Reflections on Trusting Docker: Invisible Malware in Continuous Integration Systems

Florent Moriconi, Axel Ilmari Neergaard, Lucas Georget, Samuel Aubertin, Aurélien Francillon

Reflections on Trusting Docker: Invisible Malware in Continuous Integration Systems

Florent Moriconi
EURECOM
AMADEUS

Axel Ilmari Neergaard
EURECOM
CUJO AI

Lucas Georget
EURECOM
EDF R&D
LAAS-CNRS

Samuel Aubertin
EURECOM

Aurélien Francillon
EURECOM

***Abstract*—Continuous integration (CI) is a widely adopted methodology for supporting software development. It provides automated generation of artifacts (e.g., binaries, container images) which are then deployed in production. However, to which extent should you trust the generated artifacts even if the source code is clean of malicious code? Revisiting the famous compiler backdoor from Ken Thompson, we show that a container-based CI system can be compromised without leaving any trace in the source code. Therefore, detecting such malware is challenging or even impossible with common practices such as peer review or static code analysis. We detail multiple ways to do the initial infection process. Then, we show how to persist during CI system updates, allowing long-term compromise. We detail possible malicious attack payloads such as sensitive data extraction or backdooring production software. We show that infected CI systems can be remotely controlled using covert channels to update attack payload or adapt malware to mitigation strategies. Finally, we propose a proof of concept implementation tested on GitLab CI and applicable to major CI providers.**

I. Introduction

DevOps' mindset encourages the use of Continuous Integration (CI) to quickly alert developers when a code change does not meet quality requirements. In practice, a CI system runs a set of checks (i.e., a CI build) on code changes. Quality checks can be manual (e.g., peer-review) or automated. In practice, automated steps of a CI build are often composed of a compilation, static code analysis (e.g., code linting tools, vulnerability scanners), binary scanning, tests (e.g., unit-testing, end-to-end testing, regression testing, functional validation), and deployment of artifacts (e.g., binaries, Docker images, tarball archives, pip packages) to a remote registry. Furthermore, CI build servers aim to provide a more controlled environment than developers' laptops to increase security. The CI is orchestrated by a CI orchestrator, notable examples include Jenkins, GitHub Actions, and Travis CI. Nowadays, CI orchestrators make extensive use of containers. Indeed, containers allow identical environments between a developer laptop, CI build environment, and the production servers. When a CI build starts (e.g., on code change), the orchestrator starts a new container. The container is based on a container image called the CI build image. This image is generally common to all builds of a specific family (e.g., Java-based software, Python-based software). It must contain all the tools required to perform the CI steps: compilers, linters, code analyzers, test frameworks, company-specific tools, etc. Therefore, it is common for this image to be customized by the developers to include the necessary tools. This means that the construction of the CI image is considered likewise another software and therefore, it is often built using the CI system itself. As a consequence, CI systems are considered self-hosted architectures [1]. Thompson [2] showed that self-hosted architectures raise new questions about security. This is, in particular, the case with a C compiler written in C. Thompson shows that if malware is present in the compiler and re-injects itself when the compiler is built, then detecting such malware is challenging. Therefore, to what extent should you trust that your system is free of malware? In this paper, we show that the same reasoning can be applied to continuous integration systems. In particular, we make the following contributions:

- We show that a CI system can backdoor production software without leaving any trace in the source code repository
- We show that malware can persist on updates in a self-hosted CI system
- We show that malware in a CI system can be updated using hidden channels
- We implement a proof of concept implementation targeting Docker-based CI systems

The paper is structured as follows. In Section II, we present literature on trusting software systems and CI malware. In Section III, we show how to achieve a CI malware without any trace in the source code. In Section IV, we detail a proof of concept implementation targeting Docker. In Section V, we explore the limits of the approach and mitigation strategies. In Section VI, we conclude and show future paths of improvement.

II. Related work

Our work is based on the question raised by Thompson [2] regarding self-hosted architectures. Thompson showed that one should not trust any program which was not written by oneself directly in machine code. He described how self-replicating programs, called quines [3], can learn and perpetuate knowledge over time. When applied to C compilers, he showed that it can be leveraged to backdoor the UNIX "login" command (e.g., add an attacker-controller password). Indeed, quines can learn and perpetuate knowledge without any trace in the source code. Figure 1 illustrates this property. The first step is to

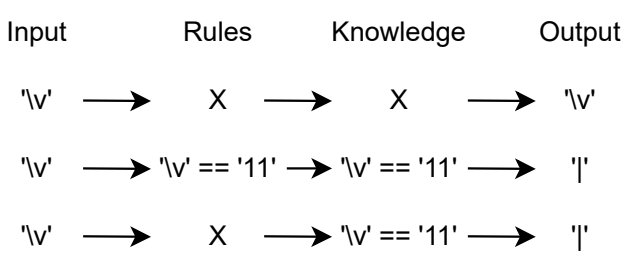

Fig. 1. Illustration of the learning process and perpetuation of knowledge in a compiler. After learning (step 2), compiler is able to apply the knowledge without explicit rule (step 3).

teach the compiler that \v is equal to ASCII character 11. Then, even after removing the rule from the source code, \v will be modified to ASCII character 11 as the rule is now part of the compiler. Furthermore, the rule will be persistent when the compiler is compiled again (e.g., compiler update). Gratzer et al. [4] show that knowledge acquired by quines is difficult to detect and remove. Compilers are an interesting target because of their ability to alter source code just before compilation. Bauer et al. [5] took advantage of compiler bugs in GCC to create a backdoor. Furthermore, two real-world attacks leveraged this concept. In 2009, the `Win32.Induc.A` virus [6] infected Delphi compilers to add malicious code to applications built using the compiler. In 2015, Xcodeghost [7] was one of the most successful attacks on the Apple App Store. It consisted of a modified Xcode compiler that makes the applications being spied.

Attacks against CI systems leverage their dependencies to a large number of components such as code dependencies or tooling (e.g., linters). Improper configuration of CI [8] can lead to vulnerabilities [9], [10]. Backdoors [11] often refer to a long-term compromise of targeted systems. Therefore, the malicious code should persist on updates. There are different ways to get initial access to the system in CI environments.

Common vulnerabilities are listed by the OWASP foundation [12]. Notable examples include hypocrite commits [13], dependency confusion [14], or supply chain attacks (e.g., attacks on upstream packages). OSC&R [15] framework describes common techniques for supply chain attacks, such as compromising legitimate artifacts or publishing and advertising malicious ones. Williams [16] shows that attackers tend to focus on popular projects largely used by the developer's community even if they are often better protected. In addition, malicious code can come directly from project maintainers [17]. Therefore, malware in CI can come from malicious code in dependency repository [18] or exploits against pipelines [19]. In addition, bypassing auto-merge rules [20] or required reviews [21] can help to merge malicious code to main branch.

Docker is widely used as it provides consistent execution environment in different systems. However, it raises new questions about security [22], [23], [24]. Indeed, Docker containers share the same kernel as the host system. Furthermore, container images can be made by the community. While images can be analysed to detect malicious or vulnerable code, analysis often rely on database of malicious content. This does not allow the detection of unknown (zero-day) attacks. Therefore, the use of containers for CI could open up new security vulnerabilities.

Wheeler [25] proposes a countermeasure to detect attacks described by Thompson through diverse double-compiling. It can be combined with other emerging solutions such as bootstrappable [26] and reproducible [27] builds (i.e., methods ensuring consistency between binaries and source codes) to improve trust in software [28]. Regarding supply chain attacks, attempts have been made to detect and mitigate them, such as forensic [29] or safeguard levels [30] artifacts. Regarding Docker, Petazzoni [31] warned about using Docker-in-Docker as it requires privileged container which makes it easier to attack the host system. Zero Trust Container Architecture [32] encourages to always consider the code within the container as potentially malicious, asking to higher level of security. In the following section we describe our approach to target container-based CI systems.

III. Our approach

We propose a new approach to compromise continuous integration systems. The approach does not leave any traces in source code repositories. Therefore, the compromise cannot be detected by auditing code repositories, a widely used way to assess software security. Figure 2 shows a general overview of the approach. Our approach requires that at least one component of the CI system use a self-hosted architecture. We detail in the following section how we identify the best location for the malware. Then we detail the self-injection process to persist on CI updates. We present possible strategies for the initial infection process. Finally, we present a few possible attack payloads and how the malware can be remotely controlled using hidden channels.

A. Self-hosted architecture

We propose malware that does not leave any trace in source code repositories. Therefore, this excludes storing the malicious code in the CI system code repository. Furthermore, the malicious code must be stored in a component (e.g., Docker client, Docker daemon, shell, test framework) that is in a self-hosted architecture. Without self-hosted architecture, the malware would be overwritten by a clean component version (i.e., build from repository source code free of malware) on CI updates. Custom build images are commonly used to avoid reinstalling all the necessary tools (e.g., linters) each time the CI build is started, leading to a self-hosted architecture. As CI updates might happen often (e.g., new features, bug fixes, upgrade tool versions), targeting a component in a self-hosted architecture is required for long-term compromise. Figure 3 shows the workflow for building the CI build image. It shows up that the CI build image is built from the CI build container. Furthermore, the CI build container is created from the CI build image. Therefore, this is a self-hosted architecture

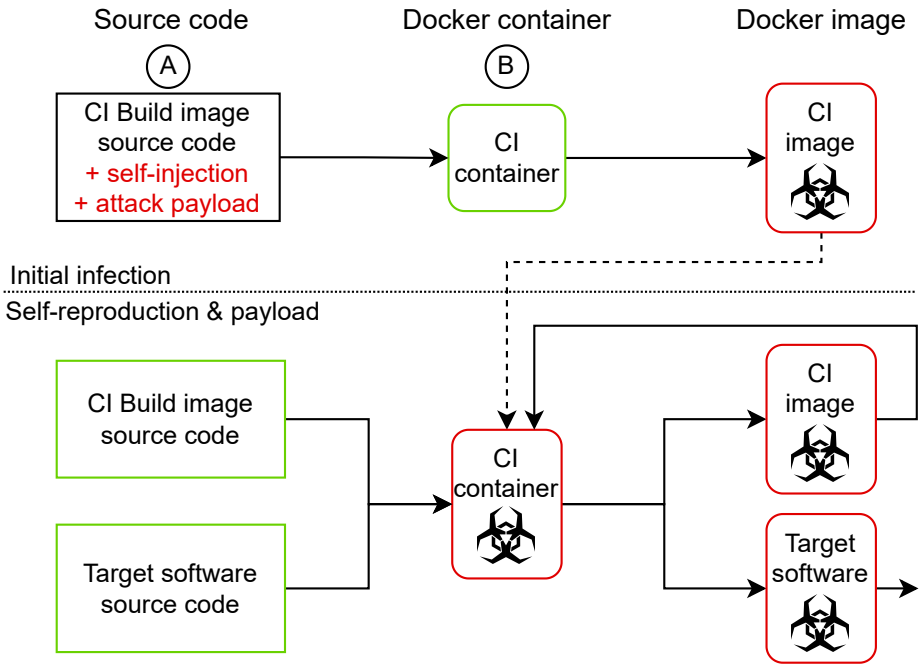

Fig. 2. Overview of initial infection and after infection modes (self-reproduction and payload). In green, the source code is free of malware. In red, the infected containers and images. (A) refers to initial infection using hypocrite commits, and (B) refers to initial infection using an in-container vulnerability (e.g., dependency confusion).

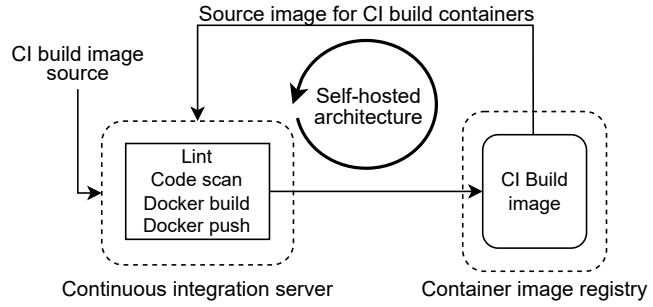

Fig. 3. Overview of a Continuous Integration self-hosted architecture. CI container is created from the CI build image, and CI build image is built in the CI container.

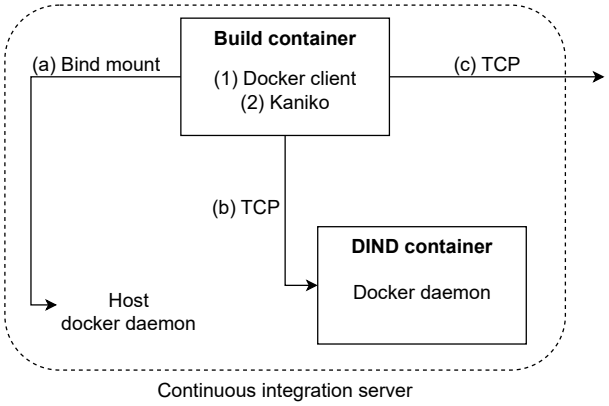

Fig. 4. Overview of different strategies for building Docker images from CI container. Docker client (1) or in-container build tool (2), such as Kaniko, can be used. The Docker client connects either to the host Docker daemon (a), a daemon in another container, i.e., Docker-in-Docker (b), or an external daemon over the network (c).

for components inside the build container. However, some components might be external to the CI container. Figure 4 shows different strategies for building Docker images in a continuous integration build container. When building a Docker image using Docker client (1) or in-container tool (2) such as Kaniko [33], this is a self-hosted architecture. However, this is not a self-hosted architecture if the building is done by an external component such as host Docker daemon, Docker-in-Docker, or distributed system. In practice, Docker-in-Docker is a very common architecture pattern for CI as it helps to target a full containerized architecture. Using the host Docker daemon is a bad practice as it gives root access to the host system from a CI build container [34]. External systems (e.g., distributed build systems) are tailored for specific needs, such as very large build volumes, and are rarely used. Therefore, to support Docker-in-Docker (dind) architecture, we must implement the malware in a self-hosted component. It means targeting software that is stored in the CI build container. This is the case with the Docker client and in-container build tools such as Kaniko. Docker client is a Command Line Interface (CLI) tool that interacts with the Docker daemon, whether the daemon is in another container (dind), on the host system over a bind mount, or in an external system. Therefore, we choose to implement the malware in the Docker client instead

of the Docker daemon. The same approach can be applied to Kaniko. However, as the Docker client is more widely used than Kaniko, we focus on the Docker client. Indeed, the Docker client interacts with the Docker daemon to send build context (i.e., Dockerfile, source files, parameters) to the daemon. In the following section, we detail how the Docker client can re-inject the malware when building the CI image that contains the Docker client.

B. Reproduction mechanism

When building a Docker image in the CI, the malware must be able to re-inject itself when a new CI build image is built: this is the self-reproduction mechanism. As presented in the previous section, we choose to hide the malware in the Docker client as it is present in the CI build image. It can be installed manually by developers (i.e., through a package manager or rarely by compilation) or taken from a base Docker image (e.g., docker:latest image). The Docker client should detect if the image being built is the CI build image or an image of interest for the attack payload. We detail in Section III-D potential attack payload actions. The Docker client might rely on the image name or build context heuristics to identify the image being built. As the Docker client has access to the build context, it can check for the presence of specific files in the build context (i.e., source code). For instance, it might check for the presence of the program `/usr/bin/docker`. We can re-inject the malware in any Docker image containing the client. However, to reduce the detection risk, we would rather try to re-inject the malware only when the CI image is built. For this purpose, we rely on image names and other rules, such as filenames that only appear in CI build images (e.g., test framework, Docker client). Before being able to self-reproduce, an initial infection must be performed.

C. Initial infection

The initial infection requires modifying the CI build image. Executing arbitrary code can accomplish this during image build or by directly altering the CI image in the registry. We detail below different strategies that internal or external attackers can apply. Therefore, our approach does not require a complicit actor (e.g., a company employee). Initial infection should only be done once, then, self-reproduction will ensure the malware will stay in the CI systems even on updates.

a) Malicious commit: The most straightforward method to execute attacker-controlled code during CI image build is to insert malicious code in the source code repository that defines the CI system. Qiushi et al. [13] show that patch proposals injecting new vulnerabilities can be accepted in large open-source software projects like the Linux kernel. Therefore, the same method can be considered in a corporate environment. However, directly committing malicious code to the repository might be challenging when enforcing peer review and code scanning. To reduce the risk of detection, malicious code can be hidden by proposing a code change that looks legitimate (e.g., adding a new code linter in CI) and relies on a remote compromised package. However, the malicious commits approach requires access to a code management system (i.e., internal network access, credentials), which might be challenging for an external attacker. Another option is to leverage a vulnerability in dependencies management.

b) Compromised dependency: Builds often rely on a large number of dependencies. These dependencies can be internal to the company or public (e.g., open-source). Public dependencies can be leveraged to execute malicious code without committing to the target repository source code. Indeed, dependency confusion [14] allows running attacker-controlled code in a continuous integration build server. The approach leverages how package managers (e.g., pip, npm) handle multiple registries. By default, they look up all configured registries and select the package with the highest version. Furthermore, public registries are commonly enabled by default (e.g., PyPI for Python pip). Therefore, creating a package on a public registry with the same name as an internal package and a higher version leads to dependency confusion. JFrog Artifactory, a widely used software to host internal artifacts, was vulnerable to dependency confusion. While attempts [35], [36] have been made to mitigate these attacks, they are difficult to deploy at scale (e.g., not backward-compatible). Therefore, dependency confusion is still applicable to a wide range of systems.

c) Image registry compromising: Another way to compromise a CI image is to directly alter the image in the registry. Every CI build runs with a user that has privileges. For the build to succeed, these privileges are required: pull source code, download dependencies, or push artifacts. However, a poorly configured CI system might allow CI users to access or modify resources out of the scope of the current build. For instance, the CI users might have full read/write access to the container image registry, allowing any build to alter any stored image. This can happen when a unique user is shared across builds, or privileges are assigned to the CI server (e.g., EC2 IAM policy [37], IP-based permission). Credentials might be wrongly populated [38] to pull requests (PRs). Furthermore, opening PRs is often allowed for a large group of users. This might lead to the compromise of registry credentials, allowing manipulation of stored images.

D. Payload

Malware has a payload, i.e., a part of code that can take action. For malware targeting desktops, the payload often allows monitoring of the user: recording keyboard keys, listening to ambient sound, copying files, etc. In the context of a CI build environment, malware can access and modify sensitive data. Indeed, CI is often used to build artifacts that will be deployed to production. It aims to provide a more secure solution than developers' laptops. An attack payload might record and exfiltrate sensitive data such as environment variables or program source codes. Furthermore, the payload can alter the source code before compilation. In practice, it might create vulnerabilities in production software, such as authentication bypass. Finally, the payload can consist of network analysis and lateral movement over the network. Indeed, CI servers are

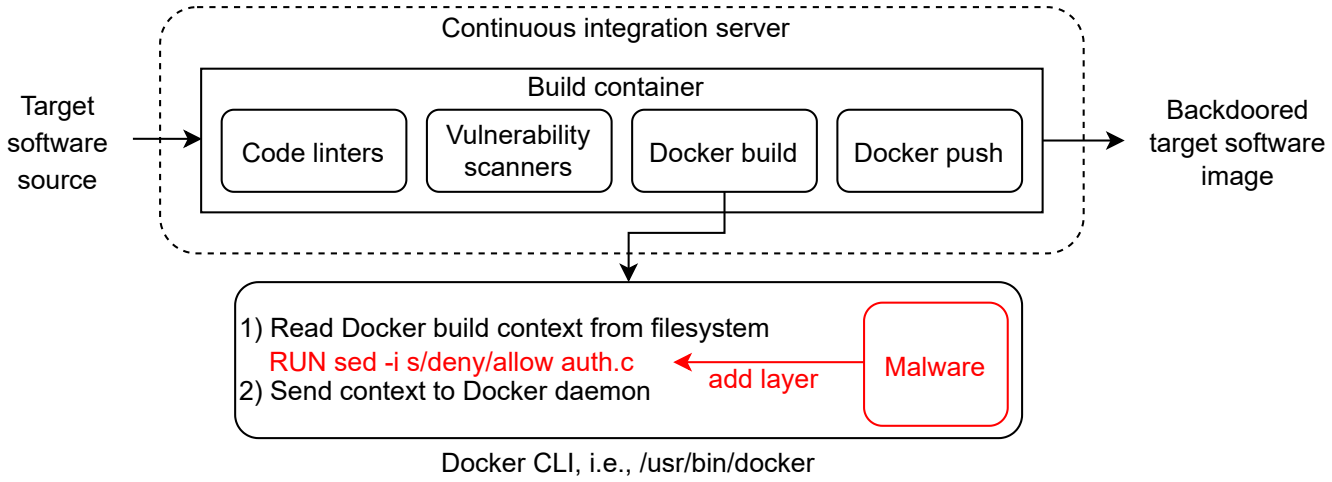

Fig. 5. Overview of on-the-fly injection of malicious code during Docker build. In red, malicious code is inside the Docker client. Backdoor code is injected in the build context before sending it to the Docker daemon.

often in the internal network perimeter. The attack's payload depends on the attacker's motivation and can therefore change over time. To this aim, we present strategies for remotely updating the malware in the following section.

E. Command & Control

After initial infection, malware might require to be updated. Indeed, target programs evolve (e.g., new code), or new security systems can be deployed. To this end, we developed a methodology for Command and Control (C&C) for the malware. C&C aims to be less detectable than reusing the initial infection workflow. In the context of a CI server, downloading packages from public registries (e.g., PyPI, npm) is common. Therefore, we can leverage this property to C&C the malware. By uploading a package to a public registry that the malware will download, we can C&C the malware. To further reduce the risk of being detected, the package should look like a standard package (e.g., code linter). However, some files inside the package will contain hidden material using steganography. Therefore, detecting that downloaded packages contain malicious code will be challenging. Other notable options for C&C include DNS tunneling [39] or git-based tunneling. Encrypted files in git (e.g., SOPS [40]) are widely used to securely store and share secrets required for software development. However, they create a blind spot for code scanners as they do not have encryption keys. This can be leveraged to hide malicious code behind encryption. Encrypted files will be decrypted by malware in the CI to update themselves. In practice, C&C will highly depend on the victim's infrastructure and security mechanisms. The next section details how we created a proof of concept implementation targeting Docker-based continuous integration systems.

IV. Implementation

To demonstrate practical self-replicating and invisible malware in CI systems, we wrote a proof of concept implementation targeting the Docker client [41]. The latest version of the artifact is available at https://doi.org/10.5281/zenodo.7777331.

Docker Engine follows a client-server architecture consisting of a client communicating with a Docker daemon, acting as the server, using the HTTP or HTTPS protocol over TCP or a socket. The daemon creates and manages Docker objects, such as images and containers, whereas the client sends commands to the daemon and shows output (e.g., logs) to the user. When the client instructs the daemon to build a new image, the client sends a file consisting of build commands to assemble an image (referred to as a "Dockerfile"), along with any other required files (referred to as "context"). As detailed in Section III-A, we target the Docker client.

A. Bootstrapping

In the context of CI systems, the Docker client runs within the CI container: this is a self-hosted architecture. This makes it an apt target for the creation of self-replicating malware; if the client rebuilds itself, then it can be patched with a self-replicating procedure like Thompson's [2]. Patching the client requires a bootstrapping [42] phase before the initial infection of any CI system. The bootstrapping phase is performed by a custom script that produces a Dockerfile containing build commands to produce a client capable of self-replication.

a) Achieving self-replication: To achieve self-replication, two separate files are needed: (1) a Dockerfile with build commands to compile a Docker client (referred to as the "genesis Dockerfile" to avoid confusion), and (2) a patch of the Docker client source code. The build commands in the genesis Dockerfile must contain all required material to build a Docker client. The patch of the Docker client source code must include a procedure that detects when a target Dockerfile sent to the daemon (1) uses a base image containing a Docker client or (2) when a Docker client is assembled from scratch. It is noteworthy that the original (i.e., non-malicious) Docker client can come from the Docker base image, from package

manager (e.g., apt), or from source. When the procedure detects one of these aforementioned scenarios, it applies the Docker client patch to the client version compiled in the build commands of the genesis Dockerfile. Afterwards, the procedure must replace the build commands in the target Dockerfile corresponding to the Docker client (base image or assembly) with the commands from the genesis Dockerfile. This procedure is based on the self-replication logic of a quine [3]. We choose to compile the malicious Docker client in the build container to avoid fetching a malicious docker client over the network (that could be detected by intrusion detection systems). However, it is also possible to fetch the malicious client from a remote registry to avoid the the complexity of compiling it during initial infection or self-reproduction.

b) Performing bootstrapping: The custom script that performs bootstrapping produces a Dockerfile with self-replicating logic based on the files introduced in the previous paragraph. The CI can be considered infected once this Dockerfile is used to assemble a CI image.Every rebuild of a similar image will then re-inject the self-replicating client patch. As illustrated in Figure 6, the custom script performs the following steps to create the aforementioned Dockerfile:

1) Insert the original genesis Dockerfile as a payload within the client patch.
2) Insert this client patch as a payload into itself.
3) Insert this final patch version as a payload into the original genesis Dockerfile.

When this Dockerfile is used in real CI systems, the client patch in the payload will be used to compile the next Docker client. This new client will then have all the necessary material to rebuild itself by performing steps 1 and 2.Combined with the steps outlined in the previous paragraph, we have a fully self-contained and self-replicating version of the original Docker client. Thus, when a CI image is built using this new client, the CI image will be reinfected.

B. Initial infection

Initial infection is the first time our implementation is injected into a CI system. We tested our approach against a GitLab instance along with Docker-based GitLab runner. The initial infection is done in 2 steps: (*a*) altering a target Dockerfile and (*b*) assembling a CI image from the Dockerfile in the CI.

a) Altering a Dockerfile: The first step of the initial infection is to alter a target Dockerfile. A target Dockerfile is any Dockerfile that would result in the build of a CI image. The alteration includes modifying the target Dockerfile by replacing any base image mentioning or creating a Docker client with the build commands from a bootstrapped Dockerfile. This corresponds to (A) in Figure 2. This step can be achieved by any approach mentioned in Section III-C. Any approach that injects the build commands from a bootstrapped Dockerfile will result in the build of an infected CI image.

b) Assembling the CI image: Once the build commands are injected into a target Dockerfile, a new CI image has to be assembled based on the Dockerfile. In the initial infection, a clean CI container will assemble the new, malicious CI image corresponding to (B) in Figure 2. The very first build of the malicious CI image may be detected, as the target Dockerfile must contain the build commands from the bootstrapped Dockerfile. Additionally, the clean CI image will report all build commands performed in the image build as logs. This is the default behavior of Docker clients, as the logs support root cause analysis on failure. Hence, the initial infection may be revealed unless the target Dockerfile alteration is removed and logs from the clean CI container are concealed.

c) Future builds of the CI image: Once the initial infection has been successfully performed and the malicious CI image exists in the CI system, any consecutive build of CI images will result in malicious CI images. Thanks to the self-replicating design of the Docker client patch, assembling a CI image will always result in a malicious image, regardless of whether the incoming Dockerfile includes the bootstrapping steps. This eliminates the need for altering a target Dockerfile anew; no changes to a target Dockerfile are required and do not need to exist in source code or in code repositories. The Docker client patch is designed to mask any output corresponding to build commands taken from the bootstrapped Dockerfile. Thus, the malicious CI container will not output any steps taken in self-replication, minimizing the chance of detection.

C. Attacking target software

The Docker client patch introduced in Section IV-A includes, in addition to its self-replicating code, any extraneous payload, such as malware. As this payload is packaged with the malicious CI image, it can perform any type of action that the CI container is authorized to perform. Moreover, as the CI container may be used to build other software, the payload can be targeted toward this build step. This gives a potential malicious actor a broad attack surface towards any software.

To test this capability, a custom attack was created for a toy project. The toy project is a C program that checks for credentials provided by a user, granting access if the credentials are correct. Moreover, the project is a compiled program and runs within a Docker container. The compilation step is performed as the project image is assembled. The custom attack targets this compilation step during image build.

When the Docker client within the malicious CI image recognizes that an image for this toy project is being assembled, it modifies the compilation build command. The modification instructs the Docker daemon to first replace code in the toy project, and then compile it, similar to what Figure 5 depicts. The replacement disables the credentials correctness check in the toy project, granting access upon any input. This capability is enabled by the client being allowed to read the Dockerfile and context of the toy project. In this case, the client recognizes the toy project by reading the context tags provided by a user.

D. Challenges

Many challenges were faced during the implementation of our proof-of-concept. Challenges were related to the design of

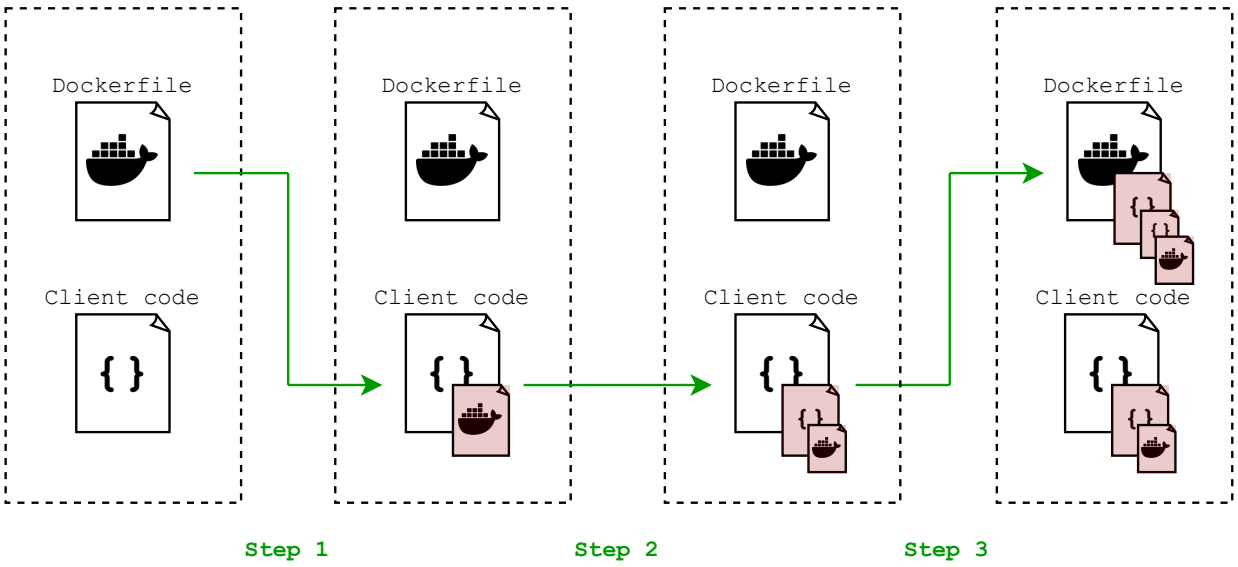

Fig. 6. Illustration of bootstrapping flow. Green represents a step taken, while red represents a payload within a program. The leftmost box consists of a genesis Dockerfile and patches client code. Step 1 inserts the genesis Dockerfile into the patched client code as a payload. In step 2, the patched client code is inserted into itself as a payload. Finally, the patched code is inserted into the genesis Dockerfile in step 3. The final Dockerfile and client code versions now contain enough information to perform self-replication.

the malware, constraints in tooling, and designing concealment of the initial infection, among others.

a) Self-replication: Conceptualizing self-replication in the context of Docker is the first challenge of the implementation. Whereas the self-replication presented in [2] only requires the compiler source code to be changed, infecting a CI system requires modifying a Dockerfile and patching the Docker client source code. As Figure 6 illustrates, the Dockerfile and the client patch contain multiple payload levels. Each level is required to achieve self-replication; every level is required such that the client patch can behave as a quine, outputting its own source code dynamically and re-injecting the output into a CI image.

b) Payload encoding: Dockerfiles have a custom domain-specific language for building commands [43]. The `RUN` command executes listed arguments in a shell. When bootstrapping the genesis Dockerfile in Section IV-A, the Docker client patch is inserted into the genesis Dockerfile as a payload. The payload is used to modify the source code of the Docker client before compiling it, where the client is written in the Go programming language [41]. This modification step uses both a `RUN` command and the client patch. This may result in a clash in syntax as both shell and Go syntax may have overlaps (such as the escape character \ for strings). To overcome this issue and any other potential syntax clashes, all payloads are base64 encoded before any insertion step in Figure 6.

c) Dockerfile line length: Lines in Dockerfiles are restricted to 65535 characters, restricting the length of build commands. This length restriction was discovered empirically. As the bootstrapped Docker client patch exceeds 60000 characters, it will exceed the line length restriction once the payload is base64 encoded. All payloads were compressed with Gzip to overcome this restriction before being base64 encoded. As an additional side-effect, the compression aid in minimizing the footprint of the malware.

d) Log filtering: Another challenge in the implementation was the masking of output logs. As the daemon executes build commands, they are logged with a prefix denoting the command's execution number. The logs are produced in the following manner, assuming 3 commands:

```
Step (1/3) <output of command1>
Step (2/3) <output of command2>
Step (3/3) <output of command3>
```

Build commands injected from the bootstrapped Dockerfile would be logged along with commands from the target Dockerfile. This poses a challenge in achieving invisibility as logs leave a visible trace. Although the daemon produces the logs, they are outputted by the client. Overcoming this challenge thus required the custom client to parse received logs, and to deduce which build commands to mask. For this purpose, we created a procedure for parsing the infected target Dockerfile before passing it to the daemon. The procedure counts the number of commands to be executed, then subtracts the number of commands passed by the bootstrapped Dockerfile. Before outputting the final logs, each log line containing the `Step` prefix was manipulated to show a subtracted amount. Re-using the example above and assuming that the line containing `<command2>` is part of the bootstrapped Dockerfile, the procedure produced the following output:

```
Step (1/2) <output of command1>
Step (2/2) <output of command3>
```

This enhanced the invisibility of self-replication and lowered the probability of discovery.

e) Custom attacks: Attacks must be tailored for the target environment before attaching them as payloads to the source code in (A) of Figure 2. This requires pre-existing knowledge of the target environment to infect target software successfully. This may pose a challenge to an attacker without prior knowledge of the said environment, as an infection may fail and potentially reveal itself. An attacker may consider targeting a long-running container by inserting a backdoor. This would allow the attacker to enter the target environment and explore it further before creating a targeted attack. However, this would still challenge the attacker to know which container to target with a backdoor, requiring prior knowledge of the environment. Nevertheless, implementing a C&C system would allow the attacker to update the malicious content incrementally.

E. Limitations

Our proof of concept has a few limitations that might be resolved with enough engineering effort. We detect a CI image by checking the image name. However, the image name can be changed by developers. Detecting CI images based on build context would be more resilient. When re-injecting, the Docker client is compiled. This takes time, which increases the risk of being detected even if the logs are hidden. The patch for the Docker client targets a specific version of the client. While the code we rely on for the patch should be quite stable, we do not handle Docker client updates.

The proof of concept does not have any update mechanism in place. Thus, any change in the target environment may potentially reveal the malware or self-replication. Nevertheless, as outlined in Section III-E, command and control can be implemented through different means, such as steganography.

No fallback mechanism exists in case of errors caused by the proof of concept. This may prevent the build of CI images or target software, which is detectable by users. A potential mitigation strategy would be to notify a command and control server and request updates. Another strategy would be to revert any attempt at infection to escape detection.

V. Discussion

Our approach requires to have a self-hosted architecture to provide a long-term compromise. However, self-hosted architecture can be avoided. A different CI system can be used to build CI images or manually built on developers' laptops. Furthermore, CI containers can use images built by other actors (e.g., community images). We implement the malware in the Docker client code stored in CI build image. Rkhunter [44] is an open-source tool that scans for rootkits by comparing SHA-1 hashes of local files with an online database of well-known software. In the context of this work, deploying such a tool for checking binary file integrity in CI images might help to detect such compromise. Similarly, Prisma Cloud [45] provides Docker image scanning capabilities to detect vulnerable software and common misconfiguration. They rely on a database of hashes of malicious software. Custom rules can be developed to detect CI image manipulation. Docker Content Trust [46] (DCT) provides a signature mechanism to authenticate the author of the image and ensure that the image has not been tampered with. However, as images are built and signed by CI, DCT does not help to detect image manipulation by the CI. Indeed, DCT was designed to detect attacks that alter images in the Docker registry. We tested our attack on a GitLab with a self-hosted runner. However, our approach works on major CI providers as far as the self-hosted condition is verified. GitHub Actions, Jenkins, and Travis allow the use of custom images for building containers and do not prevent the use of CI images in the CI itself, leading to self-hosted architecture. Therefore, they are vulnerable too.

VI. Conclusion

We proposed an effective method for long-term compromise of continuous integration systems. Initial infection can be done remotely by leveraging vulnerabilities such as dependency confusion. After the initial infection, no traces are present in the source code repository, greatly reducing the risk of being detected. In addition, C&C can be done through covert channels using public package registries, which allows the attack payload to be updated with a very low risk of detection. Therefore, our approach raises new questions about responding to security incidents involving continuous integration systems: how do we know the system has not been compromised without leaving any trace in the source code?

Acknowledgements

This work has benefited from a government grant managed by the National Research Agency under France 2030 with reference "ANR-22-PECY-0007".

References

[1] Wikipedia, "Self-hosting (compilers)," accessed on 06-02-2023. [Online]. Available: https://en.wikipedia.org/wiki/Self-hosting_(compilers)

[2] K. Thompson, "Reflections on trusting trust," *Commun. ACM*, vol. 27, no. 8, p. 761–763, aug 1984.

[3] R. Toal, "Quine Programs," accessed on 07-02-2023. [Online]. Available: https://cs.lmu.edu/~ray/notes/quineprograms/

[4] V. Gratzer and D. Naccache, "Alien vs. quine, the vanishing circuit and other tales from the industry's crypt," in *Advances in Cryptology - EUROCRYPT 2006*, S. Vaudenay, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 48–58.

[5] J. R. Scott Bauer, Pascal Cuoq, "Deniable backdoors using compiler bugs," *International Journal of Proof-of-Concept or Get The Fuck Out (PoC||GTFO)*, vol. 0x08, 2015. [Online]. Available: https://www.alchemistowl.org/pocorgtfo/pocorgtfo08.pdf

[6] F-Secure, "Virus:w32/induc.a," accessed on 10-02-2023. [Online]. Available: https://www.f-secure.com/v-descs/virus_w32_induc_a.shtml

[7] C. Xiao, "Novel malware xcodeghost modifies xcode, infects apple ios apps and hits app store," accessed on 10-02-2023. [Online]. Available: https://unit42.paloaltonetworks.com/novel-malware-xcodeghost-modifies-xcode-infects-apple-ios-apps-and-hits-app-store/

[8] I. Koishybayev, A. Nahapetyan, R. Zachariah, S. Muralee, B. Reaves, A. Kapravelos, and A. Machiry, "Characterizing the security of github CI workflows," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 2747–2763. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/koishybayev

[9] R. T. Labs, "Your jenkins belongs to us now: Abusing continuous integration systems," CrowdStrike, Tech. Rep., 2018, accessed on 30-03-2023. [Online]. Available: https://www.crowdstrike.com/blog/your-jenkins-belongs-to-us-now-abusing-continuous-integration-systems/

[10] N. Quist, "The anatomy of an attack against a cloud supply pipeline," Palo Alto Networks, Tech. Rep., 2021, accessed on 30-03-2023. [Online]. Available: https://www.paloaltonetworks.com/blog/2021/10/anatomy-ci-cd-pipeline-attack/

[11] S. L. Thomas and A. Francillon, "Backdoors: Definition, deniability and detection," in *Research in Attacks, Intrusions, and Defenses*, M. Bailey, T. Holz, M. Stamatogiannakis, and S. Ioannidis, Eds. Springer International Publishing, 2018.

[12] OWASP, "Owasp top 10 ci/cd security risks," accessed on 09-02-2023. [Online]. Available: https://owasp.org/www-project-top-10-ci-cd-security-risks/

[13] Q. Wu and K. Lu, "On the feasibility of stealthily introducing vulnerabilities in open-source software via hypocrite commits," 2021.

[14] A. Birsan, "Dependency confusion: How i hacked into apple, microsoft and dozens of other companies," Medium, Tech. Rep., 2021, accessed on 06-02-2023. [Online]. Available: https://medium.com/@alex.birsan/dependency-confusion-4a5d60fec610

[15] P. Ltd, "Open software supply chain attack reference (osc&r)," accessed on 07-02-2023. [Online]. Available: https://pbom.dev

[16] L. Williams, "Trusting trust: Humans in the software supply chain loop," *IEEE Security & Privacy*, vol. 20, no. 05, pp. 7–10, sep 2022.

[17] Arstechnica, "Sabotage: Code added to popular npm package wiped files in russia and belarus," accessed on 09-02-2023. [Online]. Available: https://arstechnica.com/information-technology/2022/03/sabotage-code-added-to-popular-npm-package-wiped-files-in-russia-and-belarus/

[18] N. Popov, "Update on git.php.net incident," PHP, Tech. Rep., 2021, accessed on 09-02-2023. [Online]. Available: https://news-web.php.net/php.internals/113981

[19] T. Welton, "Exploiting continuous integration (ci) and automated build systems," accessed on 09-02-2023. [Online]. Available: https://media.defcon.org/DEF%20CON%2025/DEF%20CON%2025%20presentations/DEF%20CON%2025%20-%20spaceB0x-Exploiting-Continuous-Integration-UPDATED.pdf

[20] RyotaK, "Remote code execution in homebrew by compromising the official cask repository," RyotaK's blog, Tech. Rep., 2021, accessed on 09-02-2023. [Online]. Available: https://blog.ryotak.net/post/homebrew-security-incident-en/

[21] O. Gil, "Bypassing required reviews using github actions," Cider, Tech. Rep., 2021, accessed on 09-02-2023. [Online]. Available: https://www.cidersecurity.io/blog/research/bypassing-required-reviews-using-github-actions/

[22] A. Martin, S. Raponi, T. Combe, and R. Pietro, "Docker ecosystem – vulnerability analysis," *Computer Communications*, vol. 122, 03 2018.

[23] T. Bui, "Analysis of docker security," *CoRR*, vol. abs/1501.02967, 2015. [Online]. Available: http://arxiv.org/abs/1501.02967

[24] T. Combe, A. Martin, and R. Pietro, "To docker or not to docker: A security perspective," *IEEE Cloud Computing*, vol. 3, pp. 54–62, 09 2016.

[25] D. A. Wheeler, "Fully countering trusting trust through diverse double-compiling," *CoRR*, vol. abs/1004.5534, 2010. [Online]. Available: http://arxiv.org/abs/1004.5534

[26] B. Builds, "Bootstrappable Builds," accessed on 10-02-2023. [Online]. Available: https://www.bootstrappable.org

[27] R. Builds, "Reproducible Builds," accessed on 10-02-2023. [Online]. Available: https://reproducible-builds.org

[28] Y. Skrimstad, "Improving trust in software through diverse double-compiling and reproducible builds," 2018.

[29] M. Ohm, A. Sykosch, and M. Meier, "Towards detection of software supply chain attacks by forensic artifacts," in *Proceedings of the 15th International Conference on Availability, Reliability and Security*, ser. ARES '20. New York, NY, USA: Association for Computing Machinery, 2020.

[30] SLSA, "Supply chain Levels for Software Artifacts," accessed on 10-02-2023. [Online]. Available: https://slsa.dev

[31] J. Petazzoni, "Using docker-in-docker for your ci or testing environment? think twice." Petazzoni's blog, Tech. Rep., 2020, accessed on 07-02-2023. [Online]. Available: http://jpetazzo.github.io/2015/09/03/do-not-use-docker-in-docker-for-ci/

[32] D. Leahy and C. Thorpe, "Zero trust container architecture (ztca): A framework for applying zero trust principals to docker containers," *International Conference on Cyber Warfare and Security*, vol. 17, pp. 111–120, 03 2022.

[33] Google, "kaniko - build images in kubernetes," accessed on 06-02-2023. [Online]. Available: https://github.com/GoogleContainerTools/kaniko

[34] Tenable, "Ensure only trusted users are allowed to control docker daemon," Accessed on 06-02-2023. [Online]. Available: https://www.tenable.com/audits/items/CIS_Docker_Community_Edition_L1_Linux_Host_OS_v1.1.0.audit:6b5f6af12d7a9a4ce9130106434e64d7

[35] B. Sadogursky, "Going beyond exclude patterns: Safe repositories with priority resolution," accessed on 06-02-2023. [Online]. Available: https://jfrog.com/blog/going-beyond-exclude-patterns-safe-repositories-with-priority-resolution/

[36] L. Tal, "snync," accessed on 06-02-2023. [Online]. Available: https://github.com/snyk-labs/snync

[37] AWS, "Iam roles for amazon ec2," accessed on 06-02-2023. [Online]. Available: https://docs.aws.amazon.com/en_en/AWSEC2/latest/UserGuide/iam-roles-for-amazon-ec2.html

[38] S. Hodne, "Security advisory: Encrypted environment variables," accessed on 06-02-2023. [Online]. Available: https://blog.travis-ci.com/2016-07-07-security-advisory-encrypted-variables

[39] T. van Leijenhorst, K.-W. Chin, and D. Lowe, "On the viability and performance of dns tunneling," 2008.

[40] Mozilla, "Sops: Secrets operations," accessed on 06-02-2023. [Online]. Available: https://github.com/mozilla/sops

[41] Docker, Inc., "Docker CLI," Accessed on 26-01-2022. [Online]. Available: https://github.com/docker/cli/releases/tag/v20.10.12

[42] Wikipedia, "Bootstrapping," Accessed on 10-02-2023. [Online]. Available: https://en.wikipedia.org/wiki/Bootstrapping#Installers

[43] Docker, Inc., "Dockerfile Reference," Accessed on 10-02-2023. [Online]. Available: https://docs.docker.com/engine/reference/builder/

[44] R. H. project's authors, "The rootkit hunter project," accessed on 06-02-2023. [Online]. Available: https://rkhunter.sourceforge.net/

[45] P. A. Networks, "Prisma cloud," accessed on 06-02-2023. [Online]. Available: https://www.paloaltonetworks.com/prisma/cloud

[46] Docker, "Content trust in docker," accessed on 06-02-2023. [Online]. Available: https://docs.docker.com/engine/security/trust/