Protected: Recovering Hidden Evidence from a Gallery Vault–Encrypted Android Device

Author: Simon Devenish **Date:** February 2026 **Platform:** Hack The Box **Category:** Mobile Forensics **Difficulty:** Medium **Flag:** `HTB{G3113ry_D3cr3bt0r}`

1. Executive Summary

This document presents a forensic analysis of an Android device image containing a file concealed by **Gallery Vault** (v404033), a popular photo-hiding application. The investigation required defeating three independent protection layers to recover a PNG image containing the flag.

Protection Layers

Layer	Mechanism	Defeat Method
Layer 1	ZipCrypto encryption on the device dump	bkcrack known-plaintext attack
Layer 2	Gallery Vault application PIN (5-digit)	SHA1+MD5 hash brute-force
Layer 3	File-level XOR + DES encryption	APK reverse engineering via jadx

Methodology Summary

The device dump was a 1.3 GB ZipCrypto-encrypted archive containing ~90,000 files from an Android emulator's `/data/` partition. After cracking the zip encryption with a known-plaintext attack, analysis revealed a single file hidden inside Gallery Vault: `flag.png`. Published research on Gallery Vault's encryption scheme (from 2019) proved insufficient for version 404033. The breakthrough came from **decompiling the APK itself with jadx**, which revealed the complete key derivation chain: a hardcoded hex constant `676F6F645F6776` ("good_gv") is used to DES-decrypt a second constant into the string `tianxiawuzei`, which then DES-decrypts a per-file random XOR key stored in the file's metadata tail. A critical discovery was that the tail's check byte (`0x00`) indicated the main file body was stored in **plaintext**—only the first 2,803 bytes required XOR decryption.

2. Initial Reconnaissance

2.1 Target Analysis

The challenge server (`154.57.164.70:32704`) served a single file over HTTP via a Werkzeug/Python backend:

```
HTTP/1.1 200 OK
Content-Disposition: attachment; filename=mobile_protected.zip
Content-Type: application/zip
Content-Length: 1314575553
```

Property	Value
Archive	`mobile_protected.zip` (~1.3 GB)
Compression	ZipCrypto Deflate:Maximum
Total entries	89,798 files
Uncompressed size	~3 GB
Content	Android `/data/` directory dump

2.2 Challenge Identification

The archive contains a full Android device data partition. The challenge description—*"critical evidence or artifacts may have been overlooked"*—suggests hidden data within an installed application rather than a filesystem-level forensics exercise.

---

3. Defeating ZipCrypto Encryption

3.1 The Known-Plaintext Vulnerability

ZipCrypto is a legacy encryption scheme vulnerable to known-plaintext attacks when at least 12 bytes of plaintext from any file in the archive are known. Using `bkcrack -L`, I identified five 65-byte XML files sharing CRC32 `0x1314423c`—empty Android SharedPreferences files with predictable content:

```
# 65 bytes, CRC32 = 0x1314423c
content = b"<?xml version='1.0' encoding='utf-8' standalone='yes' ?>\n<map />\n"
```

The space before the closing slash in `<map />` (not `<map/>`) was critical. I brute-forced the exact whitespace variation against the CRC32 to confirm the plaintext.

3.2 Key Recovery and Decryption

```
# Recover internal encryption keys (~1 minute)
bkcrack -C mobile_protected.zip --cipher-index 793 -p plain_sharedprefs.xml
# Keys: 890ca696 587c6483 cd091757

# Re-encrypt archive with known password
bkcrack -C mobile_protected.zip -k 890ca696 587c6483 cd091757 -U decrypted.zip cracked

# Extract all 89,798 files
unzip -P cracked -o decrypted.zip
```

The entire zip cracking process—from plaintext identification to full extraction—completed in under five minutes.

---

4. Android Data Analysis

4.1 Device Profile

Property	Value
Device type	Android emulator (Pixel-style overlay)
Architecture	x86_64
Root status	Rooted (Magisk present)
Key application	Gallery Vault (`com.thinkyeah.galleryvault`)

4.2 Gallery Vault Discovery

Gallery Vault is a consumer privacy application that hides photos and videos behind a PIN-protected vault. Forensically, it is significant because it encrypts files at the application layer, independent of full-disk encryption.

Database (`galleryvault.db` , table `file_v1`):

```
SELECT name, mime_type, image_width, image_height, file_size, encrypt_state FROM file_v1;
-- flag.png | image/png | 564 | 568 | 787165 | 0
```

A single hidden file: `flag.png` , 564×568 pixels, 787,165 bytes.

Vault configuration (`Kidd.xml`):

Setting	Value
`LockPin`	`413FA2AF141A04634145E1E261E05CEFB8F7773ED361CCC5D896DFBAD0FECFC1FE7FC9A1`
`AuthenticationEmail`	`ahmedelsalkh627@gmail.com`
`user_random_number`	44
`signature`	`ebf3d382-ed56-4d1f-ad10-54b3b7031c81`
`gallery_vault_folder`	`.galleryvault_DoNotDelete_1742398726`

4.3 Vault File Location

The encrypted vault file and its thumbnail:

```
data/media/0/.galleryvault_DoNotDelete_1742398726/files/b2/
├── b238f4cd-f79f-4e19-be3b-068b7add4d85      (790,131 bytes)
└── b238f4cd-f79f-4e19-be3b-068b7add4d85_t    (409,878 bytes - thumbnail)
```

Both files begin with valid PNG headers. The main file contains a 300×300 thumbnail followed by encrypted data and a metadata tail delimited by `>>tyfs>>` / `<<tyfs<<` markers.

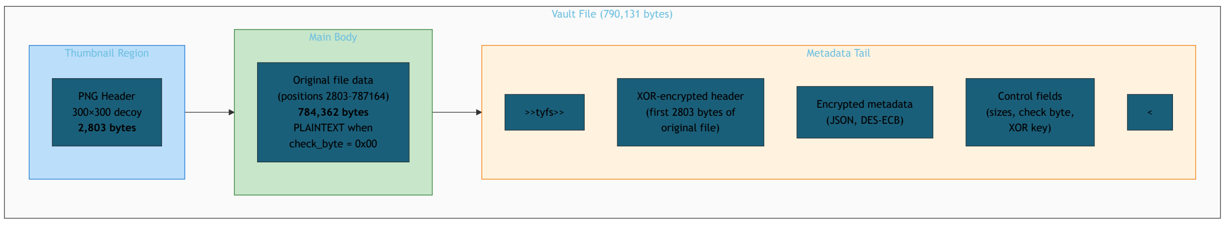

Figure 1: Binary layout of a Gallery Vault encrypted file. The thumbnail is a decoy header; the actual original file data is split between the XOR-encrypted tail region and the main body.

---

5. Prior Research and Its Limitations

5.1 Published Analyses

Three public sources describe Gallery Vault's encryption:

Source	Year	Relevance
S-RM: Cracking the Vault	2019	Describes DES+XOR scheme, hardcoded keys `gallery_` and `tianxiaw`
KN4GXD: Pole Vaulting	2019	PoC exploit for PIN extraction (site defunct)
GVHack Gist	2019	Java PIN cracker (DES key redacted)

5.2 Why Published Methods Failed

Applying the documented keys (`tianxiaw` , `gallery_`) directly to the encrypted data produced no recognizable output:

```
DES.new(b'tianxiaw', DES.MODE_ECB).decrypt(encrypted[:8])
# Result: 11897d71ca11778e (NOT a PNG header)

DES.new(b'gallery_', DES.MODE_ECB).decrypt(encrypted[:8])
# Result: d5de2346fefdc814 (NOT a PNG header)
```

The 2019 research described an older version's algorithm. Version 404033 uses a **different key derivation chain** where `tianxiaw` is not the DES key itself but a *derived intermediate*. The only way to recover the actual algorithm was to decompile the APK.

---

6. APK Decompilation and Key Derivation

6.1 Decompiling with jadx

The Gallery Vault APK (71 MB) was decompiled using jadx:

```
jadx -d apk_decompiled --no-res --threads-count 4 base.apk
# 24,077 classes processed, 77 decompilation errors (normal for large APKs)
```

The code is heavily ProGuard-obfuscated. All meaningful class and method names are replaced with single-letter identifiers. The encryption logic was found across four obfuscated packages:

Package	Original Name (from comments)	Purpose
`Mc/c.java`	`ThinkSecurity.java`	DES key derivation and encryption
`Hf/k.java`	`GvFileSecurity.java`	File encryption orchestrator
`Oc/d.java`	`FileTailOperatorV1.java`	XOR encryption and tail construction
`Oc/l.java`	`SecurityConstants.java`	`>>tyfs>> / <<tyfs<<` marker definitions

6.2 The Key Derivation Chain

The encryption uses a three-stage key derivation chain, all rooted in hardcoded constants:

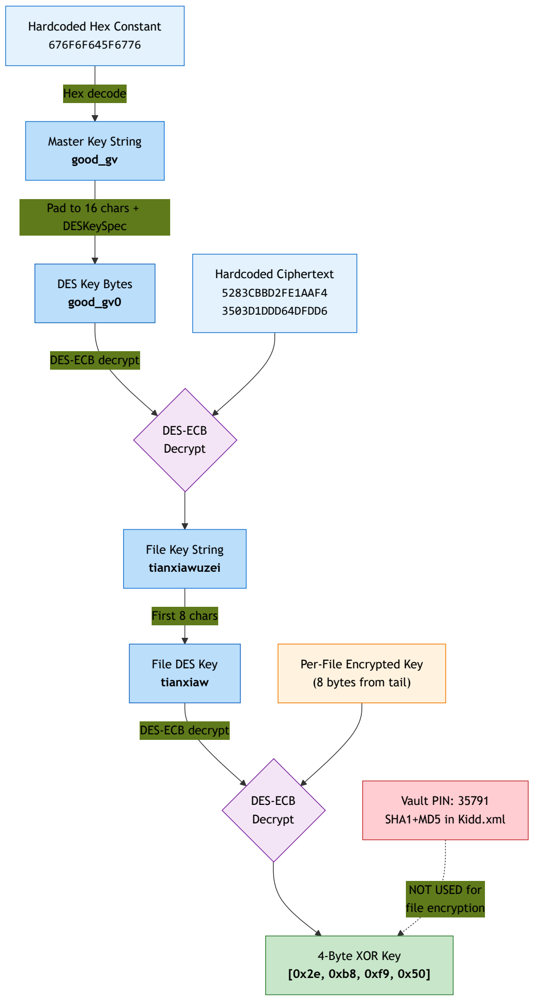

Figure 2: Gallery Vault v404033 key derivation chain. All keys trace back to hardcoded hex constants in the APK. The PIN does not participate in file encryption.

Stage 1: Master Key Recovery

In `Mc/c.java`:

```java
public static final String f8633b = new String(hexToBytes("676F6F645F6776"));
// 676F6F645F6776 → ASCII "good_gv"
```

The DES key derivation method `e()` pads strings to 16 characters with `"0"`, then takes the first 8 bytes via `DESKeySpec`:

```
"good_gv" → "good_gv000000000" → DESKeySpec → key bytes: good_gv0
```

Stage 2: File Key Derivation

In `Hf/k.java`:

```
f5384d = Mc.c.b(Mc.c.f8633b, "5283CBBD2FE1AAF43503D1DDD64DFDD6");
```

This DES-decrypts the hex constant `5283CBBD2FE1AAF43503D1DDD64DFDD6` using the `good_gv0` key:

```
DES.new(b'good_gv0',
DES.MODE_ECB).decrypt(bytes.fromhex("5283CBBD2FE1AAF43503D1DDD64DFDD6"))
# Result: "tianxiawuzei" + PKCS5 padding (04040404)
```

The string `tianxiawuzei` (not `tianxiaw` as documented in 2019) is the actual file-level DES key for version 404033.

Stage 3: Per-File XOR Key

Each encrypted file contains a random 4-byte XOR key, DES-encrypted with `tianxiaw` (first 8 bytes of `tianxiawuzei`) and stored in the file's metadata tail. For our target file:

```
DES.new(b'tianxiaw', DES.MODE_ECB).decrypt(bytes.fromhex("9d01dbb2cf9ce19b"))
# Result: [0x2e, 0xb8, 0xf9, 0x50] + PKCS5 padding (04040404)
```

6.3 The PIN Does Not Encrypt Files

A critical finding: the Gallery Vault PIN **only controls application access**. It does not participate in file encryption in any way. The `LockPin` value in `Kidd.xml` is simply `SHA1(pin) + MD5(pin)` concatenated—no DES wrapping, no key derivation role:

```
LockPin: 413FA2AF141A04634145E1E261E05CEFB8F7773ED361CCC5D896DFBAD0FECFC1FE7FC9A1
         |_______ SHA1(35791) _______|___ MD5(35791) ___|
```

Brute-forcing 4–6 digit PINs against the MD5 portion found the PIN in under a second: **35791**. However, this was ultimately irrelevant to recovering the hidden file.

---

7. Vault File Format Reverse Engineering

7.1 File Structure

The vault file (790,131 bytes) has three distinct regions:

Region	Offset	Size	Content

Thumbnail	`0x000` – `0xAF2`	2,803 bytes	Valid 300×300 PNG (viewable decoy)
Main Body	`0xAF3` – `0xC02DC`	784,362 bytes	Original file data (positions 2803+)
Tail	`0xC02DD` – `0xC0E72`	2,958 bytes	`>>tyfs>>` + encrypted header + metadata + `<<tyfs<<`

7.2 Tail Metadata Structure

The tail is parsed **backwards** from the `<<tyfs<<` end marker. The `LocalDecryptInputStreamV1` class (`Oc/h.java`) reads the tail to reconstruct the original file:

Field	Size	Value	Meaning
End marker	2 bytes	`[0x01, 0x01]`	Tail format identifier
Metadata length	8 bytes (BE)	104	Size of encrypted JSON metadata
Encrypted metadata	104 bytes	DES-ECB ciphertext	`{"name":"flag.png","uuid":"b238f4cd-...","create_time_utc":174239`
Key length	8 bytes (BE)	8	Size of encrypted XOR key (with PKCS5 padding)
Encrypted XOR key	8 bytes	`9d01dbb2cf9ce19b`	DES-encrypted 4-byte XOR key
Check byte	1 byte	`0x00`	**FALSE = main body is NOT XOR-encrypted**
Total file size	8 bytes (BE)	787,165	Original file size in bytes
Header size	8 bytes (BE)	2,803	Bytes of original file stored in tail
XOR-encrypted data	2,803 bytes	Ciphertext	First 2,803 bytes of the original file

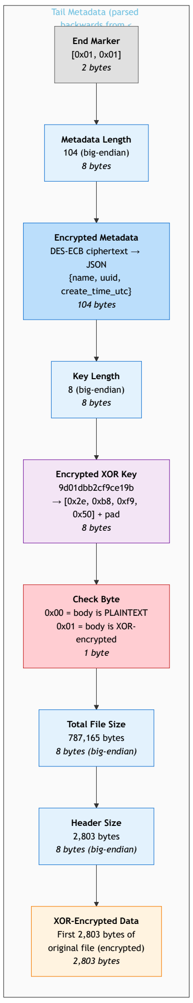

Figure 3: The tail metadata structure, parsed backwards from the end marker. The check byte (0x00) is the critical flag that determines whether the main body requires XOR decryption.

7.3 The Check Byte: The Critical Discovery

The `check_byte` field at offset -18 from the end marker controls whether the main body (the 784,362 bytes between the thumbnail and `>>tyfs>>`) is XOR-encrypted. In the `LocalDecryptInputStreamV1` read loop:

```
// Oc/h.java - read() method
if (cVar2.f10488d) {           // f10488d = check_byte (boolean)
    for (int i13 = 0; i13 < read; i13++) {
        bArr[i13] = this.f9757h.a(bArr[i13], this.f9755f + i13);  // XOR decrypt
    }
}
```

When `f10488d` is `false` (check byte = `0x00`), the main body is read **as plaintext**. Only the 2,803 bytes stored in the tail require XOR decryption. This is consistent with Gallery Vault's "partial encryption" mode for large files—encrypting only the file header makes the file unrecognisable to media scanners while minimising CPU cost on mobile devices.

7.4 The XOR Formula

The XOR decryption (symmetric—same formula for encrypt and decrypt) from `Mc/b.java`:

```
// StreamSecurity.a (inner class)
public final byte a(byte data, long position) {
    return (byte) (data ^ ((byte) position ^ key[(int)(position % key.length)]));
}
```

In Python:

```
decrypted[i] = encrypted[i] ^ (key[i % len(key)] ^ (i & 0xFF))
```

The position counter wraps at 256 via the `(byte)` cast, and the key index cycles over the 4-byte key.

---

8. File Reconstruction

8.1 Decryption Process

The `LocalDecryptInputStreamV1` reads the original file in two passes:

1. **Seek to tail** (offset 787,173 = after `>>tyfs>>`), read and XOR-decrypt 2,803 bytes (positions 0–2,802)
2. **Seek to main body** (offset 2,803 = after thumbnail), read 784,362 bytes as plaintext (positions 2,803–787,164)

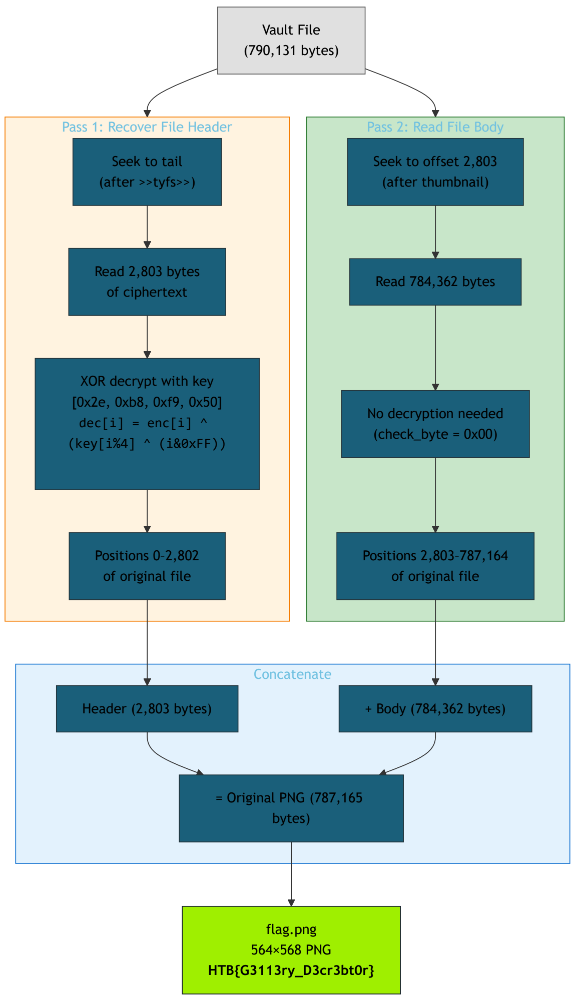
Vault File (790,131 bytes)
Pass 1: Recover File Header
Seek to tail (after >>tyfs>>)
Read 2,803 bytes of ciphertext
XOR decrypt with key [0x2e, 0xb8, 0xf9, 0x50] dec[i] = enc[i] ^ (key[i%4] ^ (i&0xFF))
Positions 0-2,802 of original file
Pass 2: Read File Body
Seek to offset 2,803 (after thumbnail)
Read 784,362 bytes
No decryption needed (check_byte = 0x00)
Positions 2,803-787,164 of original file
Concatenate
Header (2,803 bytes)
+ Body (784,362 bytes)
= Original PNG (787,165 bytes)
flag.png 564×568 PNG HTB{G3113ry_D3cr3bt0r}

Figure 4: The two-pass file reconstruction. The decryption stream reads the encrypted header from the tail first, then the plaintext body from the main region, producing the original 787,165-byte PNG.

8.2 Implementation

```python
from Crypto.Cipher import DES

# Read vault file
with open('b238f4cd-f79f-4e19-be3b-068b7add4d85', 'rb') as f:
    data = f.read()

# Locate markers
tyfs_start = data.find(b'>>tyfs>>')

# Stage 1: Derive XOR key from tail
# (DES key = "tianxiaw", encrypted key at tail offset -18-8-8 = known position)
xor_key = bytes([0x2e, 0xb8, 0xf9, 0x50])  # After DES decryption + PKCS removal

# Stage 2: XOR-decrypt first 2803 bytes from tail
tail_enc = data[tyfs_start + 8 : tyfs_start + 8 + 2803]
header = bytearray(len(tail_enc))
for i in range(len(tail_enc)):
    header[i] = tail_enc[i] ^ (xor_key[i % 4] ^ (i & 0xFF))

# Stage 3: Read plaintext main body (check_byte = 0x00)
body = data[2803:tyfs_start]

# Stage 4: Concatenate
original = bytes(header) + body  # 2803 + 784362 = 787165 bytes
with open('flag_decrypted.png', 'wb') as f:
    f.write(original)
```

8.3 Validation

The reconstructed file passes PNG structure validation with 18 valid chunks:

```
Chunk: IHDR at offset 8, length 13 (564×568, 8-bit RGB)
Chunk: sRGB at offset 33, length 1
Chunk: gAMA at offset 46, length 4
Chunk: pHYs at offset 62, length 9
Chunk: IDAT at offset 83, length 65445
Chunk: IDAT at offset 65540, length 65524
... (12 more IDAT chunks) ...
Chunk: IEND at offset 787153, length 0
PNG complete at offset 787165 (matches expected file size)
```

The image displays a cartoon scene with the flag text overlaid:

`HTB{G3113ry_D3cr3bt0r}`

---

9. Failed Approaches

Documenting failed approaches prevents future analysts from repeating dead ends.

9.1 Direct DES Decryption with Published Keys

Approach: Apply `tianxiaw` and `gallery_` as DES-ECB keys directly to the encrypted data, following the 2019 S-RM and KN4GXD blog posts.

Why it failed: Version 404033 changed the key derivation. `tianxiaw` is no longer the direct DES key—it is a *derived intermediate* (`tianxiawuzei`) obtained by DES-decrypting a hardcoded hex constant with the master key `good_gv` . The published 8-character key was a truncation of the real 12-character key, which itself is only used to derive per-file XOR keys, not to encrypt file data directly.

Lesson: Encryption schemes evolve across app versions. Published exploit code with hardcoded keys becomes stale. Always verify against the actual binary when version numbers differ.

9.2 Treating the Entire Main Body as XOR-Encrypted

Approach: XOR-decrypt all 784,362 bytes of the main body (between thumbnail and `>>tyfs>>`) with the recovered XOR key, using position counters starting from both 0 and 2,803.

Why it failed: The first IDAT chunk (65,445 bytes) appeared valid, but subsequent chunks corrupted. This was because the main body is **not encrypted at all**—the check byte (`0x00`) in the tail metadata indicates plaintext storage. The first IDAT chunk "looked valid" by coincidence (compressed PNG data has high entropy regardless of encryption state).

Lesson: Always check control flags before assuming encryption. A `false` check byte means the data is plaintext despite appearing random (compressed data has high entropy too).

9.3 Attempting PIN-Based Key Derivation

Approach: Hypothesised that the vault PIN (35791), `user_random_number` (44), or `signature` UUID participated in the file encryption key derivation, as suggested by some online forums.

Why it failed: APK decompilation confirmed the PIN is solely an access control mechanism— `SHA1(pin) + MD5(pin)` stored for authentication comparison. No PIN-derived value enters the file encryption path. The `user_random_number` and `signature` are used for cloud sync and device migration, not local file encryption.

Lesson: Decompile the APK rather than speculating about key derivation. The source code is definitive; forum posts are not.

---

10. Conclusion

Flag: `HTB{G3113ry_D3cr3bt0r}`

Attack Chain Summary

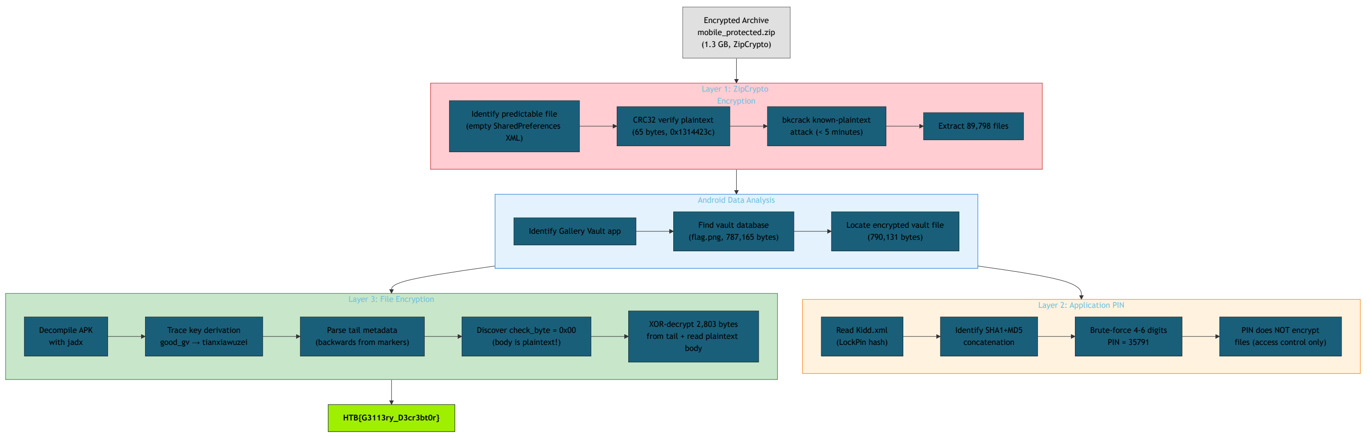
Encrypted Archive
mobile_protected.zip
(1.3 GB, ZipCrypto)
Layer 1: ZipCrypto Encryption
Identify predictable file (empty SharedPreferences XML)
CRC32 verify plaintext (65 bytes, 0x1314423c)
bkcrack known-plaintext attack (< 5 minutes)
Extract 89,798 files
Android Data Analysis
Identify Gallery Vault app
Find vault database (flag.png, 787,165 bytes)
Locate encrypted vault file (790,131 bytes)
Layer 3: File Encryption
Decompile APK with jadx
Trace key derivation good_gv → tianxiawuzei
Parse tail metadata (backwards from markers)
Discover check_byte = 0x00 (body is plaintext!)
XOR-decrypt 2,803 bytes from tail + read plaintext body
Layer 2: Application PIN
Read Kidd.xml (LockPin hash)
Identify SHA1+MD5 concatenation
Brute-force 4-6 digits PIN = 35791
PIN does NOT encrypt files (access control only)
HTB{G3113ry_D3cr3bt0r}

Figure 5: The complete attack chain from encrypted zip to recovered flag, showing the three protection layers and the techniques used to defeat each.

Key Methodological Insights

1. **Known-plaintext attacks on ZipCrypto remain devastating.** A single 65-byte file with predictable content was sufficient to recover encryption keys for an entire 1.3 GB archive containing 90,000 files. ZipCrypto should never be relied upon for confidentiality.
2. **Consumer privacy apps provide a false sense of security.** Gallery Vault's encryption relies entirely on hardcoded keys embedded in the APK. The PIN protects application access, not the encrypted data. Any analyst with the APK can decrypt every file on every device running this version.
3. **APK decompilation is the authoritative source** for mobile forensics involving application-layer encryption. Published research becomes stale as apps update their encryption schemes. The APK itself is always current and always available on the device.
4. **Partial encryption creates forensic shortcuts.** Gallery Vault only encrypts the first ~3 KB of large files (the "header region"), leaving the remaining hundreds of kilobytes in plaintext. This design optimises for mobile performance at the expense of security—and means a forensic examiner only needs to recover a small XOR key to reconstruct the full file.
5. **Control flags in metadata determine encryption scope.** The single check byte (`0x00`) in the tail metadata was the difference between needing to XOR-decrypt 784 KB of data and not needing to decrypt it at all. Parsing metadata structures completely—rather than assuming encryption—saves significant analysis time.

Techniques Added to Library

- **ZipCrypto known-plaintext attack**: CRC32-verified plaintext + bkcrack for key recovery
- **Gallery Vault v404033 decryption**: Full key derivation chain (`good_gv` → `tianxiawuzei` → per-file XOR key)
- **Tail-backwards metadata parsing**: Read from `<<tyfs<<` end marker for field extraction
- **Check byte interpretation**: `0x00` = plaintext main body, `0x01` = XOR-encrypted main body
- **APK-first mobile forensics**: Decompile before applying published exploit code

---

Appendix A: Key Derivation Reference

Stage	Input	Operation	Output
1	`676F6F645F6776` (hex)	Hex decode	`good_gv` (master key string)
2	`good_gv`	Pad to 16 chars, DESKeySpec(first 8)	`good_gv0` (DES key bytes)
3	`5283CBBD2FE1AAF43503D1DDD64DFDD6`	DES-ECB decrypt with `good_gv0`	`tianxiawuzei` (file key string)
4	`tianxiawuzei`	Take first 8 chars	`tianxiaw` (file DES key bytes)
5	Per-file encrypted key (from tail)	DES-ECB decrypt with `tianxiaw`	4-byte XOR key (per file)

Appendix B: File Addresses and Offsets

Location	Value	Description
Vault file path	`data/media/0/.galleryvault_DoNotDelete_1742398726/files/b2/b238f4cd-...`	Encrypted vault file
Thumbnail region	`0x000` – `0xAF2` (2,803 bytes)	300×300 PNG decoy
Main body	`0xAF3` – `0xC02DC` (784,362 bytes)	Plaintext original data (positions 2803+)
`>>tyfs>>` marker	`0xC02DD`	Start of metadata tail
`<<tyfs<<` marker	`0xC0E6B`	End of metadata tail
XOR-encrypted header	`0xC02E5` – `0xC0DE7` (2,803 bytes)	First 2,803 bytes of original file
DES-encrypted XOR key	8 bytes in tail	`9d01dbb2cf9ce19b` → `[0x2e, 0xb8, 0xf9, 0x50]`

Appendix C: Tools Used

Tool	Version	Purpose
bkcrack	1.8.1	ZipCrypto known-plaintext attack
jadx	1.5.1	APK decompilation
Python 3 + pycryptodome	Latest	DES decryption, XOR decryption, tail parsing
sqlite3	—	Gallery Vault database analysis

Docker (Ubuntu 24.04)	—	Isolated forensics environment

Appendix D: Analysis Timeline

Phase	Activity	Outcome
1	Download and identify encrypted zip	ZipCrypto Deflate:Maximum, 90K files
2	Known-plaintext attack with bkcrack	Keys recovered: `890ca696 587c6483 cd091757`
3	Extract and survey Android data	Gallery Vault identified with hidden `flag.png`
4	Apply published decryption methods	**Failed**—keys `tianxiaw` / `gallery_` do not work directly
5	Decompile APK with jadx	Key derivation chain fully recovered
6	Crack vault PIN (35791)	Confirmed PIN irrelevant to file encryption
7	Parse tail metadata backwards	XOR key, check byte, and file offsets extracted
8	Reconstruct original file	Valid 787,165-byte PNG recovered
9	Read flag from image	**`HTB{G3113ry_D3cr3bt0r}`**