# Project Submission

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | STUDENT NAME: | Ruben Vasconcelos | | STUDENT NUMBER: |  | | STUDENT NAME: | Daniel Rejniak | | STUDENT NUMBER: |  | | PROJECT NAME: | Linux System Call | | MODULE CODE: | CA644 | | DEGREES: |  | | LECTURER: | Brian Stone | |

|  |
| --- |
| **Declaration**  ***In submitting this project, I declare that the project material, which I now submit, is my own work. Any assistance received by way of borrowing from the work of others has been cited and acknowledged within the work. I make this declaration in the knowledge that a breach of the rules pertaining to project submission may carry serious consequences.***  ***Signature:\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*** |

**Introduction**

The assignment is divided into two sections. Section one describes the process for extending the Linux Kernel by adding a new system call.

The second section covers the specifics of our advanced system call, which is a sort of mini Docker system call utility for managing Linux containers.

**System Information**

|  |  |
| --- | --- |
| OS | Ubuntu |
| Release | 16.04 |
| Initial Kernel Version | 4.13.9 |
| New Kernel Version | 4.14.2 |

**Installing and Compiling Kernel (Part 1)**

1.1 Updating OS & Installing Required Packages

1.2 Defining the new System Call

1.3 Creating a Makefile for the new System call

1.4 Adding the new System Call’s location to the kernel’s top Makefile

1.5 Updating the System Call table and header file

1.6 Compiling and Installing the modified Kernel

1.7 Confirmation

1.8 Automated Shell Script

Before doing any work, we executed “uname –r” to verify the initial kernel version.

|  |
| --- |
| uname -r: 4.13.9 |

**1.1 Updating OS & Installing Required Packages**

The initial step is to patch the system in order to ensure that our OS is running the latest available packages. We also add some additional packages that are required to rebuild the kernel using “apt-get install -y”. Runc is also introduced at this point as it will be required by our advanced system call later on.

|  |
| --- |
| apt-get update && apt-get upgrade -y && apt-get install -y \  wget \  make \  gcc \  libssl-dev \  libncursesw5-dev \  runc |

The next step it to download the kernel from kernel.org. We opted for the latest stable kernel. We download and extract the new kernel into usr/src directory.

|  |
| --- |
| # Download kernel kernel\_major\_v="4" kernel\_minor\_v="14.2" kernel\_full\_v="$kernel\_major\_v.$kernel\_minor\_v"  rm -rf /usr/src/linux-$kernel\_full\_v linux-$kernel\_full\_v.tar.gz && wget https://www.kernel.org/pub/linux/kernel/v$kernel\_major\_v.x/linux-$kernel\_full\_v.tar.gz && tar -xvf linux-$kernel\_full\_v.tar.gz -C/usr/src/ && rm -rf linux-$kernel\_full\_v.tar.gz |

**1.2 Defining the new System Call**

After extracting the new kernel we created a directory inside it for the new System call. E.g “/usr/src/linux-4.14.2/hello/”. Then inside that directory we created a *hello.c* file with our system call’s code

|  |
| --- |
| #include <linux/kernel.h>  asmlinkage long sys\_hello(void) {  printk("Hello world\n");  return 0; } |

**1.3 Creating a Makefile for the new System Call**

The Makefile is used by the kbuild system, it will act as a binder during compilation.

|  |
| --- |
| obj-y := hello.o |

The makefile is used to define the files to be build and other specific build options. In this case we’re telling kbuild that there should be a “hello.o” file in the directory, which is to be built from our hello.c file created earlier on.

**1.4 Adding the new System Call’s location to the kernel’s top Makefile**

We need to define the location of our new “hello” directory in the top Makefile so the compiler knows where to find it during the compilation process.

|  |
| --- |
| sed -ri 's/core-y.\*kernel.\*/& hello\//' /usr/src/linux-4.4.1/Makefile |

**1.5 Updating the System call table and header file**

Now we need to create a new entry in the system call table file since we decided to go for the 64bit version of Ubuntu we need to add the new entry in the following file.

“/usr/src/linux-4.4.1/arch/x86/entry/syscalls/syscall\_64.tbl”

Inside the “syscall\_64.tbl” we added the following entry. Note that we have incremented the value of the last call since all calls in the table need to be unique.

|  |
| --- |
| # 333 common hello sys\_hello |

The next step is to update the header file located under “include/linux/syscalls.h”, this file is used to define the system calls function prototype. Using the “asmlinkage” key word lets the compiler know that all the parameters to the new system call will be put into the stack.

|  |
| --- |
| asmlinkage long sys\_hello(void) |

**1.6 Compiling and Installing the modified Kernel**

First we have to create the configuration file. There are a number of option such as oldconfig to use the existing kernel configurations which allow us to build upon the existing kernel configurations but we have opted to go with new configurations.

|  |
| --- |
| make menuconfig |

Once we have generated the .config file from the previous step we’re ready to compile the modified kernel using make.

|  |
| --- |
| make |

By default will use only one core, so in order to speed up the process you can pass in –jN flag where N is equal to the number of cores available.

|  |
| --- |
| make modules\_install install |

The next step is to install the new compiled modules.

We encountered multiple issues during the compilation and installation process. Firstly, we noticed that we require substantially more RAM to complete the kernel installation so we assigned 4GB to our VM, we also noticed some disk related issues and had to assign over 20GB of virtual disk space.

|  |
| --- |
| # Update GRUB boot order.  mkinitramfs -o /boot/initrd.img-$kernel\_full\_v $kernel\_full\_v &&  update-grub |

After compiling and installing the new kernel, some boot files were generated under “/boot”, we run the command above to tell the boot loader “GRUB” which kernel version to use.

The last step was to reboot the system.

|  |
| --- |
| reboot |

**1.7 Confirmation**

After rebooting the system, we performed some basic checks to ensure our new kernel has been installed successfully.

We ran uname –r again and confirmed that the version matched the latest stable version.

|  |
| --- |
| uname -r : 4.14.2 |

Afterwards we created a simple c program to test the new System call.

|  |
| --- |
| #include <stdio.h> #include <sys/syscall.h> #include <unistd.h>  int main() {  long int ret\_code = syscall(333);  printf("Return code from syscall: %ld\n", ret\_code);  return 0; } |

To compile and execute the simple test program we ran the following :

|  |
| --- |
| gcc test.c –o test ./test |

We ran dmesg in order to print the kernel log.

|  |
| --- |
| dmesg |

After examining the log we were able to confirm that the new system call executed successfully.

**1.8 Automated shell script**

We decided that it would be a good idea to automate the install process using grep and sed, as it would help save time and avoid manual mistakes during the developments of our advanced system call.

We created a script called “build\_and\_install.sh”, because this is a bit outside the assignment scope we’re not going to go into much detail but will submit it alongside the system call code and are happy to explain in more detail how the script works.

|  |
| --- |
| #!/bin/bash  test $EUID -ne 0 && { echo "This script must be run as root" >&2; exit 1; }  apt-get update && apt-get upgrade -y && apt-get install -y \    wget \    make \    gcc \    libssl-dev \    libncursesw5-dev \    runc  test $? -ne 0 && { echo "Failed to update and install dependencies." >&2 ; exit 1; }  # Download kernel kernel\_major\_v="4" kernel\_minor\_v="14.2" kernel\_full\_v="$kernel\_major\_v.$kernel\_minor\_v"  rm -rf /usr/src/linux-$kernel\_full\_v linux-$kernel\_full\_v.tar.gz && wget https://www.kernel.org/pub/linux/kernel/v$kernel\_major\_v.x/linux-$kernel\_full\_v.tar.gz && tar -xvf linux-$kernel\_full\_v.tar.gz -C/usr/src/ && rm -rf linux-$kernel\_full\_v.tar.gz  test $? -ne 0 && { echo "Failed to download/untar linux kernel $kernel\_full\_v, make sure you picked a valid version." >&2 ; exit 1; }  # Add new system call mkdir /usr/src/linux-$kernel\_full\_v/hello && cp hello.c Makefile /usr/src/linux-$kernel\_full\_v/hello/  sed -ri 's/core-y.\*kernel.\*/& hello\//' /usr/src/linux-$kernel\_full\_v/Makefile && sed -ri 's/^#endif$/asmlinkage long sys\_hello(void);\n&/' /usr/src/linux-$kernel\_full\_v/include/linux/syscalls.h  sed -rie 'N;s/([0-9]+).\*\n^$/&\1\n/;P;D' /usr/src/linux-$kernel\_full\_v/arch/x86/entry/syscalls/syscall\_64.tbl && sys\_call\_num=$(($(grep -E "^[0-9]+$" /usr/src/linux-$kernel\_full\_v/arch/x86/entry/syscalls/syscall\_64.tbl)+1)) && sed -ri "s/^[0-9]+$/$sys\_call\_num\t64\thello\t\t\tsys\_hello/g" /usr/src/linux-$kernel\_full\_v/arch/x86/entry/syscalls/syscall\_64.tbl  # Compile kernel and install kernel modules. cd /usr/src/linux-$kernel\_full\_v && make menuconfig && make -j$(nproc) && make modules\_install install  test $? -ne 0 && { echo "Unable to compile and install kernel, possibly a dependency issue." >&2; exit 1; }  #Update GRUB boot order. mkinitramfs -o /boot/initrd.img-$kernel\_full\_v $kernel\_full\_v && update-grub && reboot |

**Advanced System Call (Part 2)**

2.1 Overview

2.2 Create Container

2.2.1 Intro

2.2.2 Architecture

2.2.3 Security Concerns and Decisions

2.3 List Functionality

2.4 Start Functionality

2.5 Delete Functionality

2.6 Error Handling

2.7 Logging

2.9 Screenshots

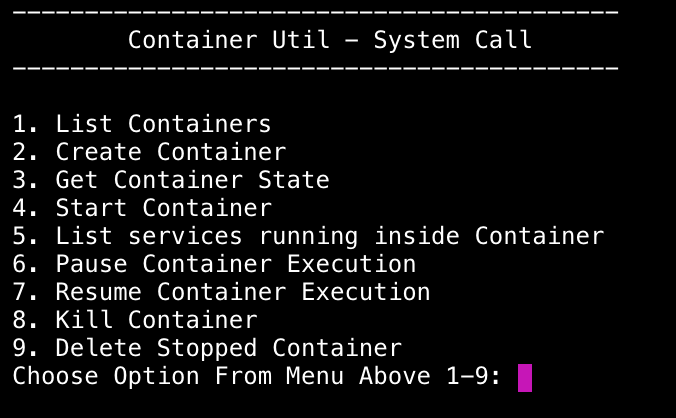
**2.1 Overview**

We decided to create a helper utility that will allow developers to create and manage Linux container without having Docker installed on the operating systems. The architecture is somewhat based on the architecture of Docker it-self, as we leverage the same container runtime technology “runc” that Docker uses. In fact, the container rootfs (more on this later) is from an Alpine Docker image. Alpine is a minimal Unix OS used by a large portion of the Docker community (Glider Labs Alpine Docker, 2017).

There are two key parts to our design, the first part is a front-end application that provides a menu where users can choose variety of options such as create, list, start, delete and list services inside the container. User does not need to know the actual commands since the command are encapsulated conveniently under each menu option.

The second part is the actual system call that receives and processes requests from the front-end application in a secure manner. Once the request has been processed by the kernel the system call returns some feedback back to the front-end application.

Below is a screenshot of our applications main menu:



The system call is designed to take in two parameters. The first parameter is the action that the user wishes to execute, the second parameter is the container id.

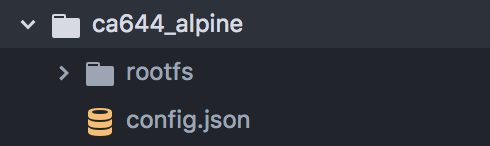
*sys\_call(int container\_action, char\* container\_id)*

There are security concerns around with creating containers so we implemented it on the client side. All other features can be implemented in the system call as non of these will allow an attacker to gain root access to the host system

**2.2 Create Container**

From the user’s perspective, all they have to do to create a container is to provide a container id (up to 20 chars long) and a new container is provisioned.

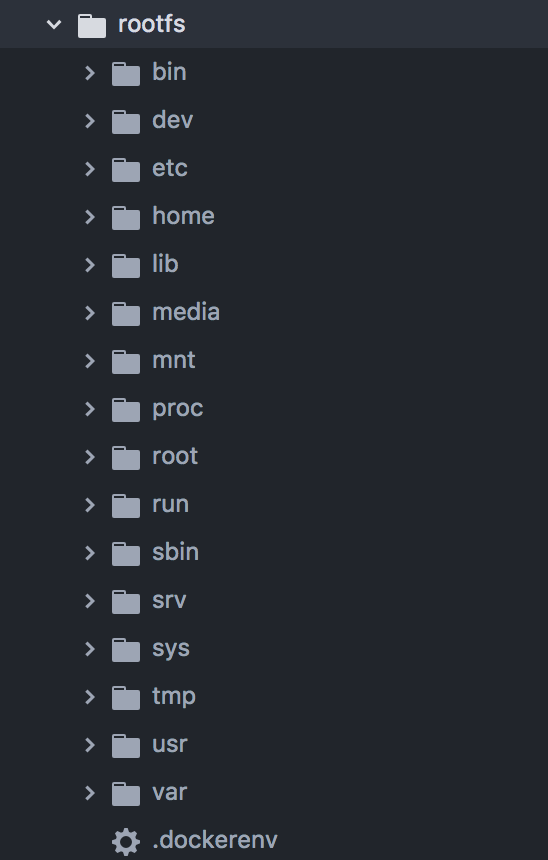
A few things must happen or be put in place before we can create a Linux container. First we created a new folder called ca644\_alpine for the container rootfs and config file which will then be used by runc. (Open Container Initiative, 2017)



The container’s rootfs defines the container’s operating system, in Docker terms this is known as the “Image”. You can extract the rootfs of any docker image available in docker hub by running the following commands.

|  |
| --- |
| mkdir rootfs && docker export $(docker create $IMAGE\_NAME) | tar -C rootfs -xvf - |

After extracting the images rootfs you will end up with something that looks like this.



Next we generated the config.json file using “runc spec”, the default config file must be updated in order to work with our System Calls architecture.



By default, terminal is equal to true, which would run the container and attach it to your shell, you can think of it as the equivalent of running a foreground process. This is clearly not going to work with our design so instead we run it as a background process also known and detached mode.

The processes are being executed inside the container as root, usually not good practice but should be ok in our case.

The args section defines the process we would like to execute inside our new container. In our case, it doesn’t do an all lot, in fact all it does is sleep for 300 seconds. Not very amazing but it’s more than adequate to show present all of our system calls features

We decided to keep the create functionality on the front-end of the application rather than having it in the kernel due to security concerns.

If the containers were being created by our system call, then the user could modify the config file to mount the ‘/’ directory or system root inside the container, at that point it would become trivial to gain root access to the host machine using “chroot” which is another operating system level virtualization that come with the linux kernel, in fact is seen by many as the predecessor of Linux container.

E.g, By adding this to the config file you would be able to see the host file system under /host-root/ inside the containers



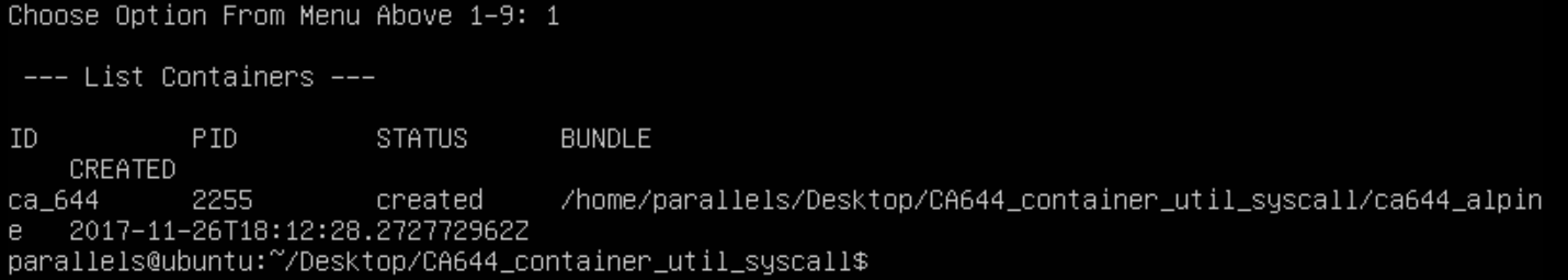
As pointed out earlier on the processes are being executed as root inside the container, you have probably put two and two together. Running as root and access to the host files system is pretty much any attackers dream.

So instead we decided to implement the functionality on the client side, this way the user will **NOT** be able to create any containers, even if they change our client code or the config.

We could have implemented some checks on the kernel to check that ‘/’ wasn’t being mounted or ensuring the config file hadn’t been changed, but we decided that potentially there would still be some unknown vulnerabilities, so decided to go with the only option that ensured security as good as docker and runc by not introducing new vulnerabilities.

**2.3 List Containers**

List functionality allows the user to view all the available container. Additional information such as container id, state, pid bundle and date are also provided



**2.4 Start Container**

This functionality allows the user to start a previously created container by simply providing the container id.

**2.5 Delete Container**

Once the container finishes executing its work (state=STOPPED), it can be removed by choosing the delete option and providing the container id.

**2.6 Error Handling**

When the system call is executed it will return a code to the front-end application, 0 for success and 1 for error. That way we can establish if the outgoing request has been processed successfully. We built a responsive mechanism where any errors are written to a file under “/tmp”, if a 1 is returned to the client application then we read the contents of that file and present them to the user.

**2.7 Logging**

Since the calls are made from user land to the kernel as a system admin we want to be able to obtain information about the incoming requests. Using the “current” struct which is made available to the system call we retrieve information such as the PID of calling process and the actual process name. This information is then logged in the kernel log which can be viewed by execution the following command.

|  |
| --- |
| dmesg |

We also log the user inputs and the executional behavior of the system call, this is useful information that helps to determine the interaction of a process with our system call.

**2.8 (Security)**

Some precautions had to be put in place when dealing with input from the user that will reach the system call.

* Buffer bounds checks
* Null termination characters

We used of fgets to read and limit user inputs in the userland application.

The System call itself will perform a check before processing the incoming values. In order to protect against a buffer overflow attack. In an event where the client side application was altered to pass it container ID of an arbitrarily length.

**3 References:**

Glider Labs Alpine Docker. (2017). Docker Alpine [online] Available at: https://hub.docker.com/\_/alpine/ [Accessed 18 Nov. 2017].

Open Container Initiative. (2017). opencontainers/runc. [online] Available at: https://github.com/opencontainers/runc [Accessed 26 Nov. 2017].