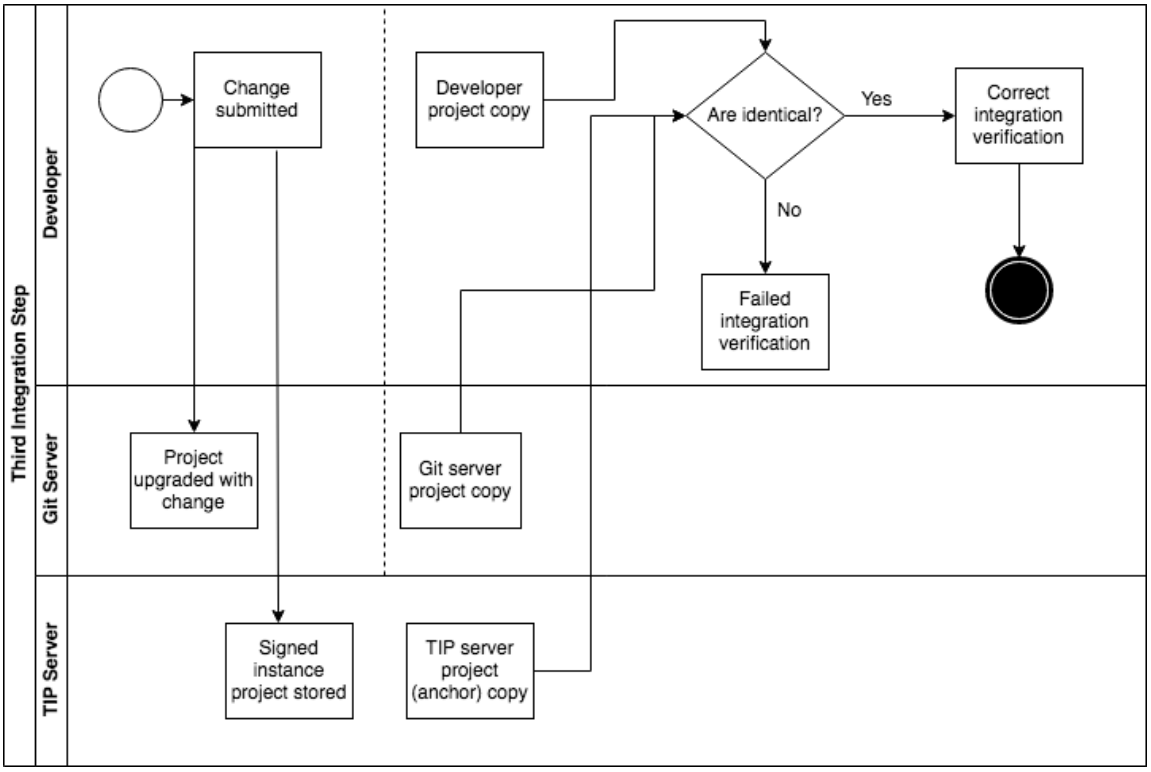

Fig. 5: Third Integration Verification Step - Developer Verification

Creating a private key inside TPM is a trivial process, while extracting this key to a hard disk is not. Indeed, until 2016 a Microsoft functionality to extract public keys in XML format was used, but this was deprecated.

```
1
2 var rsaParams = (RsaParms)keyPublic.parameters;
3 var exponent = rsaParams.exponent != 0
4      ? Globs.HostToNet(rsaParams.exponent)
5      : RsaParms.DefaultExponent;
```

```
6
7  var modulus = (keyPublic.unique as Tpm2bPublicKeyRsa).buffer;
8  // RFC 4716
9  var pemKey = Combine(
10     new byte[] {0x00, 0x00, 0x00, 0x07},
11     new byte[] {0x73, 0x73, 0x68, 0x2d, 0x72, 0x73, 0x61},
12     new byte[] {0x00, 0x00, 0x00, 0x03},
13     exponent,
14     new byte[] {0x00, 0x00, 0x00, 0x80},
15     modulus
16 );
17
18 Console.WriteLine("---- BEGIN SSH2 PUBLIC KEY ----");
19 uint i = 0;
20 foreach(char c in System.Convert.ToBase64String(pemKey)){
21     Console.Write(c);
22     if(++i == 72){
23         Console.Write("\n");
24         i = 0;
25     }
26 }
27 Console.WriteLine("---- END SSH2 PUBLIC KEY ----");
```

Listing 1.2: Creation of a private key inside TPM and extract from it

We pretended a replica of deprecated functionality computing RSA public key using its module and index data following RFC 4716 [4] standard specification, as it is shown in algorithm 1.2. The computation of this code gives us as results the public key. Hopefully we made use of TPM2 TSS Engine library that implements functionalities for OpenSSL for TPM 2.0. This library facilitates all tasks using the tpm2-tss software stack following TCG Specifications.

8 Performance Evaluation

We analyze the computational cost of our approach measuring each new added process: (i) Extract the compressed file; (ii) Check the correction of the extracted file; (iii) Extract bigHash from compressed file; (iv) Extract bigHash from downloaded server; (v) bigHash comparison and (vi) Total Integrity Proof Verification. We use Windows 10 1809 with an Intel Core i5-7200K CPU at 2.5 GHz, 16GB RAM. We measure execution time of each stage with a software project with 50 MB on a TPM simulator to present and extrapolate real results on a physical chipset. Results are shown in Table 1.

From the comparisons we can notice that our proposed implementation adds maximum 44900ms as delay against the traditional process without integrity check. This minor delay is worthy of acceptance from developers since this leads to a safer CI/CD pipeline. Moreover, check of code's integrity provides high security levels to the delivered software project.

Examined Phase Software project with 50 MB	Time in milliseconds (ms)
Traditional CI/CD pipeline	360000- 600000 ms (6-10 min)
Un-compress file (our scheme)	+16000-165000 ms
Check un-compress fold (our scheme)	+6000-7000 ms
Extract bigHask from compressed file (our scheme)	+1300-2200 ms
Extract bigHash from downloaded server (our scheme)	+1100-2100 ms
bigHash comparison (our scheme)	+1300-1700 ms
Integrity Proof Verification (our scheme)	+31000-38400 ms

Table 1: Performance Evaluation

9 Conclusions and Future Work

The paper presented an identified security gap in the CI/CD pipeline in terms of integrity validation. To confront this gap we proposed a solution and delivered a proof-of-concept implementation of it. The proposed solution achieves i) to prove code integrity validation in CI/CD pipelines; ii) to develop a lightweight implementation for secure automation and iii) to deliver a solution in which the additional cost is much less than the benefit gained by its adoption by developers. We believe that this solution may contribute to open the way for a secure CI/CD pipeline ecosystem. As future work, multi-threaded tasks in the TIP server will be considered. Since tasks are parallelizable such as getting Trusted Git repository while suspicious.zip received file is uncompressed and retrieving bigHashes values in parallel can improve the current sequential version performance. Moreover, we aim to use hard-links (Junctions in Windows) instead of a mere copy of temporal work-space saving disk-space and time.

Acknowledgment

This research has been funded by the Marie Skłodowska-Curie SealedGRID grant agreement No 777996 and the H2020-SC1-FA-DTS-2018-1 CUREX under grant agreement No 826404.

References

1. Black duck. `https://www.blackducksoftware.com/`, online; accessed 3 July 2020
2. IBM's TPM 2.0 TSS. `https://sourceforge.net/projects/ibmtpm20tss/`, online; accessed 19 June 2020
3. Jfrog. `https://jfrog.com/`, online; accessed 3 July 2020
4. The secure shell (ssh) public key file format. `https://tools.ietf.org/html/rfc4716`
5. Security monkey. `https://securitymonkey.readthedocs.io/en/latest/quickstart.html/`, online; accessed 3 July 2020
6. Servico Antibotnet. `https://www.osi.es/es/servicio-antibotnet/info/mirai`, online; accessed 19 June 2020
7. Snyk. `https://snyk.io/`, online; accessed 3 July 2020
8. Ophir Harpaz, Daniel Goldberg: The Nanshou Campaign – Hackers Arsenal Grows Stronger. `https://www.guardicore.com/2019/05/nansh0u-campaign-hackers-arsenal-grows-stronger/` (2013), online; accessed 19 June 2020

9. Arthur, W., Challener, D., Goldman, K.: Platform security technologies that use tpm 2.0. In: A Practical Guide to TPM 2.0, pp. 331–348. Springer (2015)
10. Bass, L., Holz, R., Rimba, P., Tran, A.B., Zhu, L.: Securing a deployment pipeline. In: 2015 IEEE/ACM 3rd International Workshop on Release Engineering. pp. 4–7. IEEE (2015)
11. Bass, L., Weber, I., Zhu, L.: DevOps: A Software Architect's Perspective. SEI Series in Software Engineering, Addison-Wesley, New York (2015), http://my.safaribooksonline.com/9780134049847
12. Bennetts, S.: Owasp zed attack proxy. AppSec USA (2013)
13. Bird, J.: DevOpsSec: Securing Software Through Continuous Delivery. O'Reilly Media, https://books.google.gr/books?id=dpm_vQEACAAJ
14. Chacon, S., Straub, B.: Pro git. Springer Nature (2014)
15. Deepa, G., Thilagam, P.S.: Securing web applications from injection and logic vulnerabilities: Approaches and challenges. Information and Software Technology **74**, 160–180 (2016)
16. Dheerendra, M., Sourav, M., Saru, K., Khurram, K.M., Ankita, C.: Security enhancement of a biometric based authentication scheme for telecare medicine information systems with nonce. Journal of medical systems **38**(5), 41 (2014)
17. Felten, E: The Linux Backdoor Attempt of 2003. https://freedom-to-tinker.com/2013/10/09/the-linux-backdoor-attempt-of-2003/
18. Guan, H., Chen, W.R., Li, H., Wang, J.: Stride–based risk assessment for web application. In: Applied Mechanics and Materials. vol. 58, pp. 1323–1328. Trans Tech Publ (2011)
19. Huang, Y.W., Yu, F., Hang, C., Tsai, C.H., Lee, D.T., Kuo, S.Y.: Securing web application code by static analysis and runtime protection. In: Proceedings of the 13th international conference on World Wide Web. pp. 40–52 (2004)
20. Humble, J., Farley, D.G.: Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. Addison-Wesley, Upper Saddle River, NJ (2010), http://my.safaribooksonline.com/9780321601919
21. Justin Ellingwood: An Introduction to CI/CD Best Practices. https://www.digitalocean.com/community/tutorials/an-introduction-to-ci-cd-best-practices (2013), online; accessed 19 June 2020
22. Krusche, S., Lichter, H., Riehle, D., Steffens, A.: Report of the 2nd workshop on continuous software engineering. In: CSE@ SE. pp. 1–6 (2017)
23. Kuusela, J., et al.: Security testing in continuous integration processes (2017)
24. Lee, T., Won, G., Cho, S., Park, N., Won, D.: Detection and mitigation of web application vulnerabilities based on security testing. In: IFIP International Conference on Network and Parallel Computing. pp. 138–144. Springer (2012)
25. Lipke, S.: Building a secure software supply chain (2017)
26. Microsoft: BitLocker most frequenly asked questions. https://docs.microsoft.com/es-es/windows/security/information-protection/bitlocker/bitlocker-overview-and-requirements-faq, online; accessed 19 June 2020
27. Milka, G.: Anatomy of account takeover. In: Enigma 2018 (Enigma 2018) (2018)
28. Mohan, V., Othmane, L.B.: Secdevops: Is it a marketing buzzword?-mapping research on security in devops. In: 2016 11th International Conference on Availability, Reliability and Security (ARES). pp. 542–547. IEEE (2016)
29. OWASP: pen web application security project (OWASP), howpublished = https://www.owasp.org/, note = Online; accessed 2 July 2020 ,
30. Rimba, P., Zhu, L., Bass, L., Kuz, I., Reeves, S.: Composing patterns to construct secure systems. In: 2015 11th European Dependable Computing Conference (EDCC). pp. 213–224. IEEE (2015)
31. Saru, K., Kumar, D.A., Xiong, L., Fan, W., Khurram, K.M., Qi, J., Hafizul, I.S.: A provably secure biometrics-based authenticated key agreement scheme for multi-server environments. Multimedia Tools and Applications **77**(2), 2359–2389 (2018)
32. Sathyanarayanan, N., Nanda, M.N.: Two layer cloud security set architecture on hypervisor. In: 2018 Second International Conference on Advances in Electronics, Computers and Communications (ICAECC). pp. 1–5. IEEE (2018)

33. Schneider, C.: Security devops-staying secure in agile projects. OWASP AppSec Europe (2015)
34. Stevens, M., Bursztein, E., Karpman, P., Albertini, A., Markov, Y.: The first collision for full sha-1. In: Annual International Cryptology Conference. pp. 570–596. Springer (2017)
35. Tichy, M., Goedicke, M., Bosch, J., Fitzgerald, B.: Rapid continuous software engineering. Journal of Systems and Software **133**, 159 (2017)
36. Ullah, F., Raft, A.J., Shahin, M., Zahedi, M., Babar, M.A.: Security support in continuous deployment pipeline. arXiv preprint arXiv:1703.04277 (2017)
37. XebiaLabs: Behaviour driven development security. `https://xebialabs.com/technology/bdd-security/`, online; accessed 3 July 2020