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

1 Introduction

Fuchsia is a new operating system currently being developed by Google. The project appeared on GitHub in August 2016, but was removed in the end of 2018; and it is currently hosted at *fuchsia.googlesource.com*[1]. Although the development of the project has been under way for more than two years, Google has not officially announced it yet. While the other operating system 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 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 official documentation, Fuchsia is defined as a *"modular, capability-based operatig system"*[2]. The **modular** term of the definition refers to the fact that definition suggests that the various 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 process to updates 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*.

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.

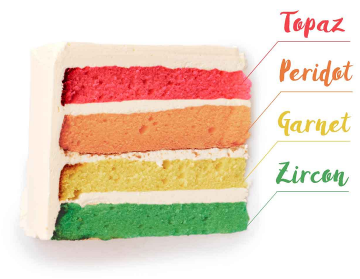

Figure 1: Fuchsia layer cake

2.1 Zircon

Zircon[3] is the lowest level of the project and includes the part of the project we worked with most directly. 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 between 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**.

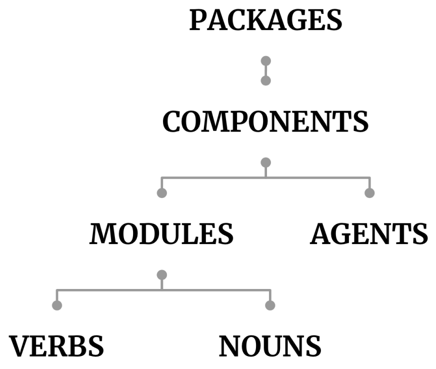

Figure 2: Fuchsia's modular approach

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. The 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 the services and activity in Android, which are working in the background and foreground, respectively. However the modular approach of Fuchsia goes even further: 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]. These 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. Modules, as mentioned above, are close to the Android apps; some examples of modules are calendar and email. Shells include the base shell and the user shell, while agents - as said before as well - 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. The role of the Flutter runner is particularly important for Fuchsia, since it would allow developers "*to build beautiful native apps on iOS and Android from a single codebase*"[9].

3 The Zircon Kernel

The microkernel used for Fuchsia 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 (via futexes)"*[10].

TODO The kernel manages a number of different types of Objects. Those which are accessible directly via system calls are C++ classes which implement the Dispatcher interface. These are implemented in kernel/object. Many are self-contained higher-level Objects. Some wrap lower-level lk primitives.

Zircon was originally a branch of **LK** (LittleKernel[11]), another kernel developed at Google for small systems typically used in embedded applications. However, it has to be noted that Zircon targets devices having less resource constraints, thus some concepts were added on top of LK, such as the concept of process, which is not present in LK, even though *"a Zircon process is made of LK-level constructs such as LK's `thread_d`"*[12]. 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 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 in a sense that is a shared library in the ELF format, however it is virtual because it's not loaded from an ELF file that is located in a filesystem, but the vDSO image is provided directly by the kernel.

HERE

The vDSO uses a simplified layout that has no writable segment and requires no dynamic relocations. This makes it easier to use the system call ABI without implementing a general-purpose ELF loader and full ELF dynamic linking semantics.

ELF symbol names are the same as C identifiers with external linkage. Each system call corresponds to an ELF symbol in the vDSO, and has the ABI of a C function. The vDSO functions use only the basic machine-specific C calling conventions governing the use of machine registers and the stack, which is common across many systems that use ELF, such as Linux and all the BSD variants. They do not rely on complex features such as ELF Thread-Local Storage, nor on Fuchsia-specific ABI elements such as the SafeStack unsafe stack pointer.

vDSO Unwind Information

The vDSO has an ELF program header of type PT_GNU_EH_FRAME. This points to unwind information in the GNU .eh_frame format, which is a close relative of the standard DWARF Call Frame Information format. This information makes it possible to recover the register values from call frames in the vDSO code, so that a complete stack trace can be reconstructed from any thread's register state with a PC value inside the vDSO code. These formats and their use are just the same in the vDSO as they are in any normal ELF shared library on Fuchsia or other systems using common GNU ELF extensions, such as Linux and all the BSD variants.

vDSO Build ID

The vDSO has an ELF Build ID, as other ELF shared libraries and executables built with common GNU extensions do. The Build ID is a unique bit string that identifies a specific build of that binary. This is stored in ELF note format, pointed to by an ELF program header of type PT_NOTE. The payload of the note with name "GNU" and type NT_GNU_BUILD_ID is a sequence of bytes that constitutes the Build ID.

One main use of Build IDs is to associate binaries with their debugging information and the source code they were built from. The vDSO binary is innately tied to (and embedded within) the kernel binary and includes information specific to each kernel build, so the Build ID of the vDSO distinguishes kernels as well.

abigen tool

The abigen tool generates both C/C++ function declarations that form the public system call API, and some C++ and assembly code used in the implementation of the vDSO. Both the public API and the private interface between the kernel and the vDSO code are specified by <zircon/syscalls.abigen>, which is the input to abigen.

The syscall entries in syscalls.abigen fall into the following groups, distinguished by the presence of attributes after the system call name:

Entries with neither vdsocall nor internal are the simple cases (which are the majority of the system calls) where the public API and the private API are exactly the same. These are implemented entirely by generated code. The public API functions have names prefixed by *zx* and zx_ (aliases).

vdsocall entries are simply declarations for the public API. These functions are implemented by normal, hand-written C++ code found in system/ulib/zircon/. Those source files #include "private.h" and then define the C++ function for the system call with its name prefixed by *zx*. Finally, they use the VDSO_INTERFACE_FUNCTION macro on the system call's name prefixed by zx_ (no leading underscore). This implementation code can call the C++ function for any other system call entry (whether a public generated call, a public hand-written vdsocall, or an internal generated call), but must use its private entry point alias, which has the VDSO_zx_ prefix. Otherwise the code is normal (minimal) C++, but must be stateless and reentrant (use only its stack and registers).

internal entries are declarations of a private API used only by the vDSO implementation code to enter the kernel (i.e., by other functions implementing vdsocall system calls). These produce functions in the vDSO implementation with the same C signature that would be declared in the public API given the signature of the system call entry. However, instead of being named with the *zx* and zx_ prefixes, these are available only via #include “private.h” with VDSO_zx_ prefixes.

Read-Only Dynamic Shared Object Layout

The vDSO is a normal ELF shared library and can be treated like any other. But it’s intentionally kept to a small subset of what an ELF shared library in general is allowed to do. This has several benefits:

Mapping the ELF image into a process is straightforward and does not involve any complex corner cases of general support for ELF PT_LOAD program headers. The vDSO’s layout can be handled by special-case code with no loops that reads only a few values from ELF headers. Using the vDSO does not require full-featured ELF dynamic linking. In particular, the vDSO has no dynamic relocations. Mapping in the ELF PT_LOAD segments is the only setup that needs to be done. The vDSO code is stateless and reentrant. It refers only to the registers and stack with which it’s called. This makes it usable in a wide variety of contexts with minimal constraints on how user code organizes itself, which is appropriate for the mandatory ABI of an operating system. It also makes the code easier to reason about and audit for robustness and security. The layout is simply two consecutive segments, each containing aligned whole pages:

The first segment is read-only, and includes the ELF headers and metadata for dynamic linking along with constant data private to the vDSO’s implementation. The second segment is executable, containing the vDSO code. The whole vDSO image consists of just these two segments’ pages, present in the ELF image just as they should appear in memory. To map in the vDSO requires only two values gleaned from the vDSO’s ELF headers: the number of pages in each segment.

Boot-time Read-Only Data

Some system calls simply return values that are constant throughout the runtime of the whole system, though the ABI of the system is that their values must be queried at runtime and cannot be compiled into user code. These values either are fixed in the kernel at compile time or are determined by the kernel at boot time from hardware or boot parameters. Examples include system_get_version(), system_get_num_cpus(), and ticks_per_second().

The last example is influenced by a kernel command line option.

Because these values are constant, there is no need to pay the overhead of entering the kernel to read them. Instead, the vDSO implementations of these are simple C++ functions that just return constants read from the vDSO's read-only data segment. Values fixed at compile time (such as the system version string) are simply compiled into the vDSO.

For the values determined at boot time, the kernel must modify the contents of the vDSO. This is accomplished by the boot-time code that sets up the vDSO VMO, before it starts the first userspace process and gives it the VMO handle. At compile time, the offset into the vDSO image of the vdso_constants data structure is extracted from the vDSO ELF file that will be embedded in the kernel. At boot time, the kernel temporarily maps the pages of the VMO covering vdso_constants into its own address space long enough to initialize the structure with the right values for the current run of the system.

Enforcement

The vDSO entry points are the only means to enter the kernel for system calls. The machine-specific instructions used to enter the kernel (e.g. syscall on x86) are not part of the system ABI and it's invalid for user code to execute such instructions directly. The interface between the kernel and the vDSO code is a private implementation detail.

Because the vDSO is itself normal code that executes in userspace, the kernel must robustly handle all possible entries into kernel mode from userspace. However, potential kernel bugs can be mitigated somewhat by enforcing that each kernel entry be made only from the proper vDSO code. This enforcement also avoids developers of userspace code circumventing the ABI rules (because of ignorance, malice, or misguided intent to work around some perceived limitation of the official ABI), which could lead to the private kernel-vDSO interface becoming a de facto ABI for application code.

The kernel enforces correct use of the vDSO in two ways:

It constrains how the vDSO VMO can be mapped into a process.

When a vmar_map() call is made using the vDSO VMO and requesting ZX_VM_PERM_EXECUTE, the kernel requires that the offset and size of the mapping exactly match the vDSO's executable segment. It also allows only one such mapping. Once the valid vDSO mapping has been established in a process, it cannot be removed. Attempts to map the vDSO a second time into the same process, to unmap the vDSO code from a process, or to make an executable mapping of the vDSO that don't use the correct offset and size, fail with ZX_ERR_ACCESS_DENIED.

At compile time, the offset and size of the vDSO's code segment are extracted from the vDSO ELF file and used as constants in the kernel's mapping enforcement code.

When the one valid vDSO mapping is established in a process, the kernel records the address for that process so it can be checked quickly.

It constrains what PC locations can be used to enter the kernel.

When a user thread enters the kernel for a system call, a register indicates which low-level system call is being invoked. The low-level system calls are the private interface between the kernel and the vDSO; many correspond directly the system calls in the public ABI, but others do not.

For each low-level system call, there is a fixed set of PC locations in the vDSO code that 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 vDSO's symbol table and used to generate kernel code that defines a PC validity predicate for each low-level system call. Since there is only one definition of the vDSO code used by all user processes in the system, these predicates simply check for known, valid, constant offsets from the beginning of the vDSO code segment.

On entry to the kernel for a system call, the kernel examines the PC location of the syscall instruction on x86 (or equivalent instruction on other machines). It subtracts the base address of the vDSO code recorded for the process at vmar_map() time from the PC, and passes the resulting offset to the validity predicate for the system call being invoked. If the predicate rules the PC invalid, the calling thread is not allowed to proceed with the system call and instead takes a synthetic exception similar to the machine exception that would result from invoking an undefined or privileged machine instruction.

Variants

TODO(mcgrathr): vDSO variants are an experimental feature that is not yet in real use. There is a proof-of-concept implementation and simple tests, but more work is required to implement the concept robustly and determine what variants will be made available. The concept is to provide variants of the vDSO image that export only a subset of the full vDSO system call interface. For example, system calls intended only for use by device drivers might be elided from the vDSO variant used for normal application code.

3.2 Syscalls

The expected number of syscall 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`.

Syscalls are used to make userspace code interact with kernel objects, almost exclusively using **handles**. In the userspace, a handle is a 32bit integer (type `zx_handle_t`). 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 call fall into three categories:

- Calls which have no limitations, of which there are only a very few, for example zx_clock_get() and zx_nanosleep() may be called by any thread. Calls which take a Handle as the first parameter, denoting the Object they act upon, which are the vast majority, for example zx_channel_write() and zx_port_queue().
- Calls which create new Objects but do not take a Handle, such as zx_event_create() and zx_channel_create(). Access to these (and limitations upon them) is controlled by the Job in which the calling Process is contained.
- 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. They are C ELF ABI functions of the form zx_noun_verb() or zx_noun_verb_direct-object().

The system calls are defined by syscalls.abigen and processed by the abigen tool into include files and glue code in libzircon and the kernel's libsyscalls.

3.3 Handles and Rights

Objects may have multiple Handles (in one or more Processes) that refer to them.

For almost all Objects, when the last open Handle that refers to an Object is closed, the Object is either destroyed, or put into a final state that may not be undone.

Handles may be moved from one Process to another by writing them into a Channel (using zx_channel_write()), or by using zx_process_start() to pass a Handle as the argument of the first thread in a new Process.

The actions which may be taken on a Handle or the Object it refers to are governed by the Rights associated with that Handle. Two Handles that refer to the same Object may have different Rights.

The zx_handle_duplicate() and zx_handle_replace() system calls may be used to obtain additional Handles referring to the same Object as the Handle passed in, optionally with reduced Rights. The zx_handle_close() system call closes a Handle, releasing the Object it refers to, if that Handle is the last one for that Object. The zx_handle_close_many() system call similarly closes an array of handles.

4 Attack

4.1 How to build fuchsia

5 Conclusions

6 Appendix

References

[1] Google, "Git repositories on fuchsia." https://fuchsia.googlesource.com.

[2] Google, "Fuchsia documentation." https://fuchsia.googlesource.com/docs/+/HEAD/development/source_code/layers.md.

[3] Google, "Zircon Layer." https://fuchsia.googlesource.com/zircon/+/master/README.md.

[4] Google, "Garnet Layer." https://fuchsia.googlesource.com/garnet/+/master/README.md.

[5] Google, "Escher." https://fuchsia.googlesource.com/garnet/+/master/public/lib/escher/.

[6] Google, "Peridot Layer." https://fuchsia.googlesource.com/peridot/+/master/README.md.

[7] Google, "Ledger." https://fuchsia.googlesource.com/peridot/+/master/docs/ledger/README.md.

[8] Google, "Topaz Layer." https://fuchsia.googlesource.com/topaz/+/master/README.md.

[9] Google, "Flutter." https://flutter.io.

[10] Google, "Zircon Kernel." https://fuchsia.googlesource.com/zircon.

[11] Google, "LK." https://github.com/littlekernel/lk.

[12] Google, "Zircon and LK." https://fuchsia.googlesource.com/zircon/+/master/docs/zx_and_lk.md.