EURECOM, Semester Project 2018-2019

Exploratory Project
Google Fuchsia

Supervisors:

Davide Balzarotti
Yanick Fratantonio
Dario Nisi
Fabio Pagani
Andrea Possemato

Students:

Andrea Palmieri
Paolo Prem

What is Fuchsia?

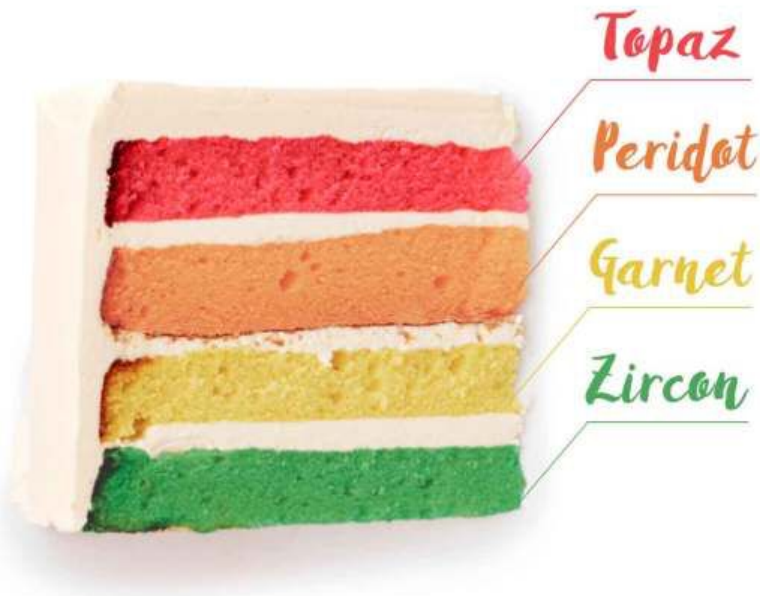
“A modular, capability-based operating system”

Modular: “*Individual parts of the platform and of applications can be developed, distributed, and updated separately by different parties.*”

Capability-based: “*A process must possess a capability (a valid **handle**) in order to perform actions upon the object to which it refers.*”

Fuchsia Documentation, *“Fuchsia is Not Linux”*: link

Fuchsia layer cake



Layer 1: Zircon

- Micro-kernel
- Userspace services, drivers and libraries (*libc* and *fdio*)
- Boots the system
- Defines Fuchsia Interface Definition Language (FIDL)
- Manages interaction with hardware
- Runs userspace programs

Fuchsia Documentation, *“Zircon Layer”*: link

Layer 2: Garnet

- Low-level system services
    - software installation
    - administration
    - communication with remote systems
- Modules:
    - **Escher**, the network service
    - **Amber**, the graphics renderer

- Update all the components of the system

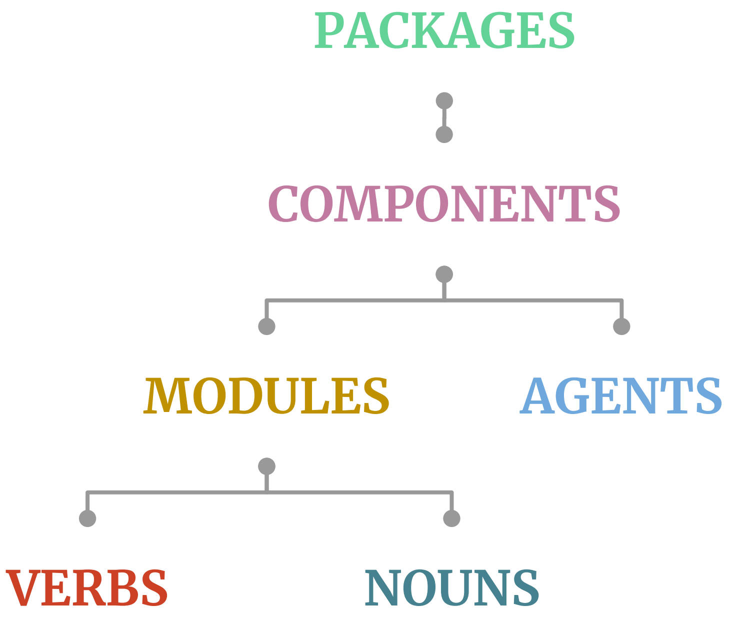
Fuchsia Documentation, *“Garnet Layer”*: link

Fuchsia's modular approach

Packages: *"almost everything [...] is stored in a package"*
Components: small and specific piece of software; currently 2 kinds of components:

- **Agents:** working in the background
- **Modules:** working on the foreground

Verbs: list of actions a module can perform
Nouns: entities a module can interact with



Layer 3: Peridot

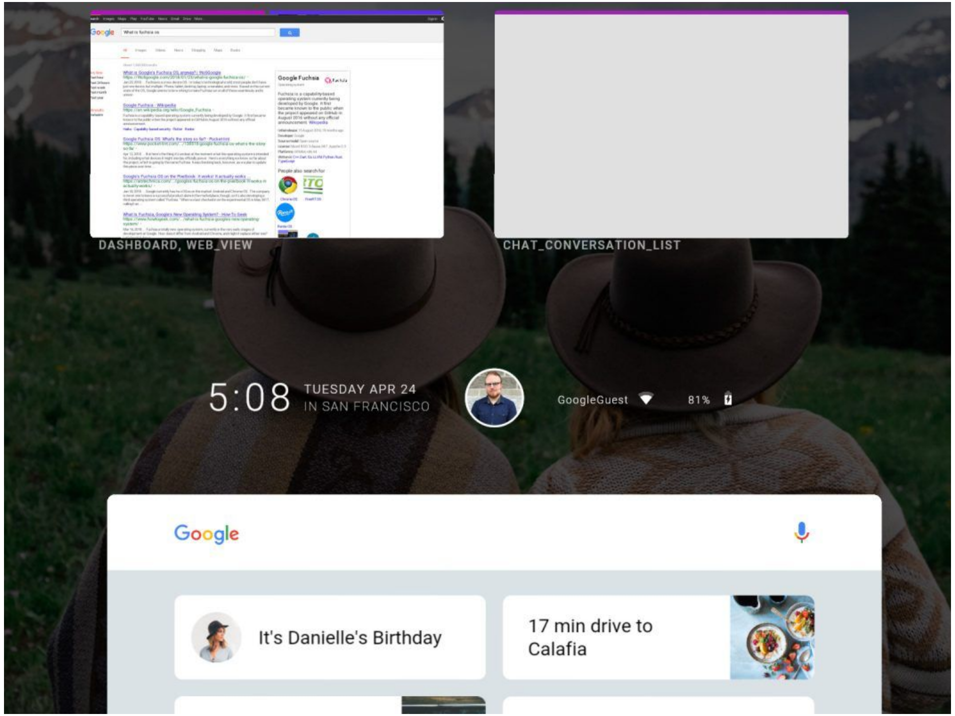
- Handle the modular approach of Fuchsia
- **Ledger**
  - manage and provide data storage to components
  - data are separated between components
  - data stores together creates the **personal ledger** of the user

Fuchsia Documentation, *“Peridot Layer”*: link

Layer 4: Topaz

Four major categories of software:

- modules: like email and calendar
- agents: working in background
- shells: base and user shell
- runners: Web, Dart and Flutter



Fuchsia Documentation, *"Topaz Layer"*: link

Use case example

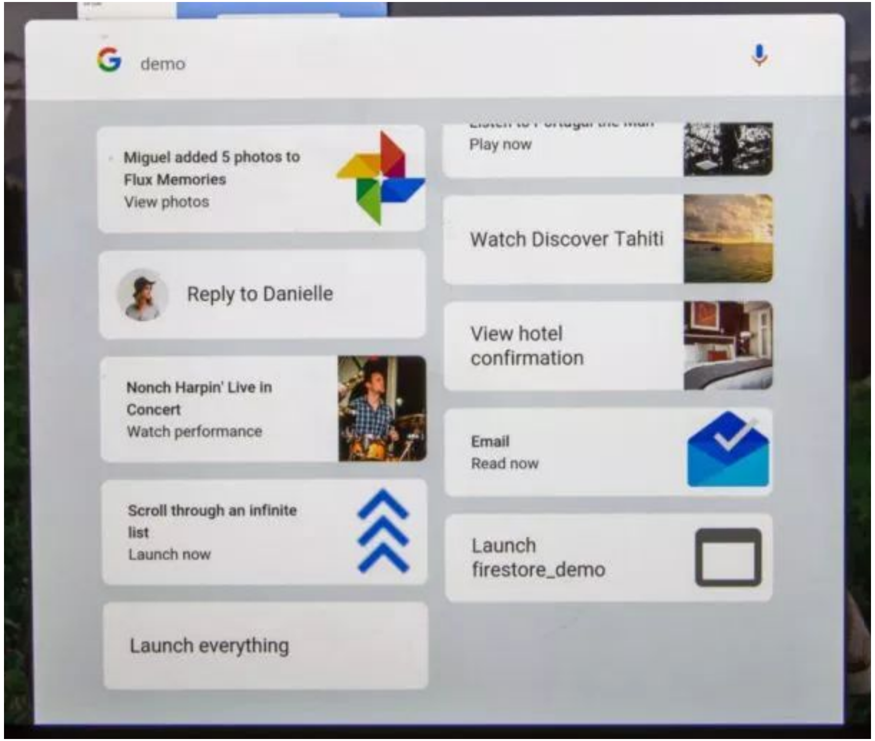
I want to upload a video:

- type “upload video” in the search bar
- Fuchsia finds the best module to do that (for example Youtube)
- you can use it directly without installing it

No more all-in-one apps

- modular-based approach

Like Android “instant apps” on steroids



Fuchsia Documentation, *“Topaz Layer”*: link

Zircon kernel

- Object-based micro kernel
- originally a fork of *LittleKernel*
- *"provides syscalls to manage processes, threads, virtual memory, inter-process communication, waiting on object state changes and locking"*
- Capability-based security model, user code interacts with OS resources using *handles*
- 64-bit only

Fuchsia Documentation, *"Zircon"*: link

Zircon kernel objects

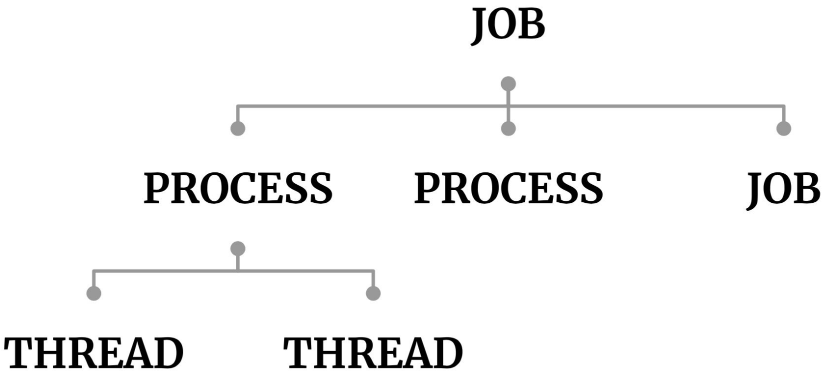
- IPC
  - **channel**: bi-directional IPC, can transport data and handles
  - **socket**: bi-directional IPC, can move only data
  - **FIFO**: first-in first-out interprocess queue
- Memory and address space
  - **VMO:** a contiguous region of virtual memory that may be mapped into multiple address spaces
  - **VMAR**: the allocation of an address space; used to map VMOs

Fuchsia Documentation, *“Zircon kernel objects”*: link

Zircon kernel objects: tasks

Tasks (*runnable* kernel objects):

- **job:** used to track privileges to call syscalls; track and limit the resource consumption
- **process**: set of instructions which will be executed by threads; include also some resources (handles and VMAR)
- **thread:** *"runnable computation entity"*; are associated to a process, which provides memory and handles necessary for computat



Fuchsia Documentation, *"Zircon kernel objects"*: [link]

Handles

- Security rights are held by *handles* (kernel objects do not perform authorization checks)
- 32bit integers (types `zx_handle_t`)
- Used to refer to kernel objects from the userspace
- Specifies also the actions which may be performed on the object
- They can be transmitted to other processes over channels (`zx_channel_write`)
- Every object may have multiple handles that refers on them, even in the same process
- Two handles that refer to the same object may have different rights

Fuchsia Documentation, *"Zircon Handles"*: link

vDSO

The **virtual Dynamic Shared Object** represents the **only** means of access to syscalls in Zircon. It is a shared library in the ELF format provided directly by the kernel, which exposes it to userspace as a *read-only* VMO.

vDSO structure:

- The first segment is read-only; it includes the ELF headers and metadata
- The second segment is executable, and it contains the vDSO code

vDSO variants (experimental feature): variants of the vDSO image which export only a subset of the full vDSO syscall interface (ex. device drivers syscall not included in the vDSO variant for normal application code)

Fuchsia Documentation, *"Zircon vDSO"*: link

vDSO Enforcement

Correct use of the vDSO is enforced by the kernel in two ways:

- vDSO mapping into a process: when `vmar_map` is called using the vDSO, offset and size of the mapping have to match exactly the vDSO's executable segment
- 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
  - Given one of these interfaces, only a fixed set of PC locations can invoke that call

Fuchsia Documentation, *"Zircon vDSO"*: link

System calls

- System calls are C functions of the form `zx_noun_verb` or `zx_noun_verb_direct-object`
- The expected number of system call is *approximately* 100
- Zircon syscalls are generally non-blocking (exceptions are `wait_one`, `wait_many`...)
- Syscalls are used to make userspace code interact with kernel objects (using handles):
  - Before the syscall, Zircon checks that the specified handle is valid in the context of the calling process
  - The kernel checks also if the handle has the rights for the requested operation

Fuchsia Documentation, *"Zircon system calls"*: link

Rights

- Associated with handles
- Privileges to perform actions either the associated handle or the object associated with the handle
- `<zircon/rights.h>` header defines default rights for each object type
- `ZX_RIGHTS_BASIC`
    - Allow primitive manipulation (*ZX_RIGHT_DUPLICATE*, *ZX_RIGHT_TRANSFER*, *ZX_RIGHT_WAIT*, and *ZX_RIGHT_INSPECT*)

Right	Conferred Privileges
ZX_RIGHT_DUPLICATE	Allows handle duplication via *zx_handle_duplicate*
ZX_RIGHT_TRANSFER	Allows handle transfer via *zx_channel_write*
ZX_RIGHT_READ	**TO BE REMOVED** Allows inspection of object state
	Allows reading of data from containers (channels, sockets, VM objects, etc)
	Allows mapping as readable if **ZX_RIGHT_MAP** is also present
ZX_RIGHT_WRITE	**TO BE REMOVED** Allows modification of object state
	Allows writing of data to containers (channels, sockets, VM objects, etc)
	Allows mapping as writeable if **ZX_RIGHT_MAP** is also present
ZX_RIGHT_EXECUTE	Allows mapping as executable if **ZX_RIGHT_MAP** is also present
ZX_RIGHT_MAP	Allows mapping of a VM object into an address space.
ZX_RIGHT_GET_PROPERTY	Allows property inspection via *zx_object_get_property*
ZX_RIGHT_SET_PROPERTY	Allows property modification via *zx_object_set_property*
ZX_RIGHT_ENUMERATE	Allows enumerating child objects via *zx_object_get_info* and *zx_object_get_child*
ZX_RIGHT_DESTROY	Allows termination of task objects via *zx_task_kill*
ZX_RIGHT_SET_POLICY	Allows policy modification via *zx_job_set_policy*
ZX_RIGHT_GET_POLICY	Allows policy inspection via *zx_job_get_policy*
ZX_RIGHT_SIGNAL	Allows use of *zx_object_signal*
ZX_RIGHT_SIGNAL_PEER	Allows use of *zx_object_signal_peer*
ZX_RIGHT_WAIT	Allows use of *zx_object_wait_one*, *zx_object_wait_many*, and other waiting primitives
ZX_RIGHT_INSPECT	Allows inspection via *zx_object_get_info*

Fuchsia Documentation, *"Rights"*: link

Insert a custom program

Location of other programs (like *vmaps.c*, *top.c*, *memgraph.cpp*)

“$FUCHSIA_ROOT/zircon/system/uapp/psutils”

Insert a custom program

I. Create *evil.c*
II. Modify *`rules.mk`*
III. Re-build fuchsia

```
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
```

Create a new process - *create-proc.c*

I. Create process
II. Create two Virtual Memory Objects (VMOs)
III. Write code in VMO
IV. Map VMOs in Virtual Memory Address Region (VMAR)
V. Create a new thread
VI. Start the process

Blog post, *"Admiring the Zircon Part 1"*: link

Create a new process - 1) create process

zx_process_create:

- Creates empty process
- Requirements:
  - handle of parent job
  - process name
  - size of the name
- Returns:
  - handle of new process

Create a new process - 2) create VMOs

zx_vmo_create:

- Creates empty VMO
- Requirements:
  - size
- Returns:
  - handle of VMO

Two VMOs:

- stack space
- code space

Create a new process - 3) write code in VMO

`zx_vmo_write`:

- Writes shellcode in VMO
- Requirements:
  - handle of VMO
  - array with code
  - size of the code

```
uint8_t code[] = {0xeb, 0xfe};
```

Create a new process - 4) map VMOs in VMAR

zx_vmar_map:

- Maps the VMO in VMAR
- Requirements:
  - permissions for the VMOs
- Returns:
  - pointer to the address in memory

stack space: *READ,WRITE*
code space: *READ,EXECUTE*

Create a new process - 5) create thread

`zx_thread_create`:

- Creates a new thread to execute the process
- Requirements:
  - handle of a process
  - name of the thread
  - size of the name

Create a new process - 6) start the process

`zx_process_start`:

- Start the process
- Requirements:
  - handle of a process
  - handle of a thread
  - pointer to the address containing the code
  - pointer to the address for the stack

Find flaws/vulnerabilities - *evil.c*

According to link, the leakage of the root job gives access to all running processes

Our approach:

1) Search for references in the code
2) Found *`ioctl_sysinfo_get_root_job`* function
3) Included *`<zircon/device/sysinfo.h>`* library
4) Called the function
5) BAAM!! We have the root job

Evil

Three main options:

- Kill a process given a koid
- Print the rights of a process given a koid
- Dump the memory of a process given a koid

Evil - main issue

- Start from root job handle
- Get tree of running processes
- Get handle of a process/job given a koid as input

Evil - get handle from koid, step 1

zx_object_get_info:

Topics: *`ZX_INFO_JOB_PROCESSES`*
`ZX_INFO_JOB_CHILDREN`

- Given these topics, returns jobs and processes directly children of a job handle
- Requirements:
  - handle of a job
  - topics defining the returning objects
- Returns:
  - array of jobs/processes koids (NOT handles)

Evil - get handle from koid, step 2

`zx_object_get_child`:

- Given a parent handle and the koid of one of its child, returns its handle
- Requirements:
  - handle of a parent
  - koid of a child
- Returns:
  - handle of the child

Evil - get handle from koid, step 3

- Wrapped everything into a recursive function
- Given a koid (command line input) returns its corresponding handle
- We can get the handle of any running process/job

Evil - option 1: kill a process

`zx_task_kill`:

- Kill a process/job given its handle
- Requirements:
    - handle of a process/job

Evil - option 2: get rights of a process

`zx_object_get_info`:

Topics: *`ZX_INFO_HANDLE_BASIC`*

- Given this topic, returns *`zx_info_basic_t`* object
  - It contains useful information, including rights
- They are provided as numbers
- We created a function that prints the corresponding string defining the rights

Evil - option 3: dump memory of a process

zx_object_get_info:

Topics: *ZX_INFO_PROCESS_MAPS*

- Given this topic, returns array of *zx_info_maps_t* objects
    - Each object contain a *base_address* value
- For each value, we printed the corresponding memory with `zx_process_read_memory` function

Shellcode execution

Idea:

- get address of vDSO
- perform a jump to a syscall there
- run shellcode

Shellcode execution - get vDSO address

1) Use `dlsym` function to get the address of the symbol `zx_deadline_after`
2) Print it and make the program crash -> get vDSO address
3) Subtract the two addresses and get an offset
4) Subtract this offset at runtime from the first address

Shellcode execution

Jump to the `nanosleep` function and execute shellcode

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

Shellcode execution

We tried two different things:

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

THE END

Filesystems

- Fuchsia's filesystems: userspace processes implementing servers
- The kernel has no knowledge about files, directories or filesystems --> filesystem clients cannot ask the kernel for filesystem access directly
- We can interact with them just like any other processes (using *channels* and *handles*)
- Current filesystems:
    - MemFS
    - MinFS
    - Blobfs

    …

Fuchsia Documentation, *"Filesystems"*: link

Life of an ‘open’

I. ‘`open`’ goes through the standard library
II. `fdio` transforms the request into a FIDL message
III. the message is sent to the filesystem server with `zx_channel_write`
IV. the client waits for the server’s response using `zx_object_wait_one` (or continue processing asynchronously). Either way, a channel has been created (one end for the client, the other for the server)
V. Once the server properly found, opened, and initialized I/O state for the given file, it sends back a “success” FIDL description object.
VI. Once the client receive the message, he can create an *fdio object* representing the handle of the file

Fuchsia Documentation, *“Life of an ‘open’”*: link

Kinda-working ‘create *peppa.txt*’

```
fbl::unique_fd fd(open("/tmp", O_DIRECTORY | O_RDONLY));
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();
```

- This is in the higher level (not creating the channel to contact the filesystem server)
- We were not able to communicate with the filesystem server with a *channel*