OATmeal on the Universal Cereal Bus: Exploiting Android phones over USB

Posted by Jann Horn, Google Project Zero

Recently, there has been some attention around the topic of physical attacks on smartphones, where an attacker with the ability to connect USB devices to a locked phone attempts to gain access to the data stored on the device. This blogpost describes how such an attack could have been performed against Android devices (tested with a Pixel 2).

After an Android phone has been unlocked once on boot (on newer devices, using the "Unlock for all features and data" screen; on older devices, using the "To start Android, enter your password" screen), it retains the encryption keys used to decrypt files in kernel memory even when the screen is locked, and the encrypted filesystem areas or partition(s) stay accessible. Therefore, an attacker who gains the ability to execute code on a locked device in a sufficiently privileged context can not only backdoor the device, but can also directly access user data.

(Caveat: We have not looked into what happens to work profile data when a user who has a work profile toggles off the work profile.)

The bug reports referenced in this blogpost, and the corresponding proof-of-concept code, are available at:

These issues were fixed as CVE-2018-9445 (fixed at patch level 2018-08-01) and CVE-2018-9488 (fixed at patch level 2018-09-01).

The attack surface

Many Android phones support USB host mode (often using OTG adapters). This allows phones to connect to many types of USB devices (this list isn't necessarily complete):

- USB sticks: When a USB stick is inserted into an Android phone, the user can copy files between the system and the USB stick. Even if the device is locked, Android versions before P will still attempt to mount the USB stick. (Android 9, which was released after these issues were reported, has logic in vold that blocks mounting USB sticks while the device is locked.)
- USB keyboards and mice: Android supports using external input devices instead of using the touchscreen. This also works on the lockscreen (e.g. for entering the PIN).
- USB ethernet adapters: When a USB ethernet adapter is connected to an Android phone, the phone will attempt to connect to a wired network, using DHCP to obtain an IP address. This also works if the phone is locked.

This blogpost focuses on USB sticks. Mounting an untrusted USB stick offers nontrivial attack surface in highly privileged system components: The kernel has to talk to the USB mass storage device using a protocol that includes a subset of SCSI, parse its partition table, and interpret partition contents using the kernel's filesystem implementation; userspace code has to identify the filesystem type and instruct the kernel to mount the device to some location. On Android, the userspace implementation for this is mostly in vold (one of the processes that are considered to have kernel-equivalent privileges), which uses separate processes in restrictive SELinux domains to e.g. determine the filesystem types of partitions on USB sticks.

The bug (part 1): Determining partition attributes

When a USB stick has been inserted and vold has determined the list of partitions on the device, it attempts to identify three attributes of each partition: Label (a user-readable string describing the partition), UUID (a unique identifier that can be used to determine whether the USB stick is one that has been inserted into the device before), and filesystem type. In the modern GPT partitioning scheme, these attributes can mostly be stored in the partition table itself; however, USB sticks tend to use the MBR partition scheme instead, which can not store UUIDs and labels. For normal USB sticks, Android supports both the MBR partition scheme and the GPT partition scheme.

To provide the ability to label partitions and assign UUIDs to them even when the MBR partition scheme is used, filesystems implement a hack: The filesystem header contains fields for these attributes, allowing an implementation that has already determined the filesystem type and knows the filesystem header layout of the specific filesystem to extract this information in a filesystem-specific manner. When vold wants to determine label, UUID and filesystem type, it invokes /system/bin/blkid in the blkid_untrusted SELinux domain, which does exactly this: First, it attempts to identify the filesystem type using magic numbers and (failing that) some heuristics, and then, it extracts the label and UUID. It prints the results to stdout in the following format:

/dev/block/sda1: LABEL="<label>" UUID="<uuid>" TYPE="<type>"

However, the version of blkid used by Android did not escape the label string, and the code responsible for parsing blkid's output only scanned for the first occurrences of UUID=" and TYPE=". Therefore, by creating a partition with a crafted label, it was possible to gain control over the UUID and type strings returned to vold, which would otherwise always be a valid UUID string and one of a fixed set of type strings.

The bug (part 2): Mounting the filesystem

When vold has determined that a newly inserted USB stick with an MBR partition table contains a partition of type vfat that the kernel's vfat filesystem implementation should be able to mount, PublicVolume::doMount() constructs a mount path based on the filesystem UUID, then attempts to ensure that the mountpoint directory exists and has appropriate ownership and mode, and then attempts to mount over that directory:

```
if (mFsType != "vfat") {
    LOG(ERROR) << getId() << " unsupported filesystem " << mFsType;
    return -EIO;
}
```

```
if (vfat::Check(mDevPath)) {
    LOG(ERROR) << getId() << " failed filesystem check";
    return -EIO;
}
// Use UUID as stable name, if available
std::string stableName = getId();
if (!mFsUuid.empty()) {
    stableName = mFsUuid;
}
mRawPath = StringPrintf("/mnt/media_rw/%s", stableName.c_str());
[...]
if (fs_prepare_dir(mRawPath.c_str(), 0700, AID_ROOT, AID_ROOT)) {
    PLOG(ERROR) << getId() << " failed to create mount points";
    return -errno;
}
if (vfat::Mount(mDevPath, mRawPath, false, false, false,
        AID_MEDIA_RW, AID_MEDIA_RW, 0007, true)) {
    PLOG(ERROR) << getId() << " failed to mount " << mDevPath;
    return -EIO;
}
```

The mount path is determined using a format string, without any sanity checks on the UUID string that was provided by blkid. Therefore, an attacker with control over the UUID string can perform a directory traversal attack and cause the FAT filesystem to be mounted outside of /mnt/media_rw.

This means that if an attacker inserts a USB stick with a FAT filesystem whose label string is 'UUID="../##' into a locked phone, the phone will mount that USB stick to /mnt/##.

However, this straightforward implementation of the attack has several severe limitations; some of them can be overcome, others worked around:

- Label string length: A FAT filesystem label is limited to 11 bytes. An attacker attempting to perform a straightforward attack needs to use the six bytes 'UUID="' to start the injection, which leaves only five characters for the directory traversal - insufficient to reach any interesting point in the mount hierarchy. The next section describes how to work around that.
- SELinux restrictions on mountpoints: Even though vold is considered to be kernel-equivalent, a SELinux policy applies some restrictions on what vold can do. Specifically, the mounton permission is restricted to a set of permitted labels.
- Writability requirement: fs_prepare_dir() fails if the target directory is not mode 0700 and chmod() fails.
- Restrictions on access to vfat filesystems: When a vfat filesystem is mounted, all of its files are labeled as u:object_r:vfat:s0. Even if the filesystem is mounted in a place from which important code or data is loaded, many SELinux contexts won't be permitted to actually interact with the filesystem - for example, the zygote and system_server aren't allowed to do so. On top of that, processes that don't have sufficient privileges to bypass DAC checks also need to be in the media_rw group. The section "Dealing with SELinux: Triggering the bug twice" describes how these restrictions can be avoided in the context of this specific bug.

Exploitation: Chameleonic USB mass storage

As described in the previous section, a FAT filesystem label is limited to 11 bytes. blkid supports a range of other filesystem types that have significantly longer label strings, but if you used such a filesystem type, you'd then have to make it past the fsck check for vfat filesystems and the filesystem header checks performed by the kernel when mounting a vfat filesystem. The vfat kernel filesystem doesn't require a fixed magic value right at the start of the partition, so this might theoretically work somehow; however, because several of the values in a FAT filesystem header are actually important for the kernel, and at the same time, blkid also performs some sanity checks on superblocks, the PoC takes a different route.

After blkid has read parts of the filesystem and used them to determine the filesystem's type, label and UUID, fsck_msdos and the in-kernel filesystem implementation will re-read the same data, and those repeated reads actually go through to the storage device. The Linux kernel caches block device pages when userspace directly interacts with block devices, but __blkdev_put() removes all cached data associated with a block device when the last open file referencing the device is closed.

A physical attacker can abuse this by attaching a fake storage device that returns different data for multiple reads from the same location. This allows us to present, for example, a romfs header with a long label string to blkid while presenting a perfectly normal vfat filesystem to fsck_msdos and the in-kernel filesystem implementation.

This is relatively simple to implement in practice thanks to Linux' built-in support for device-side USB. Andrzej Pietrasiewicz's talk "Make your own USB gadget" is a useful introduction to this topic. Basically, the kernel ships with implementations for device-side USB mass storage, HID devices, ethernet adapters, and more; using a relatively simple pseudo-filesystem-based configuration interface, you can configure a composite gadget that provides one or multiple of these functions, potentially with multiple instances, to the connected device. The

hardware you need is a system that runs Linux and supports device-side USB; for testing this attack, a Raspberry Pi Zero W was used.

The f_mass_storage gadget function is designed to use a normal file as backing storage; to be able to interactively respond to requests from the Android phone, a FUSE filesystem is used as backing storage instead, using the direct_io option / the FOPEN_DIRECT_IO flag to ensure that our own kernel doesn't add unwanted caching.

At this point, it is already possible to implement an attack that can steal, for example, photos stored on external storage. Luckily for an attacker, immediately after a USB stick has been mounted, com.android.externalstorage/.MountReceiver is launched, which is a process whose SELinux domain permits access to USB devices. So after a malicious FAT partition has been mounted over /data (using the label string 'UUID="../../data'), the zygote forks off a child with appropriate SELinux context and group membership to permit accesses to USB devices. This child then loads bytecode from /data/dalvik-cache/, permitting us to take control over com.android.externalstorage, which has the necessary privileges to exfiltrate external storage contents.

However, for an attacker who wants to access not just photos, but things like chat logs or authentication credentials stored on the device, this level of access should normally not be sufficient on its own.

Dealing with SELinux: Triggering the bug twice

The major limiting factor at this point is that, even though it is possible to mount over /data, a lot of the highly-privileged code running on the device is not permitted to access the mounted filesystem. However, one highly-privileged service does have access to it: vold.

vold actually supports two types of USB sticks, PublicVolume and PrivateVolume. Up to this point, this blogpost focused on PublicVolume; from here on, PrivateVolume becomes important.

A PrivateVolume is a USB stick that must be formatted using a GUID Partition Table. It must contain a partition that has type UUID kGptAndroidExpand (193D1EA4-B3CA-11E4-B075-10604B889DCF), which contains a dm-crypt-encrypted ext4 (or f2fs) filesystem. The corresponding key is stored at /data/misc/vold/expand_{partGuid}.key, where {partGuid} is the partition GUID from the GPT table as a normalized lowercase hexstring.

As an attacker, it normally shouldn't be possible to mount an ext4 filesystem this way because phones aren't usually set up with any such keys; and even if there is such a key, you'd still have to know what the correct partition GUID is and what the key is. However, we can mount a vfat filesystem over /data/misc and put our own key there, for our own GUID. Then, while the first malicious USB mass storage device is still connected, we can connect a second one that is mounted as PrivateVolume using the keys vold will read from the first USB mass storage device. (Technically, the ordering in the last sentence isn't entirely correct - actually, the exploit provides both mass storage devices as a single composite device at the same time, but stalls the first read from the second mass storage device to create the desired ordering.)

Because PrivateVolume instances use ext4, we can control DAC ownership and permissions on the filesystem; and thanks to the way a PrivateVolume is integrated into the system, we can even control SELinux labels on that filesystem.

In summary, at this point, we can mount a controlled filesystem over /data, with arbitrary file permissions and arbitrary SELinux contexts. Because we control file permissions and SELinux contexts, we can allow any process to access files on our filesystem - including mapping them with PROT_EXEC.

Injecting into zygote

The zygote process is relatively powerful, even though it is not listed as part of the TCB. By design, it runs with UID 0, can arbitrarily change its UID, and can perform dynamic SELinux transitions into the SELinux contexts of system_server and normal apps. In other words, the zygote has access to almost all user data on the device.

When the 64-bit zygote starts up on system boot, it loads code from /data/dalvik-cache/arm64/system@framework@boot*.{art,oat,vdex}. Normally, the oat file (which contains an ELF library that will be loaded with dlopen()) and the vdex file are symlinks to files on the immutable /system partition; only the art file is actually stored on /data. But we can instead make system@framework@boot.art and system@framework@boot.vdex symlinks to /system (to get around some consistency checks without knowing exactly which Android build is running on the device) while placing our own malicious ELF library at system@framework@boot.oat (with the SELinux context that the legitimate oat file would have). Then, by placing a function with __attribute__((constructor)) in our ELF library, we can get code execution in the zygote as soon as it calls dlopen() on startup.

The missing step at this point is that when the attack is performed, the zygote is already running; and this attack only works while the zygote is starting up.

Crashing the system

This part is a bit unpleasant.

When a critical system component (in particular, the zygote or system_server) crashes (which you can simulate on an eng build using kill), Android attempts to automatically recover from the crash by restarting most userspace processes (including the zygote). When this happens, the screen first shows the boot animation for a bit, followed by the lock screen with the "Unlock for all features and data" prompt that normally only shows up after boot. However, the key material for accessing user data is still present at this point, as you can verify if ADB is on by running "ls /sdcard" on the device.

This means that if we can somehow crash system_server, we can then inject code into the zygote during the following userspace restart and will be able to access user data on the device.

Of course, mounting our own filesystem over /data is very crude and makes all sorts of things fail, but surprisingly, the system doesn't immediately fall over - while parts of the UI become unusable, most places have some error handling that prevents the system from failing so clearly that a restart happens.

After some experimentation, it turned out that Android's code for tracking bandwidth usage has a safety check: If the network usage tracking code can't write to disk and >=2MiB (mPersistThresholdBytes) of network traffic have been observed since the last successful write, a fatal exception is thrown. This means that if we can create some sort of network connection to the device and then send it >=2MiB worth of ping flood, then trigger a stats

writeback by either waiting for a periodic writeback or changing the state of a network interface, the device will reboot.

To create a network connection, there are two options:

- Connect to a wifi network. Before Android 9, even when the device is locked, it is normally possible to connect to a new wifi network by dragging down from the top of the screen, tapping the drop-down below the wifi symbol, then tapping on the name of an open wifi network. (This doesn't work for networks protected with WPA, but of course an attacker can make their own wifi network an open one.) Many devices will also just autoconnect to networks with certain names.
- Connect to an ethernet network. Android supports USB ethernet adapters and will automatically connect to ethernet networks.

For testing the exploit, a manually-created connection to a wifi network was used; for a more reliable and user-friendly exploit, you'd probably want to use an ethernet connection.

At this point, we can run arbitrary native code in zygote context and access user data; but we can't yet read out the raw disk encryption key, directly access the underlying block device, or take a RAM dump (although at this point, half the data that would've been in a RAM dump is probably gone anyway thanks to the system crash). If we want to be able to do those things, we'll have to escalate our privileges a bit more.

From zygote to vold

Even though the zygote is not supposed to be part of the TCB, it has access to the CAP_SYS_ADMIN capability in the initial user namespace, and the SELinux policy permits the use of this capability. The zygote uses this capability for the mount() syscall and for installing a seccomp filter without setting the NO_NEW_PRIVS flag. There are multiple ways to abuse CAP_SYS_ADMIN; in particular, on the Pixel 2, the following ways seem viable:

- You can install a seccomp filter without NO_NEW_PRIVS, then perform an execve() with a privilege transition (SELinux exec transition, setuid/setgid execution, or execution with permitted file capability set). The seccomp filter can then force specific syscalls to fail with error number 0 - which e.g. in the case of open() means that the process will believe that the syscall succeeded and allocated file descriptor 0. This attack works here, but is a bit messy.
- You can instruct the kernel to use a file you control as high-priority swap device, then create memory pressure. Once the kernel writes stack or heap pages from a sufficiently privileged process into the swap file, you can edit the swapped-out memory, then let the process load it back. Downsides of this technique are that it is very unpredictable, it involves memory pressure (which could potentially cause the system to kill processes you want to keep, and probably destroys many forensic artifacts in RAM), and requires some way to figure out which swapped-out pages belong to which process and are used for what. This requires the kernel to support swap.
- You can use pivot_root() to replace the root directory of either the current mount namespace or a newly created mount namespace, bypassing the SELinux checks that would have been performed for mount().

> Doing it for a new mount namespace is useful if you only want to affect a child process that elevates its privileges afterwards. This doesn't work if the root filesystem is a rootfs filesystem. This is the technique used here.

In recent Android versions, the mechanism used to create dumps of crashing processes has changed: Instead of asking a privileged daemon to create a dump, processes execute one of the helpers /system/bin/crash_dump64 and /system/bin/crash_dump32, which have the SELinux label u:object_r:crash_dump_exec:s0. Currently, when a file with such a label is executed by any SELinux domain, an automatic domain transition to the crash_dump domain is triggered (which automatically implies setting the AT_SECURE flag in the auxiliary vector, instructing the linker of the new process to be careful with environment variables like LD_PRELOAD):

domain_auto_trans(domain, crash_dump_exec, crash_dump);

At the time this bug was reported, the crash_dump domain had the following SELinux policy:

[...]

allow crash_dump {

 domain

 -init

 -crash_dump

 -keystore

 -logd

}:process { ptrace signal sigchld sigstop sigkill };

[...]

r_dir_file(crash_dump, domain)

[...]

This policy permitted crash_dump to attach to processes in almost any domain via ptrace() (providing the ability to take over the process if the DAC controls permit it) and allowed it to read properties of any process in procfs. The exclusion list for ptrace access lists a few TCB processes; but notably, vold was not on the list. Therefore, if we can execute crash_dump64 and somehow inject code into it, we can then take over vold.

Note that the ability to actually ptrace() a process is still gated by the normal Linux DAC checks, and crash_dump can't use CAP_SYS_PTRACE or CAP_SETUID. If a normal app managed to inject code into crash_dump64, it still wouldn't be able to leverage that to attack system components because of the UID mismatch.

If you've been reading carefully, you might now wonder whether we could just place our own binary with context u:object_r:crash_dump_exec:s0 on our fake /data filesystem, and then execute that to gain code execution in the crash_dump domain. This doesn't work because vold - very sensibly - hardcodes the MS_NOSUID flag when

mounting USB storage devices, which not only degrades the execution of classic setuid/setgid binaries, but also degrades the execution of files with file capabilities and executions that would normally involve automatic SELinux domain transitions (unless the SELinux policy explicitly opts out of this behavior by granting PROCESS2__NOSUID_TRANSITION).

To inject code into crash_dump64, we can create a new mount namespace with unshare() (using our CAP_SYS_ADMIN capability), then call pivot_root() to point the root directory of our process into a directory we fully control, and then execute crash_dump64. Then the kernel parses the ELF headers of crash_dump64, reads the path to the linker (/system/bin/linker64), loads the linker into memory from that path (relative to the process root, so we can supply our own linker here), and executes it.

At this point, we can execute arbitrary code in crash_dump context and escalate into vold from there, compromising the TCB. At this point, Android's security policy considers us to have kernel-equivalent privileges; however, to see what you'd have to do from here to gain code execution in the kernel, this blogpost goes a bit further.

From vold to init context

It doesn't look like there is an easy way to get from vold into the real init process; however, there is a way into the init SELinux context. Looking through the SELinux policy for allowed transitions into init context, we find the following policy:

domain_auto_trans(kernel, init_exec, init)

This means that if we can get code running in kernel context to execute a file we control labeled init_exec, on a filesystem that wasn't mounted with MS_NOSUID, then our file will be executed in init context.

The only code that is running in kernel context is the kernel, so we have to get the kernel to execute the file for us. Linux has a mechanism called "usermode helpers" that can do this: Under some circumstances, the kernel will delegate actions (such as creating coredumps, loading key material into the kernel, performing DNS lookups, ...) to userspace code. In particular, when a nonexistent key is looked up (e.g. via request_key()), /sbin/request-key (hardcoded, can only be changed to a different static path at kernel build time with CONFIG_STATIC_USERMODEHELPER_PATH) will be invoked.

Being in vold, we can simply mount our own ext4 filesystem over /sbin without MS_NOSUID, then call request_key(), and the kernel invokes our request-key in init context.

The exploit stops at this point; however, the following section describes how you could build on it to gain code execution in the kernel.

From init context to the kernel

From init context, it is possible to transition into modprobe or vendor_modprobe context by executing an appropriately labeled file after explicitly requesting a domain transition (note that this is domain_trans(), which permits a transition on exec, not domain_auto_trans(), which automatically performs a transition on exec):

domain_trans(init, { rootfs toolbox_exec }, modprobe)

domain_trans(init, vendor_toolbox_exec, vendor_modprobe)

modprobe and vendor_modprobe have the ability to load kernel modules from appropriately labeled files:

allow modprobe self:capability sys_module;

allow modprobe { system_file }:system module_load;

allow vendor_modprobe self:capability sys_module;

allow vendor_modprobe { vendor_file }:system module_load;

Android nowadays doesn't require signatures for kernel modules:

walleye:/ # zcat /proc/config.gz | grep MODULE

CONFIG_MODULES_USE_ELF_RELA=y

CONFIG_MODULES=y

CONFIG_MODULE_FORCE_LOAD is not set

CONFIG_MODULE_UNLOAD=y

CONFIG_MODULE_FORCE_UNLOAD=y

CONFIG_MODULE_SRCVERSION_ALL=y

CONFIG_MODULE_SIG is not set

CONFIG_MODULE_COMPRESS is not set

CONFIG_MODULES_TREE_LOOKUP=y

CONFIG_ARM64_MODULE_CMODEL_LARGE=y

CONFIG_ARM64_MODULE_PLTS=y

CONFIG_RANDOMIZE_MODULE_REGION_FULL=y

CONFIG_DEBUG_SET_MODULE_RONX=y

Therefore, you could execute an appropriately labeled file to execute code in modprobe context, then load an appropriately labeled malicious kernel module from there.

Lessons learned

Notably, this attack crosses two weakly-enforced security boundaries: The boundary from blkid_untrusted to vold (when vold uses the UUID provided by blkid_untrusted in a pathname without checking that it resembles a valid

UUID) and the boundary from the zygote to the TCB (by abusing the zygote's CAP_SYS_ADMIN capability). Software vendors have, very rightly, been stressing for quite some time that it is important for security researchers to be aware of what is, and what isn't, a security boundary - but it is also important for vendors to decide where they want to have security boundaries and then rigorously enforce those boundaries. Unenforced security boundaries can be of limited use - for example, as a development aid while stronger isolation is in development -, but they can also have negative effects by obfuscating how important a component is for the security of the overall system.

In this case, the weakly-enforced security boundary between vold and blkid_untrusted actually contributed to the vulnerability, rather than mitigating it. If the blkid code had run in the vold process, it would not have been necessary to serialize its output, and the injection of a fake UUID would not have worked.
