VB2020
localhost

30 September - 2 October, 2020 / vblocalhost.com

THE FALL OF DOMINO – A PREINSTALLED HOSTILE DOWNLOADER

Łukasz Siewierski & Sebastian Porst

Google, UK & USA

lsiew@google.com

ABSTRACT

Android is an open-source operating system which allows OEMs and their subcontractors certain flexibility in adding components to the system. These add-ons may contain new and exciting features, but sometimes they also hide complex malware. This talk will deal with a malware family called 'Domino'.

Domino was discovered preinstalled on certain *Android* devices and distributed as a new operating system component on a small fraction of different phone brands, all of them low-cost devices running *Android 7* or lower. On these devices, the malware author added additional code to many *Android* components (including the default browser, the Settings app and the *Android* framework), allowing Domino to use system privileges to download additional applications later on and prevent their uninstallation by the user.

Different versions of Domino implement different behaviours, from displaying advertisements to overwriting visited URLs in order to change the default search engine or advertisement campaign referral IDs. The changes introduced by Domino also made it possible to ensure that Domino's browser was exclusively used to display all links clicked by the user.

Rather unusually, we were able to obtain a compressed archive with Domino's source code, including code comments and notes for manufacturers on how to embed Domino on their devices. Additionally, this archive includes SELinux policies crafted to allow Domino to persist and run with elevated privileges. We also obtained a test application which tried to interact with the *Google Play* store in order to test referral substitution and seems to be written by the Domino author to test some coding ideas.

BACKGROUND

The investigation of the Domino hostile downloader started with a *Malwarebytes* blog post on xHelper [1]. The xHelper package referenced in the blog post (com.mufc.fireuvw) was installed on a large percentage of three specific device models. These device models all also had two additional executable binary files which are not part of the default code of the Android Open Source Project (AOSP).

The two binaries on these devices – always stored in /bin on the system partition – had different names depending on the device model, but in all cases came in two flavours: the 'service' binary and the 'daemon' binary. Additional configuration files were stored on the /data partition of the system image.

During our investigation we also discovered modifications to the AOSP framework and application code on some devices which could be traced back to Domino.

It's important to note that the xHelper application in the specific case found by *Malwarebytes* was probably distributed using Triada and not by Domino, based on the description in the *Malwarebytes* follow-up blog post [2]. We will discuss the links between Triada, Domino and xHelper later in this document.

DOMINO BINARIES: DAMON AND SERVICE

Two Domino binaries work together to perform the malicious behaviour. The 'damon' binary (typo made by the malware authors) manages configurations, communicates with C&C servers and performs fake SMS activity. The 'service' binary is responsible for displaying advertisements and installing apps.

The 'damon' binary adds fake SMS messages to the user's inbox by opening the default *Android* SMS database file, mmssms.db, and adding an entry. The code also checks if there is a previous message thread for the same phone number so that it can continue the message thread instead of creating a new one. Unfortunately we were not able to find examples of these fake SMS messages, so their purpose is not clear to us.

Next, the binary uses an SQLite database named 'domino'. Tables created in that database store configuration options and information to be extracted from the phone, including:

- pInfo – contains fields for basic phone information like phone number, account email, carrier, locale, location, MAC, etc. Some of the values (e.g. account email) were not actually extracted from the phone, but there was still a place left for them in the database. This may mean either that the malware author decided to drop the collection of some of the data or that the data is intended to be collected in future versions.
- tbl_strategy – used to configure the advertisements displayed on the phone (e.g. the frequency). These advertisements will be displayed by the binary or by the web browser.
- ymbus – this table contains the domain name of the main C&C, which is used to download both configuration and advertisements.
- ymad – this table contains a C&C address for the SMS functionality described above.
- ympay – unused, but can also be used to store a third type of C&C. While the URL isn't used or saved anywhere in the code, the 'pay' part of the name may suggest some relation to payments.
- zxcvb – used by the 'service' binary to keep track of installed packages (described below).

The 'damon' binary uses several HTTP endpoints on the C&C domain:

- /bus-webapi/rest/service/ym_list – adds new values to the ymbus and ymad tables (the two C&C URLs) to allow Domino to switch to new C&C servers.
- /bus-webapi/rest/service/strategy – used to deliver configuration settings including the number of fake text messages to write to the SMS database. The configuration is then written to tbl_strategy.
- /domino-webapi/rest/service/cm_list – returns a phone number, text and a callback URL that is pinged to report the successful creation of a fake SMS.

The 'service' binary displays advertisements and installs *Android* apps based on the C&C responses and the configuration settings obtained by the 'damon' binary. The C&C provides a list of download URLs, app names and activity names. The apps will be downloaded and installed using the 'pm install' command. After installation, activities inside the new apps will be started using the 'am start' or 'am startservice' command. This installation flow does not require any user interaction due to the elevated privileges of the Domino binaries.

There is also a shorter version of the flow which only 'activates' the application (i.e. starts the appropriate service or activity), without installing it. This may be used as a way to make sure that the installed application or its services are still running.

ANDROID MODIFICATIONS

In addition to the two binaries, Domino also comes with AOSP framework modifications for the AOSP web browser, the PackageManagerService, the Settings app, and modifications related to the Activity class. These modifications allow Domino to display advertisements and misattribute *Google Play* store app installations to different advertising campaigns.

Browser modifications

The first set of modifications introduce additional functionality to the default AOSP browser application. One snippet of added code is responsible for downloading files based on the C&C response. The code modification in the browser allows Domino to attribute downloads to the browser process in order to hide them more effectively.

Several safety checks are performed before the download task is run:

- Check to see if the device is connected to any data network – e.g. Wi-Fi or cellular data.
- Check to see if a specific system image build property (ro.feature.browser_ext) is enabled.
- Check to see if a 'BlackListState' is set to 1. This value is set by the C&C.
- Check to see if Domino is in a 'silent period' – this is just another value set by the C&C. If that's true a download will not happen.

The list of URLs used to download files is retrieved from the C&C as a DES-encrypted string with key '13572468' (as a string) and IV 0x0102030405060708. This functionality can be used to download an APK file through the browser (the app which has the INTERNET permission) and then install the APK file using one of the binaries mentioned above. Due to the elevated privileges of the binaries, the user will not need to click through the regular app install UI. Rather, the downloaded app will be installed without the user noticing it.

The browser is also modified to display advertisements on the home screen right after the user unlocks the phone. Two other modifications also seem to be related to advertisements: adding bookmarks to the browser and changing user search queries before they are sent to the default search engine. This may allow the addition of affiliate identifiers for some search engines. The decompiled method shown below is responsible for that search query change:

```
public String getSearchUriForQuery(String arg6) {
       DeviceUtils.getInstance(this.mContext);
       String v0 = SettingsUtils.getInstance(this.mContext).getChanelValue();
       if(("none".equals(v0)) || ("".equals(v0))) {
              v0 = this.getChannelValue();
       }
       String v2 = SearchUtils.getUTF8XMLString(arg6);
       return String.valueOf(a.A) + "m=" + "" + "&c=" + v0 + "&k=" + v2;
}
```

Settings app modification

The Settings app seems to be Domino's target only for *Android 7.0*. This may point to the fact that they moved on from the AOSP browser. The AndroidManifest XML file for the Settings app has been modified to include the following metadata:

```
<meta-data android:name="BASE_B_URL" android:value="http://bus.dominoppo.in"/>
<meta-data android:name="BASE_A_URL" android:value="http://psd.dominoppo.site"/>
```

```
<meta-data android:name="YM_URL" android:value="1CQA8z9tf/mloptekPMjg8ECFoI97
                            bwiHG705Med5+J+NuNgu0kc9+g6ExM4 yIA5ccSpSQS4EgY7G8DUVjGgCg==" />
<meta-data android:name="VERSION_CODE" android:value="262" />
<meta-data android:name="VERSION_NAME" android:value="2.6.2" />
<meta-data android:name="AD_PLACE" android:value="1256" />
<meta-data android:name="CLOSED_PERIOD" android:value="0" />
<meta-data android:name="PKG_NM" android:value="com.android.htmlviewer" />
...
```

The standard Domino C&Cs URLs are set as the metadata values. Interestingly, one URL is encoded using base64, while the other two are not.

Domino communicates with the C&C server using JSON objects. It receives two lists: rList contains AdUrlBean objects and mList contains GpRefBean objects.

The Settings application registers a receiver which listens for SCREEN_OFF notifications. The application then checks if the screen is really off and sends an Action to the HTMLViewer application (the PKG_NM value defined in the AndroidManifest metadata above). This Action contains two extra fields with the values of mList and rList encoded as JSON objects.

The HTMLViewer app creates a WebView component and displays elements from the rList (AdUrlBean objects). Then, when the URL finishes loading in the onPageFinished method, the URL is parsed for the referrer= value. If that value is found, a new GpRefBean object is created which contains the referrer value and the package name (rewritten from the AdUrlBean). This object will then be used in the framework modification to replace the actual referrer value when a *Google Play* link is followed.

ActivityManagerService modifications

The first set of framework modifications is in the ActivityManagerService and other classes which handle Intents. These modifications try to intercept the INSTALL_REFERRER broadcast to misattribute the application installation to a different advertising campaign. This allows the malware authors to claim that installations from all advertising campaigns and organic sources actually come from their advertising campaign.

In order to intercept that broadcast, the broadcastIntentLocked method, which handles all broadcasts sent across *Android*, contains this additional code:

```
RefBean bean = DataUtils.checkAsoTask(intent, callerPackage, this.mContext);
if(bean != null) {
  DataUtils.initRefTask(intent.getStringExtra("referrer"), bean, this.mContext);
  intent.putExtra("referrer", bean.getRefer());
}
```

The checkAsoTask method checks if there's a GpRefBean object that contains an installed package name and was created less than three days ago (i.e. the advertisement was visited with screen off less than three days ago). If it finds such an object it generates a random number and if that number is within a specific range it will change the referrer string parameter of the Intent to one specified by a C&C. This means that after *Google Play* sends the Intent but before it has received the value of the referrer field, it will be changed to a different value. The purpose of the random number range check is to hide the fraudulent behaviour occasionally.

Some builds also have another framework modification in the Activity class. The startActivity method has some additional code which checks the content of the Intent:

```
if(intent != null && intent.getDataString() != null) {
  AdUrlBean bean = DataUtils.checkTask(intent, this.getApplicationContext());
  if(bean != null) {
    new ActivityTask(this, bean, intent).execute(new String[0]);
    return;
  }
}
```

The checkTask method checks if the data string of the Intent contains one of the two URIs: '/play.google.com/store/apps/details?id=' or 'market://details?id='. If the Intent contains one of these strings and the C&C is interested in the package name included in the Intent then an AdUrlBean will be returned. Then, instead of starting a regular Activity for the Intent, a new ActivityTask is run which starts *Google Play*, AOSP browser or *Chrome* with a different URL obtained from the matching AdUrlBean. Effectively, this means that the URL will be rewritten to a different one which is associated with the advertisement shown previously when the screen was off.

Both framework modifications and previously mentioned modifications of the Settings application and the HTMLViewer application work together in the following way:

1. The Settings application periodically polls the C&C URL for two lists: a list of ads and a list of *Google Play* referrers. Each one contains a package name, URL and some other values.
2. When the screen is off the Settings application sends a list of advertisement URLs and *Google Play* referrers to HTMLViewer.
3. HTMLViewer displays advertisements on a turned off screen and if the advertisement URL, after rendering, contains a referrer= URL then a new *Google Play* referrer object is created.
4. Some time later a user may click on a link to a *Google Play* application. Let's assume this application is named 'com.android.x'.
5. An android.intent.action.VIEW Intent is created with a deep link or URL to the app on *Google Play* in order to handle the user click.
6. startActivity is called for that Intent to view it in the *Google Play* store or the browser.
7. A list of *Google Play* referrer objects is searched for the com.android.x package name. If one is found a deep link / URL is extracted from that object.
8. This Intent is discarded and a new one is created, with a link from the *Google Play* referrer object created in step 3.
9. If the user decides to install the app from step 4 then the *Google Play* process sends the INSTALL_REFERRER broadcast to all other apps on the device.
10. This broadcast is intercepted before it gets to any other app on the device.
11. The value of the referrer string extra is changed to match the one from the list of referrer objects. This makes the referrer changes consistent with the Intent from step 8.

PackageManagerService modifications

PackageManagerService is a Java class responsible for handling app-related actions, including choosing the best activity to service an Intent. The logic is fairly simple: if there's only one app that can handle an Intent this app will be chosen, but if there's more than one then the user will get a pop-up asking which app they want to choose.

For example, if a user clicks on a link to youtube.com and the *YouTube* app is installed, a pop-up will ask if that link should be opened with a browser or with the *YouTube* app. Figure 1 shows the user interface for this functionality.

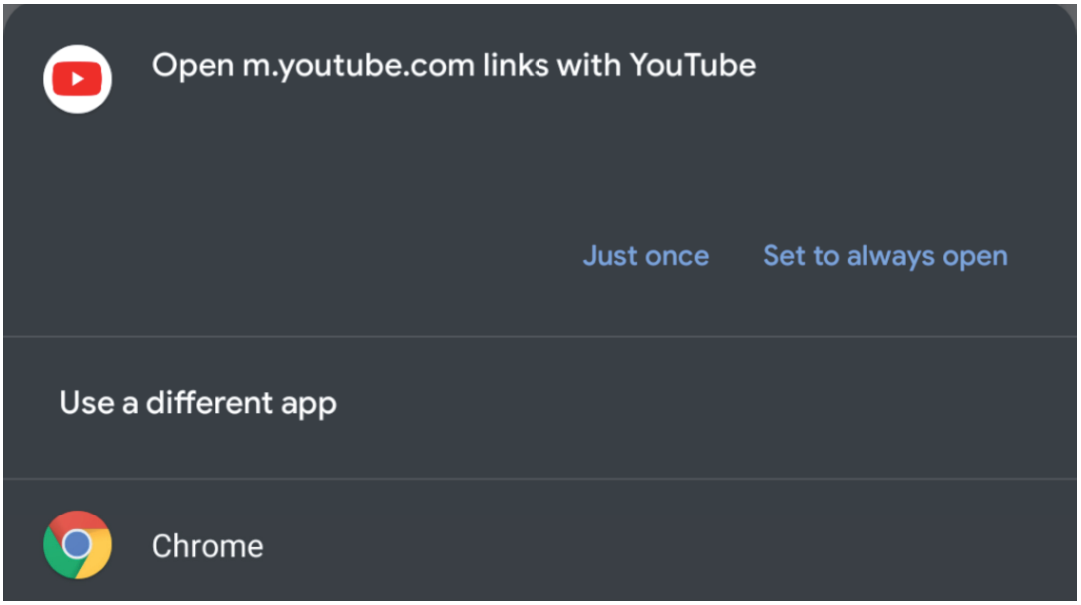

Figure 1: A pop-up asks if the link should be opened with a browser or with the YouTube app.

The modification introduced by Domino and shown below makes sure that if the Domino browser is able to handle the Intent it will be always chosen as the app to handle the Intent and no dialog will ever show up. The modification first checks the value of a Boolean system property, ro.feature.browser_ext, and if it's set to true and the browser can handle the Intent PackageManagerService will just choose the Domino browser.

```
private ResolveInfo chooseBestActivity(Intent intent, String resolvedType,
         int flags, List<ResolveInfo> query, int userId) {
  if (query != null) {
    final int N = query.size();
    boolean mBrowserSwitch =
                SystemProperties.getBoolean("ro.feature.browser_ext", true);
    if (mBrowserSwitch){
         for(int i = 0; i < N; i++ ){
         String mResolveInfoStr = query.get(i).toString();
         String mKeyPkg = "com.android.browser";
         String mAction = intent.getAction();
         String mViewAction = "android.intent.action.VIEW";
         if(mResolveInfoStr.contains(mKeyPkg) &&
                mViewAction.equals(mAction) &&
                resolvedType == null){
         return query.get(i);
         }
...
```

The 'return' statement in the last line of the conditional clause makes sure that the regular flow of 'chooseBestActivity' will not go through if one of the packages able to handle the Intent is 'com.android.browser'.

SELinux and init.rc modifications

The init.rc modifications create two services for two hostile downloader binaries, as can be seen below:

```
service htfsk /system/bin/htfsk
       class late_start
       socket htfsk stream 666 radio system
       user radio
       group system shell radio sdcard_rw sdcard_r media_rw inet wifi net_admin net_raw
       disabled

service rbn /system/bin/rbn
       class late_start
       user shell
       group system sdcard_rw sdcard_r media_rw inet wifi net_admin net_raw
       seclabel u:r:shell:s0
       disabled
```

The service that installs apps ('rbn' – the 'service' binary) runs as the shell user, which means that all app installs coming from this service are attributed to the user using Android Debug Bridge (ADB). Additionally, the groups in which the binary runs give it access to external storage (where the APK file is dropped). This is equivalent to an application having READ_EXTERNAL_STORAGE and WRITE_EXTERNAL_STORAGE permissions.

The two services are disabled by default, but there's another section of init.rc which starts them if the system property ro.feature.browser_ext is set, as can be seen below:

```
on property:ro.feature.browser_ext=true
       start htfsk

on property:ro.feature.browser_ext=true
       start rbn
```

Apart from init.rc a number of new SELinux policies and contexts are added to the system image. Domino downloader binaries get their own SELinux contexts and so do the files they need to use. There's also a domain_trans macro called to make sure that rbn_exec can transition from the init to shell domains. This makes sure that any actions performed by Domino can be attributed to ADB. This may be a way to hide behaviour from malware detection tools.

SOURCE CODE PACKAGE

We were able to find a full source code package of Domino that had been submitted to *VirusTotal*. The directory with the source code is named domino_browser_jingji_20161210 and contains two sub-directories for Domino versions 5.1 and 6.0. The numbering may correspond to the *Android* versions. The source code contains a very interesting directory tree structure, as shown in the following:

```
.
├── device
│   └── sprd
│       ├── customProject
│       │   └── mmx                 // custom system properties
│       └── scx35                   // SELinux policies
├── FilesList.txt                   // List of files to modify or add
├── frameworks
│   └── base
│       ├── core
│       │       └── (...)
│       │               └── ResolverActivity.java    // Intent changes
│       └── services
│           └── (...)
│                   └── PackageManagerService.java   // App chooser changes
├── packages
│   └── apps
│       └── Browser                                  // AOSP browser changes
├── Readme.txt                                       // Readme file
└── vendor
    └── (...)
            └── system
                └── bin
                    ├── htfsk                       // "service" binary
                    └── rbn                         // "damon" binary
```

The Readme.txt file contains a message to the OEMs in Mandarin Chinese:

> 请根据注释关键字*domino*进行集成
>
> *FilesList.txt*中是需要集成的文件列表
>
> *android.permission.READ_PHONE_STATE* 浏览器需要有这个权限的默认授权，请贵司技术>进行处理

The English translation of that message is:

> *Please integrate according to the comment keyword domino*
>
> *FilesList.txt is a list of files that need to be integrated*
>
> *android.permission.READ_PHONE_STATE The browser needs to have the default authorization for this permission, please ask your company to handle it*

The READ_PHONE_STATE permission is needed so that the advertising framework can fingerprint the device better.

The FilesList.txt file is divided into two sections: Add and Modify. Each section contains a list of files that have to be added to the system image or modified. Modifications are commented using the keyword 'domino' and 'Add-[S|E]' or 'Modify-[S|E]'. For example, in the case of the AndroidManifest.xml file of the AOSP browser application, the modifications look like this:

```
<!-- Add-S by domino -->
<uses-permission android:name="android.permission.RECEIVE_USER_PRESENT" />
<uses-permission android:name="android.permission.READ_PHONE_STATE" />
<uses-permission android:name="android.permission.GET_TASKS" />
<uses-permission
       android:name="android.permission.INTERACT_ACROSS_USERS_FULL" />
<uses-permission android:name="android.permission.REORDER_TASKS" />
<uses-permission android:name="android.permission.RESTART_PACKAGES" />
<!-- Add-E by domino -->
```

CONNECTIONS TO OTHER MALWARE FAMILIES

URL patterns seen in the 'damon' binary mentioned at the beginning have also been seen in some other malware families like Ewind. In these cases it seems like Domino's advertisement network is used as one of many different advertisement networks in applications which spam the user device with ads.

The advertisement spam seems to be connected to some rooting trojan families too. This suggests that the rooting trojans try to get elevated privileges on the device in order to persistently install applications which display advertisements coming

from the same ad network as is used by Domino. It is unclear whether the same authors are behind the Domino and the rooting trojans or whether the Domino authors just use rooting trojans as one of their distribution methods.

This whole ecosystem bears a strong resemblance to Triada and in fact may be connected to Triada. Based on the *Malwarebytes* blog post mentioned earlier in this paper [1, 2], xHelper in one case seems to stop being installed when the *Google Play* process is stopped. This points to the very likely infection vector – Triada backdoor.

As we have outlined in a blog post on the *Google Security* blog [3], Triada installed applications by backdooring the log function and waiting for the *Google Play* process to log a message. It then ran additional code which allowed Triada to install applications from the *Google Play* process context. If the *Google Play* process is stopped it will not log a message and the installation will not happen. Additionally, *Kaspersky* has confirmed [4] a connection between Triada rooting trojans (not the preinstalled backdoor) and xHelper.

We have contacted the OEMs that built devices that were preinstalled with Triada and each of them have made an updated system image available to remove Triada and xHelper from affected devices.

SAMPLE HASHES

The table below presents hashes for different binaries, applications and the source code package we were able to find. All of these samples are available on *VirusTotal*.

SHA256 hash	Description
`f00c1ddd2cd508d132415d59c243b2f4d30fd5845359fde3bb8dbb4f61c08a3e`	'Service' binary
`72e4a3ba2e64c8cf53640e2c4a11d23ec9f03b1f367c2011429ff21aa75404a9`	'Damon' binary
`cb51eabdab64e043eea37c42b6aea54f863e8cf564140b63d266235afe1744e8`	Source code package
`272c07ecabc0c3f9e2ccf761e231371fda73847126ab7aa3ce488757c1dd251f`	Adware application

INDICATORS OF COMPROMISE (IOC)

The table below shows some of the indicators of compromise which can be used to detect infection with Domino.

Description	Indicators
System property	ro.feature.browser_ext
Domain names	dominoppo.in dominoppo.site domino-ym.oss-ap-southeast-1.aliyuncs.com
URL patterns	/bus-webapi/rest/service/strategy /domino-webapi/rest/service/cm_list /bus-webapi/rest/service/ym_list
Executable file names	htfsk badamon baservice smsdamon smsservice rbn
Configuration directories	/data/rbn/ /data/smsconfig/
DES encryption key (string)	13572468

REFERENCES

[1] Collier, N. Mobile Menace Monday: Android Trojan raises xHelper. Malwarebytes. August 2019. https://blog.malwarebytes.com/android/2019/08/mobile-menace-monday-android-trojan-raises-xhelper/.

[2] Collier, N. Android Trojan xHelper uses persistent re-infection tactics: here's how to remove. Malwarebytes. February 2020. https://blog.malwarebytes.com/android/2020/02/new-variant-of-android-trojan-xhelper-reinfects-with-help-from-google-play/.

[3] Siewierski, L. PHA Family Highlights: Triada. Google Security Blog. June 2019. https://security.googleblog.com/2019/06/pha-family-highlights-triada.html.

[4] Golovin, I. Unkillable xHelper and a Trojan matryoshka. Secure List. April 2020. https://securelist.com/unkillable-xhelper-and-a-trojan-matryoshka/96487/.