# Breaking out of Docker via runC | Explaining CVE-2019-5736

In 2019-02-11 a new vulnerability in runC was [reported](https://www.openwall.com/lists/oss-security/2019/02/11/2) by its maintainers, originally found by Adam Iwaniuk and Borys Poplawski. Dubbed CVE-2019-5736, it affects Docker containers running in default settings and can be used by an attacker to gain root-level access on the host.  
Aleksa Sarai, one of runC’s maintainers, found that the same fundamental flaw exists in LXC. As opposed to Docker though, only privileged LXC containers are vulnerable. Both [runC](https://github.com/opencontainers/runc/commit/6635b4f0c6af3810594d2770f662f34ddc15b40d) and [LXC](https://github.com/lxc/lxc/commit/6400238d08cdf1ca20d49bafb85f4e224348bf9d) were patched and new versions were released.

 The vulnerability gained a lot of traction and numerous technology sites and commercial companies addressed it in dedicated posts. Here at Twistlock, our CTO John Morello wrote an [excellent piece](https://www.twistlock.com/2019/02/11/how-to-mitigate-cve-2019-5736-in-runc-and-docker/) with all the relevant details and the mitigations offered by the Twistlock platform.

 Initially, the official exploit code wasn’t to be released publicly until 2019-02-18, to prevent malicious parties from weaponizing it before users have had some time to update. In the following days though, several people decided to release their exploit code. That led the runC team to eventually [release their exploit code](https://www.openwall.com/lists/oss-security/2019/02/13/3) earlier (2019-02-13) since – as they put it – “the cat was out of the bag”.

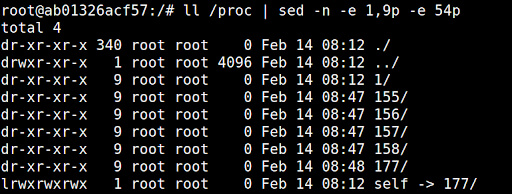
This post aims to be a comprehensive technical deep dive into the vulnerability and it’s various exploitation methods.

# What is runC?

RunC is a container runtime originally developed as part of Docker and later extracted out as a separate open source tool and library. As a “low level” container runtime, runC is mainly used by “high level” container runtimes (e.g. Docker) to spawn and run containers, although it can be used as a stand-alone tool.  
“High level” container runtimes like Docker will normally implement functionalities such as image creation and management and will use runC to handle tasks related to running containers – creating a container, attaching a process to an existing container (docker exec) and so on.

# Procfs

To understand the vulnerability, we need to go over some procfs basics. The proc filesystem is a virtual file system in Linux that presents information primarily about processes, typically mounted to /proc. It is virtual in a sense that it does not exist on disk. Instead, the kernel creates it in memory. It can be thought of as an interface to system data that the kernel exposes as a filesystem. Each process has its own directory in procfs, at /proc/[pid]:



As shown in the image above, /proc/self is a symbolic link to the directory of the currently running process (in this case pid 177). Each process’s directory contains several files and directories with information on the process. For the vulnerability, the relevant ones are:

* /proc/self/exe – a symbolic link to the executable file the process is running, and ;
* /proc/self/fd – a directory containing the file descriptors open by the process.

For example, by listing the files under /proc/self using ls /proc/self one can see that /proc/self/exe points to the ‘ls’ executable.

https://www.twistlock.com/wp-content/uploads/2019/03/procfs2.jpg  
That makes sense as the one accessing /proc/self is the ‘ls’ process that our shell spawned.

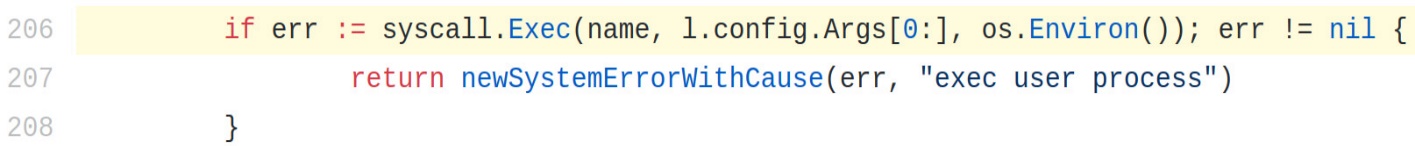
# **The Vulnerability**

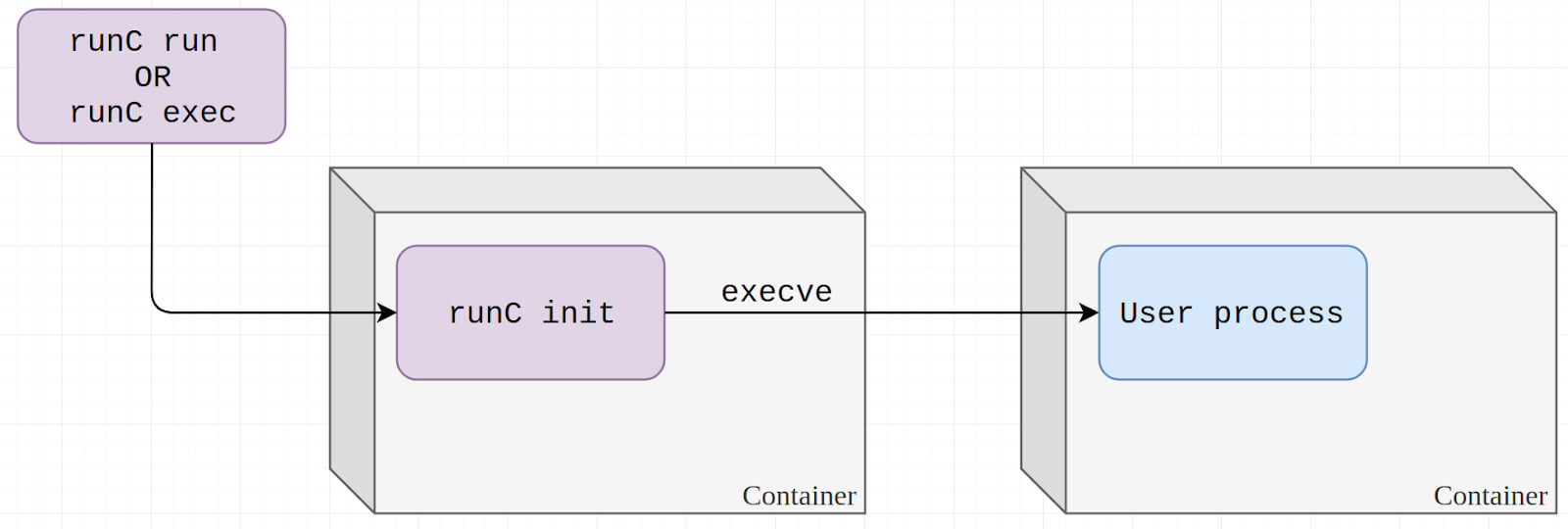
Let’s go over the vulnerability overview given by the runC team:

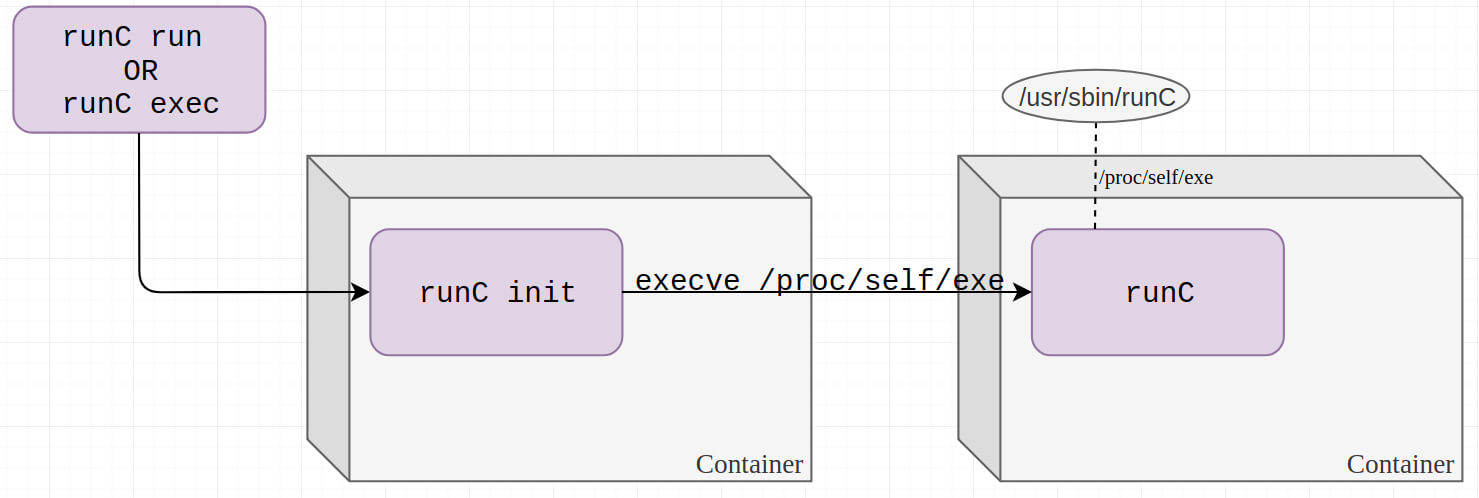
The vulnerability allows a malicious container to (with minimal user interaction) overwrite the host runc binary and thus gain root-level code execution on the host. The level of user interaction is being able to run any command ... as root within a container in either of these contexts:

* Creating a new container using an attacker-controlled image.
* Attaching (docker exec) into an existing container which the attacker had previous write access to.

Those two scenarios might seem different, but both require runC to spin up a new process in a container and are implemented similarly. In both cases, runC is tasked with running a user-defined binary in the container. In Docker, this binary is either the image’s entry point when starting a new container, or docker exec’s argument when attaching to an existing container.

When this user binary is run, it must already be confined and restricted inside the container, or it can jeopardize the host. In order to accomplish that, runC creates a ‘runC init’ subprocess which places all needed restrictions on itself (such as entering or setting up namespaces) and effectively places itself in the container. Then, the runC init process, now in the container, [calls](https://github.com/opencontainers/runc/blob/751f18de2af90495e9c5665b95bfc7adf66ddd57/libcontainer/standard_init_linux.go#L206) the execve syscall to overwrite itself with the user requested binary.

This is the method used by runC both for creating new containers and for attaching a process to an existing container.

The researchers who revealed the vulnerability discovered that an attacker can trick runC into executing itself by asking it to run /proc/self/exe, which is a symbolic link to the runC binary on the host.

An attacker with root access in the container can then use /proc/[runc-pid]/exe as a reference to the runC binary on **the host** and overwrite it. Root access in the container is required to perform this attack as the runC binary is owned by root.  
The next time runC is executed, the attacker will achieve code execution on the host. Since runC is normally run as root (e.g. by the Docker daemon), the attacker will gain root access on the host.

# Why not runC init?

The image above might mislead some to believe the vulnerability (i.e. tricking runC into executing itself) is redundant. That is, why can’t an attacker simply overwrite /proc/[**runc-init-pid**]/exe instead?  
A patch for a similar runC vulnerability, [CVE-2016-9962](https://bugzilla.suse.com/show_bug.cgi?id=1012568), mitigates this kind of attack.  
CVE-2016-9962 revealed that the runC init process possessed open file descriptors from the host which could be used by an attacker in the container to traverse the host’s filesystem and thus break out of the container. Part of the patch for this flaw was setting the runc init process as ‘non-dumpable’ before it entering the container.

In the context of CVE-2019-5736, the ‘non-dumpable’ flag denies other processes from dereferencing /proc/[pid]/exe, and therefore mitigates overwriting the runC binary through /proc/[runc-init-pid]/exe . Calling execve drops this flag though, and hence the new runC process’ /proc/[runc-pid]/exe is accessible.

# The Symlink Problem

The vulnerability may appear to contradict the way symbolic links are implemented in Linux.  
Symbolic links simply hold the path to their target. For a runC process, /proc/self/exe should contain something like /usr/sbin/runc.  
When a symlink is accessed by a process, the kernel uses the path present in the link to find the target under the root of the accessing process.  
That begs the question – **when a process in the container opens the symbolic link to the runC binary, why doesn’t the kernel searches for the runC path inside the container root?**

The answer is that /proc/[pid]/exe does not follow the normal semantics for symbolic links. Technically this might count as a violation of [POSIX](https://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap03.html#tag_03_381), but as I mentioned earlier procfs is a special filesystem. When a process opens /proc/[pid]/exe, there is none of the normal procedure of reading and following the contents of a symlink. Instead, the kernel just gives you access to the open file entry directly.

# Exploitation

Soon after the vulnerability was reported, when no POCs were publicly released yet, I attempted to develop my own POC based on the detailed description of the vulnerability given in the [LXC patch](https://github.com/lxc/lxc/commit/6400238d08cdf1ca20d49bafb85f4e224348bf9d) addressing it. Let’s break down LXC’s description of the vulnerability:

When runC attaches to a container the attacker can trick it into executing itself. This could be done by replacing the target binary inside the container with a custom binary pointing back at the runC binary itself. As an example, if the target binary was /bin/bash, this could be replaced with an executable script specifying the interpreter path #!/proc/self/exe

The ‘#!’ syntax is called shebang and is used in scripts to specify an interpreter. When the Linux loader encounters the shebang, it runs the interpreter instead of the executable.

the program finally executed by the loader is:  
interpreter [optional-arg] executable-path

When the user runs something like docker exec container-name /bin/bash, the loader will recognize the shebang in the modified bash and execute the interpreter we specified – /proc/self/exe, which is a Symlink to the runC binary.

We can proceed to overwrite the runC binary from a separate process in the container through /proc/[runc-pid]/exe.

# Shared Libraries Approach

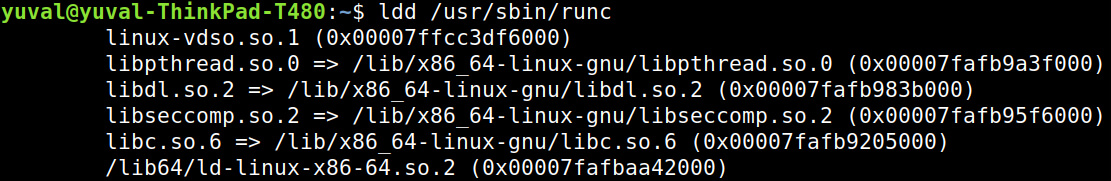
A different exploitation method is used in the official POC released by runC’s maintainers and is superior to POCs similar to mine since it can be implemented to compromise the host through two separate methods:

1. When a user execs a command into an existing attacker controlled container
2. When a user runs a malicious image

We’ll now look into building a malicious image since the previous POC already demonstrated the first scenario. The POC I wrote for this method is heavily based on q3k’s POC, which, to the best of my knowledge, was the first published malicious image POC.

Let’s go over the Dockerfile used to build the malicious image. First, the entry point of the image is set to /proc/self/exe in order to trick runC into executing itself when the image is run.



RunC is dynamically linked to several shared libraries at run time, which can be listed using the **ldd** command.

When the runC process is executed in the container, those libraries are loaded into the runC process by the dynamic linker. It is possible to substitute one of those libraries with a malicious version that will overwrite the runC binary upon being loaded into the runC process.  
Our Dockerfile builds a malicious version of the libseccomp library:



The Dockerfile appends the content of run\_at\_link.c to one of libsecomp’s source files. Subsequently, the malicious libsecomp is built.

The constructor attribute (a GCC-specific syntax) indicates that the run\_at\_link function is to be executed as an initialization function for libseccomp after the dynamic linker loads the library into the runC process. Since run\_at\_link will be executed by the runC process, it can access the runC binary at /proc/self/exe.  
The runC process must exit for the runC binary to be writable though. To enforce the exit, run\_at\_link calls the execve syscall to execute overwrite\_runc.

Since execve doesn’t affect the file descriptors open by the process, the same file descriptor trick from the previous POC can be used:

1. The runC process loads the libseccomp library and transfers execution to the run\_at\_link function.
2. run\_at\_link opens the runC binary for reading through /proc/self/exe. This creates a file descriptor at /proc/self/fd/${runc\_fd\_read}.
3. run\_at\_link calls execve to execute overwrite\_runc.
4. The process is no longer running the runC binary, overwrite\_runc opens /proc/self/fd/runc\_fd\_read for writing and overwrites the runC binary.

The built malicious image that overwrites the runC binary with a simple script that spawns a reverse shell at port 2345.

The docker run command executes runC twice. Once to create and run the container, which executes the POC to overwrite runC, and then again to stop the container using runc delete.  
The second time runC is executed, it is already overwritten, and hence the reverse shell script is executed instead.

# The Fix

RunC and LXC were both patched using the same approach, which is described clearly in the LXC patch commit:

To prevent this attack, LXC has been patched to create a temporary copy of the calling binary itself when it starts or attaches to containers. To do this LXC creates an anonymous, in-memory file using the **memfd\_create()** system call and copies itself into the temporary in-memory file, which is then sealed to prevent further modifications. LXC then executes this sealed, in-memory file instead of the original on-disk binary.

Any compromising write operations from a privileged container to the host LXC binary will then write to the temporary in-memory binary and not to the host binary on-disk, preserving the integrity of the host LXC binary. Also as the temporary, in-memory LXC binary is sealed, writes to this will also fail.

RunC has been patched using the same method. It re-executes from a temporary copy of itself when it starts or attaches to containers. Consequently, /proc/[runc-pid]/exe now points to the temporary file, and the runC binary can’t be reached from within the container.  
The temporary file is also sealed to block writing to it, although overwriting it shouldn’t compromise the host.

This patch introduced some issues though. The temporary runC copy is created in-memory after the runc init process has already applied the container’s cgroup memory constraints on itself. For containers running with a relatively low memory limit (e.g. 10Mb), this can cause processes in the container to be oom-killed (Out Of Memory killed) by the kernel when the runC init process attaches to the container.