Fine penetration tests for fine websites

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Pentest-Report *containerd* 11.2018

Cure53, Dr.-Ing. M. Heiderich, M. Wege, BSc. J. Hector, J. Larsson, MSc. D. Weißer, MSc. N. Krein

Index

Introduction

> *“containerd is available as a daemon for Linux and Windows. It manages the complete container lifecycle of its host system, from image transfer and storage to container execution and supervision to low-level storage to network attachments and beyond.”*
>
> From https://containerd.io

This report documents the findings of a security assessment targeting the *containerd* software. Carried out in November 2018, this project was commissioned to Cure53 by CNCF and entailed both a penetration test and a source code audit of the open-source *containerd* compound. Despite thorough investigations, the assessment only revealed one minor flaw on the tested scope.

As for the resources, the project was completed by six members of the Cure53 team who worked with a goals-fitting time budget of eighteen days and executed the assessment in late November 2018. It is important to emphasize that all involved testers boast considerable experience in having previously audited GoLang and being part of earlier, similar CNCF projects.

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Adopting a typical setup requested by the CNCF commissioning entity, this assessment relied on a white-box approach. This methodology means that Cure53 had full access to all relevant code and, in addition, could take advantage of a testing infrastructure, as well as detailed briefings with the *containerd* team.

Cure53 initiated the assessment with a kick-off meeting to learn more about the expected coverage and the possible threat models applicable to this rather specific piece of software. A Slack channel was made available for both teams to communicate during the test. This helped to make sure that no emerging roadblocks or technical issues could persist and negatively impact coverage.

It was quickly determined and confirmed that the actual attack surface is very small and a large output of issues is not to be expected from this project. This is directly reflected by the number and severity of the sole discovery, as only one issue could be spotted on the scope. It should be underlined that this issue clearly requires a rather strong and sophisticated attacker.

While it might seem like the audit and tests were not fruitful, they indeed managed to comprehensively verify that the *containerd* team has been following the right path in terms of keeping security promises. Cure53 ascertained robustness of implementation and successes as regards keeping the attack surface small.

In the following sections, the report first focuses on the scope and lists all relevant code repositories and similar items vital for the investigations. Next, the test methodology is described in more detail to shed light on the coverage that the Cure53 team has reached during the assessment. This also serves as means to provide transparency on how the allocated resources were spent. From there, the report proceeds to discussing the single finding before moving on to a concluding section, wherein Cure53 elaborates on the general security and privacy impressions gained from assessing the *containerd* software during this 2018 project

Scope

- ***containerd* Runtime**
  - https://github.com/containerd/containerd `branch release/1.2 commit 040e73f...`
- ***containerd* Kubernetes Plugin**
  - https://github.com/containerd/cri `branch release/1.2 commit 8671a27...`
- ***containerd* Interaction Components**
  - https://github.com/containerd/ttrpc `branch master commit f51df44...`
  - https://github.com/containerd/typeurl `branch master commit 461401d...`
  - https://github.com/containerd/go-runc `branch master commit 5a6d9f3...`

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

  - https://github.com/containerd/go-cni `branch master commit 40bcf8e...`
- ***containerd* System Abstraction**
  - https://github.com/containerd/fifo `branch master commit 3d5202a...`
  - https://github.com/containerd/console `branch master commit 0650fd9...`
  - https://github.com/containerd/cgroups `branch master commit 82cb49f...`
- ***containerd* Filesystem Interface**
  - https://github.com/containerd/continuity `branch master commit bea7585...`
  - https://github.com/containerd/aufs `branch master commit 1d75a7b...`
  - https://github.com/containerd/btrfs `branch master commit af50828...`
  - https://github.com/containerd/zfs `branch master commit 31af176...`
- **Supplied Test Targets**
  - Cure53 got access to two test systems on which the penetrations tests could be conducted in a close to real-life scenario.

Test Methodology

This section describes the methodology that was used during this source code audit and penetration tests. The project was divided into two phases with corresponding two-fold goals and focal points that had clear reference and relevance for the scope. The first phase concentrated mostly on manual source code reviews. These reviews aimed at spotting insecure code constructs marked by the potential a capacity of leading to memory corruption, information leakages and other similar flaws. The second phase of the assessment was dedicated to classic penetration tests. During this phase, it was examined whether the security promises made by *containerd* in fact hold against real-life attack situations.

Part 1 (Manual Code Auditing)

A list of items below seeks to detail some of the noteworthy steps undertaken during the first part of the test, which entailed the manual code audit against the sources of the *containerd* software in scope. This is to underline that, in spite of the almost nonexistent number of findings, substantial thoroughness was achieved and considerable efforts have gone into this test. The completed steps are listed next. Note that a given realm yielded no results unless otherwise indicated with a specific link to a finding.

- The source code and documentation of the *containerd* runtime implementation was checked for attack surface. This was necessary as a defined threat model was absent.
- The usual *os.** and *exec.** calls were checked bottom-up for leveragability (i.e. unverified execution paths, etc.), but eventually abandoned for a more top-down source-sink auditing approach.

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

- The existing *gRPC* handlers were audited to gain an understanding as to how the provided compound works together as a whole.
- The *task* and *image* endpoints were audited in particular detail to find potential command line issues and similar problems.
- The file and path handling implementation was audited for path traversal, file likes and unchecked overwrites.
- The debug and metrics for web backends were investigated for typical web problems.
- The configuration file parsing was checked for problems. While it is very simplistic and based on a TOML-parsing library, a minor flaw found in this realm resulted in the filing of CTD-01-001.
- All execution paths that resolve to *go-runc* and eventually terminate in *exec* calls were analyzed. The code flow was traced to check if injections were at all possible. Since all user-input is encapsulated in structures and arrays, no breakout points could be identified.
- Cure53 audited the implementation of *import* and *export* features to discern problems typical for such areas but no edge cases have been found.
- The *image verification* implementation was analyzed for potential verification bypasses. Again, no circumvention methods were discovered.
- The file system handlers were briefly audited and the system abstraction components were additionally studied.
- The integration glue between *cri* and *kubectl* was audited for potential command stacking. In addition, Cure53 looked for flaws in the environment and parser implementation.
- The Kubelet API within the plugin was analyzed, with the team focusing on the aspects which enumerate services (i.e. endpoints, functions, messages and remote procedure calls); the testers were looking for malleable functionality but were unable to locate any.
- The source code of the *cri* command line interface was audited to obtain an overview of its integration. The functions were mapped to potential attack vectors to no avail.

Part 2 (Code-Assisted Penetration Testing)

A list of items below seeks to detail some of the noteworthy steps undertaken during the second part of the test, which encompassed code-assisted penetration testing against the *containerd* system in scope. Given that the manual source code audit did not yield significant findings, the second approach was chosen as an additional measure for maximizing the test coverage. As for specific steps executed to enrich this phase, these can be found listed and discussed in the following bullet points.

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

- After pondering the threat model with the need to delineate an attack surface, it was decided to change the *containerd's gRPC* socket owner to a non-privileged user.
- Over the course of looking at the *containerd* file handlers, it was investigated if abusing a given functionality to modify the owner of existing files was a possibility. While theoretically feasible, the potential weakness was not leverageable.
- Endpoints interesting for attackers were enumerated (i.e. the debug interface, the metrics interface and the *gRPC* API) and tackled as regards being remotely accessible. No vulnerabilities could be identified.
- All *gRPC* calls were enumerated and their execution traced to figure out how they interacted with the file system and operating systems calls in general. A particular focus was placed on finding logical flaws in the parsing, especially looking for possible command injections.
- *Tar* archives were dissected in an attempt to create malicious archives and leak files or follow symlinks with the potential to overwrite sensitive files, particularly with respect to path traversal. Several selected archives were tested on their matching importer but did not show any unwanted behaviors.
- Further investigations of the *import* functionality were hindered by not being able to export properly working archives. Because of time constraints, the investigation of the matter was ceased.
- Exposed socket and interface creation were explored for potential information disclosure. The lack of transport security was considered insignificant since it is all locally bound.
- The Kubernetes-plugin called *cri* was locally configured on the provided test cluster. This was meant to help understand its integration and facilitate additional testing.
- It was attempted to break the container creation by extending and manipulating remote procedure calls. The approaches entailed stacking additional illicit parameters but did not lead to noticeable misbehaviors.

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Miscellaneous Issues

This section covers those noteworthy findings that did not lead to an exploit but might aid an attacker in achieving their malicious goals in the future. Most of these results are vulnerable code snippets that did not provide an easy way to be called. Conclusively, while a vulnerability is present, an exploit might not always be possible.

CTD-01-001 Config: Malicious *config* allows arbitrary directory creation *(Low)*

It was discovered that a malicious *configuration* file makes it possible to create a persistent arbitrary directory on the filesystem. What is more, the owner and group ID can also be set arbitrarily in this context.

This could potentially allow a low-privileged user to write *configuration* files. For instance, a new directory in */etc/cron.d/* can be created and set to be owned by a non-*root* user, effectively allowing said user to write *cron* files.

On the test system provided, *cron* refused to execute such a *cron* job due to ownership of the directory (i.e. non-*root* owner). However, other applications may implicitly trust the content of */etc/* since it is usually owned by *root*.

The code excerpt below - with the relevant parts highlighted - demonstrates the issue.

Affected File:
containerd/sys/socket_unix.go

Affected Code:

```
func GetLocalListener(path string, uid, gid int) (net.Listener, error) {
	// Ensure parent directory is created
	if err := mkdirAs(filepath.Dir(path), uid, gid); err != nil {
		return nil, err
	}

	l, err := CreateUnixSocket(path)
[...]
func mkdirAs(path string, uid, gid int) error {
	if _, err := os.Stat(path); err == nil || !os.IsNotExist(err) {
		return err
	}

	if err := os.Mkdir(path, 0770); err != nil {
		return err
	}

	return os.Chown(path, uid, gid)
```

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

```
}
```

As can be seen above, during the creation of the *parent* directory, the owner of the directory is simply changed to the provided user and group ID. A Proof-of-Concept (PoC) *configuration* file furnished below results in a directory created in */etc/cron.d/.* The user and group ID of that directory are set to 1000, meaning the Cure53 user on the test system.

PoC Config:

```
[grpc]
        address = "/etc/cron.d/new_dir/containerd.sock"
        uid = 1000
        gid = 1000
```

It is recommended for the *parent* directory to always inherit the owner of the directory used for the creation of the new directory.

Conclusions

The results of this CNCF-commissioned assessment testify to security being a well-handled priority in the development and deployment of the *containerd* software. After investigating the scope for eighteen days in November 2018, six members of the Cure53 team could only identify one*, Miscellaneous, Low*-ranking finding.

While the scope of this test and the architecture of the system were rather well-defined, a threat model was not communicated to Cure53 by the development team. Therefore, it took more effort than originally expected to delineate an adequate attack surface. To enable fulfillment of the project goals, a completely unhindered access to two test setups was given to Cure53. One of them was a more traditional container-based test system and one entailed a more advanced, Kubernetes-based test cluster.

The assessment status and progress were reported on a dedicated messaging channel, furnished and managed by the Docker community. All communications were fluent, highly productive and kept to a required minimum. The headlines of the discovered issues were posted in the messaging channel before the end of the audit, well in advance of the delivery of this final report.

Both the *containerd* container runtime and the Kubernetes *cri* plugin are very well written from a security standpoint. The choice of the Go language made it difficult to find any sort of memory corruptions or similar bugs during this assignment, ultimately leading to no such problems being uncovered. The majority of the code was written in a clean manner, thus easing the process of the code audit.

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

To give some details on the proceedings, the *containerd* container runtime*,* along with the Kubernetes *cri* plugin were audited and penetration tested after an initial examination of the documentation and the two provided test setups. As side items, Cure53 looked at the interaction components, namely *ttrpc*, *typeurl*, *go-runc* and *go-cni*, as well as further skimming the system *fifo*, *console* and *cgroups* abstractions. Further, the filesystem interfaces, notably *continuity*, *aufs*, *btrfs* and *zfs*, were also briefly analyzed.

In the core assessment, the maintainer-provided test setups were modified to enable unprivileged interface access with a potential for privilege escalation. The included testing code was used as a basis to explore different abstractions and interfaces offered by the system. The only discovery was of a *miscellaneous* nature and has "*Low".* As such, it does not raise any concerns about the general shape of the application.

As security issues in Go implementations are mostly logic flaws and occasionally race conditions, not much could be discovered within *containerd* and *cri*. This is because neither of them is marked by complicated or complex logic. The overall amount of code interfacing with the container environment and the orchestration cluster is significant and of most unusual quality for a project of these dimensions.

Overall, a nearly complete absence of findings proves that the entire *containerd* runtime, along with the Kubernetes-plugin, provide a robust and mature platform that can be recommended for the wide, general deployment. While the assessment certified to good security premise, it is recommended to finalize and extend the existing documentation within the source code of the application and the markdowns in the repositories, as this approach would facilitate practical maintenance in the future. To conclude, from a security perspective the project is fit for purpose and can be used safely.

Cure53 would like to thank Philip Estes and Michael Crosby from the *containerd* team as well as Chris Aniszczyk of The Linux Foundation, for their excellent project coordination, support and assistance, both before and during this assignment. Special gratitude also need to be extended to The Linux Foundation for sponsoring this project.