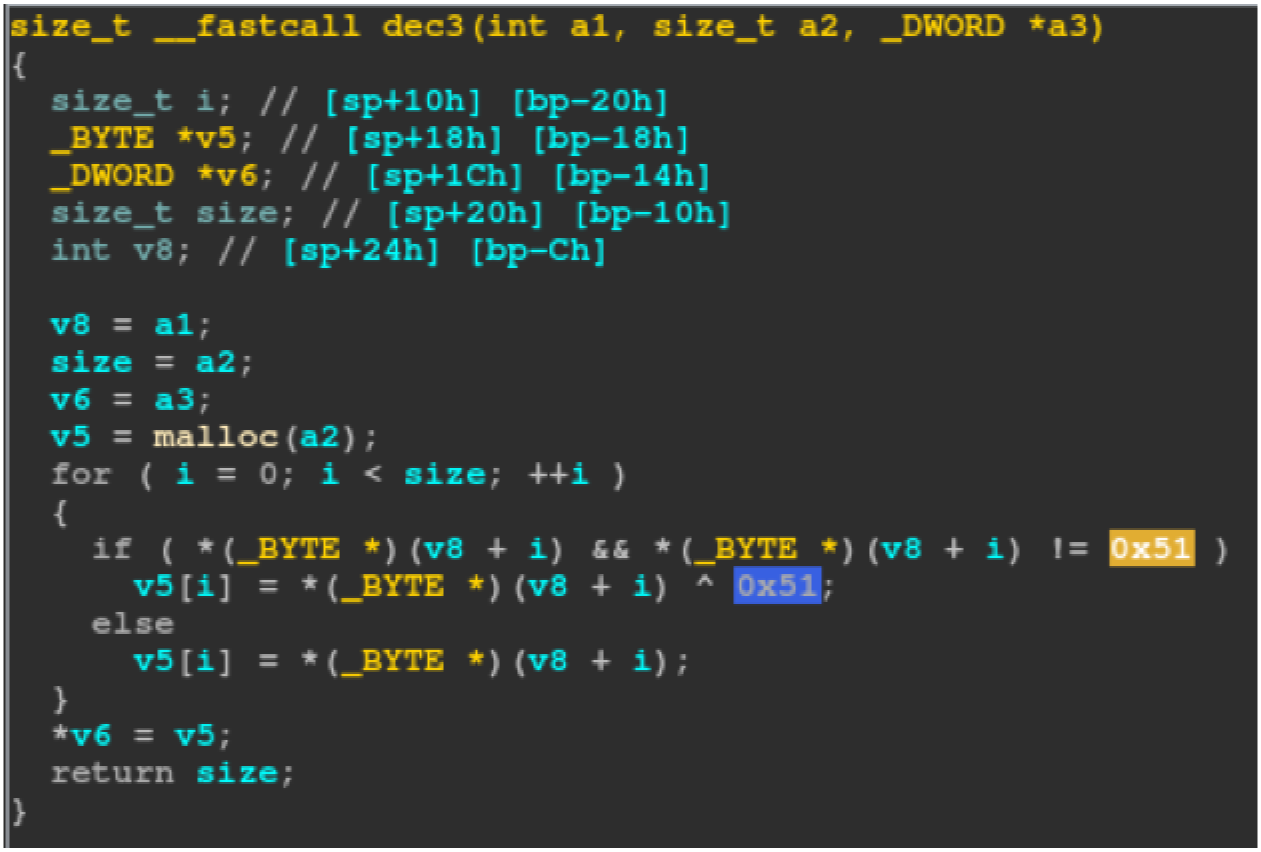
FunkyBot: A New Android Malware Family Targeting Japan

fortinet.com/blog/threat-research/funkybot-malware-targets-japan.html

September 4, 2019



```
size_t __fastcall dec3(int a1, size_t a2, _DWORD *a3)
{
  size_t i; // [sp+10h] [bp-20h]
  _BYTE *v5; // [sp+18h] [bp-18h]
  _DWORD *v6; // [sp+1Ch] [bp-14h]
  size_t size; // [sp+20h] [bp-10h]
  int v8; // [sp+24h] [bp-Ch]

  v8 = a1;
  size = a2;
  v6 = a3;
  v5 = malloc(a2);
  for ( i = 0; i < size; ++i )
  {
    if ( *(_BYTE *)(v8 + i) && *(_BYTE *)(v8 + i) != 0x51 )
      v5[i] = *(_BYTE *)(v8 + i) ^ 0x51;
    else
      v5[i] = *(_BYTE *)(v8 + i);
  }
  *v6 = v5;
  return size;
}
```

Last year, FortiGuard Labs identified a malware campaign targeting Japanese users. The campaign impersonated a logistics company and deployed an Android malware called FakeSpy.

We have been monitoring these actors and the phishing websites they created, and recently we noticed that they have started deploying a different Android payload.

As in their previous campaigns, this payload consists of a packer and a payload. However, both of these are different from the ones we have encountered previously.

In the following blog, we will provide a deep analysis of both the packing mechanisms as well as the deployed payload, which to the best of our knowledge is a new malware family. It may have been developed by the same people behind the campaign as a substitute for the too-well known FakeSpy malware family they have been using up to now. Based on logging strings found in the persistence mechanism of the payload, as seen in figure 7, we have decided to call this new malware family FunkyBot.

We will be analysing the following sample:

152be211ecd21c8abfd7c687a5ca8a17906f589c59055516e5482ff3fcf42dbf

Packer

The Packer is made of two separate parts:

- Java code contained in the *classes.dex* file
- Native code contained in the *libcsn.so* file

Java Functions

The code of the packer in the sample we analysed was obfuscated. Luckily, after some searching we were able to find an un-obfuscated version of the code, and we will be using that.

The reference sample for the packer is the following:

b4f3b7850c4332bcf85bbd64ebd6d837a3de64a03c1150cdd27e41599d2852b6

The first interesting function that is executed is *_attachBaseContext(Context base)*. This function accesses the configuration file contained in the asset folder of the APK. In this case, it is a JSON file called ‘*_dcfg_.data’*, and it loads the following parameters:

- “*size”*: identifies the number of ‘*.dex*’ payloads that the packer has to generate
- “*payloadType”*: identifies where the encrypted data of the payload is located
- “*isTestIn”*: flag used for testing purposes
- “*type”*: identifies the kind of encryption used

In the samples we analysed, we found the following two configurations:

{"size":2,"payloadType":0,"isTestIn":"0","type":3}
{"size":2,"payloadType":1,"isTestIn":"0","type":3}

The packer determines which version of Android it is running on in order to generate the proper payload. It also goes the extra mile and generates some fake dex files to possibly confuse malware analysts.

Figure 1: Creation of fake Dex files

It then checks the ‘*payloadType’* value, and if the value is equal to 1, it will copy the asset data to another folder. Otherwise, it will proceed without moving anything, as it uses the *classes.dex* file loaded in memory instead.

Figure 2: Extraction of Dex encrypted file

JNI Functions

The Class *JNITools* declares a set of native functions that are contained in *libcsn.so*.

Figure 3: JNITools native functions declaration

The native *JNI_OnLoad* function, which is run when the library is loaded, registers the native functions declared in *JNITools*, allowing them to be called differently than the usual scheme of *Java_<className>_<FunctionName>*, probably to make the reversing process harder.

If the value of the configuration variable '*type'* is different from 0 (which means the payload needs to be decrypted), the code accesses the folder */data/data/<appname>/app_csn0/* and creates a folder '*.unzip'* in it. Note that the name of the folder contains a dot as the first character, making it invisible to a normal ls command.

The decryption routines are run on a file generated from the encrypted payload data. This data is obtained from one of two sources, based on the value of '*payloadType'*:

- 0: the '*classes.dex'* file, containing all the executed code
- 1: an asset file, in this case *assets/csn-enc.data*

In the first case, the packer accesses the */proc/<pid>/maps* file to locate the memory where the *classes.dex* file is loaded, and then looks for a specific set of characters in memory that identify the beginning of the encrypted data. In this case, the magic word is `csn_`. When found, it starts copying from that point onwards.

Figure 4: Searching for Magic 'csn_'

The different values of the configuration variable '*type'* correspond to different decryption routines. The code supports the following values:

- 0: No encryption
- 2/3 : variations of XOR based decryption with the value `0x51` (81)

Figure 5: Decrypt Switch

In the samples presented here, the configuration always had a value of 3, which corresponds to the following decryption function:

Figure 6: Decryption 'type' 3

These routines then generate a '*classes.dex'* payload file that is loaded by the *ClassLoader*.

Payload: FunkyBot

In the sample analysed, the payload consisted of two *.dex* files. One being a copy of the original legitimate application that the malware is impersonating, and the other being the malicious code.

The payload is started by calling the method `*runCode*` class `*com.wfk.injectplugin.EntryPoint*` through Java reflection. This method starts `*KeepAliceMain.start()*`*.*

Figure 7: EntryPoint

KeepAliceMain

This Class is used as persistence mechanism by the malware. It uses an open source library that can be found on Github to keep the service alive on the device. It also allows the malware to mute sounds from the device, even though in this specific instance this functionality is not used.

This class periodically re-launches the main service used by the malware to create a gRPC connection to a remote server.

GRPC Client

Command and Communication Address

The server address is not hardcoded in the `classes.dex` file, but it is retrieved during execution. The code executes the function `*GprcsUtils.Regist_Server(String str)*`, which calls `*UrlTool.loadIPAddrFromIns()*` to extract the C2 URL.

Figure 8: Loaded IP from Instagram

Much like Anubis used to do with fake Telegram and Twitter accounts, this malware uses social media to obtain its C2: it downloads the webpage of a photo-less Instagram account. It then extracts the biography field of this account and decodes it using Base64.

Figure 9: Fake Instagram account

Finally, the resulting string is decrypted using DES and a key is generated using the value `d2a57dc1d883fd21fb9951699df71cc7` as its seed (which happens to be the MD5 hash corresponding to the word ‘app’), which can be seen in figure 8 under the String variable str3.

The resulting URL is *149.28.24.166:11257* and the fake accounts have been reported to Instagram.

SMS Service

After the connection to the server is started, the malware proceeds to collect and send the following information about the device:

- IMEI
- IMSI
- Phone number
- List of contacts

The amount of exfiltrated information is relatively limited, especially when compared to bigger families like Anubis, Cerberus, or Hydra. However, like previous campaigns, it also features aggressive spreading techniques.

After having sent all of the device’s contacts to the C2, it waits for it to respond with a telephone number and a message body to construct an SMS. This strategy has been used by multiple campaigns, including FakeSpy and MoqHao, to enable the malware to spread in a worm-like fashion. It is logical to assume that this sample would do the same.

It is interesting to note that the malware identifies the provider of the SIM card and looks specifically for a specific Japanese telecommunication provider. To do so, it checks the IMSI (International Mobile Subscriber Identity) value of the device. This value is composed of two halves: the first identifies the provider, and the second is unique to the specific device.

The malware checks to see if the first half corresponds one of its listed values, which are all connected to the aforementioned provider.

Figure 10: Checking if the device is served by a specific provider

At the beginning, we thought the function was going to possibly be used for some targeted action towards the customers of this provider. Instead, if the function *`is<Provider>()`* returns *true*, then the malware simply proceeds to increase the value controlling the maximum number of SMS messages it allows itself to send.

After some research, we concluded that this behaviour might just be because the provider enables customers to send free SMS messages to each other, increasing the amount of traffic a single infected device is capable of generating before arousing suspicion.

Finally, the malware is able to set itself as the default SMS handler application, and uses this to upload to the C2 all the received messages. This functionality can be very dangerous, considering that most banks currently use two-factor authentication through SMS.

Conclusion

By monitoring the campaign primarily targeting Japanese service providers, FortiGuard Labs was able to identify this campaign and what, to the best of our knowledge, is a new malware family.

During our analysis, we also encountered other samples that were not completely developed and lacked some of the functionalities discussed in this blogpost, suggesting that the malware is currently under development and is being tested in the wild.

The capabilities of this family are limited at the moment, but the fact that we were able to find different samples that showed significant improvement in the span of a few weeks shows that this family should not be underestimated.

FortiGuard Labs will continue to monitor this campaign as it evolves.

-=FortiGuard Lion Team=-

Solutions

Fortinet customers are protected against this malware by the following Signatures:

- Packer: Android/Agent.DDQ!tr
- Payload: Android/Funky.A!tr and Android/Funky.B!tr

Acknowledgements

I would like to thank Evgeny Ananin for his help in the research needed for this blogpost.

IOC List:

Packers:

b4f3b7850c4332bcf85bbd64ebd6d837a3de64a03c1150cdd27e41599d2852b6
152be211ecd21c8abfd7c687a5ca8a17906f589c59055516e5482ff3fcf42dbf

Payloads:

02036825d69208612fd281b3d4fd9be06fc315addeac1fe8872eb2cc9f6f1fcd
beb6cb245f6597b6d2b9e9232774329b94f2eada5980a3cb28f9100cc161f4a4

CCs:

149[.]28[.]24[.]166[:]11257
108[.]61[.]187[.]156[:]11257