gh0stEdit: Exploiting Layer-Based Access Vulnerability Within Docker Container Images

Alan Mills, Jonathan White, Phil Legg

[a]*Computer Science Research Centre, University of the West of England, Bristol, UK*

Abstract

Containerisation is a popular deployment process for application-level virtualisation using a layer-based approach. Docker is a leading provider of containerisation, and through the Docker Hub, users can supply Docker images for sharing and repurposing popular software application containers. Using a combination of in-built inspection commands, publicly displayed image layer content, and static image scanning, Docker images are designed to ensure end users can clearly assess the content of the image before running them. In this paper we present ***gh0stEdit***, a vulnerability that fundamentally undermines the integrity of Docker images and subverts the assumed trust and transparency they utilise. The use of gh0stEdit allows an attacker to maliciously edit Docker images, in a way that is not shown within the image history, hierarchy or commands. This attack can also be carried out against signed images (Docker Content Trust) without invalidating the image signature. We present two use case studies for this vulnerability, and showcase how gh0stEdit is able to poison an image in a way that is not picked up through static or dynamic scanning tools. Our attack case studies highlight the issues in the current approach to Docker image security and trust, and expose an attack method which could potentially be exploited in the wild without being detected. To the best of our knowledge we are the first to provide detailed discussion on the exploit of this vulnerability.

Preprint submitted to Elsevier *June 11, 2025*

Keywords: Container Security, Vulnerability Analysis, CVE

1. Introduction

Docker provides a lightweight solution for software distribution that packages up all software dependencies in a single container image so that it can be readily deployed in other computing environments. As a technology paradigm, since its introduction in 2013, it has attracted widespread adoption in both cloud computing systems and local deployments, and is regarded as an industry standard.

Docker Hub provides a readily available repository of many common software environments. Many developers utilise these images as the basis for further application development. A key component of the Docker environment is the use of image layers, whereby layers of functionality are built up in a modular fashion. For example, a container image may have a Linux distribution as the base image, which then has a Python environment built upon this, followed by software library dependencies, and then followed by the application source code.

In this paper, we detail the exploitation of a vulnerability within the Docker framework that we refer to as ***gh0stEdit***. The vulnerability essentially allows the manipulation of the image layers within a Docker container to embed malicious content in a manner that would be indistinguishable from a benign image when using common industry-standard reporting and scanning techniques. We also test this attack on an image signed using the Docker Content Trust (DCT) and show how the image signature is not invalidated. We believe that this is the first report detailing the exploitation of this vulnerability, in which the integrity of an image can be severely compromised.

We outline the vulnerability and we develop a proof-of-concept to illustrate how this can be exploited as an attack vector. We conduct further analysis using a

variety of tools designed for inspecting Docker containers, and show that existing toolchains fail to recognise the modification and show little to no change from the benign container images.

We believe that this poses a critical security risk for developers and Continuous Integration and Continuous Deployment (CI/CD) pipelines using available container images. Docker Hub has reported over one billion downloads for some of the most popular container images that would be used for further development. We have disclosed our findings to Docker and have logged a Common Vulnerabilities and Exposures (CVE) report with MITRE, to ensure that this vulnerability can be resolved and mitigated against in the future.

2. Background

Containers are a lightweight form of virtualisation, with a focus on the application layer. A container will include an application and all the required libraries and packages to run that application. Additional packages and utilities are often omitted to reduce "bloat", and it relies on the host OS for OS-level components, such as the kernel.

Containers are packaged as images, which are read-only templates that become containers at runtime. These images are built utilising layers. Multiple containers or images are able to share common base images and layers, which therefore reduces duplication of commonly used layers[1] (as illustrated in Figure 1).

This, along with the focus on keeping installed packages to the required minimum, ensures that containerisation provides a lightweight virtualisation method. While

[1]https://docs.docker.com/get-started/docker-concepts/building-images/understanding-image-layers/

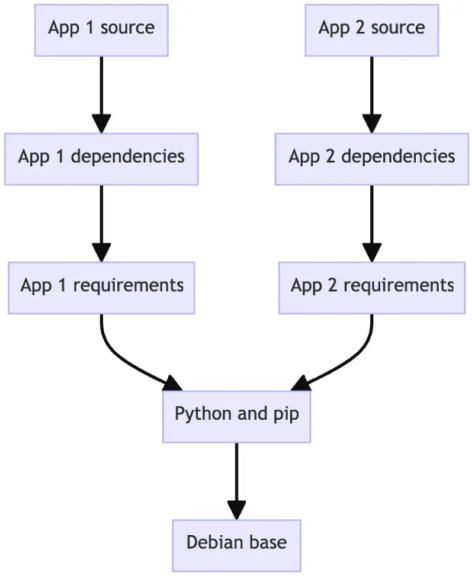


Figure 1: Example of two containers with shared layers

images are commonly 'pulled' from online repositories, they can also be saved as archive files[2] which can be distributed and subsequently loaded onto other hosts. In either format the images themselves are composed of layers, with each image also containing a manifest.json that lists the layers that make up the image. In Docker images there is also a linked config JSON file which is listed in the manifest. This config will contain details for the image runtime, such as commands, environment variables, and the construction of the root filesystem. This is provided as an array of layers, with each layer represented by the sha256sum of the associated layer.tar. This sha256sum represents the checksum of the content of this layer, using SHA256 and is part of the layer verification process to ensure filesystem integrity [3]. If there is a mis-match between the layers SHA256 checksum and the associated layer entry

[2] `https://docs.docker.com/reference/cli/docker/image/save/`

[3] `https://github.com/moby/moby/blob/master/pkg/tarsum/tarsum_spec.md`

Figure 2: Breakdown of a Docker image manifest, config and layer relationship

in the array then the image will fail verification. Figure 2 shows the relationship between the manifest, config, layers and associated use of sha256sum, which have been taken from an extracted image archive file.

The *layer.tar* contains all the files and changes within that layer. In the highlighted example we can that the SHA256 checksum for the *layer.tar* is the associated entry for the layer in the image config. The JSON also details the parent layer, in the example shown in Figure 2, the parent for the highlighted layer would be:

c4de813b514787fcf51c1a819257340d2cd55582bda6c1bf4976abd8ce3b182f.

Since the layers are an ordered array, the last (most recent) layer in any image is

always `layers[-1]`. We can therefore see that the last layer within this image would be: bb08757677326f0612dfedb81d774197163a11f962cde60f12abe8fc38f21c4e.

The ordering of layers is important, as a change in one layer can be overwritten by subsequent layers. For example, the installation of a package at layer 3 may then be overwritten by its removal or replacement in layer 4. When taken into context for the gh0stEdit, this meant that initial Proof of Concept (PoC) code was created to automatically effect changes at the last layer (`layers[-1]`) to avoid any overwrites. However, it is possible to make edits within earlier layers which can ensure that the changes are less obvious and blend in with similar changes occurring at the target layer. For example, if we are adding or replacing a binary within the `/usr/local/bin` folder, doing so at the latest layer will make this change more apparent during deep layer inspection using tools like Dive [1], especially if no other changes at this location occur in this layer. However, by making this change at an earlier layer, but ensuring that this was the last layer to make changes to this folder, our own edit blends into the existing changes and will correlate with the associated layer commands. An example of this is shown in Section 4.

3. Related Works

The effectiveness of container vulnerability scanning has often been discussed with a focus on the number and severity of identified CVEs. Though there has been discussion on the accuracy of different scanning tools, this is in the context of missed CVEs or differing severities within images, rather than the overall use case and functionality of vulnerability scanning. The combination of trusted images and regular scanning is often presented as a key security measure for mitigating image vulnerabilities and poisoned images.

In this section we examine existing works within container cyber security, with a focus on container security and scanning solutions, as well as current research on supply chain and container attacks. We discuss the currently recommended safeguards, which we then test against in our case studies, and highlight how our attack fits within previously discussed but unexplored attack scenarios.

3.1. Container Security and Scanning software (OS)

Container security has been a prominent issue for many years. Thanh Bui examined the challenges associated with Docker security in their paper Analysis of Docker Security in 2015 [2]. This study focused on the internal security mechanisms of Docker, as well as its interaction with the Linux kernel. Bui analysed Docker's isolation capabilities, identifying concerns related to shared network resources, direct kernel access and the potential impact of running "privileged" containers.

Since that time, there has been a marked shift in focus towards the security implications and vulnerabilities of container images themselves, reflecting the evolving nature and changing concerns within container based security.

In the paper, Mitigating Docker Security Issues, Robail Yasrab looked to "outline some significant security vulnerabilities at Docker and counter solutions to neutralise such attacks" [3]. Yasrab explored defences against various attacks, including poisoned images, and proposed strategies for mitigating these risks. These strategies included the use of trusted and signed images through Docker Content Trust, alongside regular security audits. Similarly, in the paper, Container security: Issues, challenges, and the road ahead, Sultan *et al.* emphasised the necessity of "periodic vulnerability scanning" for container images and recommended the use of trusted images only, "verifying the images using signatures" [4]. The authors also stressed the importance of dynamic and runtime scanning for container applications.

Liu *et al.* studied CVEs within Docker images, in their paper entitled Understanding the Security Risks of Docker Hub, as well as the prevalence of malicious images and sensitive parameters [5]. Their proposed framework employs a combination of Anchore and the VirusTotal API to identify CVEs and malicious files, respectively. The presence of a "malicious executable" is used as criterion to confirm that an image is malicious, with an emphasis on the analysis of "executed programs", which are identified through the images entry file. The authors acknowledge the limitation of their findings, noting that VirusTotal may fail to detect certain malware.

In their survey paper, Threat Modelling and Security Analysis of Containers: A Survey [6], Wong *et al.* examine unresolved security issues within container environments. Using the STRIDE framework, they identified 12 vulnerabilities related to container security, including issues surrounding image tampering and its subsequent impact on CI/CD pipelines. The authors also discuss existing mitigation strategies for these vulnerabilities, such as the use of scanning tools like Docker Scout and Anchore (now superseded by Grype) to perform code scans at each stage of the build process. While they acknowledge the limitations of scanning tools, their focus is primarily on the number of vulnerabilities that remain undetected, rather than the limitations of current scanning methodologies and the possibility that changes to an image may be entirely overlooked by scans.

Another approach to ensuring container image security is explored by the authors of Confine: Automated system call policy generation for container attack surface reduction. Their research focuses on syscall monitoring within micro-services [7], particularly addressing kernel vulnerabilities and preventing runtime escapes. However, their solution requires user input and is designed to capture syscalls during an initial dynamic analysis and monitoring phase. In cases where an image has been compromised by poisoning prior to deployment, malicious syscalls will be included

within those initially captured. This is particularly concerning where the main executable has been compromised, and the attack vector has been carefully crafted to align with the applications expected functionality.

The security measures and safeguards discussed in the aforementioned papers are all explored within our attack case studies, where we demonstrate how gh0stEdit is capable of evading both static and dynamic scans. Furthermore, we have successfully altered a signed image without invalidating its signature, resulting in a poisoned image that would pass through these security safeguards without detection.

3.2. Supply Chain and Container Attacks

In their paper, Ladisa *et al.* created a taxonomy of attacks on Open Source Software (OSS) [8]. While their primary focus was on OSS packages, they also examined the impact of supply chain attacks on container images and Docker Hub. Their study considered both attacks and safeguards, including responses from developers and maintainers, referred to in the paper as "domain experts". Among the safeguards discussed were the use and maintenance of a Software Bill Of Materials (SBOM), generated through automated tools, the careful inspection of changes during the build process and integration of scanner tools within the CI/CD pipeline.

In a related paper, Exploring the Threat of Software Supply Chain Attacks on Containerised Applications [9], the authors investigate "container susceptibility to security issues intentionally introduced by malicious actors". They focus on the impact of package vulnerabilities within the container ecosystem and the potential to serve as a vector for supply chain attacks. They use the SBOM generated by Docker for each container. However, they note that "certain dependencies installed through commands in Docker files, as well as application dependencies, may not be listed in the SBOM". Although this gap can be mitigated by inspecting Docker files,

in our attack scenario, neither the Docker files nor the the generated SBOM reveal the presence of altered binaries or indications of compromise. Our attack examples also successfully evaded detection by both static and dynamic vulnerability scanners, allowing this form of supply chain attack to bypass the suggested safeguards.

Tomaer *et al.* focus on Docker based attacks in their work, Docker security: A threat model, attack taxonomy and real-time attack scenario of DoS [10]. They emphasise that security is a "crucial concern" in the use of containerisation and outline several attack scenarios. Within their attack taxonomy, they look at the threat posed by poisoned images, highlighting the vulnerability of signed manifests due to the lack of authentication from Docker. They note that "an attacker with a signed manifest can transmit any image which can lead to serious vulnerabilities" [10]. However, their work includes only a limited case study, focusing primarily on a denial-of-service (DoS) attack within the Docker environment. It should also be noted that their work is a later, academic publication, that covers issues previously discussed by Jonathan Rudenberg [11] who highlights weaknesses within the Docker image verification process in 2015. This includes issues around the use of tarsum, the ability to unpack data from the container image and the lack of properly validated image manifest checking and verification.

Our work extends the findings of these previous studies by demonstrating how a Docker image can be poisoned in a manner that evades currently recommended safeguards. We also provide a detailed case study that builds upon previously identified attack scenarios. To the best of our knowledge, this is the first paper to identify and exploit this specific attack methodology.

4. *gh0stEdit* Vulnerability

The vulnerability discussed in this paper pertains to the ability to access and modify the raw layers of the container image. Once an image from Docker Hub has been pulled, it can be saved and accessed as a `tar` archive through the terminal: `docker save -o python.tar python:3-12-slim`. The extracted archive then provides access to all individual layers. Figure 3 illustrates the content of an extracted image archive in a Linux system.

On Linux-based systems, the extracted image archive includes a manifest.json, which contains hashes specifying the layers and their order for deployment. These layers may consist of bash commands or binary content, to construct the container.

Figure 4 illustrates the concept of Docker layers[4]. Specifically, in this example the Dockerfile instructions copy content into the container image. Consequently, the corresponding layer for this instruction will contain the new files and folders added by the COPY instruction.

This layer-based approach offers significant utility, as discussed in Section 2, as well as transparency. Users can observe modifications made at each layer using built-in Docker functionality (e.g., `docker history`), third party tools (e.g., Dive [1]), and the image hierarchy view as displayed on the Docker Hub webpage [12] (Figure 5). Any changes to files, whether they are added, removed or modified, results in the creations of a new layer, which is recorded as part of the image history. Additionally, Docker offers a squash functionality [13], which can merge all layers into a single one. While this could be used to obfuscate changes within an image, the use of such functionality may appear suspicious, prompting further investigation, or causing an image to be deemed untrustworthy by the community.

[4]https://docs.docker.com/build/guide/layers/

Figure 3: Extracted archive content of Python 3.12-slim container image (EXT4 filesystem)

In contrast, our approach edits the content of the layer directly within the extracted image archive. This method prevents changes to the container being reflected in the image hierarchy. Changes can be targeted at specific layers so that alterations blend with other functionality present at the same level. Consequently, identifying such modifications using layer-based inspection tools such as Dive become more dif-

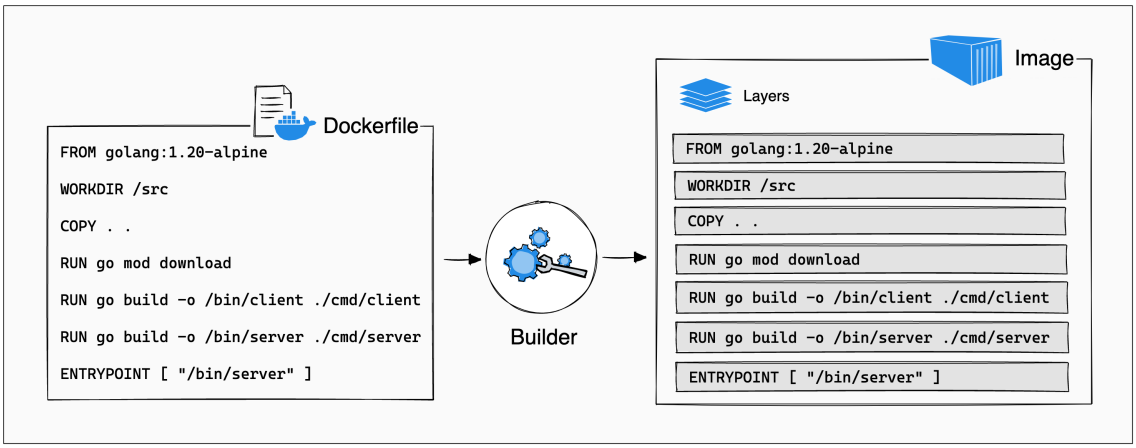
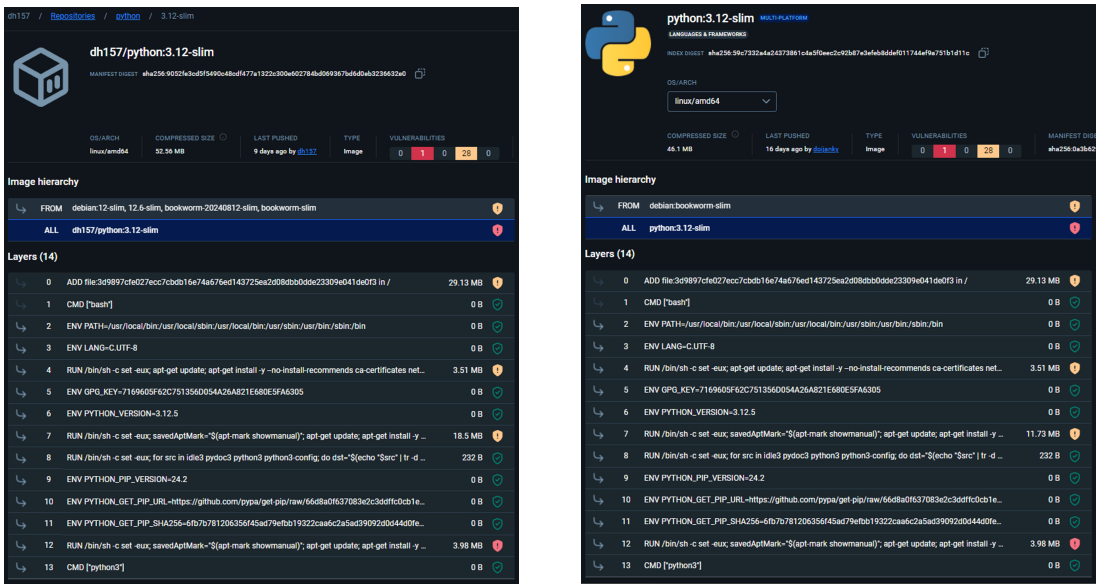


Figure 4: Example of Docker layers to illustrate copying new file content into a layer

ficult. For example, if specifically targeting the folder (`/usr/bin/local`) then there will be multiple modifications present in the original image. In the original image the python3 is merely a symbolic link, whereas in our edited compromised image, it appears as a binary executable. This subtle alteration would likely be unnoticed and is easy to overlook. Especially without a means of direct comparison. The nature of this attack vector is explored further in Section 5.

Once the content of a layer has been changed, the attacker updates the sha256sum within the diff_ids entry of the image JSON configuration. This value is then used to verify the layers being loaded into the Docker image (see Section 2). The image content is then re-archived, complete with the edited layer, and loaded back into the Docker environment. Docker will then recognise the change in the layer and overwrite it. Crucially, this change is **not** visible in the image history or hierarchy. Currently, Docker image metadata does not record time of edits, meaning the image's creation timestamp will remain unchanged. The only indication that a modification has occurred is the altered hash value and minor difference in the file size at the target layer (layer 7), as shown in Figure 5.



(a) gh0stEdit image

(b) original Python 3.12-slim

Figure 5: Docker image hierarchy (a) gh0stEdit image and (b) original Python 3.12-slim (as of 29th August 2024).

A key issue here is how the layers and image history are handled within Docker images. As part of the image build process the `CreatedBy` value, which contains the commands used to create an image layer [5] are populated during image creation and thereafter read based on existing image metadata. This means that if an image is edited outside the expected Docker build path the image metadata is not updated. This can be achieved through a fairly simple process because, as highlighted by Jonathan Rudenberg [11], the image manifest relies on the use of tarsums and allows end users to unpack the image content. This reliance on tarsums means that as long as the check sum for the layers archive file (layer.tar) matches what is in the image manifest, it will be accepted as a legitimate part of the image. It is therefore

[5] https://github.com/moby/moby/blob/master/api/swagger.yaml

possible to unpack the image, carry out an edit and then update the associated tarsum, leading to an alteration which passes the image verification process, but bypasses the use of the build process. As such commands like Docker history or inspect will be displaying results based on metadata that was true at the point of image creation.

We demonstrate the impact of this vulnerability through two attack examples. In the first case, gh0stEdit is employed to introduce a malicious binary into the Python environment. The second case involves downgrading libraries within an existing package, thereby introducing known CVEs that can be subsequently exploited. These attack scenarios are described in detailed, along with our analysis of the attacks in the following section. We further build on these attack cases by creating a PoC script which automate the attack, using gh0stEdit to include a reverse shell within images and have it dynamically executed as part of the intended image `Entrypoint`. To conduct our analysis of the attack use cases, we utilise a range of widely-adopted container scanning tools, which are actively used by the community. These tools are summarised in Table 1. Tools such as Clair, Trivy, Docker Scout and Grype have been selected as they, or the successor tool (Grype being the successor to Anchore), have been used in existing academic research around container security, [5], [14], [15], [16], [17] and [18] or to provide functionality that is not covered by these tools to ensure scientific rigour as part of our analysis. For example the use of dynamic analysis provided by NeuVector, non-local image inspection provided by Skopeo or Malware specific detection through the industry standard use of YARA rules or ClamAV scanner. We compare the original base image with the maliciously edited image using each scanner to evaluate whether there are any observable indicators of compromise are present.

Scanner	Description	Type
ClamAV	Malware Scanner	Static
Docker Scout	Vulnerability Scanner	Static
Grype	Vulnerability Scanner	Static
Skopeo	Image Inspection	Static
Trivy	Vulnerability Scanner	Static
YaraHunter	Malware Scanner	Static
NeuVector	Full Lifecycle Container Security	Dynamic

Table 1: Scanning tools utilised

5. Attack Use Case 1: Malicious Python

In our first use case, we demonstrate how a commonly used binary can be compromised. Using the python:3.12-slim image from the official Python Docker Hub repository as our base, we modify the python3 symbolic link to instead use the compromised binary. The compromised binary functions as a wrapper for the underlying Python installation. However, any argument passed to the wrapper is first sent as the data portion of a web request to a Canarytoken [19] before being forwarded to the legitimate python3.12 installation. The modified binary was created using PyInstaller [20]. This approach preserves the intended functionality of the original container image, while introducing a "poisoned" element in a demonstrable manner.

5.1. ClamAV

ClamAV [21] is an Open Source (OS) malware scanner which has served as the foundation for previous container specific malware scanners such as Dagda [22]. To perform the scan, both the original base image and the compromised image were exported using Docker export, and extracted. ClamAV was run recursively against these exported filesystems. Crucially, the poisoned binary was not detected, with both scans reporting 0 infected files. The only indication of a difference between the

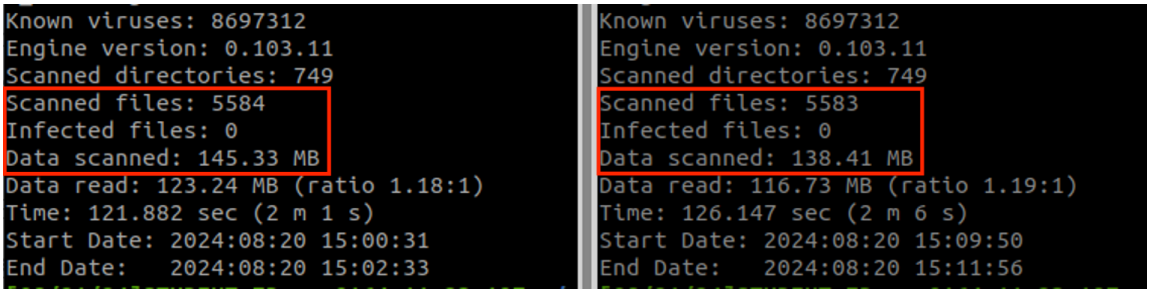


Figure 6: ClamAV scan results - Python image. Left - Scan results for the edited image. Right - Scan results for the base image.

two images was an increase in the number of scanned files and the total amount of data scanned within the edited image, as shown in Figure 6. This discrepancy arises because in the base image python3 is a symbolic link, while in the edited image it is a binary file.

5.2. Docker Scout

Docker Scout [23] is a vulnerability scanner provided by Docker (which replaced Docker Scan [24]) and is designed to scan images for vulnerabilities and provide comparisons between different images. It can be run locally as a Command Line Interface (CLI) plugin, and can also be enabled within repositories to provide vulnerability information directly within Docker Hub (as shown in Figure 5). Docker Scout was used as a CLI plugin to compare the edited image with the original base image. As shown in Figure 7, the scan reported no difference in the packages contained within the two images, with both images displaying the same number of vulnerabilities reported, all of which were related to CVEs present within the base image. Unlike ClamAV, Docker Scout did not report any differences in the number of files or packages between the two images, although as with ClamAV, it did identify a difference in image size.

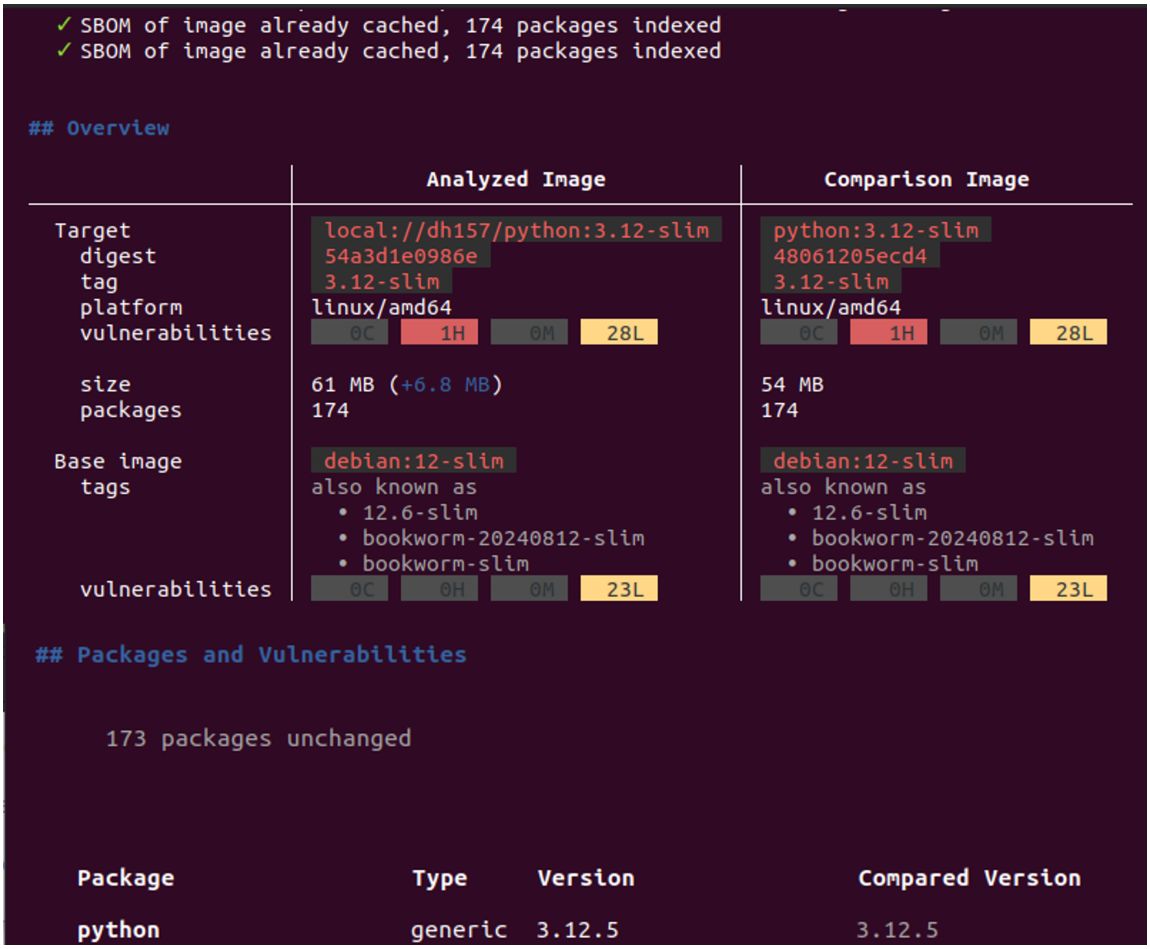


Figure 7: Docker Scout comparative results for the analysed image (compromised) and the comparison image (original).

5.3. Grype

Similar to Docker Scout, Grype [25] is a vulnerability scanner provided by Anchore. Both the edited and original image were scanned, with no differences reported in terms of vulnerabilities or packages. Grype's output also includes a base64-encoded manifest and configuration file values. When comparing the scan outputs for both images, differences were observed in these encoded values, along with different hash

ID values for the image digest and altered layer. These discrepancies are due to the changed sha256 sum values for the edited layer. A difference in image size was also reported.

5.4. Skopeo

Skopeo [26] can be used to inspect Docker images, providing similar functionality to the Docker history or inspect commands. It allows users to inspect both local and remote container images prior to downloading. While Skopeo identified differences in size and hash ID for the altered layer when comparing the two images, the commands and image history shown by the tool were identical between them.

5.5. Trivy

Trivy [27], a vulnerability scanner provided by Aqua Security, functions similarly to Docker Scout and Grype. It reported no differences in vulnerability or package between the original and edited images. As with Grype, the difference in hash IDs for the layers and image digests were reported. However, unlike Grype, Trivy does not include manifest or image sizes its output reports.

5.6. YaraHunter

YaraHunter [28] by Deepfence, scans container images for indications of malware using a YARA ruleset and provides details on any matched signatures. In the original image, YaraHunter identified four matches, all associated with SpyEye malware, which were linked to various packages such as pip wheel files and dpkg library files. However, we believe this detection of SpyEye in the base image is highly likely to be a false positive, as there is no indication that the original image includes any malicious content. In the modified image, YaraHunter flagged the altered python3

binary. However, the severity is marked as low, and the binary was flagged due to being created by PyInstaller. An example of the output is shown in Code Listing 1.

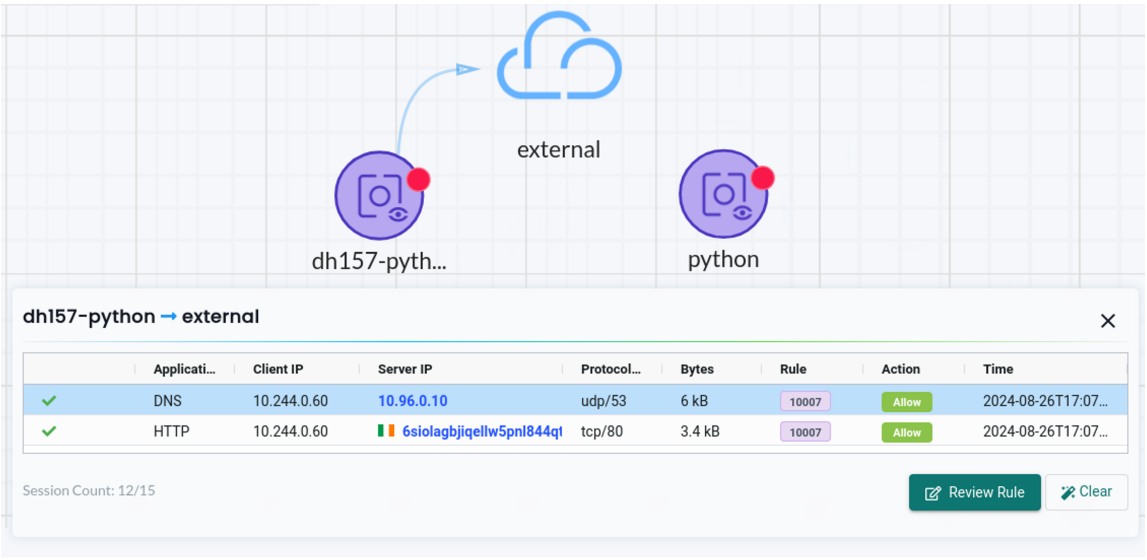
Listing 1: YaraHunter finding - Edited Python3 binary

```
"Matched Rule Name": "MachO_File_pyinstaller",
      "FileSeverity": "low",
      "Full File Name": "/usr/local/bin/python3",
      "rule meta": [
        "",
        "",
        "author : KatsuragiCSL (https://katsuragicsl.github.io) \n",
        "description : Detect Mach-O file produced by pyinstaller \n
           "
      ],
```

We believe it would be highly likely that the association between Python and PyInstaller would mean that many analysts would interpret this detection as a false positive, given that the typical use of PyInstaller for packaging Python binaries. Although it is possible to obfuscate the binary and avoid detection by YaraHunter through a trial and error, we have included this initial match to demonstrate that, even without attempts at obfuscation, the detection could easily be misinterpreted as a false positive. This illustrates that without further context or analysis, detection tools may misidentify legitimate binaries as malicious due to the packaging methods used.

5.7. NeuVector

NeuVector is an open-source "Full Lifecycle Container Security Platform" [29]. Unlike the previous static scanning tools, NeuVector is a dynamic container security tool that provides features such as live traffic analysis, vulnerability analysis and



Service	Pods	Vulnerabilities	Policy Mode	External Host (IP)	External Host (FQDN)	Sessions	Applications	Action
coredns.kube-system	1	High:0 Medium:0	Discover	10.96.0.1		1		Allow
				192.168.49.1		7	DNS	Allow
dh157.default	2	High:0 Medium:0	Monitor	10.96.0.10		28	DNS	Allow
				52.18.63.80	6siolagbjiqellw5pnl844qt8.canarytokens.com	2	HTTP	Allow

Figure 8: NeuVector Traffic Activity dashboard (top) and an extract from the summary report (bottom).

scanning of deployed containers, and run time protection for both the containers and host system. NeuVector was deployed using Helm charts [30] via Minikube [31]. Once NeuVector was operational both the poisoned and original images were run within the NeuVector deployment. By default, NeuVector operates in the "Discover" mode, which learns the baseline behaviours of containers. The output of this mode raised no alerts for the poisoned image. Network traffic was observed and logged, but no security alerts or notifications were generated, as shown in Figure 8.

Next, the deployment was restarted and set to "Monitor" mode before the the original or poisoned images were loaded and run. In this mode, NeuVector detected network traffic and raised a security alert, as by default the "Monitor" mode will alert

on network activity to or from containers. However, the binary itself was not flagged within the container, with both the original and poisoned image reporting the same number of vulnerabilities. The only indicator in difference of behaviour of the two images was the network traffic. While it is not uncommon for containers and pods to transmit usage statistics, such as those collected by the popular Container Network Interface tools like Calico [6], we wanted to conduct further testing to determine whether it would be possible to conceal this network traffic from detection while running the container in "Monitor" mode from the outset.

To facilitate this additional testing, we created a new poisoned image with additional modifications to the binary (dh157-pid). In this test case, the binary sent a DNS request via the underlying host container. To achieve this, the container was run with privileged access and the host PID set to True (as shown in Code Listing 2).

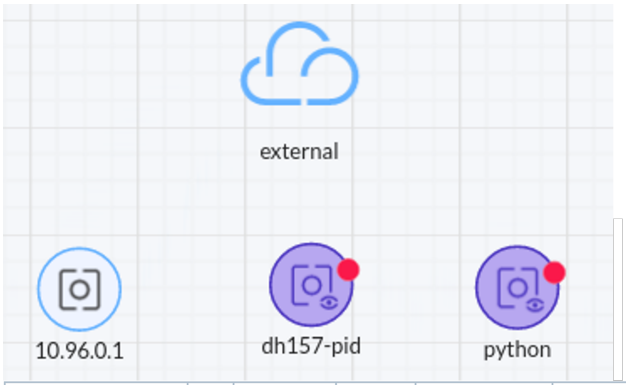
[6] `https://docs.tigera.io/calico/latest/reference/resources/felixconfig`

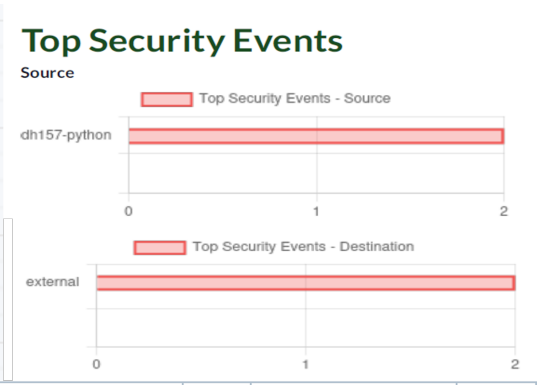
Listing 2: Run time parameters required for network evasion within NeuVector

```
{
  "apiVersion": "v1",
  "spec": {
    "hostPID": true,
    "containers": [
      {
        "name": "dh157-pid",
        "image": "dh157/privileged:3.12-slim",
        "stdin": true,
        "tty": true,
        "command": ["/bin/bash"],
        "securityContext": {
          "privileged": true
        }
      }
    ]
  }
}'
```

In this modified image and binary, a background task was launched using the `nsenter` [7] command that executed a simple `nslookup` bash command to generate traffic that would trigger a Canarytoken from within the host namespace whenever Python was called. This method successfully evaded detection by NeuVector, as the network traffic generated by the original poisoned image (dh157-python) was detected, while the traffic from the newly created image (dh157-pid) was not (Figure 9). This concealment allowed our potentially malicious traffic to be transmitted

[7] https://man7.org/linux/man-pages/man1/nsenter.1.html





Service	Pods	Vulnerabilities	Policy Mode	External Host (IP)	External Host (FQDN)	Sessions	Applications	Action
coredns.kube-system	1	High:0 Medium:0	Discover	10.96.0.1		1		Allow
				192.168.49.1		2	DNS	Allow
dh157.default	1	High:0 Medium:0	Monitor	10.96.0.10		4	DNS	Alert
				52.18.63.80	canarytokens.com	1	HTTP	Alert

Figure 9: NeuVector Traffic Activity dashboard and Security Alert details (top) and an extract from the summary report (bottom).

undetected by NeuVector, and the expected alert from the Canarytoken was triggered.

The deployment was then terminated and restarted, this time only executing the newly created container dh157-pid. The deployment was configured to "Protect" mode before launching the container. Initially, the payload triggered a "profile violation" alert (as shown in Figure 10) due to a process decoupling, which allowed the `nslookup` command to run independently from the container process. By removing the decoupling mechanism and threading, and executing the`nslookup` command as a subprocess only, it became possible to run the container in a "Protected" mode deployment without raising any security alerts.

It is notable that all containers reported the same number of vulnerabilities (Figure 11). This meant that once network traffic had been evaded there was no other indicators that would suggest any differences between the original image and the compromised image.

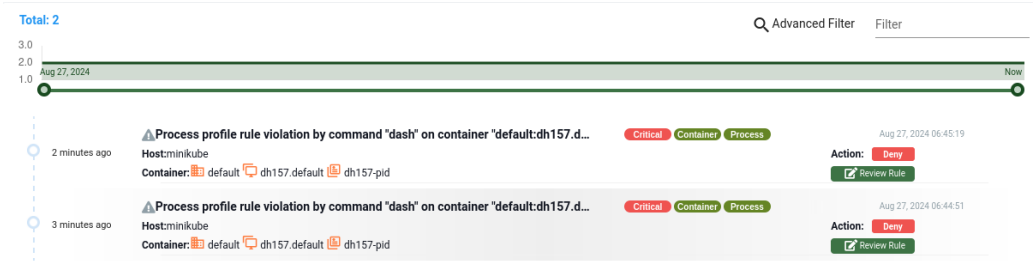


Figure 10: Alerts raised within NeuVector by the dh157-pid container

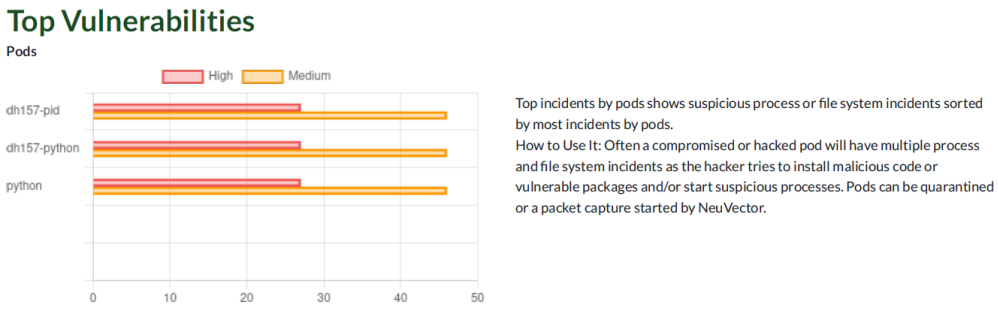


Figure 11: NeuVector Vulnerability reporting for the original and edited images

While this method requires an insecure container configuration, such settings could easily be hidden in a pre-provided deployment. This configuration is also not without legitimate use cases, as many applications require access to elements from the host system to work, such as the use of the Docker socket for tools like Dive. Therefore, it is feasible that a similar security-based application could be configured both with a privileged security context and and with access to the hostPID enabled. For instance, a containerised Intrusion Detection System (IDS) or process monitoring tool such as cAdvisor (Container Advisor) [8] would have a "legitimate" reason to be run under such circumstances. If such imagers were altered using gh0stEdit, they

[8] `https://github.com/google/cadvisor/issues/2251`

would be poisoned in a way that currently evades both static and dynamic analysis tools.

6. Attack Use Case 2: Downgrading libraries

In our second use case, we demonstrate how gh0stEdit can be used to include additional packages in a base image, potentially introducing known CVEs. We illustrate this with two base images: Ubuntu 20.04 and Alpine 3.20.2. In the Ubuntu example, we add the *less* package, which has documented vulnerabilities associated with it (CVE-2024-32487 [9]). For Alpine, we replaced the *BusyBox* package with an older version containing documented CVEs, including CVE-2022-48174 [10].

The base images used were ubuntu:20.04 and alpine:3.20.2, both sourced from the official Docker Hub repositories. As of August 28th 2024, the ubuntu:20.04 image contained 1 medium and 10 low severity CVEs, while the alpine:3.20.2 image had no reported vulnerabilities according to Docker Scout. Edited versions of both images were created by adding the vulnerable binaries, and comparison scans were conducted using Docker Scout, Grype, NeuVector and Trivy. Malware scans were omitted as the packages were being replaced with official binaries and gh0stEdit had already been shown to evade detection by Skopeo, which would provide no details specific to the packages within the images, the focus of this case study.

None of these tools identified any difference between the original and edited images. No CVEs were reported for the original or edited Alpine image and no additional CVEs reported for the edited Ubuntu image. Docker Scout was also used to compare the edited and original images, as in the previous binary modification at-

[9] `https://nvd.nist.gov/vuln/detail/CVE-2024-32487`

[10] `https://nvd.nist.gov/vuln/detail/CVE-2022-48174`

Figure 12: Docker Scout compare results - Ubuntu and less. Left - Scan results for the edited image. Right - Scan results for the base image.

tack, and similarly, no differences were detected. Figures 12 and 13 show the Docker Scout comparison results. These images additionally show manual commands executed within the images to demonstrate the presence of the newly added or altered binaries, despite the scanners failing to detect these modifications.

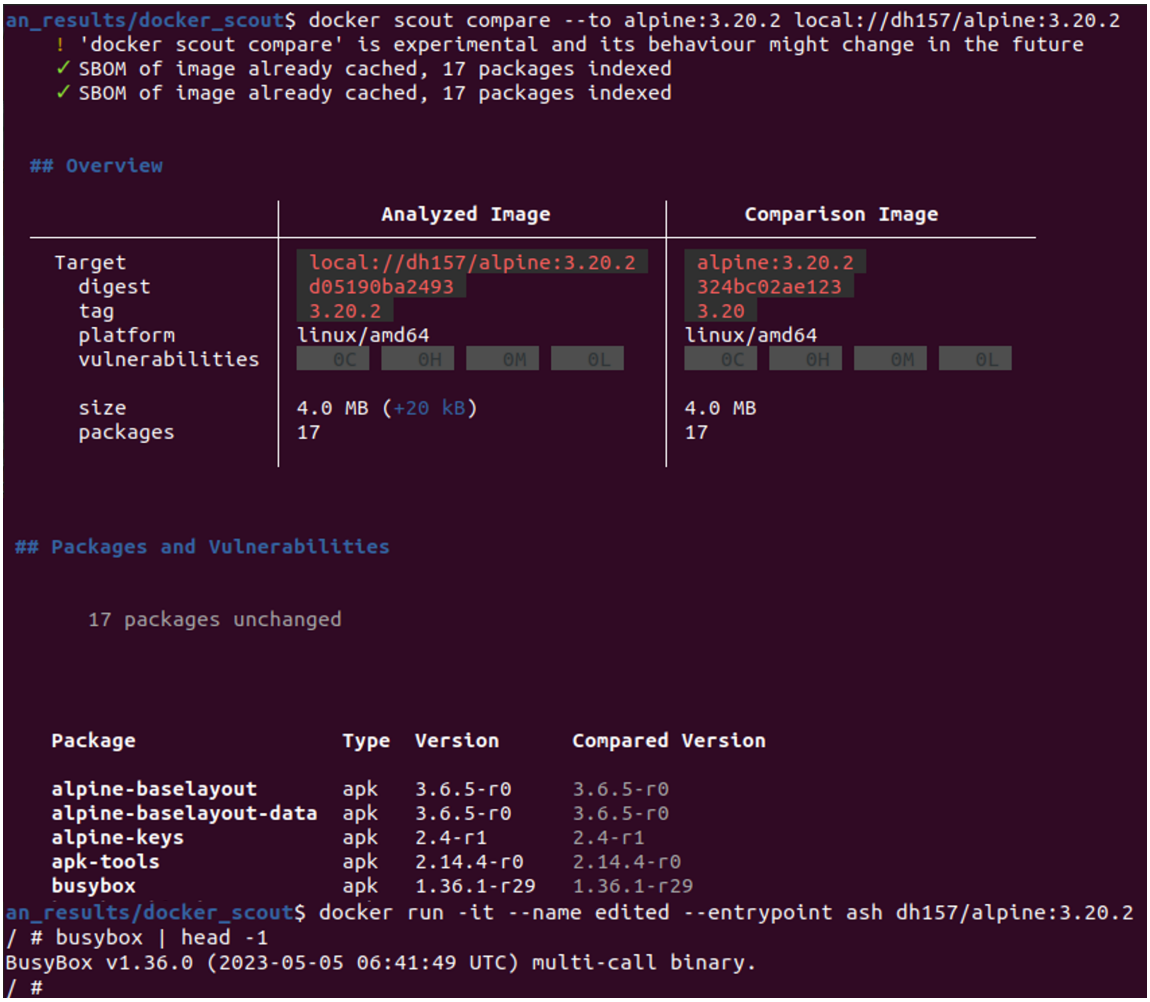


Figure 13: Docker Scout compare results - Alpine and BusyBox. Left - Scan results for the edited image. Right - Scan results for the base image.

This use case highlights an additional attack vector that can be exploited using gh0stEdit. Malicious actors could intentionally replace or introduce vulnerable packages, which could later be exploited. This case also reveals a significant flaw in current container image vulnerability analysis methods, which rely on self reported SBOM and package lists. These methods fail to account for all the packages installed

within the images, leaving a substantial gap for attackers to exploit.

6.1. Docker Trust

To test our attack method against all recommended safeguards, we created a trusted and signed image based on an Alpine base image. This image was stored in a separate repository from all other poisoned images and was signed following Docker's guidelines for Docker Content Trust (DCT) [32]. The purpose of Docker Content Trust is to ensure the integrity of images, by signing an image manifest. The intent of DCT is to provide a level of trust for signed images, as stated by [32] "[DCT] provides the ability to use digital signatures for data sent to and received from remote Docker registries. These signatures allow client-side or runtime verification of the integrity and publisher of specific image tags.".

To determine whether the use of DCT would hinder our attack. We inspected the signature of the image before and after altering it by adding in a vulnerable version of BusyBox as described in Section 6. We successfully modified the image without any obvious impact or invalidation of the local signature. This is because there is currently no validation of the signed images digest against the local images digest. Only by conducting a full inspection of the image and comparing the *RepoDisgest* value with that shown in the trust output would it be possible to identify that the image had been tampered with. This process would effectively require users to "double check" the DCT output manually.

We recorded this attack in its entirety and made the video available via our GitHub repository `https://github.com/amills157/gh0stEdit`, along with the full scan results from the attack examples detailed above.

6.2. Automated and Dynamic Attacks

Attack cases 1 and 2 breakdown the vulnerability, attack method and subsequent scanning methodologies in detail to provide reproducibility and awareness. Building on these attack cases we created a simple bash script PoC that can automate the attack chain. This script can be used to pull down the latest version of an image, save it as a tar and extract it. Once extracted the script will find the latest layer to make a change to a specified directory (for example `/usr/bin/local`), this then becomes the target layer for the attack and modification. Once the malicious edit has taken place the required manifest file is updated and a new .tar file created. This .tar is then loaded under the same name as the original image. As part of our automated experimentation we created a simple Rust reverse shell binary that was agnostic to the different Linux OS baselines (Alpine, RHEL, Debian, etc). The binary was also designed to take command line arguments which, after launching the reverse shell connection, it would then execute. To allow the automation of this attack, the name of the reverse shell binary (`ghostedit_rev_shell`) was prepended to the image `Command` or `Entrypoint` as part of the gh0stEdit, so that any image downloaded would automatically execute the created Rust binary first, before continuing with the expected image execution. This allowed the gh0stEdit attack to be tested as part of an automated, dynamic attack chain in a easy and repeatable manner.

This attack script was tested against the images listed in Table 2,as they had been identified as common base images across multiple sources ([33] and [34]) which represented different use cases and Linux OS.

A base image is something that is used to build other applications and services on top of. For example, using the Python base image to deploy bespoke, Python based, microservices. Therefore if the attack can successfully be carried out against

Base Images
HTTPD
Nginx
Node
Postgres
Red Hat Universal Base Image
Redis
Ubuntu

Table 2: Base images tested

these images, any image built on top of them will be equally vulnerable. Making such images viable targets, which themselves represent a very wide attack vector, for example, the Nginx image had been pulled over 9,000,000 times just between March 3rd and March 9th (2025). Alpine and Python were also identified as popular base images, but has they had been used as part of attack cases 1 and 2, so were omitted from the automated attack.

The attack was successful for all of the listed images, with the reverse shell connecting to the waiting listener, before continuing the normal execution of the image. This would give the attack access to the container files-system as the default user, often root. In the case of the Nginx image this allowed the attacker to dynamically alter the default index page whilst the container was running (see Figure 14).

This helps highlight how the gh0stEdit vulnerability can be exploited as part of a automated attack chain, for example in a CI/CD pipeline, which is discussed and expanded on further in 7.

7. Discussion

The results from our attack examples and subsequent analysis demonstrate that this vulnerability can be exploited to compromise Docker images in ways that bypass

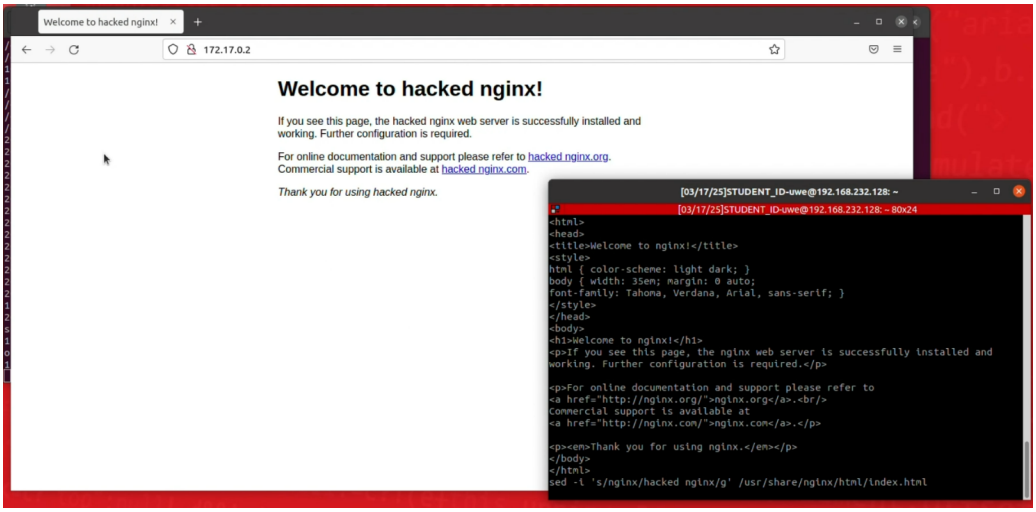


Figure 14: Malicious edit of the running Nginx container, via a gh0stEdit deployed RCE

both static and dynamic analysis. The only detectable changes in the edited images are the altered layer, image digest, and the increased image size. The image creation time, commands, and history remain unchanged, and newly added binaries or CVEs are not detected during scans. We have also demonstrated that by exploiting the gh0stEdit vulnerability, it is possible to create an image capable of sending out network traffic without raising any security alerts. The full scanner results can be found on our GitHub page[11].

The detectable indications of change rely on having a known 'original' comparison image for reference. In a real-world attack scenario, such as a poisoned image in a Docker repository or an image maliciously altered within a development environment, detecting these changes would be extremely difficult. Image update are common, as seen in the official Python repository, which updated multiple images between 4th-8th and 13th-15th August 2024. These updates result in different image digests and layer

[11] `https://github.com/amills157/gh0stEdit`

IDs, particularly for images which are not tagged with a specific patch version, such as 3.12-slim or latest. In this scenario, once an image compromised using the gh0stEdit vulnerability is present in a repository, direct comparison becomes infeasible. The most reliable method of detecting such an attack would involve manual, in-depth layer-based analysis combined with reverse engineering and version checking of all installed binaries against the known SBOM.

Within a production environment the exploitation of this vulnerability could occur at multiple stages. Manual exploitation could be carried out by a malicious insider with access to a repository at almost any stage. If a legitimate change is required, such as updating the code base for a container image, then gh0stEdit could be deployed simultaneously, to affect a malicious change that is masked by the legitimate update, providing "cover" for pushing a new version of an image to a repository.

The use of tags and labels [12] within production environments [13] also provides another perfect "cover" for the use of gh0stEdit. If image (re)tagging is automated within a CI/CD pipeline then a malicious edit to this pipeline will impact all images, potentially lead a supply chain attack on a massive scale.

External attackers may also make use of gh0stEdit in combination with typo squatting or repository hijacking attacks. Making malicious edits to a legitimate image and then uploading it under similar but different names, confident in the knowledge that (currently) it will pass all industry recommended scanning practices, elevating this attack path to dangerous new heights compared to previous attempts

[12] https://www.docker.com/blog/docker-best-practices-using-tags-and-labels-to-manage-docker-image-sprawl/

[13] https://blog.nashtechglobal.com/docker-tagging-strategies-for-deploying-to-production

which have been identified through the use of static scanning tools or image layer inspection [35].

This type of attack evasion, combined with increased popularity of tools such as Kubernetes and Docker in DevOps [36], provides a wide attack surface that requires a low level of technical complexity to achieve. The main mitigating factor for the successful use of gh0stEdit is access to either a trusted repository or deployment pipeline. However any developer operating on a process that open source images which pass static or dynamic scans, such as those covered in this paper, are safe to use, is at risk of a significant supply chain attack through the exploitation of gh0stEdit.

As gh0stEdit is itself a way to exploit an existing vulnerability within Docker images, the attacks that it can be utilised for are varied. As demonstrated as part of our case study it is possible to include malicious binaries which can connect out from the container image itself. This can take the form of CryptoMiners, a commonly deployed malware within container images, or binaries designed to exfiltrate logging or confidential data as part of a targeted attack (such as an insider threat). It could also be used to deploy backdoors or downloaders as part of a staged attack, however attackers would still be confined to the container environment, at least initially.Attacker motivations and targets may also vary, for example gh0stEdit could be used in combination with Kubernetes specific ransomware attacks [37] or other destructive attacks. The main mitigation to such attacks is the potential for network traffic to be identified stopping a long running attack in its tracks. Something which can be overcome, but would require either insider knowledge of the targets defensive posture or a wide spread attack aiming to infect and capitalise on poor cyber security practice and defences.

The attack examples and analysis presented in this paper highlight significant

vulnerabilities within the current container ecosystem and approaches to Docker-based security. Fundamentally, the recommended safeguards for container-based security rely on an assumed transparency and inherent trust in the container image. End users and scanning tools trust that the image hierarchy and self-reported SBOMs are accurate, a trust that gh0stEdit exploits.

We have also demonstrated that the gh0stEdit exploit is effective on images signed as part of the Docker Content Trust (DCT) process, underscoring the severity of this attack and revealing issues within the DCT image signing process. To mitigate this risk, image signatures should be verified and invalidated locally to ensure signed images are not subsequently tampered with. Currently a signed image could be maliciously altered without any obvious indications of modification. In a CI/CD environment, this could result in the poisoned image being deployed as part of a production-ready asset, having passed all recommended safeguards.

7.1. Cross Compatibility

The primary environment used during the testing and investigation of gh0stEdit was an Ubuntu 20.04 LTS, which uses (by default) the EXT4 filesystem. Testing was however carried out within a CentOS 9 stream Virtual Machine, using the XFS filesystem. Within the XFS filesystem the structure of the docker images when saved and extracted differed, with additional nesting layers and JSON files. Whilst this meant our automated script did not work, having been designed for the EXT4 environment, manual testing of the gh0stEdit vulnerability was successful. Figure 15 shows the comparison between the Redis image which has been altered, using gh0stEdit, on an XFS system and the official image. The edited image had the reverse shell added, as described in 6.2 and we can see that there are no obvious changes to either the layers or the image `Command`.

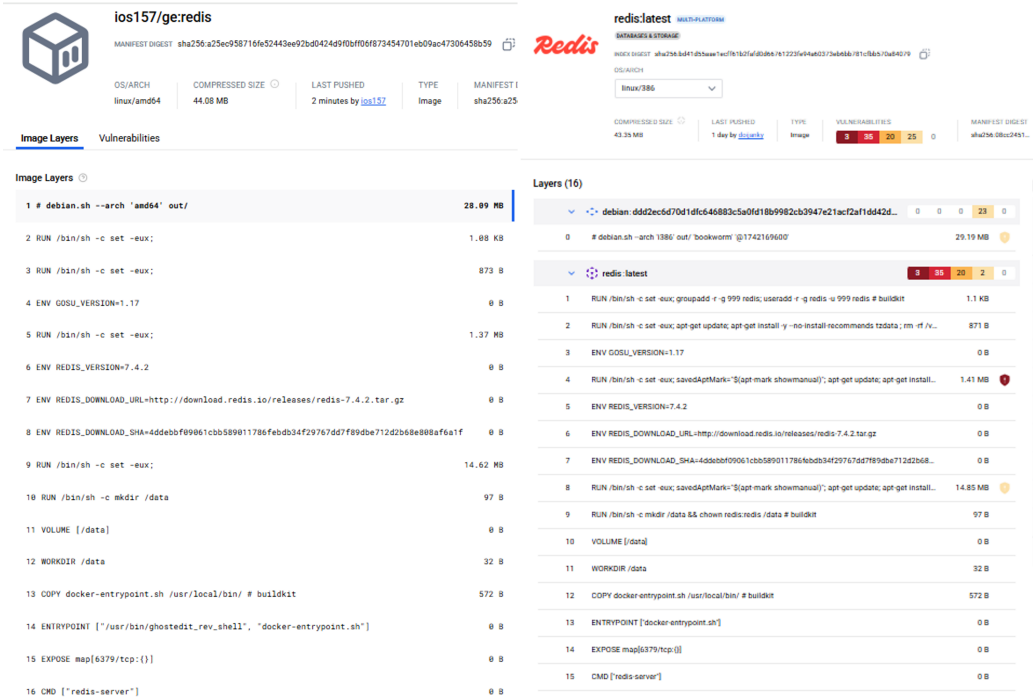


Figure 15: Comparison of edited and official Redis images - XFS gh0stEdit

8. Responsible Disclosure and Ethical Considerations

The gh0stEdit exploit was responsibly disclosed to Docker on the 14th of August 2024. Our disclosure included a Proof-of-Concept (PoC) bash script for editing the final layer within a Docker image, a video demonstration of the exploit, and an edited Docker image for analysis. All artefacts were shared privately with Docker to avoid third-party exposure or reverse engineering. Subsequent updates and further attack examples, such as the attack on signed images, were also communicated privately with Docker under their vulnerability disclosure policy [14].

We acknowledge that the attack methods discussed in this paper could be mis-

[14] `https://www.docker.com/trust/vulnerability-disclosure-policy/`

used, but we believe that publicly disclosure is necessary to raise awareness of these vulnerabilities. At present, no comprehensive measures are in place to prevent such attacks occurring. Without open discussion and demonstration, developing a solution will be unlikely. We encourage the academic and cyber security communities to act on these findings and work towards enhancing the security of the container ecosystem.

9. Conclusions and Further Work

In this paper we present gh0stEdit, the exploitation of a vulnerability in Docker images that enables attackers to compromise the integrity of an image. By saving an image as an archive file, an attacker can directly edit the layers, overwriting existing layers with malicious edits. Our case studies demonstrate that these edits are extremely difficult to detect and can bypass recommended industry-standard safeguards, including static and dynamic scanning, as well as image signing via Docker Content Trust (DCT). Using this exploit we successfully introduced a poison binary and intentionally added CVEs into images, which went undetected by multiple vulnerability scanners that are commonly used. We also showed that an image signed through DCT can be maliciously altered without invalidating its signature. To the best of our knowledge we are the first to publicly detail and discuss the exploitation of this vulnerability.

This exploit presents a serious threat, particularly in the context of software supply chain attacks and invalidates existing trust in image transparency. These malicious edits do not appear in an image's history, hierarchy, commands, or SBOM. It would therefore be challenging to determine if this vulnerability has been exploited in practice without conducting an in-depth manual inspection of the image.

To mitigate against gh0stEdit, container image scanners and safeguards need to be restructured. Security checks that rely on self-reported data should be replaced with a "zero trust" approach. Content must be directly examined to ensure installed packages are free of known vulnerabilities, rather than relying on self-reported SBOMs.

One possible solution would be the introduction of an "image or layer edit time" field, which would be reported alongside existing image metadata, such as the image creation date. This would ensure that any post-creation changes to the container are evident, and could mitigate against gh0stEdit.

Additionally, the logic behind image signing and Docker Content Trust needs to include a verification check between the signed *RepoDigest* and the digest of the inspected image. This would ensure that any changes made to an image would invalidate its signed status, providing an additional layer of security.

Acknowledgements

This work was supported by the College of Arts, Technology and Environment at the University of the West of England.

References

[1] Dive, `https://github.com/wagoodman/dive` (Last accessed: 28 August 2024).

[2] T. Bui, Analysis of docker security, arXiv preprint arXiv:1501.02967 (2015).

[3] R. Yasrab, Mitigating docker security issues, arXiv preprint arXiv:1804.05039 (2018).

[4] S. Sultan, I. Ahmad, T. Dimitriou, Container security: Issues, challenges, and the road ahead, IEEE Access 7 (2019) 52976–52996.

[5] P. Liu, S. Ji, L. Fu, K. Lu, X. Zhang, W.-H. Lee, T. Lu, W. Chen, R. Beyah, Understanding the security risks of docker hub, in: European Symposium on Research in Computer Security, 2020, pp. 257–276.

[6] A. Y. Wong, E. G. Chekole, M. Ochoa, J. Zhou, Threat modeling and security analysis of containers: A survey, arXiv preprint arXiv:2111.11475 (2021).

[7] S. Ghavamnia, T. Palit, A. Benameur, M. Polychronakis, Confine: Automated system call policy generation for container attack surface reduction, in: 23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020), 2020, pp. 443–458.

[8] P. Ladisa, H. Plate, M. Martinez, O. Barais, Sok: Taxonomy of attacks on open-source software supply chains, in: 2023 IEEE Symposium on Security and Privacy (SP), IEEE, 2023, pp. 1509–1526.

[9] M. Mounesan, H. Siadati, S. Jafarikhah, Exploring the threat of software supply chain attacks on containerized applications, in: 2023 16th International Conference on Security of Information and Networks (SIN), IEEE, 2023, pp. 1–8.

[10] A. Tomar, D. Jeena, P. Mishra, R. Bisht, Docker security: A threat model, attack taxonomy and real-time attack scenario of dos, in: 2020 10th International Conference on Cloud Computing, Data Science & Engineering (Confluence), IEEE, 2020, pp. 150–155.

[11] Docker image insecurity, `https://titanous.com/posts/docker-insecurity` (Last accessed: 17 March 2025).

[12] Docker hub, https://www.docker.com/products/docker-hub/ (Last accessed: 28 August 2024).

[13] Docker build — docker docs, https://docs.docker.com/reference/cli/docker/build-legacy/#squash (Last accessed: 28 August 2024).

[14] O. Javed, S. Toor, Understanding the quality of container security vulnerability detection tools, arXiv preprint arXiv:2101.03844 (2021).

[15] R. Shu, X. Gu, W. Enck, A study of security vulnerabilities on docker hub, in: ACM Conference on Data and Application Security and Privacy, 2017, pp. 269–280.

[16] K. Wist, M. Helsem, D. Gligoroski, Vulnerability analysis of 2500 docker hub images, in: In: Advances in Security, Networks, and Internet of Things, Springer, 2021, p. 307–327.

[17] L. Chen, Y. Xia, Z. Ma, R. Zhao, Y. Wang, Y. Liu, W. Sun, Z. Xue, Seaf: A scalable, efficient, and application-independent framework for container security detection, Journal of Information Security and Applications 71 (2022) 103351.

[18] A. Mills, J. White, P. Legg, Ogma: visualisation for software container security analysis and automated remediation, in: 2022 IEEE International Conference on Cyber Security and Resilience (CSR), IEEE, 2022, p. 76–81.

[19] Introduction — canarytokens, https://docs.canarytokens.org/guide/ (Last accessed: 28 August 2024).

[20] Pyinstaller manual, https://pyinstaller.org/en/stable/ (Last accessed: 28 August 2024).

[21] Clamavnet, https://www.clamav.net/ (Last accessed: 28 August 2024).

[22] Dagda, https://github.com/eliasgranderubio/dagda (Last accessed: 28 August 2024).

[23] Docker scout, https://github.com/docker/scout-cli (Last accessed: 28 August 2024).

[24] Docker scan, https://github.com/docker/scan-cli-plugin (Last accessed: 15 May 2023).

[25] Grype, https://github.com/anchore/grype (Last accessed: 28 August 2024).

[26] Skopeo, https://github.com/containers/skopeo (Last accessed: 28 August 2024).

[27] Trivy, https://github.com/aquasecurity/trivy (Last accessed: 28 August 2024).

[28] Yara hunter, https://github.com/deepfence/YaraHunter (Last accessed: 28 August 2024).

[29] Neuvector, https://github.com/neuvector/neuvector (Last accessed: 28 August 2024).

[30] Helm! — charts, https://helm.sh/docs/topics/charts/ (Last accessed: 28 August 2024).

[31] Welcome! — minikube, https://minikube.sigs.k8s.io/docs/ (Last accessed: 28 August 2024).

[32] Content trust in docker, https://docs.docker.com/engine/security/trust/ (Last accessed: 28 August 2024).

[33] Top ten most popular docker images each contain at least 30 vulnerabilities, https://snyk.io/blog/top-ten-most-popular-docker-images-each-contain-at-least-30-vulnerabilities/ (Last accessed: 17 March 2025).

[34] Top 15 docker containers in 2024, https://analyticsindiamag.com/ai-trends/top-docker-containers/ (Last accessed: 17 March 2025).

[35] Analysis on docker hub malicious images: Attacks through public container images, https://sysdig.com/blog/analysis-of-supply-chain-attacks-through-public-docker-images/ (Last accessed: 17 March 2025).

[36] Popular technologies in the devops stack 2024, https://www.statista.com/statistics/1292382/popular-technologies-in-the-devops-tech-stack/ (Last accessed: 17 March 2025).

[37] Siloscape, software s0623, https://attack.mitre.org/software/S0623/ (Last accessed: 17 March 2025).