Exploratory Project on **Google Fuchsia**

EURECOM, Semester Project 2018-2019

Supervisors
Davide Balzarotti
Yanick Fratantonio
Dario Nisi
Fabio Pagani
Andrea Possemato

Students
Andrea Palmieri
Paolo Prem

Contents

1 Introduction

Fuchsia is a new operating system currently being developed by Google. The project appeared on GitHub in August 2016 and it is currently hosted at *fuchsia.googlesource.com*[1]. Although the development of the project has been going on for more than two years, Google has not officially announced it yet. While the other operating systems developed by Google - Chrome OS and Android - are based on the Linux kernel, Fuchsia is based on a new *micro-kernel*, called **Zircon**.

The description of the project suggests that Fuchsia will be able to run on different platforms: from embedded systems to smartphones, tablets and even personal computers. In May 2017 a user interface was added to the main project, which contributed to media speculation about Google's intentions with Fuchsia, including the possibility that it could replace Android in the future.

In the official documentation, Fuchsia is defined as a *"modular, capability-based operating system"*[2]. The **modular** term of the definition refers to the fact that the components of the system are divided into distinct process, as opposed to the traditional monolithic operating systems. This approach introduces different advantages, such as an easier update mechanism and features addition, as well as the possibility to download parts of components only when needed. The second part of the definition, instead, refers to the **capability-based security** enforced in Fuchsia by the use of constructs called *handles*.

In this report we are going to discuss the results of our exploratory semester project. It has to be considered that the operating system is still under development, thus the topics examined on this report and in project itself are continually subject to changes. In order to be updated with the development of Fuchsia, we recommend to follow the official documentation available on *fuchsia.googlesource.com*[2].

2 Fuchsia Layers

In the Google documentation, it is often used the analogy of a *4 layer cake* to describe the organization of the Fuchsia project. The names of the layers are: **Zircon**, **Garnet**, **Peridot** and **Topaz**. In this section we are going to briefly describe their purpose and main features.

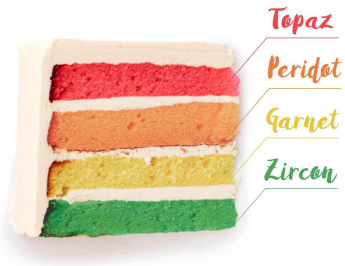

Figure 1: Fuchsia layer cake

2.1 Zircon

Zircon[3] is the lowest level of the cake and includes the part of the project we worked most directly with. It contains the micro-kernel behind Fuchsia, also called Zircon, together with a small set of userspace services, drivers, and libraries (like *libc* and *fdio*). Zircon also defines the **Fuchsia Interface Definition Language** (FIDL), which is the protocol used to create bindings between different languages, such as Dart, Go, C/C++ and Rust. The main role of this layer is to ensure the boot of the system, handle the interaction with the hardware, load and run userspace processes.

2.2 Garnet

The layer built on top of Zircon is called Garnet[4]. It contains *"device-level system services for software installation, administration, communication with remote systems and product deployment"*. Some of the modules included in Garnet are: the network service (*Escher*[5]), the graphics renderer (*Amber*),

the package management and update system. The idea behind this layer is to be able to update all the components running on the Fuchsia system, including apps and the Zircon kernel.

2.3 Peridot

The main role of Peridot[6] is to handle the modular approach of Fuchsia. In the documentation it is written: "*almost everything that exists on a Fuchsia system is stored in a Fuchsia* **package**." This includes software and system files, which, instead of being all-in-one programs, are made up of **components**.

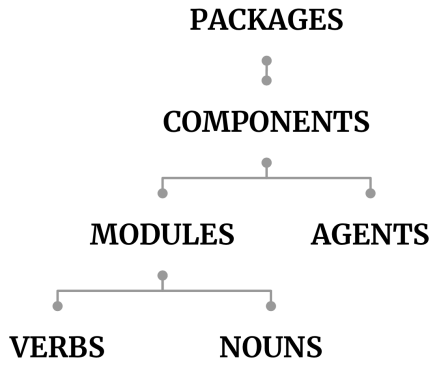

Figure 2: Fuchsia's modular approach

These components are small piece of software designed to perform specific jobs; currently, there are 2 kinds of components available: **agents** and **modules**. An agent is a component working in the background, which provides information to other components. Modules, instead, are the components working on the foreground and tagged with a specific **task**. When looking at the roles of agents and modules, it is easy to draw a parallel with services and activities in Android, which are working, respectively, in the background and foreground. However the modular approach of Fuchsia goes even further; indeed every module includes a list of action it can perform (**verbs**) and the entities it can interact with (**nouns**). In this way, if the user wants to perform a specific action, Fuchsia will find the best tool for that action by converting

it into a noun and a verb; then it will look through the modules matching the verb and filter the list on the ones that can handle the specified noun.

The Peridot layer contains Ledger, which manages and provides the data storage to each component. These data are separated from one another at component and user level: *"the data store for the particular component/user combination is private – not accessible to other apps of the same user, and not accessible to other users of the same app"* [7]. All the data stores together creates the *personal ledger* of the user, which is synchronized across different devices through the cloud. In this way, users could select an action to take on a device, while actually performing it on another device and having a distributed storage system.

2.4 Topaz

Topaz[8] is the top layer of the cake and it currently contains four major categories of software: modules, agents, shells and runners. The shells include the base shell and the user shell, while agents - as previously mentioned - are working in background and can be seen as Android's services. The runners Fuchsia is going to include are the Web, Dart, and Flutter runners.

3 The Zircon Kernel

In this section we are going to discuss some of the interesting aspects of the Fuchsia kernel.

Fuchsia's micro-kernel has the same name of the layer in which it is placed: Zircon. The micro-kernel *"provides syscalls to manage processes, threads, virtual memory, inter-process communication, waiting on object state changes and locking"*[9].

Zircon was originally a branch of **LK** (LittleKernel[10]), a micro-kernel specifically developed for small systems and typically used in embedded applications. However Zircon targets devices have less resource constraints, thus some concepts were added on top of LK, such as the concept of process. Other main differences are that Zircon is 64-bit only, while LK is 32-bit and

that Zircon has a capability-based security model, while in LK all code is trusted.

3.1 Zircon Kernel objects

Zircon is a *object-based kernel* where user mode code almost exclusively interacts with OS resources via object handles. Kernel objects do not have a notion of security and do not perform authorization checks: thus security rights are held by handles. We are going to discuss the role of the handles in the following paragraphs, but, for now, it is enough to think of a handle as a sort of an active session with a specific OS subsystem scoped to a particular resource.

The Zircon Kernel objects are divided into 6 categories. For the purpose of this report we are going to briefly analyze only 3 of them, however the documentation covers all of them in the *Zircon Kernel objects* page[11].

IPC

One of the object categories is dedicated to Inter-Process Communication, and it contains 3 types of kernel objects: channels, sockets and FIFOs. The first two objects are both means for bi-directional inter process communication, but the former can transport data and handles, while the latter can move only data. FIFOs are the classic first-in first-out interprocess queue, and their read and write operations are more efficient than sockets or channels, but there are some restrictions on the size of elements and buffers.

Tasks

This object category includes the *runnable* kernel objects: jobs, processes and threads. They all share the ability to be suspended, resumed and killed.

A Zircon job is a set of processes and child jobs. They are used to track privileges to call syscalls and perform other kernel operations, as well as track and limit the resource consumption. Every process belongs to a single job and every job (except the root one) belongs to a single parent job.

A Zircon process, instead, is defined as *"a set of instructions which will be executed by one or more threads, along with a collection of resources"*. Processes are owned by jobs and they contains handles, Virtual Memory

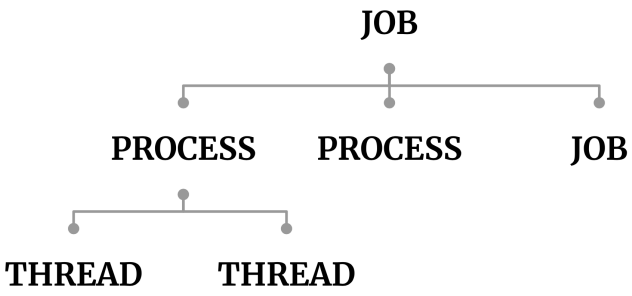

Figure 3: Hierarchical structure of Zircon tasks

Address Regions and threads. A process is created via `zx_process_create` and its execution begins with `zx_process_start`. The execution of the process is interrupted when:

- the last thread is terminated or exits
- the process calls `zx_process_exit`
- the parent job terminates the process or is destroyed

In Zircon, a thread is defined as a *"runnable computation entity"* and it is the kernel object representing a time-shared CPU execution context. Threads are associated to a process object, which provides the memory and the handles to other objects necessary for computation. A thread object is created using `zx_thread_create`, but the execution starts when `zx_thread_start` is called. It is possible to terminate a thread in these ways:

- by calling `zx_thread_exit`
- when the parent process terminates
- by calling `zx_task_kill` with the thread's handle

Memory and address space

This object category includes Virtual Memory Objects (VMO) and Virtual Memory Access Regions (VMAR).

A Virtual Memory Object represents a contiguous region of virtual memory that may be mapped into multiple address spaces. They are used in by the kernel and userspace to represent paged and physical memory as well as for sharing memory between processes and between the kernel and userspace.

VMOs are created with `zx_vmo_create` and basic I/O can be performed on them using `zx_vmo_read` and `zx_vmo_write`. The size of a VMO can be explicitly defined and will be rounded up to the next page size boundary by the kernel.

Virtual Memory Address Regions represent the allocation of an address space and they are used to map VMOs. Every process starts with a single VMAR that spans the entire address space. Each VMAR can be logically divided into any number of non-overlapping parts, each representing a child VMAR, a virtual memory mapping, or a gap. Moreover, VMARs have a hierarchical permission model for allowable mapping permissions: for example, if the root VMAR allows read, write, and executable mapping, it is possible to create a child VMAR that only allows read and write mappings. By default, all allocations of address space are randomized.

3.2 Handles

Handles are used to refer to kernel objects from the userspace and they can be transmitted to other processes over channels, by using either `zx_channel_write` or `zx_process_start` and passing the handle as the argument of the first thread into a new process. Every object may have multiple handles that refers on them, even in the same process. Typically, when the last open handle referring to a specific object is closed, the object is destroyed or put into a final and permanent state.

In addition to pointing to kernel object, a handle specifies also the actions which may be performed on the object; two handles that refer to the same object may have different rights. In the userspace, a handle is a 32bit integer (type `zx_handle_t`). `zx_handle_duplicate` and `zx_handle_replace` may be used to obtain additional handles referring to the same object as the Handle passed in, optionally with less rights. The `zx_handle_close` system call, instead, closes a given handle. In case that handle is the last one pointing to that object, the object is released.

3.3 vDSO

vDSO stands for **virtual Dynamic Shared Object** and it represents the only means of access to system calls in Zircon. It is a *dynamic shared object* because it is a shared library in the ELF format, however it is virtual because it is not loaded from an ELF file that is located in a filesystem, but the vDSO image is provided directly by the kernel, which exposes it to userspace as a *read-only Virtual Memory Object*.

Whenever a program loader sets up a process, the only way to make system calls is for the program loader to map the vDSO into the new process's address space before its first thread starts running. Therefore, each process that will launch other processes capable of making system calls must have access to the vDSO VMO.

vDSO Structure

The vDSO structure consists of consecutive segments, each containing aligned pages:

- The first segment is read-only, and includes the ELF headers and metadata for dynamic linking along with constant data private to the implementation of the vDSO.
- The second segment is executable, and it contains the vDSO code.

vDSO Enforcement

As mentioned above, the vDSO entry points are the only means to enter the kernel to use system calls. Indeed the machine-specific instructions used to enter the kernel (such as syscall on x86) are not part of the system ABI, so it is invalid for user-code to directly execute such instructions. The interface linking the kernel to the vDSO code is *"a private implementation detail"*. Considering that the vDSO is itself normal code that is executed in userspace, the kernel is responsible to handle all the possible entries into the kernel mode from the userspace.

The correct use of the vDSO is enforced by the kernel in two ways; the first one being about how the vDSO can be mapped into a process. Indeed, when `vmar_map` is called using the vDSO and requesting `ZX_VM_PERM_EXECUTE`, the kernel requires that the offset and size of the mapping match exactly the vDSO's executable segment. Moreover, only one of such mapping is allowed

and, once the valid vDSO mapping has been established in a process, it cannot be removed. Trying to map the vDSO a second time into the same process, to unmap the vDSO code, or even make an executable mapping that does not include the correct offset and size, will return `ZX_ERR_ACCESS_DENIED`. At compile time, the offset and size of the vDSO's code segment are obtained directly from the vDSO ELF file and used as constants to check the correct mapping of the kernel. When a process establish a valid mapping of the vDSO, the kernel stores the address of such process, so it can be retrieved quickly.

The second method to enforce the correct use of the vDSO consists in constraining what PC locations can be used to access the kernel. When a user thread enters the kernel for a syscall, a register indicates which of the private interface between the kernel and the vDSO is being invoked. Many of these interfaces, often called *low-level syscall*, correspond directly to the system calls in the public ABI, but others do not. Given a low-level system call, only a fixed set of PC locations in the vDSO code can invoke that call. The source code for the vDSO defines internal symbols identifying each such location. At compile time, these locations are extracted from the symbol table and used to generate the kernel code that defines a PC validity predicate for each low-level system call. Considering that the vDSO used by all user process is defined only once, these predicates check for valid, known offsets from the beginning of the vDSO code segment. While entering to the kernel for a system call, the PC location of the syscall instruction are examined on on x86 - or on other machines' instruction. The base address of the vDSO code used to call `vmar_map` is subtracted from the PC, and the resulting offset is passed to the validity predicated for the specific system call which is being invoked. In case the predicate says the PC value is invalid, the calling thread will not be allowed to proceed with the system call.

vDSO Variants

vDSO variants are an experimental feature that was proposed in the Fuchsia documentation. Currently only a proof-of-concept implementation and some simple tests are available, but the general approach would include to provide variants of the vDSO image which export only a subset of the full vDSO syscall interface. In this way, system calls that would be used by device drivers might not be included in the vDSO variant used for normal application code.

3.4 Syscalls

As mentioned above, system calls are provided by `libzircon.so`, which is a **virtual shared library** that the Zircon kernel provides to userspace, better known as the virtual Dynamic Shared Object or vDSO. The system calls are C functions of the form `zx_noun_verb` or `zx_noun_verb_direct-object`. The expected number of system call that Zircon will include is approximately 100, however some temporary syscalls are currently available, mostly for debug purposes; these temporary syscalls will be removed once *"API/ABI surface is finalized"*. Zircon syscalls are generally non-blocking, some exceptions are `wait_one`, `wait_many`, `port_wait` and `thread_sleep`.

System calls are used to make userspace code interact with kernel objects, almost exclusively using **handles**. Before executing a syscall, Zircon checks that the specified handle parameters refer to an handle that exists in the handle table of the calling process. Moreover, the kernel checks that the handle has the required rights for the requested operation.

From an access standpoint, system calls fall into three categories:

- Calls having no limitations: there are only a very few, such as `zx_clock_get` and `zx_nanosleep`; these may be called by any thread.
- Calls which take a handle as the first parameter: the specified handle denotes the object the syscall acts upon. This category includes most of the system call.
- Calls which create new objects without taking a handle: access and limitations of these calls is controlled by the job in which the calling process is contained. Two examples of these category of syscalls are `zx_event_create` and `zx_channel_create`.

The system calls are defined by `syscalls.abigen` and processed by the `abigen` tool. The complete list of system call is available on *fuchsia.googlesource.com*[12].

4 Filesystem

Fuchsia's filesystems live entirely within userspace as simple processes which implement servers. In this way they can be changed easily and their modifica-

tions don't require recompiling the kernel. The primary mode of interaction with a filesystem server is achieved using the handle primitive rather than system calls. The kernel has no knowledge about files, directories, or filesystems, so filesystem clients cannot ask the kernel for filesystem access directly.

4.1 Life of an `open`

The `open` call is a function provided by a standard library; for C/C++ programs, this will normally be declared in `unistd.h`. While on a monolithic kernel an open would be a lightweight shim around a system call, the Zircon kernel intentionally has no such system call. Indeed clients access filesystems through channels.

The standard library is responsible for taking a handle and making them appear like file descriptors, so that the *"file descriptor table"* is a notion that exists within a client process. A library called *fdio* is responsible for providing a unified interface to a variety of resources, such as files, sockets, services, pipes, and more. This layer defines a group of functions, such as *read*, *write*, *open*, *close*, *seek*... that may be used on file descriptors. Each supported protocol is responsible for providing the required client-side code to interpret the specifics of their interaction. For example, sockets provide multiple handles to clients: one acting for data flow, and one acting as a control plane, while files typically use only a single channel.

An `open` call go through the standard library, acting on the current working directory *fdio object*, which transformed the request into a FIDL message, which is sent to the server using the `zx_channel_write` system call. The client can optionally wait for the server's response using `zx_object_wait_one`, or continue processing asynchronously. Either way, a channel has been created, where one end lives with the client, and the other end is transmitted to the server.

Once the filesystem server properly found, opened, and initialized I/O state for the specified file, it sends back a *success* FIDL message over the channel. This would be read by the client, identifying that the call completed successfully. At this point, the client could create an *fdio object* representing the handle to the given file and reference it with an entry in a file descriptor table.

5 Contribution

In this section are discussed the various phases of the work we performed during the Semester Project.

5.1 Insert our program in Fuchsia

The step following the compilation of Fuchsia was to insert a custom program inside the OS. Since we decided to focus on the *Zircon* layer, we started by looking there for the location of some programs we knew being part of that layer only. The source code of some of them (like *vmaps.c*, *top.c*, *memgraph.cpp*) can be found in the following directory: `\$FUCHSIA_ROOT/zircon/system/uapp/psutils`.

In this folder we created our custom program that we wanted to add in Zircon and we called it `evil.c`. At this point, we needed to tell Zircon to build this file. In order to do that, we modified the `rules.mk` file in the same folder. This is the code we added:

```
include make/module.mk
MODULE := $(LOCAL_DIR).evil
MODULE_TYPE := userapp
MODULE_GROUP := core
MODULE_SRCS += $(LOCAL_DIR)/evil.c
MODULE_NAME := evil
MODULE_LIBS := \
  system/ulib/fdio \
  system/ulib/zircon \
  system/ulib/c
MODULE_STATIC_LIBS := \
  system/ulib/pretty \
  system/ulib/task-utils
```

After that, we built the project again, using with the `fx full-build` command and run it. Finally, we were able to execute our custom `evil` program inside Fuchsia. We added another program to be compiled inside the OS, called `create-proc`.

We decided to focus our work on these two programs: we used the `create-proc` to create a new process inside Fuchsia (so we knew the content of its memory) and `evil` to tamper with the OS and its system calls. The content of the two custom programs are going to be discussed in the following paragraphs.

5.2 `create-proc`

In order to create a new process inside Fuchsia, we followed a blog post called *"Admiring the Zircon Part 1: Understanding Minimal Process Creation"*[13]. This requires few steps:

1) An empty process is created using the `zx_process_create` function, which will be executed with the following arguments:
   - The handle of a parent job. The new process is created as child of this job which, in our case, was the *default job* in Fuchsia. We obtained its handle using the `zx_job_default` function.
   - The name of the process.
   - The size of the name.
   - An integer for options (`0` in our case). This function returns two handles: one to the new process, and a handle to the root of its address space.

Two Virtual Memory Objects (VMOs) are required to proceed, one for the stack space, the other one for the code space.

2) It is possible to create two VMOs using the `zx_vmo_create` function. The following arguments are required:
   - The size of the VMO
   - An integer for options (`0` in our case) This function returns an handle pointing to the newly created VMO.

To make the new process not being killed after the `create-proc` program is terminated, it is required to execute a *jump loop* instruction, which will keep it alive until another program closes it.

3) We can create array containing the jump loop in shell code (i.e. [`0xeb`, `0xfe`]) and write it into the first VMO using the `zx_vmo_write` function. The following arguments are required:

- The handle of the VMO object
- A pointer to an array containing the code
- An integer for options (`0` in our case)
- The size of the code

After the creation of the two VMOs we need to map them in the Virtual Memory Address Region (VMAR).

4) In order to map the VMOs to the VMAR we can use the function `zx_vmar_map`. In this function we need to pass also some permissions, which will be *READ* and *WRITE* for the stack space and *READ* and *EXECUTE* for the code space. This function returns a pointer to where the VMO is mapped in the given address region. In our case, we passed the handle pointing to the root of our process' address spaces.

5) In order to start our process, we needed to create a new thread as a requirement of the `zx_process_start` function as well as a new event, which is an object that is *signalable*.

6) Finally, to start the process, we used the function `zx_process_start`. The following arguments are required:

- The handle of a process object
- The handle of a thread object
- The pointer to the address containing the code
- The pointer to the address for the stack
- The handle of an event object
- An integer for options (`0` in our case)

We used this program to create a new process in Fuchsia called `my_proc` for which we knew exactly its content. Therefore, we proceeded to find a way to dump its content and the content of other processes, so we started working on the `evil` program.

5.3 Get the privileges associated to a given handle

As mentioned previously, the handles, in addition to pointing to a kernel object, specify certain permissions that indicate the actions that the calling process can perform against the object to which the handle refers. The permissions are defined in `<zircon/rights.h>` and their meaning can be found

in the fuchsia documentation [14]. It is interesting to note that among the *"basics"* permissions (`ZX_RIGHTS_BASIC`) there is also `ZX_RIGHT_INSPECT`, which allows us to obtain information about a given kernel object. Currently the Fuchsia source code does not include any function that allows us to see what permissions are associated with a handle. For this reason we decided to take advantage of the `ZX_RIGHT_INSPECT` permission to call `zx_object_get_info` and get information about a given handle, including the rights associated to it. However, they are not in an human readable format and therefore we had to use `<zircon/rights.h>` to map the id associated to the rights to their name. In the end, we obtained a function which, given a handle having at least the basics rights, is able to print all the privileges associated to that handle.

5.4 Get the root job handle

While looking for useful information, we found a video[15][16] tackling the problem of security flaws in Fuchsia. As part of the presentation, the speaker discussed a flaw in the current implementation of the kernel, which allows a program who is able to get the *"root job"* to obtain all its children recursively, by calling the function `object_get_child`. This means that all the handles of running processes and jobs could be obtained with a handle to the root job.

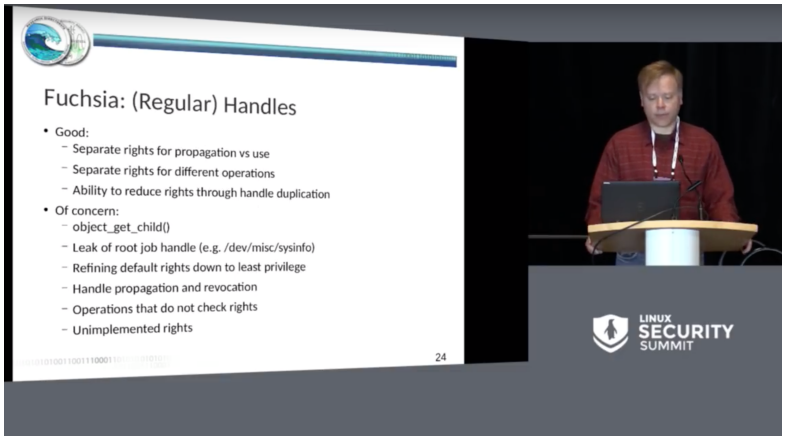

Figure 4: Part of the video addressing some concerns about handles

We started looking for the function to obtain the root job handle and (a lot of `grep` later) we found `ioctl_sysinfo_get_root_job`. Unfortunately the function does not work without the library contained in `<zircon/device/sysinfo.h>`, thus we had to include it. Using this function, we obtained a piece of code that retrieves a handle to the root job for the calling process. We were also able to verify that the handle of the root job, similarly to other handles, is different for every calling process.

We tried our function to retrieve the privileges associated to the root job handle and we found that we could call on it `zx_object_get_info`. This could allow us to navigate the tree of running processes and, using `object_get_child`, obtain a handle to every Fuchsia process.

5.5 `evil`

The `evil` program can perform 3 actions:

- Get the privileges associated to a process given its `koid`
- Kill a process given its `koid`
- Dump the memory of a process given a `koid`

To do each of the mentioned task, the program has to navigate the tree of running jobs and retrieve the corresponding handle starting from the root job, which we retrieve using the following code:

```
int fd = open("/dev/misc/sysinfo", O_RDWR);
if (fd < 0) {
    fprintf(stderr, "ERROR: Cannot open sysinfo: %s (%d)\n",
            strerror(errno), errno);
    return ZX_ERR_NOT_FOUND;
}

ssize_t n = ioctl_sysinfo_get_root_job(fd, root_job);
close(fd);
```

We copied it directly from the source code of Fuchsia and modified it so that we can retrieve the root job. Then, we used its handle to recursively get all its children (both jobs and processes) in the following way:

1) As an initial step, all the jobs and processes which are directly children of the root job are gathered using the `zx_object_get_info` function. Unfortunately, this function does not work recursively, so we had to do it on our own. Its requirements are:
   - The handle for which info are retrieved
   - A topic, which defines what this function will return In our case, we used the `ZX_INFO_JOB_PROCESSES` and `ZX_INFO_JOB_CHILDREN` to get, respectively, the processes and job children.

This function returns an array of `koids` (ids of processes and jobs in Fuchsia) and from these, we had to get the corresponding handles. This step is required because we needed to make a recursive function, which instead of the root job handle, takes one of its children and recursively gathers all the others.

2) The function `zx_object_get_child` returns the handle of a child, given the parent handle and the `koid` representing it. In this way, for each child of the root job we could get the corresponding handle and recursively gather all the currently running processes and jobs.

After that, we wrapped these functions into a single one that, given a `koid` as input, returns its corresponding handle. Now we are going to briefly describe each of the tasks our program is doing.

Get privileges associated to a given handle

We already had a function which is able to retrieve the privileges the calling process have over a given handle. The description of the function is discussed in the previous paragraph.

Kill a process

Once we have a handle pointing to a process/job and having `ZX_RIGHT_DESTROY`, we can kill the pointed object calling the function `zx_task_kill`, which only requires the handle of the runnable object.

Dump memory of a process

To dump the memory of a process we need to get where its memory is mapped. Again, we called the `zx_object_get_info` function, with the `ZX_INFO_PROCESS_MAPS` topic. This time, the result is an array of `zx_info_maps_t` objects, which, according to the documentation, is *"a depth-first pre-order walk of the target process's Aspace/VMAR/Mapping*

tree". For each element of this array we got the `base_address` and we read its associated memory with the function `zx_process_read_memory`. We check the status of these calls and if it is successful we print all the memory of a buffer using a function that dumps it in a more human-readable form. The length of the buffer (which by default is set to 100) can be changed via an option when running the `evil` program.

We tested our program using the process we created with `create-proc` and we were successfully able to see the *jump loop* code which is executing.

5.6 Shellcode execution

During the project we decided to take a look also into the syscall checks performed by the vDSO. We took the code of `create-proc` as a starting point to play with syscalls using shellcode. Since the Zircon kernel is accepting system calls which are coming only from the vDSO address space, we decided to get its base address and perform a `nanosleep` call (which makes the process hang for 5 seconds) jumping at the right address in memory. When a process crashes in Fuchsia, some information are displayed on screen, including the vDSO address. This is randomized; indeed, each time the same program crashes, the address of the vDSO changes.

We used the function `dlsym` to obtain the address of the symbol `zx_deadline_after`, a Zircon syscall. We printed it in the console, and we could obtain the offset between that function and the beginning of the vDSO by subtracting it to the vDSO base address which is printed when a process crashes. In this way we could obtain the vDSO base address dynamically from the code.

After that, we used the following code for the shellcode execution:

```
asm("movq %1, %%rdi;\*/
     movl $3, %%eax;\*/
     jmp *%2;"
    : "=a" (res)
    : "d" (tmp), "r" (sys_location));
```

This piece of code moves a value (`tmp` in our case, which is equal to 1) to the `rdi` register; this step is required for the `nanosleep` to work. Then, we

move 3, which is the number of the syscall we want to call, to the `eax` register. Finally, we jump to `sys_location`, which is the base address of the vDSO plus the offset of the `nanosleep` system call that we obtained using IDA.

Everything works and the process hangs for 5 seconds before quitting. Therefore, we decided to move on in two different ways:

- Jump to another syscall and check if the process still hangs
- Change the shellcode and check if it works with an arbitrary syscall

Unfortunately, both these directions resulted in a policy error which we were not able to solve. However, at this point of the project we discovered the bug related to leakage of the root job handle, thus we decided to focus on that direction.

5.7 Not really interesting tests on files

In the very last part of the project we tried to address some issues related to file creation. Since there is no `open` in Zircon, we started from obtaining the code to create a file.

The following piece of code allow us to create a `peppa.txt` file:

```
fbl::unique_fd fd(open("/tmp", O_DIRECTORY | O_RDONLY));
// Create a file
const char* filename = "peppa.txt";
fd.reset(openat(fd.get(), filename, O_CREAT | O_RDWR));
fd.get();
const char* data = "malicious payload";
ssize_t datalen = strlen(data);
write(fd.get(), data, datalen);
fd.reset();
```

However, this is the implementation on an higher layer: we are not creating a channel to contact the filesystem server and obtain the handle of the file. We tried to figure out how to perform the same action on a lower level, but we were not able to do so. The main issue was related to the creation of the channel we have to pass to the *filesystem server*.

6 Conclusion

In this report we discuss the results of the exploratory semester project about Google Fuchsia. We started with a general description of the 4 levels the *Fuchsia cake*, and continued with an more specific analysis on the Zircon kernel. In particular, the third section is focused on kernel objects, handles, the vDSO and system calls. In the fourth section we reported a brief study of Fuchsia filesystems and an example of how an *open* call is handled.

In the last section we illustrated our contribution, which can be summarized in two programs that can be called directly from Fuchsia. The first one (`create-proc`) allows the creation of a new process. The second one (`evil`) exploit the leakage of the root job handle with high level rights, to recursively navigate the tree of processes. Given a `koid`, `evil` is able to kill the respective process, dump its memory and return the associated rights we have with respect to that process.

An important part of the semester project was dedicated to reading and understanding the Fuchsia documentation, which, to date, is still incomplete and subject to frequent changes. The absence of documentation and material has slowed down part of the objectives, but has also contributed positively to the originality of the project and has allowed us to enrich our knowledge with new and state-of-the-art concepts.

References

[1] “Git repositories on fuchsia.” https://fuchsia.googlesource.com.

[2] “Fuchsia documentation.” https://fuchsia.googlesource.com/docs/+/HEAD/development/source_code/layers.md.

[3] “Zircon Layer.” https://fuchsia.googlesource.com/zircon/+/master/README.md.

[4] “Garnet Layer.” https://fuchsia.googlesource.com/garnet/+/master/README.md.

[5] “Escher.” https://fuchsia.googlesource.com/garnet/+/master/public/lib/escher/.

[6] “Peridot Layer.” https://fuchsia.googlesource.com/peridot/+/master/README.md.

[7] “Ledger.” https://fuchsia.googlesource.com/peridot/+/master/docs/ledger/README.md.

[8] “Topaz Layer.” https://fuchsia.googlesource.com/topaz/+/master/README.md.

[9] “Zircon Kernel.” https://fuchsia.googlesource.com/zircon.

[10] “LK.” https://github.com/littlekernel/lk.

[11] “Zircon Kernel objects.” https://fuchsia.googlesource.com/zircon/+/HEAD/docs/objects.md.

[12] “Zircon System calls.” https://fuchsia.googlesource.com/zircon/+/HEAD/docs/syscalls.md.

[13] “Admiring the Zircon Part 1: Understanding Minimal Process Creation.” https://depletionmode.com/zircon-process.html.

[14] “Rights.” https://fuchsia.googlesource.com/zircon/+/HEAD/docs/rights.md.

[15] J. Carter, “Security in Zephyr and Fuchsia.” https://www.youtube.com/watch?v=Jov4dTnjm2o.

[16] E. Brown, "Redefining Security Technology in Zephyr and Fuchsia." https://www.linux.com/blog/2018/9/redefining-security-technology-zephyr-and-fuchsia.