Android Binder Security Note
On >Passing Binder Through Another Binder<

(extended abstract, November 11th 2011)

updated: November 16th 2011

Tomáš Rosa

http://crypto.hyperlink.cz

Binder related sub-page: http://crypto.hyperlink.cz/abinder.htm

Keywords: Android, binder, OpenBinder, security, cross-binder reference forgery (XBRF).

This memo describes certain details of the Android mechanism of *passing binder through another binder*. The aim of this paper is to describe certain technical details of this mechanism as well as to point out some obvious security weaknesses. For the sake of simplicity, we do not include description of the whole Android binder framework here. The interested reader may check the OpenBinder documentation by Dianne K. Hackborn that is still available in [1]. Despite it not being compatible with the Android binder framework, a lot of OpenBinder general ideas seem to still apply.

Before describing our observations, one clarification is apposite: OpenBinder is obviously more general concept. Therefore, when Android documentation talks about *calling a remote binder* and *passing on a binder reference*, OpenBinder documentation talks about *invoking a remote object method* and *passing on an object reference*, respectively. Obviously, the OpenBinder terminology describes more clearly what is going on when we call/pass a "binder". Anyway, if Android people ever talk about a *binder* they are often referring to `android.os.IBinder` Java interface or, in particular, to the code that is serving the remote side of the binder pipe (i.e. not to the kernel driver of the same name [2], [3]). In Android, there is also a term for the code stub that pretends to be a fully capable local object, but it actually only hands incoming requests over to the remote side – it is called a *proxy binder* (in OpenBinder, this is a *proxy object*). Keeping these distinctions on mind actually makes the available documentation a bit more readable. In the following, we will stay with the Android-related terminology. So, in the following, a *binder* does not refer to the kernel driver itself ([2], [3]), it refers to a code that we can invoke remotely through this driver.

Furthermore, if not stated otherwise, for the sake of simplicity and regarding the way threads are implemented in the contemporary Linux kernels [12], we use the words process and thread interchangeably in the following text. Note, however, that the binder framework implementation referred here not only supports threads – it actually directly assumes that the whole transaction processing is generally assisted by several working threads of each particular process.

For calling a remote binder method, there is an important rule saying that a process (that has opened the `/dev/binder` driver) can invoke only such remote binders that it was *explicitly invited to call*. In the structure `binder_proc` defined in [2], the kernel driver keeps sorted trees of allowed remote binders per each attached process (cf. `refs_by_desc` and `refs_by_node` fields of `binder_proc`). In particular, there is an independent `binder_proc` structure kept per each open file descriptor of `/dev/binder`. It is referred to by a pointer stored in `private_data` of the respective `file` structure object [12]. We can see the creation of `binder_proc` in `binder_open()` [2]. Note that the driver is written using the Miscellaneous Character Drivers (misc) framework [4].

The set of allowed binders for each process (i.e. certain transitive relation of "*process P is invited by process Q to call binder B*") is almost automatically updated by the kernel driver basing on the binder references exchanged within transactions flowing through the *others*, already established binder pipes. Simply saying (cf. bellow for more details), the kernel driver actively sniffs the data being sent in any binder communication. When it finds a binder reference, it verifies the binder already belongs to the set of allowed binders for the target process. If no, the kernel driver introduces its reference into `refs_by_desc` and `refs_by_node` r-b trees in the target `binder_proc`. This way, the

receiving process gets invited to call that binder later on. To be able to send such invitation, the inviter itself must have been invited to call that binder before.

There is one special binder reference that is inherently allowed by default - this is the handle 0 (typed as `NULL` pointer) that refers to the ServiceManager. This is a Linux daemon implemented in `service_manager.c` [9]. In Java world, it is encapsulated by `android.os.IServiceManager` Java proxy binder interface. An interesting communication graph showing ServiceManager at work is presented in [7] (cf. part “14. Binder”). It is worth noting that the ServiceManager not only plays the role of global binder locator. It also plays an important role in “seeding” the whole invitation relation.

When the particular service (e.g. ActivityManager) registers its binder object at ServiceManager, it in fact passes its binder reference to ServiceManager. The kernel driver recognizes this and introduces that reference into the set of allowed binders for ServiceManager (in `refs_by_desc` and `refs_by_node` trees in its `binder_proc` structure). That means the ServiceManager is now invited (allowed) to call ActivityManager. Later on, when another process *P* asks a binder reference for ActivityManager and the ServiceManager replies with this reference, then again the kernel driver sees this and it introduces that particular binder reference into the respective trees in `binder_proc` of process *P*. In this way, process *P* gets also invited to talk to ActivityManager. Furthermore, it has the same effect as if *P* has been invited to call this binder directly by the ActivityManager itself. Formally speaking, we see how the transitivity property of the relation *is invited to call* is employed to start up to whole IPC communication. Therefore, ServiceManager is not only the global locator, it is also the global inviter.

It is interesting to look at how the mechanism of "reference sniffing" or "invitation capturing" works. The answer is in the structure `binder_transaction_data` defined in `binder.h` (of the kernel driver, cf. [3]). This structure describes the particular transaction command/reply data being exchanged with a binder object. Beside `data.ptr.buffer` pointing to the data payload, we can see `data.ptr.offsets` pointer. This is an array of data offsets that should index all binder references being passed on in the data payload. It is the user space code that prepares the data payload (parcel) which is responsible for providing these indices to the kernel driver. The particular “parceled” data structure for the binder reference is `flat_binder_object` defined in [3]. When processing the data payload, the kernel driver looks at those particular offsets and manipulates the references being passed on together with eventually introducing them into the respective trees in the target `binder_proc` structure, hence inviting the target process to call these binders later on. The same mechanism works for both command and reply data transfers.

Of course, there is much more to say about this topic. For instance, when it comes to reference counting (to keep the referenced binders alive), the situation starts to be a bit fuzzy. It definitely requires to reverse-engineer the whole kernel driver together with its helping stuff in the user space to fully understand this.

The binder reference representation in `flat_binder_object` [3] also deserves a few words. In short - local binders (i.e. those ones being hosted in the current process) are represented directly by pointers to the user space objects, while remote binders (i.e. those ones being hosted by a “remote” process) are referred by handles. In the kernel driver, there is a system-wide unique representation of each single binder by `binder_node` structure instance [3]. In the kernel space, local binders of a process are referenced directly by `binder_node` pointers sorted in `nodes` r-b tree in the particular `binder_proc`, while remote binders are primarily referenced indirectly via *descriptors* (it is just driver’s word for the `handle` appearing in `flat_binder_object`) sorted into `refs_by_desc` and `refs_by_node` trees in `binder_proc`.

Because of the unique binder object representation in the kernel (as `binder_node`), the whole mechanism of binder passing has one interesting feature: A process can create a binder and send it (its reference) somewhere to the Android ecosystem world. Later on, when this process possibly receives the same binder back, it will get the same user space pointer. Therefore, the process can recognize it is its own binder it has sent to the Android world before. The same works for binders referenced by handle – a process can e.g. recognize it has just received the binder it had already received before. All this can be done by examining solely the binder reference data, the integrity of which *should be* (cf. bellow) preserved by the kernel driver.

This was a relatively short description of the binder-through-binder passing mechanism. This observation, however, does not explain whether it is possible to pass on a binder reference from process *Q* to process *P* in such a way, that once being invoked from *P*, it will still seem like *Q* is calling the binder (instead of *P*). By examining `binder.c` [2], it is easy to see the mechanism described above still preserves PID/eUID-based caller authentication. On the other hand, the kernel driver is also capable of passing on open file descriptors. It would be interesting to verify what would happen if *Q* would pass on *P* directly its file descriptor of `/dev/binder`. Actually, this might work, since the kernel

driver uses PID and eUID noted in `binder_proc` (cf. its `pid` and `tsk->cred->euid` fields) which seems to be passed together with the file descriptor. Well, this is an open question. All we can say now is that the kernel driver uses the same mechanism and data structure for locating file descriptors among the parceled data being transferred as for the binder references described above. The interested reader may try to search for `BINDER_TYPE_FD` in [2]. In particular, this is the value of `type` field in `flat_binder_object` in case of a file descriptor is to be sent.

Actually, the open file descriptor passing mechanism can be easily observed right in the `dumpsys` utility [5]. It is employed here to pass on the `STDOUT_FILENO` of the `dumpsys` Linux process within the `DUMP_TRANSACTION` data payload (this is the transaction D3 in the binder communication graph [7]). It is interesting to check [6] to see how the binder transaction gets formed and invoked via the proxy binder method `BpBinder::dump()`. The remote side binder uses this file descriptor to write its output data on behalf of the `dumpsys` process right into its terminal output. To see how far this file descriptor really propagates, we may look at the method `public void dump (FileDescriptor fd, String[] args)` of `android.os.Binder` [8]. Well, that is quite nice mechanism.

As was already noted in [10], the Android operating (eco)system is based on intensive object-oriented client-server communication. It is the binder framework that is in the center of this communication most of the time. From the security viewpoint, it is reasonable to ask on how far is the binder framework resistant against attacks being typically prevalent in this area. One such typical attack is a session hijacking – an attacker steals a communication pipe that was opened by some application component and continues in commanding the remote side (e.g. a service). From papers like [10], one can deduce that explicit cryptography-based techniques are seldom employed by user applications to preserve authentication and integrity of the binder communication. Actually, the state of the art seems to be represented by the Android operating system that uses the authentication based on caller's PID/eUID for (at least some of) its own system services. The userland applications seem to simply rely on "security by obscurity" approach by hoping that it is somehow hard to hijack the binder communication.

From the session hijacking viewpoint, the paradigm of explicit invitation described above seems to defeat the most straightforward attacks based on stealing or simply guessing a binder reference. Until the particular binder reference is inserted into the respective trees in `binder_proc` of the process, it is useless in direct communication (of that process). Furthermore, it should get introduced there only by the mechanism of explicit reference passing - i.e. the *invitation to call*. So, the whole concept looks good.

We should be, however, a bit worried about the mechanism of reference sniffing described above. The potential weakness is that it relies on a hint from the user space that helps the kernel driver to locate binder references in the data payload (cf. `data.ptr.offsets` in `binder_transaction_data`). If a dishonest sending process will not explicitly index the particular binder reference being passed on, it will probably bypass the sniffing mechanism of the kernel driver. Therefore, the reference data will be handed unmodified over to the receiver. The receiver will, however, still regard it as a binder reference because of its AIDL (or equivalent) template. In this way, the dishonest sender may transfer a raw binder reference data that will be evaluated later on with respect to the r-b trees in `binder_proc` of the *target* process (or to say – in the context of the remote binder). It actually allows an attacker to further indirectly reference those binders that the target process was already invited to call before. By fooling the target process this way, the attacker can manage the target to further pass on or call its "protected" binder (that the attacker is not invited to call directly). We call this a *cross-binder reference forgery* (*XBRF*).

On a first sight, it may seem the cross-binder reference forgery is a minor or even artificial problem. It may, however, lead to practical attacks on many naive userland application. In the worst case, there can be some attacks right on the Android application framework mechanisms – i.e. on the object-oriented kernel that makes the Android an operating system in itself (sitting on the top of an Embedded Linux kernel). It, however, requires further deep investigation. There is, for instance, a strange-looking field `cookie` in `flat_binder_object` that could theoretically prevent these attacks (or at least some of them). Unfortunately, this mechanism seems not to work (as a security measure) for binder references being passed on as handles, since the correct value gets filled in automatically by the kernel driver or is not important at all (cf. `binder_transaction()` in [2] and search for `BINDER_TYPE_HANDLE`). In other words, cookies seem to be important only when referring to local binders (i.e. those ones being hosted in the current process) by using user space pointers to their serving objects (cf. discussion of `flat_binder_object` above). Finally, this is a condition the attacker can avoid in the cross-binder reference forgery.

Besides making the application components themselves use reliable cryptographic authentication techniques, there is a promising approach aimed at improving the kernel driver itself [11]. An important assumption for a successful XBRF attack is that the attacker has at least certain idea on which binder references are already valid in the context of the target process. Therefore, employing reference handle randomization seems to be a natural way on how to harden the

attacker's task considerably. It must be, however, checked carefully on how much entropy we can really introduce into these handles and whether there are some other ways on how the attacker could learn valid references in the target process regardless their random nature. Apparently, there are still a lot of interesting open questions.

Update (November 16th 2011)

To support the ServiceManager daemon [9], there is a small local module `binder.c` (cf. [13], [14]). Attention should be paid not to confuse this module with the main kernel driver of the same name (cf. [2], [3]). The `binder.c` [13] is in fact a simple toolbox that is used by ServiceManager to handle its binder communication. We note it is worth it studying this module in itself, since several aspects of the binder framework communication can be learned here in considerably easier way than by studying the deep forest of Java world objects.

Important function regarding the potential XBRF is `bio_get_ref()` of [13] that is used by ServiceManager to parse a binder reference to be registered at the manager. This function further calls the static C function `_bio_get_obj()` of [13] which is actually the core of binder reference parsing here. To better describe its relation to XBRF, we copy and paste this particular function here.

```
static struct binder_object *_bio_get_obj(struct binder_io *bio)
{
    unsigned n;
    unsigned off = bio->data - bio->data0;

        /* TODO: be smarter about this? */
    for (n = 0; n < bio->offs_avail; n++) {
        if (bio->offs[n] == off)
            return bio_get(bio, sizeof(struct binder_object));
    }

    bio->data_avail = 0;
    bio->flags |= BIO_F_OVERFLOW;
    return 0;
}
```

Apparently, the programmer who wrote that piece of code was probably aware of the XBRF threat. It is easy to see that to successfully parse the flat binder object this function requires its data pointer to be already noted in the array of binder reference offsets. This way, the XBRF scenario is mitigated for any code that is using this function to parse flat (parceled) binder objects in its transactions.

Well, this is another kind of countermeasure that can defeat XBRF attack strategy effectively. It is, however, again based on a user space code. This time, of course, it is not a code that the attacker should have under its control, since it is the recipient's code (i.e. the remote binder) that performs the check. On the other hand, there is still a residual risk that programmers will simply omit to do such a check in their codes. Furthermore, the conceptual drawback of the kernel driver relying solely on a user space hint is still not solved by a countermeasure of this kind. Therefore, in some situations, a careful manipulation with the transaction data prepared by an attacking process may still lead to a successful XBRF exploit.

Anyway, it is an interesting observation to see that somebody was probably already aware of XBRF risk before. It would be also interesting to further explore whether such a countermeasure is really employed in the whole Android operating system.

References

/* The source tree refers to Android Gingerbread 2.3.5 (android-2.3.5_r1 tag). */

[1] Hackborn, D.-K.: *OpenBinder version 1.0*, http://www.angryredplanet.com/~hackbod/openbinder/docs/html/index.html, checked 11/11/11

[2] gingerbread/kernel/drivers/staging/android/binder.c

[3] gingerbread/kernel/drivers/staging/android/binder.h

[4] Rubini, A.: *Miscellaneous Character Drivers*, Linux Journal, June 30, 1998, http://www.linuxjournal.com/article/2920, checked 11/11/11

[5] gingerbread/frameworks/base/cmds/dumpsys/dumpsys.cpp

[6] gingerbread/frameworks/base/libs/binder/BpBinder.cpp

[7] Yaghmour, K.-J.: *Embedded Android Workshop*, ELCE 2011, http://www.slideshare.net/opersys/embedded-android-workshop-at-embedded-linux-conference-europe-2011, checked 11/11/11

[8] gingerbread/frameworks/base/core/java/android/os/Binder.java

[9] gingerbread/frameworks/base/cmds/servicemanager/service_manager.c

[10] Chin, E., Felt, A.-P., Greenwood, K., and Wagner, D.: *Analyzing Inter-Application Communication in Android*, MobiSys'11, June 28-July 1, 2011

[11] Buček, J.: *Personal communication on a former version of this note*, November 2011

[12] Bovet, D.-P. and Cesati, M.: *Understanding the Linux Kernel*, Third Edition, O'Reilly Media, Inc., 2006

[13] gingerbread/frameworks/base/cmds/servicemanager/binder.c

[14] gingerbread/frameworks/base/cmds/servicemanager/binder.h