Security-Review Report runc 11.-12.2019

Cure53, Dr.-Ing. M. Heiderich, M. Wege, N. Hippert, J. Larsson,
MSc. D. Weißer, MSc. N. Krein, MSc. F. Fäßler

Index

Introduction

> *"runc is a CLI tool for spawning and running containers according to the OCI specification."*
>
> From https://github.com/opencontainers/runc

This report describes the results of a security assessment and a review of the general security posture found on the *runc* software complex and its surroundings. The project was requested and sponsored by CNCF as a common part of the CNCF project graduation process. Within the frames of well-established cooperation, the project was awarded to Cure53, which investigated the *runc* scope in terms of security processes, response and infrastructure.

The work was carried out by seven members of the Cure53 team in November and December 2019, with a total budget standing at eighteen person-days. From the star, the work was split into two different phases, with *Phase 1* focused on *General security posture checks* and *Phase 2* dedicated to *Manual code auditing* aimed at finding implementation-related issues that can lead to security bugs. Cure53 worked in close collaboration with the *runc* team and the communications during the engagement took place in a dedicated channel of the Docker Slack workspace. The Cure53 team was invited to join the exchanges in that channel by the maintainers. In general, communications were productive, yet the scope was very clear and not many questions had to be asked.

By coincidence, Cure53 received information about a possible race condition vulnerability present in the *runc* codebase at the time when this assessment was in progress. This unusual opportunity was used to first analyze the alleged issue and create a working PoC. Secondly, it served as a specific case of a bug initially found by a third-party, giving Cure53 a front-row seat to observing and evaluating the disclosure process. Perspectives of the original finder and the reaction of the *runc* team upon getting access to the bug report could be investigated and, thanks to this real-life example of an actual vulnerability spotted by a third party, Cure53 gathered empirical evidence on optimizing the process at the *runc* entities in the future.

In the following sections, the report will first present the areas featured in the test's scope in more detail, zooming in on the proposed structure of the two phases delineated above. The report is enriched by Cure53 describing the evaluated areas and explaining the methodology of the executed tests in more detail. The aforementioned accidentally covered real-life issue, together with the relevant PoC and credit for the original finder, is then documented. Cure53 additionally furnishes mitigation advice, so to ascertain that the *runc* team can address this hard-to-find and tricky problem. The report closes with a conclusion in which Cure53 summarizes this 2019 project and issues a verdict about the security premise of the investigated *runc* scope.

Scope

- ***runc v1.0.0-rc9***
  - *runc* codebase
    - https://github.com/opencontainers/runc/tree/master
    - Commit: *46def4cc4cb7bae86d8c80cedd43e96708218f0a*
  - *runc* project's security posture and maturity levels

Test Methodology

The following paragraphs describe the metrics and methodologies used to evaluate the security posture of the *runc* project and codebase. In addition, it includes results for individual areas of the project's security properties that were either selected by Cure53 or singled out by other involved parties as needing a closer inspection.

As noted in the Introduction*,* the test was divided into two phases, each fulfilling different goals. In the first phase, the focus was on the general security posture of the code and the project. Furthermore, Cure53 examined the processes that the *runc* development team has made available for security reports, also as relates disclosure and general hardening approaches. In the second phase, the work has shifted to the manual source code review of specific code areas.

Phase 1: General security posture checks

In this component of the assessment, Cure53 looked at the General security posture of the *runc* project and inspected the overall code quality from a meta-level perspective. Some of the indicators taken into account encompassed test coverage, security vulnerability disclosure process, approaches to threat modeling and general code hardening measures. The sum of observations from across these areas have been used to describe the maturity levels of this project at a meta-level, independently of the security qualities of the provided code and created binaries.

Later chapters in this report will dive into the details of the inspected items, justifying these choices and presenting the results in the specific case of the *runc* software project.

Phase 2: Manual code auditing

For this component, Cure53 performed a Small-scale code review and attempted to identify security-relevant areas of the project's codebase and inspect them for common flaws.

Unlike standard processes in a usual penetration test and code audit, this phase only took a few days. As such, it was a brief rather than an in-depth inspection. It should be seen as an initial probing aimed at evaluating whether more thorough code audits should be recommended. The goal was not to reach an extensive coverage but to gain an impression about the overall quality. The completed tasks assist Cure53 in making a judgment call as to whether *runc* needs additional tests and what kinds of tests these could be.

Later chapters in this report will shed more light on what was being inspected, why and with what implications for the *runc* software complex.

Phase 1: General security posture checks

This phase is meant to provide a more detailed overview of the *runc* project's security properties that are seen as somewhat separate from both the code and the *runc* software itself. To facilitate clear flow and understanding, this section is divided into two subsections, where the first part consists of elements specific to the application and the project. The second part looks at the elements linked more strongly to the organizational/team aspect. Lastly, each aspect below is taken into account and an evaluation of the overall security posture is based on cross-comparative analysis of all observations and findings.

- A general high-level code audit was undertaken to arrive at a solid judgment of the entire *runc* project, in particular with the task of checking for unsafe patterns and coding styles.
- The complete project structure was analyzed; the main call flow was mapped; the individual sub-components were enumerated and the supported platforms were checked.
- The project's external and third-party dependencies were cross-checked for problematic components.
- The provided documentation was examined in order to learn about the provided functionality and the depth of instructions available to the developer.
- Relevant runtime- and environment-specifications were examined in connection with the general project solution domain.

- Past vulnerability reports and postings were checked to see in which areas certain errors had previously emerged, also in assessing the likelihood of their reappearance.
- An in-depth static code analysis was carried out to check for applicability of automated measures. The scan results were verified for usability.
- The project’s maturity was evaluated; specific questions about the software were compiled from a general catalog according to individual applicability.

Application/Service/Project Specifics

In this section, Cure53 will describe the areas that were inspected to get an impression on the application-specific aspects that lead to a good security posture, such as choice of programming language, selection and oversight of external third-party libraries, as well as other technical aspects like logging, monitoring, test coverage and access control.

Language Specifics

Programming languages can provide functions that pose an inherent security risk and their use is either deprecated or discouraged. For example, *strcpy()* in C has led to many security issues in the past and should be avoided altogether. Another example would be the manual construction of SQL queries versus the usage of prepared statements. The choice of language and enforcing the usage of proper API functions are therefore crucial for the overall security of the project.

runc is written in Go, which inherently provides memory safety and broadly offers a higher level of security in comparison to e.g. C/C++. This is further underlined by only making use of the Go’s *unsafe* package if absolutely necessary, in particular when interfacing with the operating system. The code is written with best practices in mind, which helps not only with auditing, but also with maintenance. The above indicators contribute to a healthy security posture and seem well-understood and properly spread throughout the *runc* codebase. Specific examples include:

- Nesting being avoided by handling errors first;
- Separating test-cases from code;
- Documenting all relevant code;
- Keeping documentation/items concise;
- Separating independent packages;
- Avoiding unnecessary repetitions.

The usage of *unsafe* is limited on *runc* to *syscall* functionality where unsafe pointers are absolutely required and implementation cannot be achieved otherwise. The *unsafe* constructs are solely used as pointers for return values from *prctl()* and similar low-level system calls.

Though some vendor code containing dangerous-looking marshalling code (see *cilium/ebpf/marshalers.go*) was identified, those fragments are apparently not actively used anywhere in the production codebase.

External Libraries & Frameworks

While external libraries and frameworks can also contain vulnerabilities, it is nonetheless beneficial to rely on sophisticated libraries instead of reinventing the wheel with every project. This is especially true for cryptographic implementations, since those are known to be prone to errors.

runc makes use of external libraries, therefore avoiding reimplementation of already existing solutions. Since *runc* is heavily dependent on functionalities that are exposed by the Linux kernel, it makes extensive use of third-party packages like *golang.org/x/sys/unix* to provide a more portable interface to the underlying operating system. To safeguard a good alternative for file system interactions, packages like *filepath-securejoin* are used as well. Generally no concerns were found to be present in the used third-party packages. All appear to be widely recognized by the community and appear to be under active development.

Configuration Concerns

Complex and adaptable software systems usually have many variable options which can be configured according to the actually deployed application necessities. While this is a very flexible approach, it also leaves immense room for mistakes. As such, it often creates the need for additional and detailed documentation, in particular when it comes to security.

Containers created with *runc* can have security pitfalls due to the flexibility of the configurations. Those pitfalls are not explicitly addressed by the documentation and require deep knowledge about Linux to even have a general awareness about them. Examples for such pitfalls are described in the following paragraphs.

In a default setup, *runc* makes the hosts *dmesg* output available inside the containers unless *kernel.dmesg_restrict=1* is set on the host system. It includes information about the kernel, which means that in some cases it might disclose certain details to an attacker. This especially holds for an adversary who already has access to a *runc* container and needs more information in order to escape the environment. There is no reason why this information should be retrievable from a sealed container. Such information leaks can easily be prevented by blocking the respective *syslog* system call via *seccomp* and this is highly advised.

The */proc* filesystem is a special virtual filesystem included in Linux. It generally requires being mounted inside a container because a lot of software relies on it. It is also the place where security settings like *AppArmor* are applied and files that have direct effect on the underlying system are exposed. In essence, the */proc/sys/kernel/core_pattern* file controls how core files are handled when a process crashes. It also allows to specify a command that is executed, thus it could potentially be used to break out of the container. That is why the default *config* generated by *runc spec*, as well as *configs* from Docker and *podman*, specify that paths like */proc/sys* need to be remounted as read-only. Because this is not documented, new projects that build upon *runc* might not be aware of this danger and may forget to include the read-only settings. It is recommended to follow up on this issue.

Another nuance is shown in RUN-01-001, where the order of mounts can have a large security impact. *podman* mounts shared volumes before the */proc* filesystem, enabling the race condition in the first place, even though the Docker *configs* mount the */proc* filesystem first. The proposed revision of the sequence is not a guaranteed fix to the underlying issue, but it does mitigate the specific PoC.

It should also be noted that the race condition uses a shared volume, though *runc* configs can also define the same *rootfs* for multiple containers. Using the same *rootfs* allows a race condition independent from the mount order. Implementations such as Docker and *podman* define unique *rootfs* for each container, and thus do not suffer from this issue. However, a new project using *runc* might not be aware of the risk in a shared *rootfs.*

Because of the described security-relevant issues, It is recommended to provide better default configuration files and add exhaustive explanations of security considerations to the shipped documentation.

Access Control

Whenever an application needs to perform a privileged action, it is crucial that an access control model is in place to ensure that appropriate permissions are present. Further, if the application provides an external interface for interaction purposes, some form of separation and access control may be required.

runc has a divided access control model in place, which makes the topic of access control rather complex. The framework itself offers a subset of functionality that can be configured in order to limit access for running containers. The general access control model is shared between the hosts responsible for implementation and *runc*. By default, *runc* offers the possibility to run *rootless* containers as well as *build* tags, which control individual access for a given container.

Logging/Monitoring

Having a good logging/monitoring system in place allows developers and users to identify potential issues more easily or get an idea of what is going wrong. It can also provide security-relevant information, for example when a verification of a signature fails. Consequently, having such a system in place has a positive influence on the project.

runc makes use of *logrus*[1], a structured logging system that is compatible with Golang's standard library logger. Although *logrus* offers a transparent API that replaces the standard logging functionality, *runc* explicitly invokes *logrus* when needed. It would make sense to centralize its usage by globally overriding the intrinsic *log* feature. This is because then uniform logging could be consistently applied throughout the entire codebase. Despite this rather small point of critique, *runc* tries to make sure that every error is explicitly caught in log files. This additionally counts for the newly created container namespaces and child processes which rely on *logpipes*. Container events are equally treated via the *events* command line interface. A useful addition, though likely hard to implement, would be a mechanism for logging exploitation attempts. Considering previous breakout exploits that abuse vulnerabilities such as *CVE-2019-5736,* it might make sense to include a warning mechanism that makes administrators aware of attackers that try to abuse previous vulnerabilities.

Unit/Regression Testing

While tests are essential for any project, their importance grows with the scale of the endeavor. Especially for large-scale compounds, testing ensures that functionality is not broken by code changes. Further, it generally facilitates the premise where features function the way they are supposed to. Regression tests also help guarantee that previously disclosed vulnerabilities do not get reintroduced into the codebase. Testing is therefore essential for the overall security of the project.

A containerized unit- and integration-tester are shipped by *runc* and can easily be invoked via the package-provided *makefile*. This speeds up building test environments by making sure that necessary environments are present. While integration tests are centralized in *runc*'s codebase, unit-testing is spread out across a multitude of project files, thus making it harder to recognize whether specific functionalities are covered by unit-testing scripts or not. At the same time, unit-testing looks fine and covers areas ranging from *config* parsing to *cgroups* handling, as well as filesystem tests for containers. However, regression testing is missing, especially as regards assessment of

---

[1] https://github.com/sirupsen/logrus

whether fixes for previously disclosed vulnerabilities remain valid. Despite requiring additional engineering effort, reifying that is highly recommended.

Documentation

Good documentation contributes greatly to the overall state of the project. It can ease the workflow and ensure final quality of the code. For example, having a coding guideline which is strictly enforced during the patch review process ensures that the code is readable and can be easily understood by a spectrum of developers. Following good conventions can also reduce the risk of introducing bugs and vulnerabilities to the code.

The *runc* project makes a relatively positive impression as far as the existing documentation is concerned. While a good amount of information is present, it is somewhat scattered around and makes it hard to obtain a full picture of the software complex. Thus, it is possible to overlook one or the other documented pitfall. On the one hand, it may also be advantageous to get more detailed descriptions of the function of the namespaces and *cgroups* in relation to *runc*. On the other hand, highly sensible development principles[2] and equally detailed maintainer guidelines[3] underline the earnest approach the developers are taking.

Organization/Team/Infrastructure Specifics

This section will describe the areas Cure53 looked at to find out about the security qualities of the *runc* project that cannot be linked to the code and software but rather encompass handling of incidents. As such, it tackles the level of preparedness for critical bug reports within the *runc* development team. In addition, Cure53 also investigated the degree of community involvement, i.e. through the use of bug bounty programs. While a good level of code quality is paramount for a good security posture, the processes and implementations around it can also make a difference in the final assessment of the security posture.

Security Contact

To ensure a secure and responsible disclosure of security vulnerabilities, it is important to have a dedicated point of contact. This person/team should be known, meaning that all necessary information such as an email address and preferably also encryption keys of that contact should be communicated appropriately.

---

[2] https://github.com/opencontainers/runc/blob/master/PRINCIPLES.md
[3] https://github.com/opencontainers/runc/blob/master/MAINTAINERS_GUIDE.md

Alongside security notes[4], *runc* offers the relevant contact's email address (security@opencontainers.org). However, the document omits important details, such as the respective PGP keys and an outline of the disclosure process. Upon handling the reporting of the vulnerability described in RUN-01-001, Cure53 found the response times unsatisfactory. Specifically, an answer was only issued after additional inquiry. A seemingly completely unrelated email address for handling security requests (secalert@redhat.com) was provided by the developers later in the process. As such, it is highly recommended to improve this area, first by making sure security researchers can encrypt their reports by including a PGP key and, secondly, by making sure that the reporting and disclosure processes are transparently outlined. A revised strategy in this realm will make it easier for the researchers to understand the type of information to include and which answers they can expect when.

Security Fix Handling

When fixing vulnerabilities in a public repository, it should not be obvious that a particular commit addresses a security issue. Moreover, the commit message should not give a detailed explanation of the issue. This would allow an attacker to construct an exploit based on the patch and the provided commit message prior to the public disclosure of the vulnerability. This means that there is a window of opportunity for attackers between public disclosure and wide-spread patching or updating of vulnerable systems. Additionally, as part of the public disclosure process, a system should be in place to notify users about fixed vulnerabilities.

Both *SECURITY.md* and *CONTRIBUTING.md* of *runc* discourage filing of vulnerabilities directly into GitHub. They rather propose sending an email to the appropriate security contact. *runc* additionally employs a mailing list meant for distribution vendors to share actionable information when severe security issues occur. This is a good practice and makes sure that distributions are notified early on about upcoming security fixes. Usually, this increases the pace of supplying updated packages. Fixed vulnerabilities are easily identified in their respective GitHub commits. While not being tagged accordingly, they typically mention the related *CVE* number, so that a clear connection between the fix and vulnerability can unfortunately be made easily.

Bug Bounty

Having a bug bounty program acts as a great incentive in rewarding researchers and getting them interested in projects. Especially for large and complex projects that require a lot of time to get familiar with the codebase, bug bounties work on the basis of the potential reward for efforts.

---

[4] https://github.com/opencontainers/org/tree/master/security

The *runc* project does not have a bug bounty program at present, however this should not be strictly viewed in a negative way. This is because bug bounty programs require additional resources and management, which are not always a given for all projects. However, if resources become available, establishing a bug bounty program for *runc* should be considered. It is believed that such a program could provide a lot of value to the project.

Bug Tracking & Review Process

A system for tracking bug reports or issues is essential for prioritizing and delegating work. Additionally, having a review process ensures that no unintentional code, possibly malicious code, is introduced into the codebase. This makes good tracking and review into two core characteristics of a healthy codebase.

In *runc*, bugs which are not security-related should be handled via GitHub and users are able to directly submit *pull* requests. The developers seem to have a firm grip on the process of submitting, triaging and reviewing such changes.

Evaluating the Overall Posture

In general, the security posture of *runc* makes a good impression, as it can be derived from the judgments made about the individual items above. The short code audit and the history of previous vulnerabilities clearly show that there is not too much reason for concern. The handling of a security issues should probably be improved and could benefit from the incentives for reporting security issues. Nevertheless, the project has a good stance when it comes to its overall security posture.

Choosing Go has been a great decision and automatically reduces the potential for introducing memory safety-related issues. Additionally, the rather complete documentation along with the established processes for patch reviews further reduce the risk of security vulnerabilities. A topic worth-mentioning is that of a bug bounty program: since these require good funding, it is understandable that smaller projects are likely unable to secure these. However, with future growth of the project and potentially increased resources, bug bounty scheme should definitely be considered.

Phase 2: Manual code auditing & pentesting

This section comments on the code auditing coverage within areas of special interest and documents the steps undertaken during the second phase of the audit against the *runc* software complex. Cure53 describes the key aspects of the manual code audit together with manual pentesting and, since only one major issue was spotted, attests to the thoroughness of the audit and confirms the high quality of the *runc* project.

- *runc* was partially manually pentested on the server generously provided by the development team and partially examined on local systems.
- A variety of container setups have been created to gain a deeper understanding of the outer and inner workings of *runc*.
- The code responsible for dealing with mount-points was explicitly audited for common exploitation possibilities.
- Derailing *runc* by using symlinks was attempted in the context of the */dev* filesystem but this could not be achieved.
- Abusing core dumping to the host via symlinks along with spawning *zombie* process while killing their *parent* processes has led nowhere.
- It was audited to what extent interfering with the */proc* filesystem could lead to *AppArmor* not being applied correctly.
- It was checked if any privileged processes were reaching into containers with the intent of escaping the respective container by default.
- Several typical *runc* invocations were traced to see which operations and especially system calls are being used to create a container.
- It was attempted to locate TOCTOU errors in handling files/path; the *EnsureProcHandle()* is used properly.
- It was investigated what the impact of shared namespaces would be, but failing to join *mount/pid* namespaces[5] stopped these efforts.
- The access controls handled by *runc* were audited to figure out what the expectations on the host system are.
- The integration of *CRIU* with respect to abusing its invocation via *runc-checkpoint* and *runc-restore* was investigated.
- The codebase was audited for all aspects of terminal attachment functionality, in particular process invocation and the *recvtty/console* code were examined.
- The *runc* code was audited for problems in check-pointing and handling of *root* filesystems.
- The code handling of Intel RDT was given extra care, especially as regards file-handling and filesystem/scheme writing.
- The file path normalization code down the Go core library was audited. It was seen as pretty straightforward and purely lexical, with no potential for affecting symlinks.
- The *securejoin* code was analyzed and pentested with respect to symlinks in file paths, including some core library functionality.

---

[5] https://github.com/opencontainers/runc/issues/1700

Mounting/Binding and Symlinks

The code responsible for mounting volumes and binding filesystems was inspected, particularly in connection with symlinks and filename/path traversal/normalization. Cure53 was making sure it was impossible to escape out of a container via the filesystem. This was given extra care, since it had been pointed out as a focus area by the *runc* development team. In essence, one maintainer requested looking for 'ways to escape containers via symlinks and mounts from within the root filesystem' and 'binding */mnt* to */mnt* inside the container, */mnt* being a traversing link akin to ../../'.

While no obvious vulnerabilities in the core library could be identified, the discussion of these efforts nevertheless indirectly led to the discovery of RUN-01-001 by the independent party, namely Leopold Schabel (*leoluk*).

It has to be noted that the applied path-based approach used within the *runc* codebase is generally not race-safe, even as far as the application of the *filepath-securejoin* package is concerned. These path aspects, quite prone to race-conditions, should eventually be reworked to handle paths via filedescriptors and cease using textual file paths, as evidenced by the related issue[6].

Identified Vulnerabilities

The following sections list both vulnerabilities and implementation issues spotted during the testing period. Note that findings are listed in chronological order rather than by their degree of severity and impact. The aforementioned severity rank is simply given in brackets following the title heading for each vulnerability. Each vulnerability is additionally given a unique identifier (e.g. *RUN-01-001*) for the purpose of facilitating any future follow-up correspondence.

RUN-01-001 Race-condition bypassing masked paths (*High*)

Through relationships to various security researchers, an interesting opportunity for a security review emerged. Leopold Schabel (aka *leoluk*), who had previously reported an issue as a publicly disclosed *runc*[7] vulnerability, discovered a race-condition involving two containers that can be used to bypass the read-only remounting of the potentially dangerous */proc* filesystem paths.

The issue was reported by the discoverer at midnight UTC on the 26th/27th of November 2019 and has been included in the documented security mailing list of security@opencontainers.org. The steps to reproduce the issue with *podman* can be

---

[6] https://github.com/containers/crun/issues/111
[7] https://github.com/opencontainers/runc/issues/2128

found on the respective gist. Additional reproduction steps for *runc* are available by following the link.

The attack requires a *rootfs* in *container-2* where */proc* is symlinked to a folder like */evil/layer/proc.* This target folder has to be inside of a volume shared by *container-1* and *container-2*. When *container-2* is started, *runc* will first mount *procfs* by following the symlink into the shared volume to */evil/layer/proc*. Then *container-1* has to win a race condition, whereas *container-1* switches the mount-point from */evil/layer/proc* to */evil/layer~/proc*. This means the *procfs* in *container-2* is now in */evil/layer~/proc,* and not in */evil/layer/proc*. However, *runc* trusts the path and continues with the setup. Eventually *runc* will remount dangerous *procfs* paths as read-only, yet does so by following the symlink into the normal folder at */evil/layer/proc*. This means that the dangerous *procfs* paths were not remounted as read-only. After *container-1* switches the mount point back, *container-2* gains a writable access to the dangerous *procfs* paths.

During this investigation, the discoverer also realized that the fix for issue *#2128*, specifically the function *EnsureProcHandle()*, could also be bypassed with this attack. Symlinks can be used to point critical files like */proc/self/attr/%s* to other *procfs* files, thus passing the checks. However, *runc* will write *AppArmor* settings to the wrong file.

It should be noted that these issues are very difficult to fix because file paths are inherently prone to race conditions. For general file handling, it is advised to work with filedescriptors rather than paths, but there is no equivalent mount syscall that takes filedescriptors. Documenting these risks and attack surface can help projects building on top of *runc* while mitigating such issues. This can be done, for example, by using much more restrictive configurations.

Conclusions & Verdict

This assessment of the *runc* complex generally concludes on a positive note. Cure53, represented in this assessment by a team of seven testers, can conclude that the project held well to scrutiny and exhibits numerous indicators of taking security seriously. After being commissioned to perform this assessment by CNCF and having spent eighteen days on the scope, Cure53 arrived at positive verdict for both phases of the project. Consequently, it can be stated that the *runc* complex passed general security posture checks (*Phase 1)* and exposed no major mistakes in its coding practices (*Phase 2).*

To give some context, this is one in a series of high-level assessments created by Cure53 for a CNCF-selected project, which contraposes classic code audits and pentests. From the meta-level of code quality and project structure to the employed coding patterns and coherent style, the *runc* project is quite impressive. Offering a verdict, Cure53 must underline that the *runc* processes and documentation are of high quality, even though some room for improvement has been identified. The state of the software system is sound and mature.

Among the main positive conclusions, Cure53 wishes to point out that the static code analysis did not reveal any problems of significance, meaning that automated testing will most likely not yield results with the current state of technology. The choice of implementation language and external components further attests to the solid stance of the system. The general design principles and development guidelines are highly sensible and unusual in that sense.

In terms of items that are currently evaluated as possibly calling for further attention, Cure53 needs to note the lack of proper regression testing and the seemingly unstructured application of unit-tests within the codebase. Those should be reconsidered together with the redesign of documentation, which was found somewhat difficult to maneuver, in particular as regards the configuration notes being scattered.

The sole security issue discovered by a third-party during this engagement was used to test the security incident handling processes. This generally appeared to be subpar, since the reaction times were slow and the actual handling was referred to a contact person who is not documented in the project’s security guidelines. In this context, even if the resources for the project are clearly limited, the creation of a bug bounty program would be greatly beneficial to incentivize security researcher community. The race condition described in RUN-01-001 uncovered a general problem of handling file paths textually. It is recommended to rethink the approach and possibly replace it with a filedescriptor-based solution to make it race-safe.

Drawing on the findings stemming from this 2019 CNCF-funded project, Cure53 can state that the *runc* project is mature and safe, even though improving some aspects would lead to a greater praise. Notably, *runc* is being used widely in the container- and orchestration-realm, and this seems to be for very good reasons. The project can only be recommended for continued large-scale deployment. Ongoing development according to the minimal design principles and maintainer guidelines should keep the system solid in a long-term

Cure53 would like to thank Michael Crosby and Philip Estes from the *runc* 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 needs to be extended to The Linux Foundation for sponsoring this project.