Man in the Binder: He Who Controls IPC, Controls the Droid

Nitay Artenstein and Idan Revivo
Malware Research Lab, Check Point

September 29, 2014

Abstract

While built on top of Linux, Android is a unique attempt to combine the principles behind microkernel OS architectures with a standard monolithic kernel. This offers many advantages in terms of flexibility and reliability, but it also creates unique liabilities from a security standpoint. In this paper, we'll have a detailed look at Binder, the all-powerful message passing mechanism in Android, then examine the far-reaching consequences if this component is compromised by an attacker. We'll explain how to parse, utilize and exfiltrate the data passed via Binder, and conclude with a demonstration of three different types of attack previously thought difficult to implement on Android, but shown to be easily done when controlling Binder.

Introduction

In recent years, Android malware authors have put considerable effort into taking their game to the next level: The first Android bootkit[1], the first Android-centric botnet[2] and the first Android ransomware[3] are all a part of this trend.

However, in contrast to the maturity and advanced technical craftsmanship evident in contemporary PC malware, Android malware still seems to be in its infancy. The symptom of this is that malicious code in Android will usually attack at a level which is too high and therefore application-specific and non-portable, or use low-level Linux kernel techniques which focus on subverting the system at a place far below the Android framework, which is where the interesting stuff takes place[4].

The one thing in common to all Android malware found in the wild is that they are not based on deep knowledge of Android internals, and as a result do not integrate well into the Android framework.

[1] `http://blogs.360.cn/360mobile/2014/01/17/oldboot-the-first-bootkit-on-android/`
[2] `http://securelist.com/blog/mobile/57453/`
[3] `http://labs.bitdefender.com/2014/05/reveton-icepol-ransomware-moves-to-android/`
[4] A good example of this approach is this classic article by dong-hoon you from Phrack 0x44: `http://www.phrack.org/issues/68/6.html`

In this paper, we focus on what we believe will be the next target for Android malware authors: A crucial Android-specific system component which forms the main bridge between the high-level application framework and the low-level system layer. Subverting this component allows an attacker to see and control almost all important data being transferred within the system. Welcome to the Binder.

In the following pages, we will give an overview of Binder and its role in the OS architecture, explain why this component is an optimal target for malware, and discuss possible subversion techniques. We will then show how these techniques can be used for taking Android malware to the next stage of its evolution, by either making known-types of attack more global and effective, or by opening the door for new types of attacks not possible before.

This paper assumes that the reader is generally familiar with Android programming, and has basic understanding of how the Android application sandbox is implemented. We also assume that the reader already knows about the DVM, the Zygote, and how managed bytecode interfaces with native code via JNI. We should also note that Binder, and the Android OS as a whole, are complicated subjects. This paper focuses on Binder in the context of black hat techniques that can be used to subvert it. Some additional reading, which discusses miscellaneous aspects of Binder, is suggested at the end of the paper.

Most importantly, all the techniques described in this paper require running with root permissions.

Android and Binder

The architecture of the Android OS can best be understood as a compromise between two opposing philosophies of operating system design: The traditional monolithic kernel approach, which is based on implementing the operating system entirely in supervisor mode and using system calls as the main interface to OS services, and the microkernel approach, which relies on the heavy use of message passing between user applications and user space based system servers that serve, in turn, as the bridge to a minimalist kernel.

From a security standpoint, the main advantage of a microkernel based architecture is that it exposes a smaller attack surface: A normal application has no business speaking directly to the kernel, and the kernel, as a result, needs to handle less untrusted input. The flip side of this is that an attacker does not need access to supervisor mode in order to completely subvert the system. All it takes is control of the system servers or the message passing mechanism.

Binder was designed by Dianne Hackborn as the centerpiece of a hybrid OS architecture. The idea, first implemented in Palm OS Cobalt, was to run a microkernel-inspired, object oriented OS design on top of a traditional monolithic kernel, mainly through the use of Binder as an optimized IPC mechanism which will present an object oriented abstraction of system resources to the upper layers. Google hired Hackborn in 2006, and her ideas greatly influenced Android's architecture.

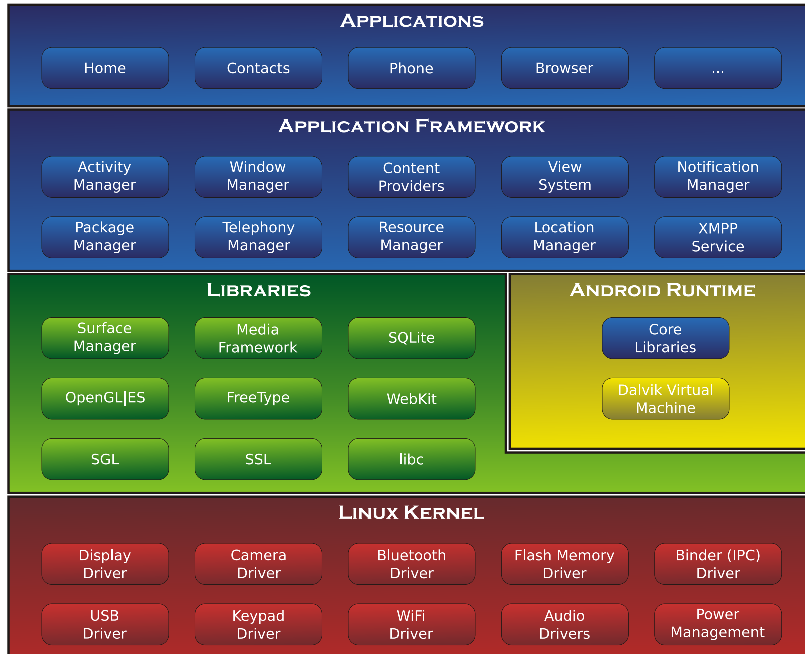

Figure 1: Android's architecture, the classic diagram

To explain Android's architecture, the diagram in figure 1 is normally used, and it is probably familiar to anyone who has ever worked with Android. This diagram is helpful in understanding the basic architecture of the OS: standard user applications (depicted in the uppermost layer) interact with various system services in the Application Framework layer, in what is a classic server-client pattern.

These services, such as the Telephony Manager, the Location Manager, and the View System, provide access to various hardware resources in accordance with the application's permissions. The system servers are the only components with sufficient permissions to interact directly with the kernel and to provide access to the required resources.

This classic diagram, while useful, is of limited assistance to a security researcher, who will require a more fine-grained understanding of how things work.

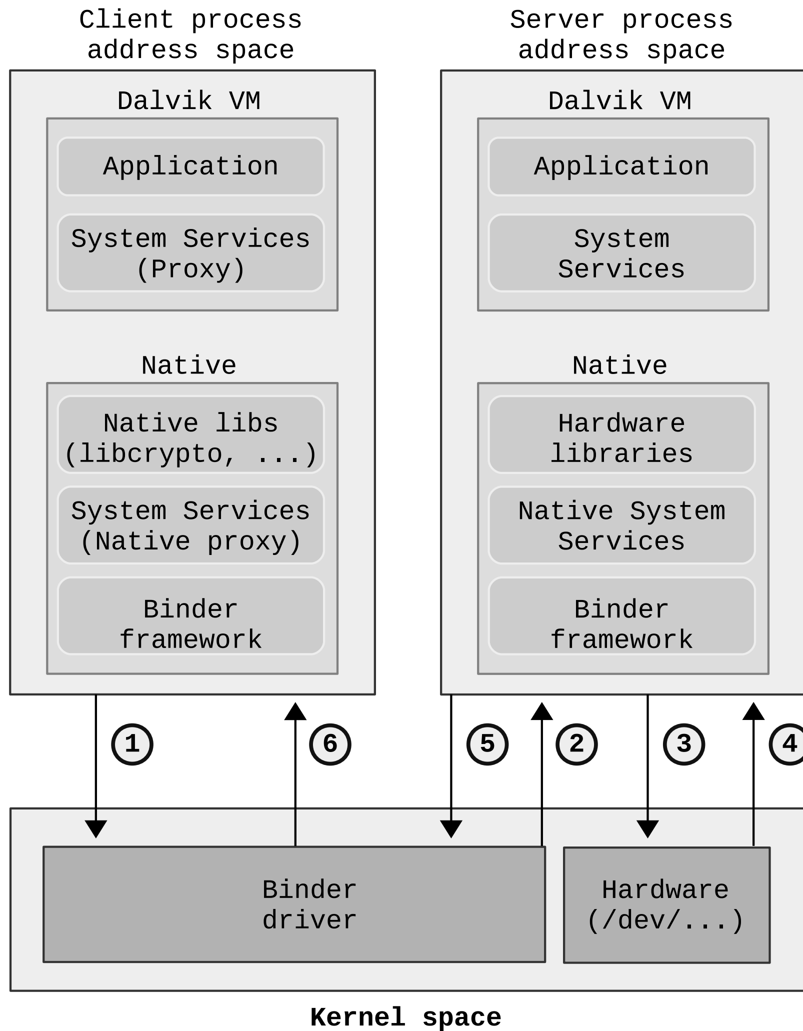

Figure 2: A down-to-earth view of how processes talk to each other

Figure 2 depicts the flow from a low-level, process to process view. We can see that a typical client's address space contains an instance of the DVM, which runs the user application as well as proxies for the Java system services. These Java services communicate with their native counterparts (also proxies) via JNI. Since the system services are not actually implemented in the client's address space, the client will need to use IPC to request a system service from the appropriate server.

This is where Binder comes in. Binder is a two-headed monster: A lion's share of its functionality is implemented in "the Binder framework", a user space library (`libbinder.so`) which is loaded into most processes in Android.

This library handles, among other tasks, most of the grunt work of wrapping and un-wrapping complex objects into simplified, flattened objects referred to as Parcels, before they are sent across to another process or received by it.

The Binder driver, on the other hand, carries out the critical, kernel-level tasks involved in IPC, such as the copying of data from one process to another and maintaining a record of which handle corresponds to which object in a specific process.

Figure 2 illustrates a simplified flow of data between the client and server processes, via the Binder driver. After handling all tasks such as marshalling objects into Parcels, the Binder framework calls an `ioctl` syscall with the file descriptor of `/dev/binder` as a parameter (1) and transfers the relevant data to the kernel. The driver then looks up the required service, copies the data to the system server's address space, then wakes up a waiting thread in the server process to handle the request (2).

After unmarshalling the `Parcel` objects and verifying that the client process has the relevant permissions to carry out the required task (for example, make a network connection), the server performs the requested service, and if necessary calls into the kernel to interact with the relevant hardware (3) and receive a response from it (4). Afterwards, the copy of `libbinder` which is loaded within the server's own address space marshals the response data and sends it back to the driver (5), which hands it back to the client process (6).

It is necessary to keep this architecture in mind when trying to wade your way through the mind-boggling, undocumented swamp that is Android's source code.

The code of a typical service in Android is split into two parts: the service itself and its interface. And since we're discussing IPC, even that is a simplified view. In fact, each interface has a dual implementation - a proxy on the client side and a stub on the server side. This allows a developer creating an app for Android to call services transparently, without even realizing that they are invoking IPC.

A call on the client side for a system service will, under the surface, invoke the corresponding function in the proxy interface. This will trigger a Binder call into the server process, where the stub interface will be waiting for incoming transactions. From that point, the stub interface will call into the actual implementation on the server side, passing the arguments transferred via Binder.

Any type of data can be transferred via a `Parcel`. If the data type is non-primitive (an object), that object will be flattened into what is defined as a "flat binder object". A flat binder object can be used to send across objects of arbitrary complexity as well as file descriptors. The driver performs the heavy work, as it keeps a translation table between pointers to the real objects in the originating process' memory space, and the handles assigned to these objects so that remote processes can refer to them.

A user application running on Android is severly limited in what it can do on its own. Generally speaking, any action outside of its virtualized address space requires interaction with one of dozens of system services. An app on Android may call into Binder, and receive replies from it, thousands of times a minute.

This information highway going into and out of an app will contain massive amounts of data - and an attacker who can tap into this data will immediately gain immense power over the device. Furthermore, controlling the information flow to Binder is a uniquely portable way to steal and modify user data in Android. An attacker does not need to know anything about the implementation of a specific app: regardless of the application's internal complexity, it will eventually have to call a limited set of system services.

A key to Binder's value for attackers is that Android developers and security personnel are generally not aware of the sheer extent of data they are sending across via IPC. For example, developers routinely use Intent objects to send data between different Activities within the same app. Little do they know, however, that by doing so they are in fact using Binder to send their data to the remote process running Activity Manager.

Another example: A developer using HTTPS might assume that the data being sent is encrypted. However, before being sent across the network, the data will first be delivered in plaintext to the Network Manager. How to intercept, and possibly modify, interesting data that is passed through Binder, is the focus of the next section.

Subverting Binder

To control Binder, we first need to find the point where the Binder framework finishes wrapping up the data in Parcels, and passes it on to the driver. Which process we'll choose as our target really depends on our purpose: if we aim to trap the global data flow associated with a single system service - this is something we'd like to do if, say, we want to install a system-wide keylogger - we'll choose to attack the relevant server. If we want to grab the data used by a single client process, and achieve more stealth while we're at it, we'll attack that process alone.

To get a stranglehold on the exact point where the data gets sent to the Binder driver, we'll use the classic library injection technique. After injecting our code into the target process and running from the context of the victim's address space, we'll put a hook in place to divert the control flow to our own code.

The function we'll need to hook is `IPCThreadState::talkWithDriver`, exported by `libbinder`. This function is the only place in the process' address space where a ioctl is being sent to the Binder driver. This is what it looks like:

```
ioctl(mProcess->mDriverFD, BINDER_WRITE_READ, &bwr)
```

Our next steps will require an understanding of the structs used by Binder and the ways these structs are handled. In the above function call, an ioctl is sent to the Binder device's file descriptor. `BINDER_WRITE_READ` is one of several request codes that can be sent to the driver, and the only one which concerns us here. The final argument is the address of a struct of the type `binder_write_read`,

which is the first data structure we'll need to examine. It has the following declaration in `binder.h`:

```
struct binder_write_read {
  signed long write_size; /* bytes to write */
  signed long write_consumed; /* bytes consumed by driver */
  unsigned long write_buffer;
  signed long read_size; /* bytes to read */
  signed long read_consumed; /* bytes consumed by driver */
  unsigned long read_buffer;
};
```

When calling into kernel space, this structure will contain a pointer to a write buffer which will hold the transaction code and its parameters. Upon return from the ioctl call, the read buffer will be filled with the driver's reply, prefixed by a code which corresponds to the type of reply.

The transaction code can be one of a possible range of codes defined in `enum binder_driver_command_protocol`. We are generally interested only in `BC_TRANSACTION`. When this code is at the beginning of the buffer pointed to by `write_buffer`, we know that we are dealing with a Binder transaction, and we can parse the buffer accordingly.

The key struct in a Binder transaction is `struct binder_transaction_data`, declared in `binder.h`:

```
struct binder_transaction_data {
  union {
    size_t handle;
    void *ptr;
  } target;
  void *cookie;
  unsigned int code;
  unsigned int flags;
  pid_t sender_pid;
  uid_t sender_euid;
  size_t data_size;
  size_t offsets_size;
  union {
    struct {
      const void *buffer;
      const void *offsets;
    } ptr;
    uint8_t buf[8];
  } data;
};
```

Let's go over some of the more useful fields in this struct:

`target` - This union will contain either a handle to the referred object if the object is in a remote process' address space, or an actual pointer to the object if it is within the current process' address space. The Binder driver will keep a mapping between each object and its handles, and will do the appropriate translation.

For example, a client process can ask a server process to initialize a certain object which will represent a required service (for instance, an audio recorder). After creating the requested object, the server process will write the object's address to the `target` field, and pass the data to the Binder driver. The driver will then map the pointer to a specific handle, and pass on that handle to the client process.

From this point on, whenever the client process wishes to refer to that object, it will pass the handle back to the Binder driver. The driver will then swap the handle for the actual memory address and pass it on to the server process.

`code` - This is the code of the function which the server is requested by the client to execute. Further on, we will see how to match the value in this field to an actual function, and how to parse the arguments being passed.

`flags` - The flags for this bitfield are defined as follows:

```
TF_ONE_WAY = 0x01; /* this is a one-way call */
TF_ROOT_OBJECT = 0x04; /* the component's root object */
TF_STATUS_CODE = 0x08; /* contents are a 32-bit status code */
TF_ACCEPT_FDS = 0x10; /* allow replies with file descriptors */
```

The flags we'll usually encounter in the transactions we wish to intercept are `TF_ACCEPT_FDS`, signifying that file descriptors can be passed within flat binder objects, and `TF_ONE_WAY`, which means that we should not wait for a reply after performing the transaction.

`data` - This is the most important member, as it points us to the actual data buffer being sent. First of all, you can generally ignore the fact that this is a union, as the `buf` union member is very rarely used in the types of transactions that we care about. Focusing on the `ptr` struct, we can see that it contains two pointers: `buffer` and `offsets`. `buffer` holds a pointer to the raw data sent via Binder. `offsets` points to a separate buffer, which holds the positions within `buffer` in which we'll find flat binder objects that Binder will need to convert to real objects.

As described above, `data.ptr.buffer` points to the buffer which holds all the good stuff that we want. Understanding how to read it is our next goal. And the best way to achieve that goal is to focus on a real transaction - a Media Player function call passed from the proxy interface on the client process to the stub interface on the server process, and from there to the actual implementation.

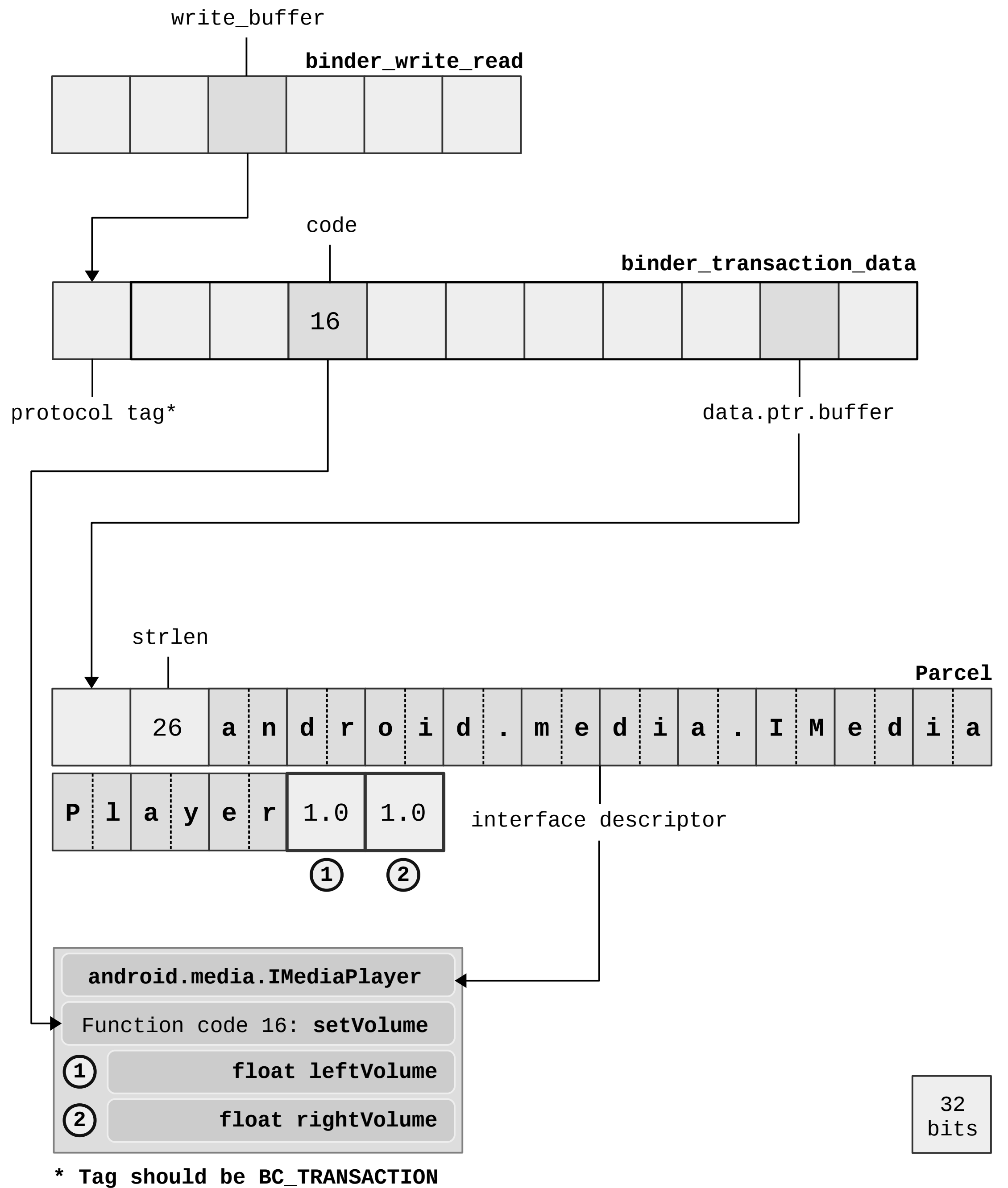

Figure 3: Dissecting a typical Binder transaction

Unwrapping A Parcel Object

The buffer that we'll look at next is in fact a `Parcel` object. And like every parcel that has a right to expect that it will reach its destination, this one has an address stamped on it. The interface descriptor is a 16-bit Unicode string that is appended to the start of a `Parcel` and simply states which service it is being delivered to. We'll just note in passing that Binder uses an elaborate system to determine where to deliver the data, and does not depend upon this string, which was added as a security measure.[5]

In the example given in figure 3, the interface descriptor appended to the beginning of the `Parcel` is `android.media.IMediaPlayer`. The IMediaPlayer interface passes requests to the almighty Media Player service, which is one of the main dispensers of audio output in Android.

Let's focus on figure 3 for a moment, and read it in lockstep with this walkthrough on how to understand the data being passed in a `Parcel`. A `Parcel` object has no pre-defined size: A variable amount of data is stored in each `Parcel`, in accordance with the number and types of arguments required by the remote function being invoked.

So, we only need to figure out the prototype of the function being called. That could be easy or hard, depending on whether or not we have the source code. And since this is Android, we usually will have the source.[6]

Referring back to the example in figure 3, we can see that the `code` member in the `binder_transaction_data` struct is, in this sample, 16. Let's open the source code of the interface that's being talked to - in this case, IMediaPlayer - and dig in.[7]

First of all, let's go for the easy catch. At the beginning of the file, you'll find an enum similar or identical to the one below. Here it is with line numberings in comments:

```
enum {
  /* 1 */ DISCONNECT =
            IBinder::FIRST_CALL_TRANSACTION, // Defined as 1
  /* 2 */ SET_DATA_SOURCE_URL,
  /* 3 */ SET_DATA_SOURCE_FD,
  /* 4 */ SET_DATA_SOURCE_STREAM,
  /* 5 */ PREPARE_ASYNC,
  /* 6 */ START,
  [...]
  /* 16 */ SET_VOLUME
};
```

[5] While out of scope for this paper, further details on Cross-Binder Reference Forgery (XBRF) attacks can be found at: `http://crypto.hyperlink.cz/files/xbinder.pdf`

[6] If you are dealing with a unique build for which no source code is available, the best approach to take is to still rely on the official source code from AOSP, while looking for any small difference in implementation that might arise. It is extremely rare to see significant changes to the code at this fundamental level of the architecture.

[7] The full path to the source file is `frameworks/av/media/libmedia/IMediaPlayer.cpp`

Each member of the enum corresponds to an action that is performed by the service. There is an exact correspondence between the order of the members in the enum and the code member in the binder_transaction_data struct. Here, member number 16 (which is the code we are looking for) is SET_VOLUME.

Let's take a step back and see how this is translated into a real function call. We can see that the source code file contains implementations for two classes: BnMediaPlayer ("Bn" stands for "Binder native"), which contains the stub interface in the server process, and BpMediaPlayer ("Binder proxy"), which contains the proxy interface in the client process.

This code gives us a very good general idea on how IPC works at the native level: The Binder proxy class exposes a variety of methods, such as disconnect() and setDataSource() (as seen in the enum above), which are called in the client process via JNI. The method in the proxy class then performs a Binder transaction that contains the function code and the argument data. On the other side of the divide, the stub interface in the server process receives the transaction, parses the Parcel, and calls the relevant function in the implementing classes, where the real work of the service is being done.

A few code snippets should make this clearer. Let's begin at the top, and see the call chain that eventually triggered our sample transaction in figure 3. Here is how the code is called from the managed MediaPlayer class in Java:

```
public native void setVolume(float leftVolume,
                             float rightVolume);
```

So, whenever any Java code instantiates a MediaPlayer class, then calls its setVolume method, what it really does is call a native method via JNI. Let's go another step down the ladder, to the native level, and look at the real code that's being run. The code we're looking for is in the BpMediaPlayer class which we observed earlier:

```
status_t setVolume(float leftVolume, float rightVolume) {
  Parcel data, reply;
  data.writeInterfaceToken(IMediaPlayer::getInterfaceDescriptor());
  data.writeFloat(leftVolume);
  data.writeFloat(rightVolume);
  remote()->transact(SET_VOLUME, data, &reply);
  return reply.readInt32();
}
```

This code, which runs on the client process as part of the proxy interface, prepares the transaction and sends it across to the server process in these simple steps:

1. It initializes two Parcel objects on the stack: one for sending the transaction data, one for getting a reply.

2. It writes an interface token (which is composed of a 32-bit integer which defines a "strict mode policy" - out of scope for our purposes - followed by the interface descriptor) to the `data` buffer.

3. It writes the two arguments required for this function call - `leftVolume` and `rightVolume` - to the `data` buffer.

4. It retrieves a reference to the remote service being called, then calls its `transact` member. This method prepares the `binder_transaction_data` struct, as well as `binder_write_read`, with all relevant pointers and members; this is how the `SET_VOLUME` value ends up as the value in the `code` member in our struct in figure 3.

5. `transact` then calls an ioctl into Binder, and the transaction is sent to the server process.

If you'll look at the `Parcel` buffer in figure 3, you should see that it reflects the steps carried out by the code above: The buffer is composed of an interface descriptor, followed by the two arguments, `leftVolume` and `rightVolume`.

To close the circle, let's see what happens on the other side, when the transaction reaches the server process. This code is from the `onTransact` method of the `BnMediaPlayer` class, which implements the stub interface on the server process side.

```
status_t BnMediaPlayer::onTransact( uint32_t code,
          const Parcel& data, Parcel* reply, uint32_t flags) {
  switch (code) {
    [...]
    case SET_VOLUME: {
      CHECK_INTERFACE(IMediaPlayer, data, reply);
      float leftVolume = data.readFloat();
      float rightVolume = data.readFloat();
      reply->writeInt32(setVolume(leftVolume, rightVolume));
      return NO_ERROR;
    } break;
```

This an exact mirror image of what we saw in the proxy interface, except that the `setVolume` method called now is a member of the class `MediaPlayer`, which is responsible for the actual implementation of Android's media player.

And now, at the end of this journey, you should have a pretty good idea of how IPC works in Android on the userland level, and how to read the `Parcel` buffer and understand the data being sent.

We're now prepared for the fun part: attacking the system. But before we take off our gloves, please note: We're only touching the tip of the iceberg. We'll grab data from the most obvious places, mainly to show how versatile is this method of attack. However, you can easily see that the possibilities are almost endless. The only limit is how imaginative you're willing to be, and how far you want to dig into the system's internals.

First Attack: Keylogger

Keyloggers have always been problematic on Android. Not equipped with adequate knowledge of the system's internals, malware authors had a hard time understanding where and how to trap a user's keyboard input.

A prevalent approach currently used by malware is to replace the default keyboard with a custom keyboard application which saves the entered input to a buffer or sends it back to the C&C server. However, this attack is easily detected, even by a non-technical user. Using a Man in the Binder attack would be a much more robust and stealthy solution.

To receive keyboard data, an application has to register with an Input Method Editor (IME) server. An IME is the actual keyboard implementation in Android; A user can swap IMEs and install new ones, but only one IME can be enabled at a time. The default IME in most Android images is `com.android.inputmethod.latin`.

When an application registers with a server - in this case the IME - to receive data from it, the client/server model we described earlier is turned on its head: Now, the application becomes the server, and the server becomes its client. Whenever new data is available for the application, the service delivers the goods by using Binder to call into a callback method initialized by the application. In this case, the arguments for this callback will be our keyboard input.

To grab data in this new model, we have to take our Binder kung-fu up a notch: We no longer need to intercept the data going down from the application into Binder - we have to intercept the reply tossed up from Binder to the app. To do so, we'll need to do something along the lines of this pseudo-code:

```
int hooked_ioctl(int fd, int cmd, void *data) {
  do_evil_deeds_with_transaction(data);
  int ret = ioctl(fd, cmd, data);
  do_evil_deeds_with_reply(data);
  return ret;
}
```

Parsing the reply buffer isn't different from parsing the transaction data. The first four bytes of the reply are a code that tells us what kind of buffer to expect. The protocol codes are defined in `enum binder_driver_return_protocol`. We are generally only interested in `BR_REPLY` and `BR_TRANSACTION`.

`BR_REPLY` is a raw buffer that doesn't have any specific format: it contains the return data from a function we have called. A `BR_TRANSACTION`, on the other hand, holds the information sent from a service when we have registered a callback method to handle incoming data.

This is the kind of transaction we'll get when we receive keyboard data from the IME, and it is parsed exactly the same as a `BC_TRANSACTION`, as in figure 3 above. We are merely looking at the other side of the transaction, this time from the server angle.

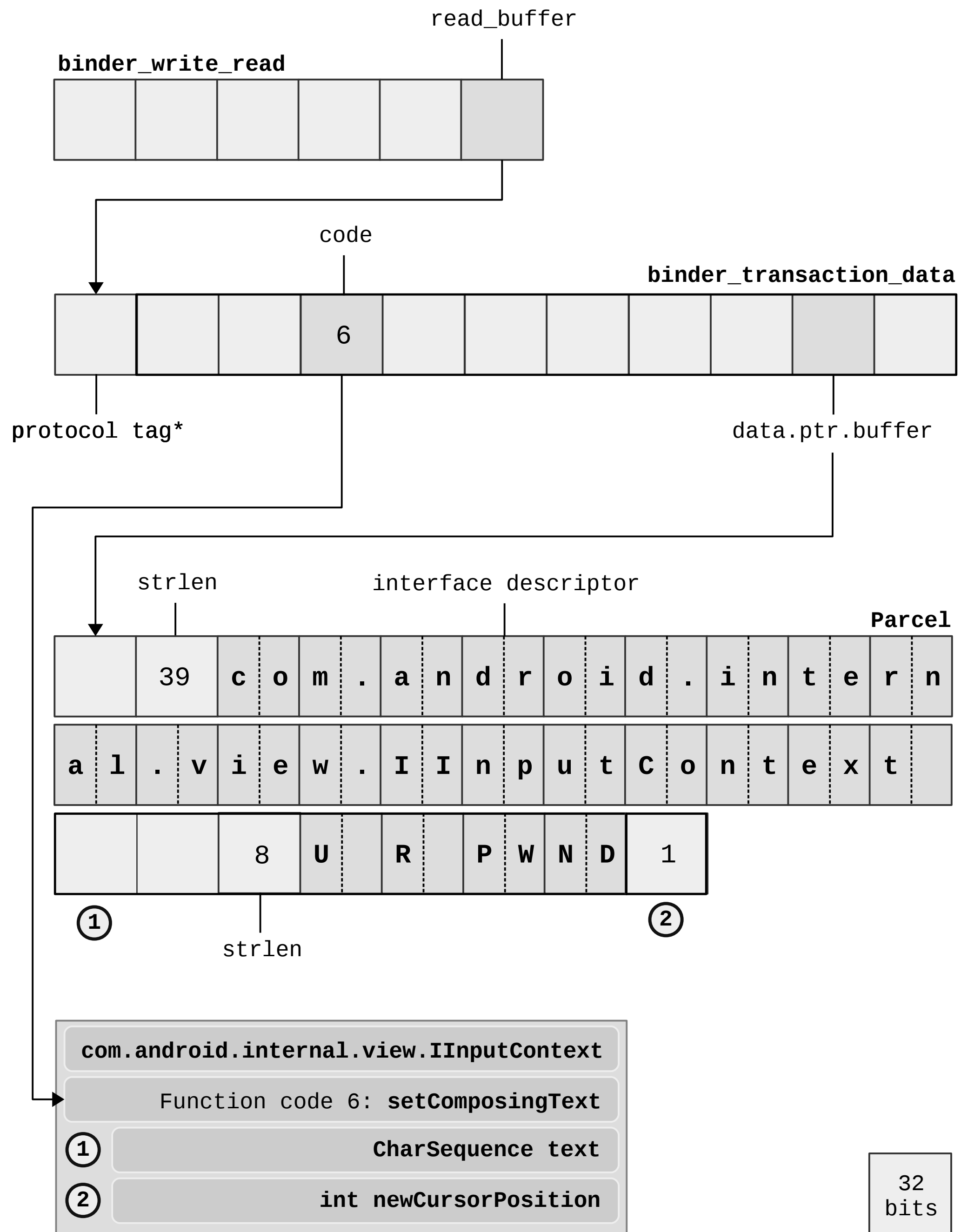

Figure 4: Getting at that juicy keyboard data

By hooking all Binder calls in the application process, and typing some test input, you'll quickly find exactly what you're looking for. The buffer you'll see is illustrated in figure 4, which demonstrates how we would parse a transaction coming in from the IME.

As you can see, `read_buffer` in `binder_write_read` points to the reply we got via Binder. Parsing the beginning of the buffer, we know that we need to continue if the protocol code is `BR_TRANSACTION`. The remainder of this buffer can be parsed as a `struct binder_transaction_data`.

The Binder packet which contains the keypress data is delivered to an internal interface called `com.android.internal.view.IInputContext`. This interface sends the data up to the `InputContext` class, which handles received keyboard input within the client process.

Remember that in this transaction, the client process acts as a server, and receives calls from the IME whenever keyboard strokes are detected. Each time a key is pressed, another callback to IInputContext is triggered, and a fresh buffer is sent via Binder. So we know that the target of this transaction is `IInputContext`. Digging through the source code, we'll find that `IInputContext` has no actual implementation; the only thing we have is an AIDL file.

AIDL is a domain-specific language meant for defining Android IPC interfaces. The methods which can be called in a remote process are defined as a series of function prototypes. When an AIDL file is processed by the AIDL compiler, it generates the required native classes that together form the proxy interface on the client side and the stub interface on the server side.

Looking at the `code` member in the `binder_transaction_data` struct, we see that the function code for the transaction in question is 6. How do we connect this number to a real function prototype? The key is in `IInputContext.aidl`, the AIDL source file.

The AIDL compiler will enumerate the function prototypes in the source file one by one, then generate code similar to the snippets we've seen in the Media Player example. There is a direct correspondence between the order of the functions in the AIDL file and the function code as it appears in the transaction. Let's have a look at the AIDL source code for `IInputContext`, with function numbers in the comments:

```
oneway interface IInputContext {
  /* 1 */ void getTextBeforeCursor(int length, int flags,
                int seq, IInputContextCallback callback);
  /* 2 */ void getTextAfterCursor(int length, int flags,
                int seq, IInputContextCallback callback);
  /* 3 */ void getCursorCapsMode(int reqModes, int seq,
                IInputContextCallback callback);
  /* 4 */ void getExtractedText(ExtractedTextRequest request,
                int flags, int seq,
                IInputContextCallback callback);
  /* 5 */ void deleteSurroundingText(int leftLength,
```

```
                  int rightLength);
  /* 6 */ void setComposingText(CharSequence text,
                  int newCursorPosition);
  [...]
}
```

As you can see, the code 6 corresponds to the prototype for `setComposingText`. We can now parse the buffer from the point after the interface descriptor. The first argument, which implements the `CharSequence` interface, holds a member for the string length, followed by the string itself as an array of 16-bit characters. It is here that we'll find the keyboard input. Q.E.D.

Now let's continue to something more elaborate.

Second Attack: Playing with in-app data

There is a dark secret common to all Android applications. Android developers who try to understand how the system works are often quite surprised at the sheer extent of their application's use of IPC. But even this is not the whole truth: Hardly anyone is aware that an Android app uses Binder to pass data *within the same application.*

A typical Android application of reasonable scale is divided into dozens of different Activities. An Activity is simply an interaction unit within the application, composed of a GUI and the logic used to manage it. From a 10,000 foot view, an Android app is merely a collection of Activities that transfer data among each other, occasionally receiving input from the user or communicating over the network.

Let's look at a security-critical application - a banking app that, among other things, allows a user to transfer money from their account into a different account. Such functionality will normally be implemented in three Activities: The first Activity would let the user type in the recipient's account details; The second Activity would ask the user to confirm the details; And a third Activity would present confirmation that the transaction was made.

Most serious banking apps take precautions that aim to minimize the damage that can be done when the application is running on a compromised device - for example, working with an in-app keyboard which obviates the need for communicating with the IME, or encrypting data within the app before handing it to the Network Manager for delivery to the bank's servers.

However, these measures are mostly copied wholesale from Windows anti-malware methods; they are not based on understanding the factors unique to the Android OS.

In this case, we need to understand that the Activity Manager is the only component of the system that is permitted to initiate new Activities, whether within an individual app, or across different apps. This means that to start a new Activity within the same process, your application has to make a Binder

call to Activity Manager and pass it all the data that needs to be available to the second in-app Activity.

And this is where secure apps often fail: Not aware of what is really taking place, even a security-minded developer will inadvertently pass sensitive data, in plaintext, via Binder. In the case of our cookie-cutter banking application, what would happen is that after implementing a secure keyboard, and encrypting the account details within the application so they cannot be tampered with, the account details will still be sent in plaintext to the Activity Manager. An attacker controlling Binder could easily modify the account details, the amount, or anything else, while they are in transit.

To show a simple example of how this works, we whipped up a mock banking application which idiotically does nothing except query the user for an amount to be transferred in a transaction, then send that amount to an Activity called `TransactionActivity`, which actually carries out the transaction. By hooking all the app's Binder transactions which were meant for the Activity Manager, and dumping their raw contents to file, we got the output in figure 5.


```
0x000000: 04 03 00 00 1c 00 00 00 61 00 6e 00 64 00 72 00 ........a.n.d.r.
0x000010: 6f 00 69 00 64 00 2e 00 61 00 70 00 70 00 2e 00 o.i.d...a.p.p...
0x000020: 49 00 41 00 63 00 74 00 69 00 76 00 69 00 74 00 I.A.c.t.i.v.i.t.
0x000030: 79 00 4d 00 61 00 6e 00 61 00 67 00 65 00 72 00 y.M.a.n.a.g.e.r.
0x000040: 00 00 00 00 85 2a 62 73 7f 01 00 00 e0 42 4d 71 .....*bs....?BMq
0x000050: c0 42 4d 71 0d 00 00 00 63 00 6f 00 6d 00 2e 00 ?BMq....c.o.m...
0x000060: 62 00 61 00 6e 00 6b 00 2e 00 74 00 65 00 73 00 b.a.n.k...t.e.s.
0x000070: 74 00 00 00 ff ff ff ff 00 00 00 00 ff ff ff ff t...????....????
0x000080: 00 00 00 00 ff ff ff ff 0d 00 00 00 63 00 6f 00 ....????....c.o.
0x000090: 6d 00 2e 00 62 00 61 00 6e 00 6b 00 2e 00 74 00 m...b.a.n.k...t.
0x0000a0: 65 00 73 00 74 00 00 00 21 00 00 00 63 00 6f 00 e.s.t...!...c.o.
0x0000b0: 6d 00 2e 00 62 00 61 00 6e 00 6b 00 2e 00 74 00 m...b.a.n.k...t.
0x0000c0: 65 00 73 00 74 00 2e 00 54 00 72 00 61 00 6e 00 e.s.t...T.r.a.n.
0x0000d0: 73 00 61 00 63 00 74 00 69 00 6f 00 6e 00 41 00 s.a.c.t.i.o.n.A.
0x0000e0: 63 00 74 00 69 00 76 00 69 00 74 00 79 00 00 00 c.t.i.v.i.t.y...
0x0000f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0x000100: 68 00 00 00 42 4e 44 4c 01 00 00 00 00 00 00 00 h...BNDL........
0x000110: 06 00 00 00 61 00 6d 00 6f 00 75 00 6e 00 74 00 ....a.m.o.u.n.t.
0x000120: 00 00 00 00 0a 00 00 00 00 00 00 00 12 00 00 00 ................
0x000130: 41 00 20 00 74 00 68 00 6F 00 75 00 73 00 61 00 A. .t.h.o.u.s.a.
0x000140: 6E 00 64 00 20 00 64 00 6F 00 6C 00 6C 00 61 00 n.d. .d.o.l.l.a.
0x000150: 72 00 73 00 00 00 00 00 14 00 00 00 00 00 00 00 r.s.............
0x000160: 00 00 00 00 12 00 00 00 21 00 00 00 00 00 00 00 ........!.......
0x000170: ff ff ff ff 85 2a 68 73 7f 01 00 00 05 00 00 00 ????.*hs........
0x000180: 00 00 00 00 ff ff ff ff ff ff ff ff 00 00 00 00 ....????????....
0x000190: ff ff ff ff 00 00 00 00 00 00 00 00             ????........
```


Figure 5: Raw data in transit to Activity Manager

This raw hex dump can seem a little disconcerting at first, so let's see how we can take it apart.

By dissecting the parameters in the `binder_transaction_data` struct as we did in previous samples, we can determine that this `Parcel` buffer holds the arguments for a function whose code is 3. Figuring out what is the function in

question is the first step towards parsing the blob above. In the case of the ActivityManager, the IPC protocol is defined in Java, via the IActivityManager interface, which is implemented by the ActivityManagerNative class.

These definitions in IActivityManager.java give us a big hint:

```
int START_RUNNING_TRANSACTION =
      IBinder.FIRST_CALL_TRANSACTION;
int HANDLE_APPLICATION_CRASH_TRANSACTION =
      IBinder.FIRST_CALL_TRANSACTION+1;
int START_ACTIVITY_TRANSACTION =
      IBinder.FIRST_CALL_TRANSACTION+2;
```

Bearing in mind that FIRST_CALL_TRANSACTION is defined as 1, we can determine that the function code corresponds to START_ACTIVITY_TRANSACTION. Our next steps are to check the implementation in ActivityManagerNative.java, find its onTransact() method, and happily read this code, which tells us exactly how to parse our blob:

```
public boolean onTransact(int code, Parcel data,
                Parcel reply, int flags) {
  switch (code) {
    case START_ACTIVITY_TRANSACTION: {
      data.enforceInterface(IActivityManager.descriptor);
      IBinder b = data.readStrongBinder();
      IApplicationThread app =
        ApplicationThreadNative.asInterface(b);
      String callingPackage = data.readString();
      Intent intent = Intent.CREATOR.createFromParcel(data);
      [...]
      int result = startActivity(app, callingPackage, intent,
        resolvedType, resultTo, resultWho, requestCode,
        startFlags, profileFile, profileFd, options);
      [...]
```

In the above code, we see that an Intent is generated from the Parcel data. Grabbing it is just what we're after, since Intent is the abstraction that is used to pass data and requests to the Activity Manager. The Intent constructor is too elaborate for our current scope, but you are welcome to check out its source at Intent.java.

For our purposes, we need to know that the Intent holds the name of the Activity being started (in this case, com.bank.test.TransactionActivity), and that in the above buffer, we can easily see that the Intent's Bundle object starts at offset 0x100. A Bundle object is used to transfer key/value pairs within Intents. It's identified by its magic number, the ASCII sequence "BNDL", preceded by the size of the object in bytes.

We show the general method of parsing it in figure 6. And we couldn't resist having some fun with the transferred amount.

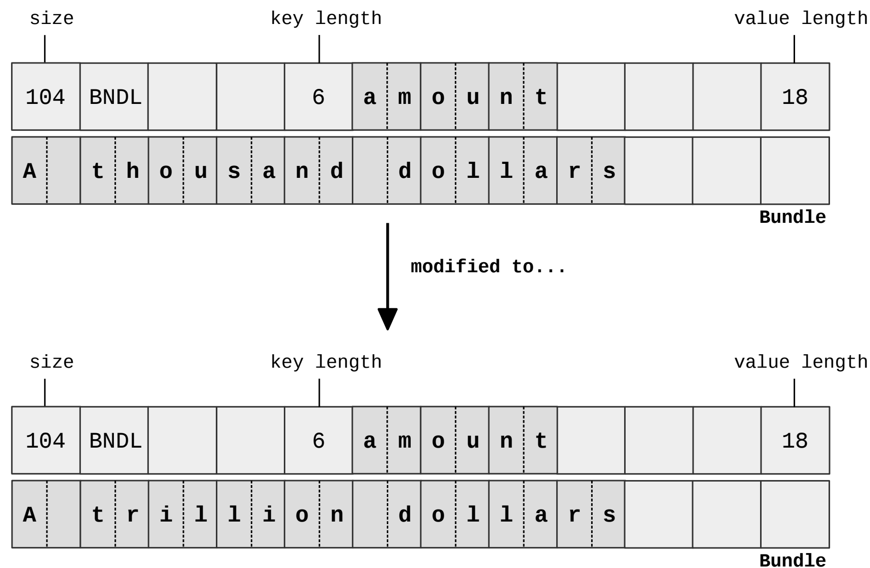

Figure 6: Caught in transit: Capturing and changing in-app data

As you can see from figure 6, the amount to be transferred was stored by the banking application in a key/value pair, then wrapped up in a `Bundle` object. This `Bundle` object, in turn wrapped up in an `Intent`, is what gets sent via Binder to the Activity Manager. The Activity Manager then delivers this data to the new Activity as it is started.

This is the main mechanism used to build persistence between different Activities in a single app. Grabbing the values passed around in this manner, and possibly modifying them, is quite simple once an attacker knows what to look for. In the case of a banking app, we can modify the values in transit, and (for example) transfer arbitrary sums to an account of our choosing. And since this data is usually not encrypted, this is literally a back door into the user's data.

Third Attack: SMS Grabbing Made Easy

Traditionally, malware on Android would steal or intercept SMS messages on the device by simply trying to fool the user into installing a rogue app with the `SEND_SMS` and `RECEIVE_SMS` permissions. Other than the fact that it relies on social engineering, the weakness of this technique is that an anti-malware application installed on the device could easily detect that something is amiss and prompt the user to uninstall the suspect app. Using a Man in the Binder attack would circumvent these problems, but first we require better understanding of how SMS messages are handled within the system.

The first place where an SMS reaches userland in Android is via the RIL

layer.[8] From there, the SMS data is passed to the Telephony Provider, running as the process com.android.phone. This is done through a Unix domain socket, not through Binder. com.android.phone then sets itself up as a Content Provider for the SMS data.

A Content Provider in Android is any component that chooses to respond to queries from other processes and provide information, based on adequate permissions. Requests to a Content Provider are usually translated to (or originally formatted as) SQL queries. The Provider queries its database, and if possible returns a result. This result is abstracted into a Cursor object, which "points" at the correct rows in the database. Under the surface, a Cursor is in fact just a piece of shared memory where the result data is copied, then passed to the querying process.

And here it is again, the same client-server architecture that is prevalent throughout Android. And who's the client listening for the data on the other side? This would be any application which is registered to receive SMS. In most system images, this would be the default SMS/MMS application that comes with the device, running as com.android.mms.

And now, three guesses how Content Provider queries and replies are sent between processes.[9]

So, our preferred tactic for intercepting SMS is to put ourselves in the middle of the communication between com.android.phone and com.android.mms. com.android.mms sets up a thread to act as a Content Observer, and receives notifications (via the BR_TRANSACTION mechanism as discussed above) whenever an SMS has been received. The client process then initiates a transaction with the Telephony Provider. Identifying this transaction is easy, since the interface descriptor is IContentProvider[10], with a function code of 1. This corresponds to the following function in IContentProvider.java:

```
public Cursor query(String callingPkg, Uri url,
                String[] projection, String selection,
                String[] selectionArgs, String sortOrder,
                ICancellationSignal cancellationSignal)
```

For our purposes, what we really care about is that the return value of this function is a Cursor object. And since a Cursor object is in fact an abstraction for a file descriptor[11], let's see how we obtain that file descriptor and read the

[8] Currently, the best place to learn more about the Radio Interface Layer is "Android Hacker's Handbook", Chapter 11 ("Attacking the Radio Interface Layer")

[9] Uh, Binder

[10] Note that any process can register itself as a Content Provider, so there are situations in which Binder transactions with this same interface descriptor will be directed at several different processes. Binder identifies which service to communicate with by looking at the handle field of the binder_transaction_data struct, not at the interface descriptor. However, the interface descriptor would normally be enough in order to identify which server is being talked to.

[11] The file descriptor is a handle to /dev/ashmem, which is an Android-specific implementation of shared memory, mainly meant to facilitate the translation of shared memory regions to file descriptors that can be transferred from one process to

reply sent from the Telephony Provider database. This is the place to mention that structs of type `flat_binder_object`, which are normally used to enable one process to refer to objects that exist in another process (Binder will assign handles to each object), can also be used to transfer file descriptors from one process to another. The Binder driver will do the necessary translation.

Take a look at what a `flat_binder_object` type looks like:

```
struct flat_binder_object {
  unsigned long type;
  unsigned long flags;
  union {
    void *binder; /* local object */
    signed long handle; /* remote object */
  };
  void *cookie;
};
```

Flat binder objects that hold a file descriptor are preceded with a type code of `BINDER_TYPE_FD` (defined in `binder.h`). In that case, the file descriptor itself will be held in the `binder` member (defined here as a union). The reply for the `query()` function is pretty long and complex, but since all you really need is the file descriptor, you could just parse it as follows:

```
int i, temp;
for (i = 0; i < reply_length; i++) {
  temp = *(reply_buffer + i); // reply_buffer is a uint32_t*
  if (temp == BINDER_TYPE_FD) {
    got_fd = true;
    break;
  }
}
if (got_fd) {
  flat_binder_object* fbo =
    (flat_binder_object *) (reply_buffer + i);
}
int fd = fbo->binder;
```

From this point, you can just read the Content Provider's reply as you would read any Linux file. The hex dump in figure 7 shows the results of reading this file.

---

another. Information about ashmem is pretty scarce, but here's a nice intro: `http://notjustburritos.tumblr.com/post/21442138796/`

```
0x000000: 00 00 00 00 01 00 00 00 01 00 00 00 00 00 00 00 ................
0x000010: 01 00 00 00 50 c3 65 b6 48 01 00 00 03 00 00 00 ....P??H.......
0x000020: ec 01 00 00 0e 00 00 00 00 00 00 00 00 00 00 00 ?..............
0x000030: 00 00 00 00 03 00 00 00 fa 01 00 00 42 00 00 00 ........?...B...
0x000040: 01 00 00 00 00 00 00 00 00 00 00 00 2b 39 37 32 ............+972
0x000050: 35 38 36 32 32 31 37 30 31 00 43 6f 6d 65 20 6f 586221701.Come o
0x000060: 6e 2c 20 43 6f 68 61 61 67 65 6e 21 20 59 6f 75 n, Cohaagen! You
0x000070: 20 67 6f 74 20 77 68 61 74 20 79 6f 75 20 77 61  got what you wa
0x000080: 6e 74 2e 20 47 69 76 65 20 74 68 6f 73 65 20 70 nt. Give those p
0x000090: 65 6f 70 6c 65 20 61 69 72 21 20 00 01 00 00 00 eople air! .....
```

Figure 7: The intercepted SMS. To be read aloud in a heavy accent

It is even easier to intercept an outgoing SMS (at least when the message is sent from the default application, `com.android.mms`), since in this case there are no file descriptors involved. The client process simply initiates a transaction with the Telephony Provider using a function code of 3, which corresponds to the `insert`() function. This function inserts an SMS into the main database, and at the same time triggers the sending procedure. The full text contents of the SMS are sent as the arguments to this function. Parsing them, and extracting the SMS text, is left as an exercise to the reader.

Conclusion

In this paper, we have described Binder from a security angle, and demonstrated how its functionality could be subverted and integrated into a new kind of Android malware.

Defending against this threat is a complex task. As is evident from the attack descriptions, each attack can have several different implementations: The client side of a transaction could be attacked, or the server side; The ioctl function itself could be hooked, or any function which is above it in the call chain. For example, an attacker could hook a specific server if they suspect that the target application is performing some security checks.

The prevalence of Binder in Android means that a lot more data is sent via IPC than you might suspect. To properly defend an application, it is first necessary to thoroughly audit its IPC transactions (using methods such as those we have described above), and then encrypting any critical information within the application before it gets sent to Binder. This should particularly include any data which is sent between different Activities of the same app.

To protect against keylogging attacks, an application should also implement its own keyboard with the application context. This should be done carefully, since the in-app keyboard itself could be leaking data if the keyboard output is sent unencrypted via Activity Manager. SMS grabbing is hard to protect against if your application is using plaintext SMS. To properly defend against this threat, it is necessary to use a binary SMS and encrypt its contents.

Throughout this paper, we have been stressing the idea that Android is a unique operating system, a result of several architectural concepts that have never been realized before on such a massive scale. The threats it is facing are, likewise, unique. The security community still does not have the depth of understanding in Android's internals that is required to adequately defend the system. Now is the time to change that.

References

[1] Aristide Fattori, Kimberly Tam, Salahuddin J. Khan, Lorenzo Cavallaro and Alessandro Reina. "On the Reconstruction of Android Malware Behaviors". This is pioneering work which uses Binder as a central component of an Android malware analysis system. http://www.isg.rhul.ac.uk/sullivan/pubs/tr/MA-2014-01.pdf

[2] Aleksandar (Saša) Gargenta, "Deep Dive into Android IPC/Binder Framework". This is the go-to resource for anybody wishing to get started with Binder. Also watch the video. https://thenewcircle.com/s/post/1340/Deep_Dive_Into_Binder_Presentation.htm

[3] Constanze Hausner, "Binderwall: Monitoring and Filtering Android Interprocess Communication". This thesis offers rich technical information about Binder, while focusing more on the defensive side. https://www.sec.in.tum.de/assets/Uploads/MAConstanzeHausner.pdf

[4] Thorsten Schreiber, "Android Binder". A shorter, more general work, but good for an overview of Binder. http://www.nds.rub.de/media/attachments/files/2012/03/binder.pdf