**Table of Contents**

* 1. Introduction
  2. Background
     1. Survey of Android IPC
     2. Security Layers
  3. Compromising the System
  4. Our LKM’s API
  5. Attacks
     1. UID
     2. System Call Hooking
     3. Becoming the Service Manager
  6. Related Works
  7. Discussion
  8. Conclusion

**a. Introduction**

The Android OS is the most popular mobile operating system to date. Moreover, the Android OS's popularity is continually on the rise1. Being as such, it is a desirable target for malware, as more and more of our personal information is being stored on smart phones. As time goes by, the malware that targets android is getting increasingly sophisticated and complex2 .The software market reacts by developing better anti-malware detection and prevention methods3. Looking at the PC world, we can reasonably assume that the kernel will become the next attacked component of the system, as this gives malware the same or better abilities as existing defending software.

We will present a number of novel attacks which will mostly take place at the kernel level in a way that makes it difficult for anti-malware software to detect them, under the assumption such attacks were not known to the defending agent (zero-day attacks). In section TBD we detail our assumptions about the android framework, such as that their exists a way to gain root privileges, among others. We justify these assumptions in the same section. In section TBD we lay the framework for the following attacks, building the first steps for compromising the system. In section TBD, we go on to describe the implementation and usage of each attack. In Section TBD, we discuss our results and in the final section we conclude by summarizing our article.

Note: our attacks are non-control data attacks which don’t hijack the execution flow of the system4. However, control data attacks could have worked just the same. Better way to incorporate this?

**b. Background**

small general description

**b.i. Survey of Android’s IPC**

The Binder driver is a Linux kernel module which handles most of Android's IPC. Other common IPC methods in Linux, such as sockets and pipes, are not available for the use of user applications, which leaves the binder as the primary IPC method.

The Binder driver is implemented as a file, as with any Linux driver. When first opening the binder driver for communication, each process has a 1MB buffer allocated for it within the binder driver. This buffer is used as shared memory in the kernel for moving data from one process to another.

When process A wants to send data to process B, it calls the Binder driver which sends the data to process B. The Binder driver does so by copying A's data into A's buffer within the driver, copies it to process B's buffer, and from there into B's memory, it then wakes B up.

The calls are done through an ioctl call to the binder for both reading and writing. There is a special structure (binder\_read\_write) for that purpose, where the data A wants to write and the data B replies with are both placed within it.

As a security measure, two processes can not talk unless they have already communicated with each other, or a third process has invited them to communicate with each other. The Binder driver enforces this by keeping for each process a list of its communicable processes.

The Service Manager is a process which uses an exception to the above rules of the Binder, in order to seed communications between processes that haven't talked to each other. The Service Manager holds handles to all running services, and whenever the Binder gets a target-less message from a process, it references it to the Service Manager which returns the correct handle to the sending process.

The Service Manager is the only handle kept by the Binder driver. It is set during the system startup through the binder's *BINDER\_SET\_CONTEXT\_MGR* API which can only be called if the handle isn't set.

During the system startup the System Server is created after the Service Manager. This process initializes most system services (Such as the PackageManager or the ActivityManager), which run under him as threads. These services all register themselves with the Service Manager.

**b.ii. Android's Security Layers**

expand on same section from the presentation

**c. Compromising the System:**

In order to initiate our attack we must assume the following:

a. There is an exploit for getting root privileges without triggering any malware

detection.

b. There is no malware detection for usage of the *insmod* API, which is used to load

dynamic code to the kernel by root users.

These assumptions are reasonable seeing as root exploits are found in different Android versions5, and the *insmod* API is present in the Android kernel6.

First, a malicious application is downloaded by the user, which upon activation proceeds to gain root privileges through a known vulnerability (Assumption a).

Then, as root, the application loads our LKM (Loadable Kernel Module) using the *insmod* API.

In order for the application to be able to communicate with the LKM, it must create a special device file (As with any driver). This can easily be achieved by installing BusyBox, an application that enables few basic shell commands which are disabled in android by default, and calling its *mknod* API.

There probably are other ways to get the last part done, but as root it is no problem installing BusyBox (Which is a rather small application) and running this single command without the user knowing. Make this last part a footnote?

After the last step is completed, we now have gained the ability to run our own code in the kernel. The next steps will use that facility to activate themselves.

**d. Our LKM’s API**

In order to communicate with the kernel module we use the *ioctl* system call, which allows us to define the following calls to the module:

*IOCTL\_UID* - Change the current process’ user id to any given id.

Can be trivially expanded to change group id.

*IOCTL\_SYS\_CALL* - Change a given system call into a pre-defined function within the

module. Can add whichever system call that comes to mind.

*IOCTL\_CLR\_CTX\_MNG* - Clear the pointer to the original service manager and its

saved user id within the binder driver.

**e. Attacks**

**e.i. Setting User Credentials (UID, GID...)**

This attack accesses the current process structure held by the scheduler and changes its credentials.

The malicious application calls *IOCTL\_UID* in order to activate the attack.

Then, the LKM calls the function “prepare\_creds” which returns a new credentials structure (This structure holds all ids of a process, as expected).

Into this new credentials structure the LKM assigns the wanted id and then calls “commit\_creds” which applies the new credentials structure to the current process structure held by the scheduler.

Changing credentials as you wish gives you the option to look like a regular user to the system in order to avoid routine malware detection (For example), become root again, or act as a system service (With UID 1000 which is used by all system services).

**e.ii. Hooking System Calls**

This attack searches for the system call table and replaces a selected system call function pointer to a hook function pointer.

On some versions of the linux kernel, this attack may not work because the system call table is defined in read-only space, but bypassing that only requires us to change the system tablespace to a writable location.

When first loaded, the LKM finds the address of the system call table and saves the original system call pointers in order to undo any damage we may cause.

The malicious applications calls *IOCTL\_SYS\_CALL* in order to activate the attack, which then replaces the selected system call pointer in the saved table with a pre-defined function within the our LKM.

Hooking certain system calls allows you to spy on the system, reduce anti-malware hooks, inserting logic-bombs, etc.

**e.iii. Becoming the Service Manager**

This attack replaces the original service manager’s node in the binder driver and starts a process to become one itself.

First, a new process is spawned which gets all current running services from the service manager and saves references to them. This is done to maintain the flow of the system because applications continuously ask for services (Like the SystemUI or the ActivityManager), and the system would crash if none of them got any. this included parsing binder messages and not so trivial saving of the references, to elaborate?

Then the process calls *IOCTL\_CLR\_CTX\_MGR* to clear the original service manager node within the binder driver (This also clears the corresponding saved UID in the driver), andalso calls *BINDER\_SET\_CONTEXT\_MGR* in order to become the service manager through the binder driver’s API.

As the new service manager, you have the ability to decide which applications have access to which services and who can add a new service. It is also possible to replace a default service such as the camera or GPS to our own in a way that whoever asks for it for the first time, will receive our hooked version.

**f. Related Works**

VMM Based Rootkit Detection on Android - made a static KM to verify integrity of certain aspects of the system (system calls, sockets, etc) and used a layer below the kernel to check the integrity of the KM itself.

**g. Discussion**

**h. Conclusion**

**Notes**

1.<http://techcrunch.com/2013/07/01/android-led-by-samsung-continues-to-storm-the-smartphone-market-pushing-a-global-70-market-share/?ncid=tcdaily>

2. **\*\*\*reference to dreamdroid and userspace rootkit\*\*\***

3. **\*\*reference to userspace rootkit, it has vendor specific defences, and VMM article\*\***

4. reference to the paper eitan sent on non-control data attacks

5. **\*\*footnote for rageagaintthecage, zygote exploit and more\*\***

6. **\*\*Note that I checked different kernels\*\*.**