Secure Execution Environments 2023-24

Practical Exercises: Linux namespaces

February 22, 2024

Changelog

- v1.0 - Initial version.

1 Introduction

The goal of this laboratory guide is to show how to make use of Linux namespaces and the result of their application.

These exercises must be executed in Linux systems, which can run on virtual machines.

2 Process namespace

A top-most process of a new process namespace is created with the `clone` syscall. This call accepts some configuration flags, and the one to use to initiate a new namespace is `CLONE_NEWPID`.

Edit, compile and run the following C code, in order to observe the results of creation a new process namespace. Notice that you need to add the flag `CLONE_NEWUSER` to enable the new process/thread to create new namespaces.

```c
#define _GNU_SOURCE
#include <sched.h>
#include <unistd.h>
#include <stdio.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/wait.h>

static char child_stack[1048576];

int child()
{
    printf( "Child: PID = %d, PPID = %d, EUID = %d\n", getpid(), getppid(), geteuid() );

    return 0;
}

int main()
{
    int child_pid, status;

    printf( "Parent: PID = %d, PPID = %d\n", getpid(), getppid() );

    child_pid = clone( child, child_stack + sizeof(child_stack), CLONE_NEWUSER |
        CLONE_NEWPID, 0 );

    printf( "Parent: child PID = %d (errno = %d)\n", child_pid, errno );

    waitpid( child_pid, &status, 0 );

    return 0;
}
```

3 Network namespace

A process created with a new network namespace cannot directly use the network interfaces existing in the outer namespace. A new network namespace is created with the `CLONE_NEWNET` flag of the `clone` syscall. The process created this wait will have, at start, a new network namespace initialized with a virtual, loopback interface (starting in the DOWN state). This loopback interface is different from the one of other network namespaces.

```c
#define _GNU_SOURCE
#include <sched.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>

static char child_stack[1048576];

int child()
{
    printf( "Child network namespace\n" );
    system( "ip link" );

    return 0;
}

int main()
{
    int status;

    printf( "Parent network namespace\n" );
    system( "ip link" );

    clone( child, child_stack + sizeof(child_stack), CLONE_NEWUSER | CLONE_NEWNET, 0 );

    wait( &status );

    return 0;
}
```

4 User namespace

A new user name enables a process to gain all system capabilities while loosing a mapping between their UID/GIDs and some value usable to derive access privileges. Such lack of mapping yields a value of 65534, with is ordinarily mapped to the user/group `nobody`.

Compile and run the following application:

```c
#define _GNU_SOURCE
#include <sched.h>
#include <unistd.h>
#include <stdio.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <signal.h>

static char child_stack[1048576];

int child()
{
    for (;;) {
        printf( "Child: PID = %d, EUID = %d, RUID = %d\n", getpid(), geteuid(), getuid() );
        sleep( 1 );
    }

    return 0;
}

int main()
{
    int child_pid, status;

    printf( "Parent: EUID = %d, RUID = %d\n", geteuid(), getuid() );

    child_pid = clone( child, child_stack + sizeof(child_stack), CLONE_NEWUSER, 0 );

    do {
        waitpid( child_pid, &status, 0 );
    } while (!WIFEXITED(status));

    return 0;
}
```

Then, is another console, find the PID of the child process, the one with the new user namespace, and execute:

```
echo '2 UID 1' > /proc/PID/uid_map
```

where UID should be the UID of the father process (see initial output of the application running) and PID the PID indicated by the child application. Notice that the output changes. Try the command above with different value; you should not succeed. This is a one-time operation.

5 UTS namespace

A Linux host posesses a set of names: host name, domain name, operating system name, etc. These names define a so-called UTS (UNIX Time Sharing) name. A new UTS namespace is created with the `CLONE_NEWUTS` flag of the `clone` syscall. The process created this wait will have, at start, a new UTS namespace initialized with the values used in the outer namespace. A UTS namespace allows processes to modify UTS components without affecting the names used in outer namespaces.

```c
#define _GNU_SOURCE
#include <sched.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/utsname.h>

static char child_stack[1048576];

void uts_name( char * prefix )
{
    struct utsname name;

    uname( &name );
    printf( "%s: %s\n", prefix, name.nodename );
}

int child()
{
    uts_name( "original child UTS name" );
    sethostname( "AES_host", 8 );
    uts_name( "new child UTS name" );

    return 0;
}

int main()
{
    int status;

    uts_name( "parent UTS name" );

    clone( child, child_stack + sizeof(child_stack), CLONE_NEWUSER | CLONE_NEWUTS, 0 );

    wait( &status );

    sleep( 1 ); // Just to wait for the flush of children I/O

    uts_name( "parent UTS name" );

    return 0;
}
```

6 The unshare tool

The `unshare` Linux tool can be used to run an application with an arbitrary set of new namespaces.

6.1 process namespace

Execute the following command:

```
sudo unshare --fork --pid su ‘id -u -n‘
echo $$
```

The first command creates a login shell (with `su`) for your current username, but under a new process namespace.

The `$$` variable contains the PID of the current shell interpreter. You should see the value 1.

List the directory `/proc`. You should see many directories with a name that is a number: them represent processes, and the numbers their PID. In this case, these processes belong to a different name-space, so the process of your command interpreter is there but not under PID 1.

Assuming that your command interpreter is `bash`, execute the following command:

```
pidof bash
```

It lists the PID of all the `bash` processes. Now, which one is yours?

Execute the following command for showing the process namespace of each of the command interpreter instances:

```
for p in ‘pidof bash‘; do
ls -la /proc/$p/ns/pid
done
```

Can you see your command interpreter now? Now you know the PID of your command interpreter in its outer namespace, in this case in the namespace of the one that executed the `unshare` command.

Without exiting from the current namespace, execute again the `unshare` command and list again the process namespaces of `bash` instances. Now you should see two `bash` instances with a different process namespace.

Terminate all the command interpreters created in different process namespaces and start again the exercises proposed in this section, but this time with the `--mount-proc` option (before the command to be ran by `unshare`, `su`, in this case). This option forces the remounting of the `/proc` filesystem for the current namespace, by default in the same directory (`/proc`). Now you should be able to see the PID of processes from outer process namespaces under this directory.

Terminate the current shell before proceeding.

6.2 Network namespace

Make sure you have to shell consoles active (with will name them as C1 and C2). In C2 execute the following command:

```
sudo unshare --fork --net su ‘id -u -n‘
ip link
```

The first command creates a login shell (with `su`) for your current username, but under a new network namespace.

The second command lists the network interfaces visible to the shell, which is only a new loopback interface, still down:

In an inner network name space you can create virtual network interfaces (called `veth`) to link to other virtual network interfaces belonging to outer network name spaces. However, this can only be done from the outer namespaces to the inner ones, and not the other way around, for security sake.

For doing this, one needs to identify the inner network namespace from the outer one. This can be done through the PID of the process that belongs to the inner network namespace. To do that, execute the following comamand from either C1 or C2:

```
for p in ‘pidof bash‘; do
ls -la /proc/$p/ns/net
done
```

and look for the PID of the `bash`-running process that has a network namespace identifier different from all the other. For the following examples, we will assume it is 12345.

Thus, run the following command in C1:

```
sudo ip link add name veth_outer type veth peer name veth_inner netns 12345
```

This command creates two `veth` interfaces, directly connected with each other: `veth\_outer`, in the outer namespace, and `veth\_inner`, in the inner namespace. You can observe each of them by executing the command

```
ip link
```

in both C1 and C2 consoles.

Now, you can configure them, activate them and experiment the communication betwwen them. In C1, execute:

```
sudo ip addr add 10.1.1.1/24 dev veth_outer
sudo ip link set veth_outer up
```

and in C2 execute:

```
sudo ip addr add 10.1.1.2/24 dev veth_inner
sudo ip link set veth_inner up
```

Now, in C1 you can ping the interface on the inner namespace:

```
ping 10.1.1.2
```

whilst in C2 you can ping the interface on the outer namespace:

```
ping 10.1.1.1
```

Now, processes in the outer network namespace can control the communication of the inner one. For instance, you can provide direct access to the Internet. In C1, execute the following commands to allow a NAT forwarding from the inner namespace:

```
sudo bash -c 'echo 1 > /proc/sys/net/ipv4/ip_forward'
sudo iptables -t nat -A POSTROUTING -s 10.1.1.0/24 -j MASQUERADE
```

and in C2 execute these to instruct how to address other networks from the inner namespace:

```
sudo ip route add default via 10.1.1.1
```

Now, you should succeed in reaching any Internet host from C2:

```
ping 1.1.1.1
```

Terminate the shell on console C2 and observe that the `veth_outer` interface disapears. However, the rest of the configuration, which is independent of this interface, remains (iptable rules and IP forwading). Remove them with these commands:

```
sudo bash -c 'echo 0 > /proc/sys/net/ipv4/ip_forward'
sudo iptables -t nat -D POSTROUTING -s 10.1.1.0/24 -j MASQUERADE
```