HT core-android analysis

Martin Pozdena
Zhongying Qiao

Introduction

We analysed leaked Hacking Team repository `core-android` containing HT's Remote Control System resources related to Android platform backdoor. The fundamental backdoor resources are to be found in folder `RCSAndroid`. The remaining folders and files in the repository are either related to the build system (HT used Gradle system to build its Android backdoor), application obfuscation (`crypter`) or they are related to signing the resulting `apk` file (`./export/sign.bat`). Presence of `bat` files suggests that most of development work inside Hacking Team was done on Windows machines. The following sections describe our findings concerning the main backdoor components in `RCSAndroid`.

RCSAndroid

First glance reveals that the most interesting resources of backdoor in `RCSAndroid` are located in folder `src` (backdoor resources developed in Java) and `jni` (backdoor resources developed in C to run natively on Android). We calculated that `RCSAndroid` consists of 656 Java source files with total count of 116,626 lines of code and 118 C source files summing up to 9,117 lines of code. Although this figures also contain external open source resources it was clear to us that given the limited timeframe our analysis of Android backdoor needs to be more high-level than that of Linux backdoor.

Java resources

Most of relevant Java resources are located in folder `RCSAndroid/src/com/android/dvci`. Java part of a backdoor runs as Android Service, component that unlike Activity provides background services and does not provide any graphical interface to the user. It starts of by determining whether the application runs inside of a debugger or virtual machine (if yes then backdoor immediately terminates):

```
// ANTIDEBUG ANTIEMU
if (!Core.checkStatic()) {
    if (Cfg.DEBUG) {
        Check.log(TAG + " (onCreate) anti emu/debug failed");
    }
    if (Cfg.DEMO) {
        Status.self().makeToast(M.e("RUNNING"));
    }

    return;
}
```

`checkStatic()` method checks whether the application runs inside the emulator or inside debugger. Emulator check (implemented in `AntiEmulator.java`) checks for multiple parameters in hunt to determine whether it is running in emulated device including:

- Build.FINGERPRINT
- Build.TAGS
- Build.PRODUCT
- Build.DEVICE
- Build.BRAND
- Build.MANUFACTURER
- tm.getDeviceId()
- tm.getSubscriberId()
- tm.getSimOperatorName()
- tm.getLine1Number()

Anti debugger is implemented in `AntiDebug.java` class and it determines whether application runs in the debugger environment based on information it can receive from the system. For instance it checks if flag `FLAG_DEBUGGABLE` is set at `Status.self().getAppContext().getApplicationInfo().flags` or it uses `android.os.Debug` library as follows:

```
public boolean checkConnected() {
    if (Cfg.DEBUGANTI) {
        Log.w("QZ", " checkConnected: " + Debug.isDebuggerConnected());
    }
    return Debug.isDebuggerConnected();
}
```

If application determines that it is not running inside the emulator or debugger it creates 5 broadcast receivers checking for change of state of the following system components:

```
bst = new BSt();    //BroadcastReceiver standby
bac = new BAc();    //BroadcastReceiver AC connected
bsm = new BSm();    //BroadcastReceiver SMS
bc = new BC();      //BroadcastReceiver Call
wr = new WR();      //BroadcastReceiver Wifi
```

Receivers change the behavior of backdoor according to the broadcasts spreading across the Android system.

Another interesting feature of Android backdoor is that it contains features for demonstrations to Hacking Team's customers (Law Enforcement Agencies). The following snippet of code was taken from the main background service. It informs user that agent (running silently in the background) was successfully created through Android Toast (pop-up message that appears at the bottom of screen).

```
if (Cfg.DEMO) {
    Toast.makeText(this, M.e("Agent Created"), Toast.LENGTH_LONG).show();
}
```

Actual capture of victim's data is again implemented in various backdoor modules, each targeting different sort of information. Majority of modules also reveal that they cannot steal user's data without having root privileges on the phone (thanks to Android OS application sandboxing). On the other hand, Android backdoor is able to escalate privileges in using some of its natively running components. For example `ModuleDevice.java` implements functionality allowing to capture detailed information about infected device including IMEI, batery charge, how much free space is available or whether backdoor has root priviliges or not:

```
sb.insert(0, M.e("Model:") + Build.DISPLAY + "\n");
sb.insert(0, M.e("IMEI: ") + Device.self().getImei() + "\n");
sb.insert(0, M.e("Root: ") + (root ? "yes" : "no")  + M.e(", Su: ") +
        (su ? "yes" : "no") + M.e(", Admin: ") + (admin ? "yes" : "no") +
        M.e(", Persistence: ") + Status.getPersistencyStatusStr() + "\n");
sb.insert(0, M.e("Free space: ") + freeSpace + " KB " +
        M.e("Installation: ") + "\n");
sb.insert(0, M.e("Battery: ") + battery + "%" + "\n");
```

Another interesting module is `ModuleChat.java` which allows attacker to collect victim's messages directly from messanger's storage inside the phone. It supports the following messenger applications:

```
if (Cfg.ENABLE_EXPERIMENTAL_MODULES) {
    subModuleManager.add(new ChatTelegram());

} else {
    subModuleManager.add(new ChatBBM());
    subModuleManager.add(new ChatFacebook());
    subModuleManager.add(new ChatWhatsapp());
    subModuleManager.add(new ChatSkype());
    subModuleManager.add(new ChatViber());
    subModuleManager.add(new ChatLine());
    subModuleManager.add(new ChatWeChat());
    subModuleManager.add(new ChatGoogle());
    subModuleManager.add(new ChatTelegram());
}
```

For example in case of Telegram, Android backdoor tries to access the following files:

```
String pObserving = M.e("org.telegram.messenger");
String dbFile = M.e("/data/data/org.telegram.messenger/files/cache4.db");
String dbAccountFile = M.e("/data/data/org.telegram.messenger/shared_prefs/userconfing.xml");
```

Data is not being transmitted from victim's device to command and control server immediately when recorded, but it is rather stored in the local storage and synchronized with C&C server in batch. This is ensured through several Java classes where backdoor modules first utilize the class `EvidenceBuilder.java` for example as follows:

```
final EvidenceBuilder log = new EvidenceBuilder(EvidenceType.DEVICE);
log.write(WChar.getBytes(content, true));
log.close();
```

`EvidenceBuilder` stores the data in form of class `Packet`, which is subsequently passed to the class `EvDispatcher`. This class stores data in form of `Packets` in the queue and ensures that data in the queue is periodically flushed from memory to the device storage. Victim's data are again encrypted and stored in files that are hard to locate on the filesystem. It was complicated to determine the storage without running the application, but file paths are generated in class `EvidenceCollector` and method `makeNewName` as follows:

```
final String basePath = Path.logs();
final String blockDir = prefix + (progressive / LOG_PER_DIRECTORY); //$NON-NLS-1$

// http://www.rgagnon.com/javadetails/java-0021.html
final String mask = M.e("0000"); //$NON-NLS-1$
final String ds = Long.toString(progressive % 10000); // double to
// string
final int size = mask.length() - ds.length();
if (Cfg.DEBUG) {
    Check.asserts(size >= 0, "makeNewName: failed size>0"); //$NON-NLS-1$
}
final String paddedProgressive = mask.substring(0, size) + ds;

final String fileName = paddedProgressive + "" + logType + "" + makeDateName(timestamp);
final String encName = encryptName(fileName + LOG_EXTENSION);
```

C resources

Android runs on the top of Linux kernel and many aspects of Linux and Android architecture are similar on the native level. Android allows developers to implement some computationally intensive application tasks in C and access it through Java Native Interface (JNI). It is therefore possible to exploit for example some known Linux kernel vulnerabilities using natively running code in order to escalate privileges to root or escape SELinux Mandatory Access Control mechanism (SELinux is configured in enforcing mode since Android version 5.0).

As it was already mentioned in the previous section overwhelming majority of Java backdoor modules required root access in order to function properly. The reason for this is that every single Java Android application runs in Operating System level sandbox. Each application runs under different Linux user and as a such is unable to write or read information stored in the folders of other Applications (“Linux users”). This security mechanism can be only defeated by escalating privileges to root. As Android device manufacturers are infamous for providing delayed or no system updates at all, majority of Android devices run with outdated and vulnerable native components including Linux kernel. Therefore, such a privilege escalation does not need to be necessarily so complicated to achieve.

Relevant C source codes from `RCSAndroid` are to be found in `jni` folder and they indeed seem to be trying to achieve local privilege escalation. They indicate that Hacking Team was able to escalate privileges or break SELinux using its Android backdoor. For example file `exploit.c` checks whether device:

1. is already rooted:

```
int fildes = open("/system/bin/su", O_RDWR);
status = fstat(fildes, &sustat);
close(fildes);

mode_t mode = sustat.st_mode;
uid_t uid = sustat.st_uid;
gid_t gid = sustat.st_gid;
off_t size = sustat.st_size;

int executable = (mode & S_IXOTH) && (mode & S_IROTH);
int suidded = (mode & S_ISUID);
int root = uid == 0;
int regular = S_ISREG(mode);

sprintf(buf, "[*] executable: %d suidded: %d root: %d regular: %d\n",
        executable, suidded, root, regular);
LOG(buf);

return executable && suidded && root && regular;
```

2. is of version 2.2-3.0 and can be exploited using known GingerBreak privilege escalation (http://c-skills.blogspot.de/2011/04/yummy-yummy-gingerbreak.html)

There are more versions of privilege escalation exploit which we believe is targeting more recent Kernel versions. They are all invoked from file `exploit_list.c` and upon successful escalation runs the following code:

```
/* ask for root */
ret = setresuid(0, 0, 0);

if(ret) {
  cleanup(exp);
  return 0;
}

// If everything was ok we are root now
exec_payload(args, cmd);
```

Although `exec_payload` implementation is not to be found among the source codes it is likely that backdoor executes for instance code from `suidext.c` which provides the following functionality:

```
LOG("Usage: ");
LOG("%s", argv[0]);
LOG(" <command>\n");
LOG("fb - try to capture a screen snapshot\n");
LOG("vol - kill VOLD twice\n");
LOG("reb - reboot the phone\n");
LOG("blr - mount /system in READ_ONLY\n");
LOG("blw - mount /system in READ_WRITE\n");
LOG("rt - install the root shell in /system/bin/rilcap\n");
LOG("ru - remove the root shell from /system/bin/rilcap\n");
LOG("rf <mntpoint> <file> - remove <file> from <mntpoint>");
LOG("sd - mount /sdcard\n");
LOG("air - check if the shell has root privileges\n");
LOG("qzx \"command\" - execute the given commandline\n");
LOG("fhc <src> <dest> - copy <src> to <dst>\n");
LOG("fhs <mntpoint> <src> <dest> - copy <src> to <dst> on mountpoint <mntpoint>\n");
LOG("fho <user> <group> <file> - chown <file> to <user>:<group>\n");
LOG("pzm <newmode> <file> - chmod <file> to <newmode>\n");
LOG("adm <package name/receiver>\n");
LOG("qzs - start a root shell\n");
LOG("lid <proc> <dest file> - return process id for <proc> write it to <dest file>\n");
LOG("ape <content> <dest file> - append text <content> to <dest files> if not yet present\n");
LOG("srh <content> <file> - search for <content> in <file>\n");
```

Apart from privilege escalation exploits `jni` folder also contains exploits allowing to break SELinux (in folder `selinux_exploit`).

Conclusion

Hacking Team's Android backdoor was at the first glance complex piece of malware. In fact some Internet resources including Trend Micro Security Intelligence Blog describes it as one of the most sophisticated Android malwares ever exposed. Its functionality goes far beyond the functionality of Linux backdoor we also investigated. Given the malware complexity and limited timeframe for malware analysis it was impossible to provide in-depth analysis of its functionality. Nevertheless, we believe that we provided a basic insight into some of the fundamental malware components.