Small footprint inspection techniques for Android

Damien Cauquil, Pierre Jaury

29C3
December 29, 2012

Introduction

Damien Cauquil

Company Sysdream (head of research)

Twitter @virtualabs

Blog `http://virtualabs.fr`

Pierre Jaury

Company Sysdream

Twitter @kaiyou_

Blog `http://kaiyou.org`

Sysdream, IT security services

Location Paris, France

Website `http://sysdream.com`

Table Of Contents

1 Reverse engineering and side effects

2 Reverse engineering on Android

3 Minimal footprint techniques

4 Fino approach and implementation

5 Demo

Reverse engineering and side effects

1. Reverse engineering and side effects
   - Why reverse engineering?
   - Static or dynamic analysis?
   - It is all a matter of physics
   - Side effects amplification
2. Reverse engineering on Android
3. Minimal footprint techniques
4. Fino approach and implementation
5. Demo

Why reverse engineering?

- Curiosity
- Security assessment
- Cracking
- Interoperability
- ...

→ Exploring the internals

→ Understanding the program

Static or dynamic analysis?

Static analysis

- Look at the program
- Explore the binary
- Use disassembly tools
- Read some low-level bytecode
- Make plenty of assumptions

Dynamic analysis

- Monitor what is available
- Run the program
- Run the program, again
- . . . (much like fuzzing)
- Make some other assumptions

It is all a matter of physics

And those very annoying side effects

Generalizing about the internals given observations

Physics

- Consider a system
- Monitor the system
- Apply various actions
- Generalize a law
- **Measure uncertainty**

Dynamic reverse engineering

- Consider a program
- Monitor the program
- Apply various actions
- Generalize about the program
- **Side effects**

Side effects amplification

Anti-debugging and other very nice techniques

Side effects are bad, yet one might enjoy...

- amplifying them on purpose
- making them terrible in non-native environments
- creating new sources of side effects
- targetting tricky sources of side effects
- putting analysts in terribly hairy situations

$\longrightarrow$ anti-debugging

Reverse engineering on Android

1 Reverse engineering and side effects

2 Reverse engineering on Android
- State of the art
- Android reverse cookbook
- Why so unsatisfied?

3 Minimal footprint techniques

4 Fino approach and implementation

5 Demo

State of the art

(awe)?Some tools

Static analysis

- Smali/Baksmali
- APK-tool
- dex2jar
- jd-gui
- ...

Dynamic analysis

- Android virtual machine
- ARM emulators
- DDMS
- APKill
- ...

Android reverse cookbook

The daily life of a reverse analyst

- Wake up
- Run the application on a standard device
- Run the application inside an emulator
- Inspect the memory
- Inspect network traffic
- Fetch and disassemble the package
- Read the dalvik dex bytecode and match it to behaviors
- Inject some home-cooked hooks with Smali
- ...

Why so unsatisfied?

We remain bulls in china shops

- No proper anti-anti-debugging tools

→ Spend hours patching Smali code to bypass protections

- Heavy debugging tools that are easily detected
- Many unexpected side effects due to virtulization
- More side effects due to execution path/memory inspection
- Patches adding even more side effects

→ Biased reports

Minimal footprint techniques

1 Reverse engineering and side effects

2 Reverse engineering on Android

3 Minimal footprint techniques
- Why go minimal?
- Measuring the footprint
- Minimizing the footprint

4 Fino approach and implementation

5 Demo

Why go minimal?

- Side effects are bad
- Be faster (less overhead)
- Be stealthier
- Go further

Measuring the footprint

How much do these side effects really annoy you?

Side effects are bad. How bad?

Most of the time

- Time overhead (slow down the program)
- Space overhead (use more memory)
- Concurrency constraints

Worst case scenario

- State inconsistencies, deadlocks
- Access conflicts
- Application crashing
- Device freezing

Minimizing the footprint

(((Anti-){2})+)debugging techniques, and more

Many technical responses:

- minimizing the space footprint

→ go modular!

- minimizing the time overhead

→ live aside, do not hook!

- avoiding state inconsistencies

→ always prefer pure functions!

- avoiding concurrency conflicts

→ always check the current thread!

Minimizing the footprint

(((Anti-){2})+)debugging techniques, and more

A general approach:

- no patch of existing bytecode
- simple and modular payload
- no interaction with unknown threads
- as little memory interaction as possible
- stick with pure functions and read access as far as possible
- communication only through covert channels
- no unintended user interaction (no graphical popup, . . .)

$\longrightarrow$ remain as silent as possible

Fino approach and implementation

1 Reverse engineering and side effects

2 Reverse engineering on Android

3 Minimal footprint techniques

4 Fino approach and implementation
- Minimal from scratch
- Dead code injection
- Covert communication
- Entry point discovery
- Fino

5 Demo

Minimal from scratch

Because patching is great, but...

Usual solution for debuggers:

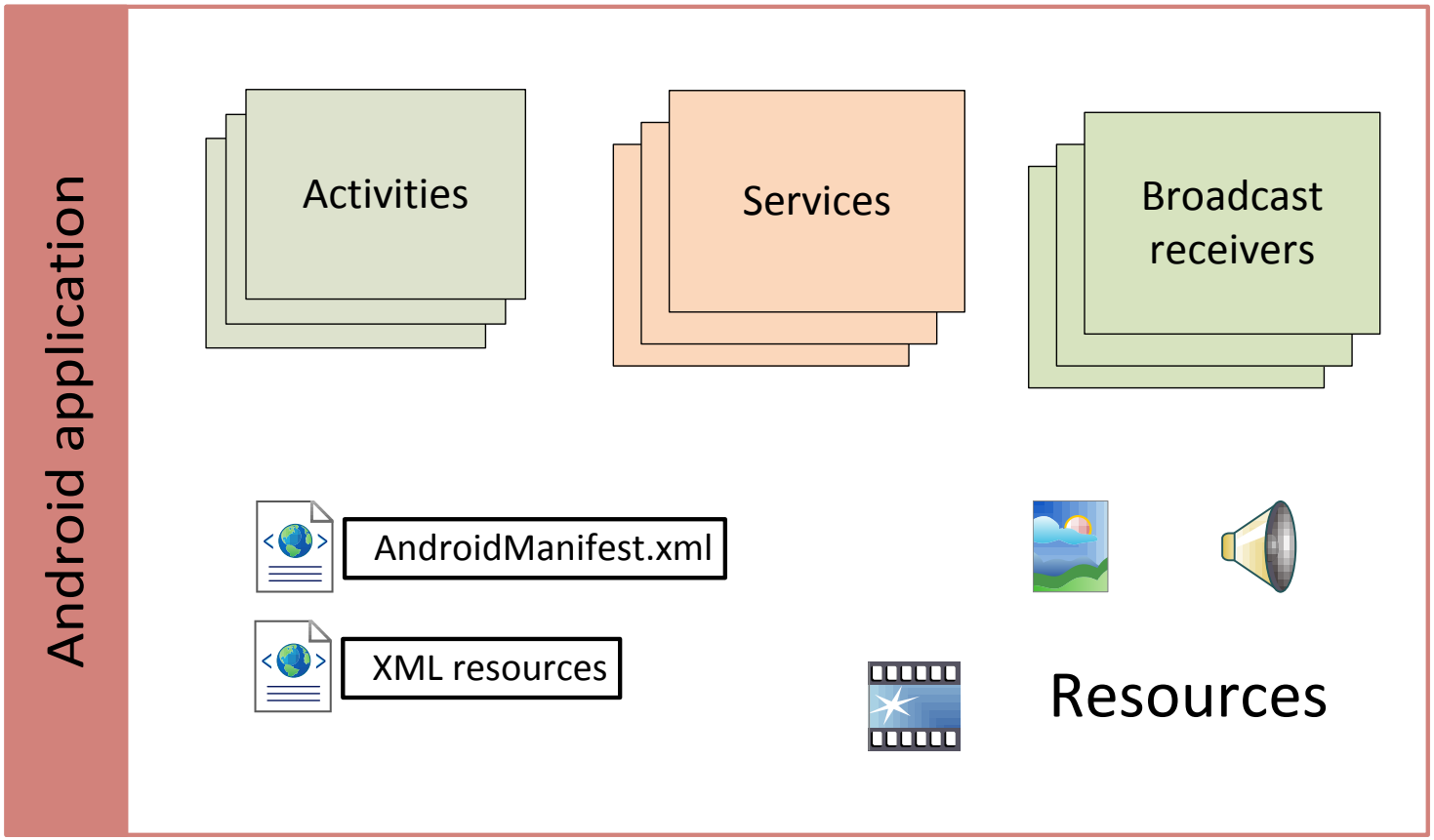
1. write some sketchy debugging code
2. add plenty of modules for execution and memory inspection
3. note the many side effects and anti-debugging snippets
4. patch the debugger, then go to 2

A somehow different approach:

1. put *avoiding side effects* as a core design choice
2. write a modular debugging framework
3. add less modules because of the design constraints

Dead code injection

What does an Android application look like?



Dead code injection

. . . which appears to be undead

Dead code injection

- Inject some code in the application
- The code is never referenced
- Invoked by a system mechanism

→ event handler

→ broadcast receiver

→ bound service

Service injection

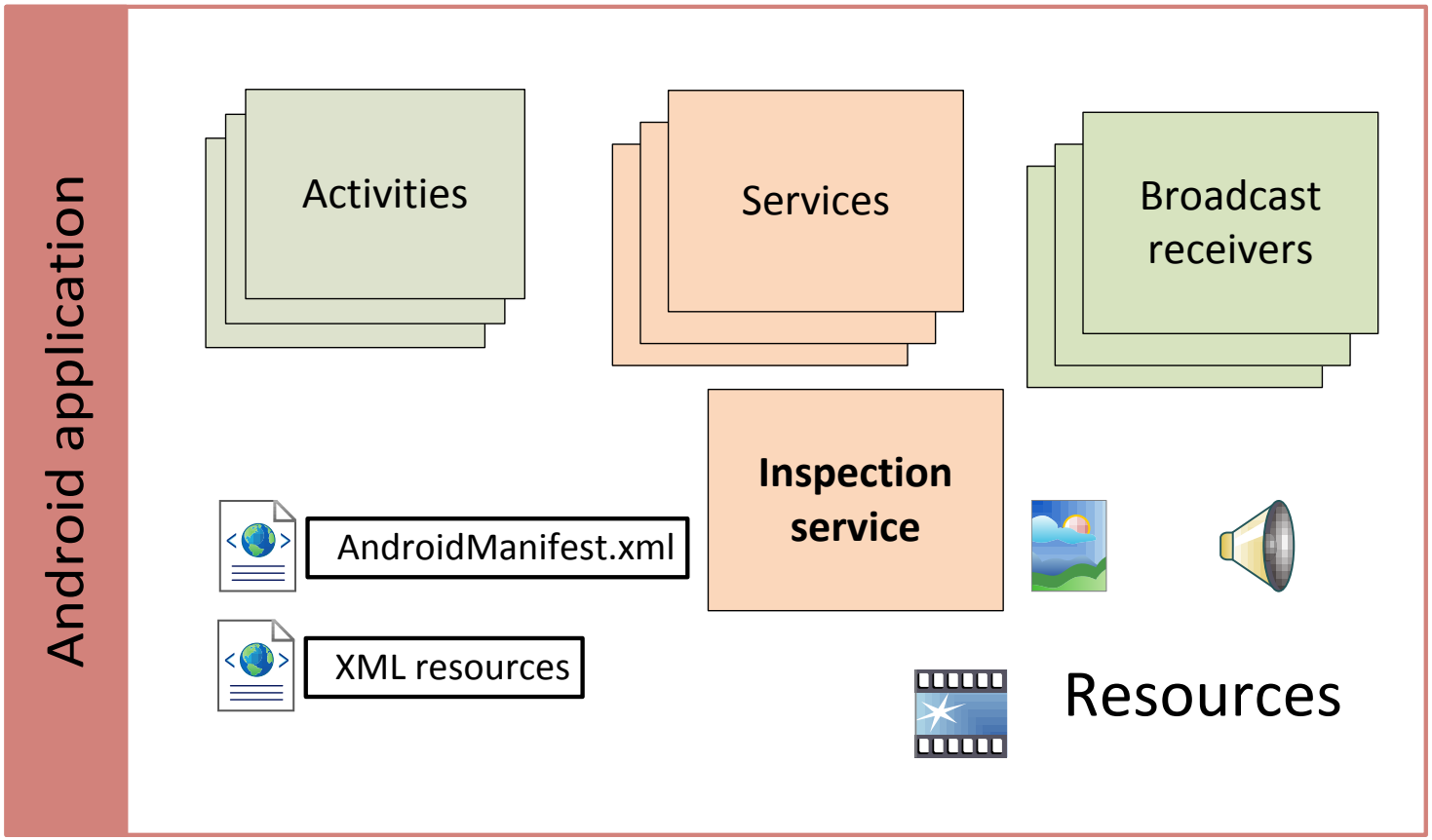
- Service injected in the APK
- Never referenced in the code
- Action filtered declared
- Invoked by the system with service binding

→ Silent until invoked

→ Launched in the application thread

Dead code injection

What does it look like once injected?



Covert communication

You really do not want side effects, do you?

How to communicate with the injected code?

- Through network sockets: system/device dependant
- Same goes for local sockets
- Through the graphical interface: out of the question

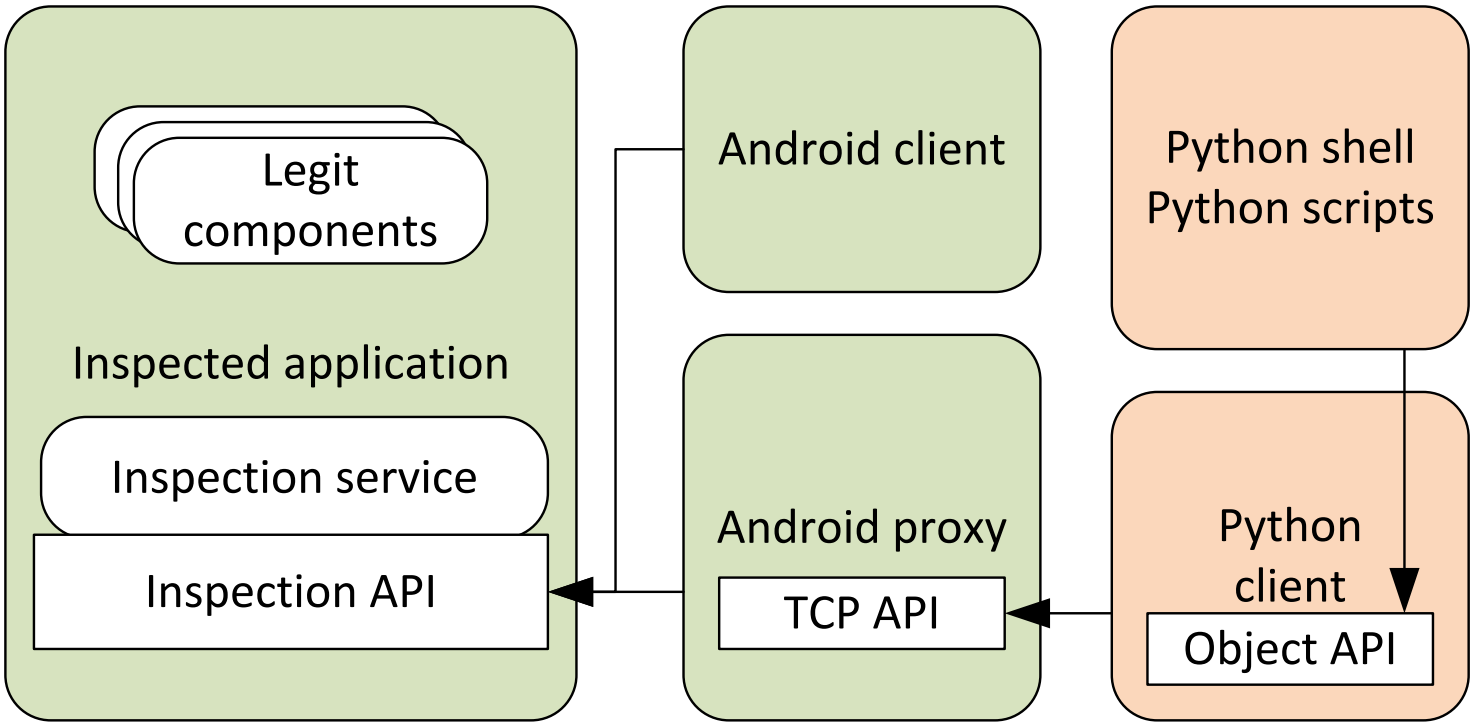
→ Through plain service remote procedure calls

→ Only native types as arguments and returns

→ A client or a proxy is necessary

Covert communication

Client? Proxy?



Entry point discovery

The story of a poor lonesome service

- Communication with some dead code
- Goal: memory inspection, function call, ...
- Mean: mostly Java reflection API

→ Necessary to get some entry points

→ `Application.ActivityLifecycleCallbacks`

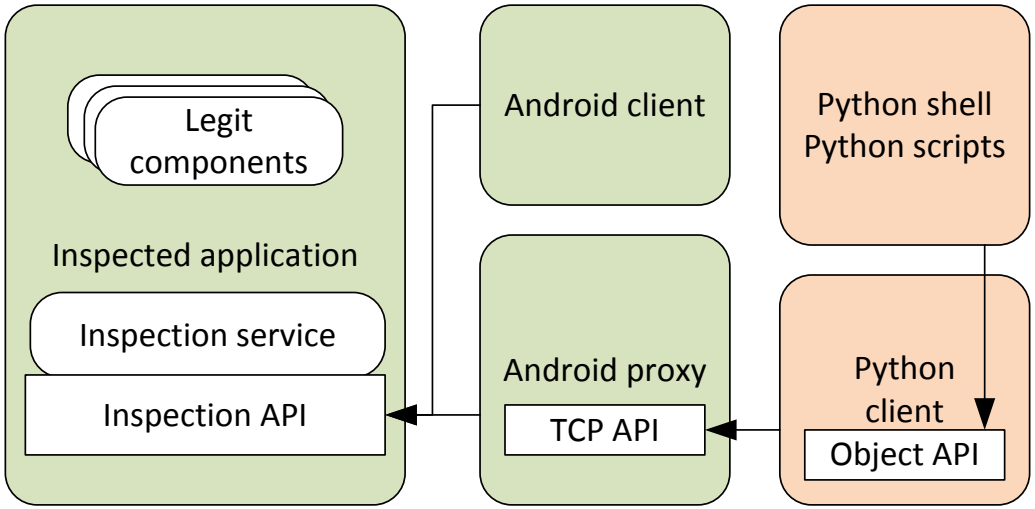
Fino

'cause we finally built some tool

Fino Low footprint inspection service

Gadget Android-side API proxy

Client Python object oriented API and interactive shell



Demo

1 Reverse engineering and side effects

2 Reverse engineering on Android

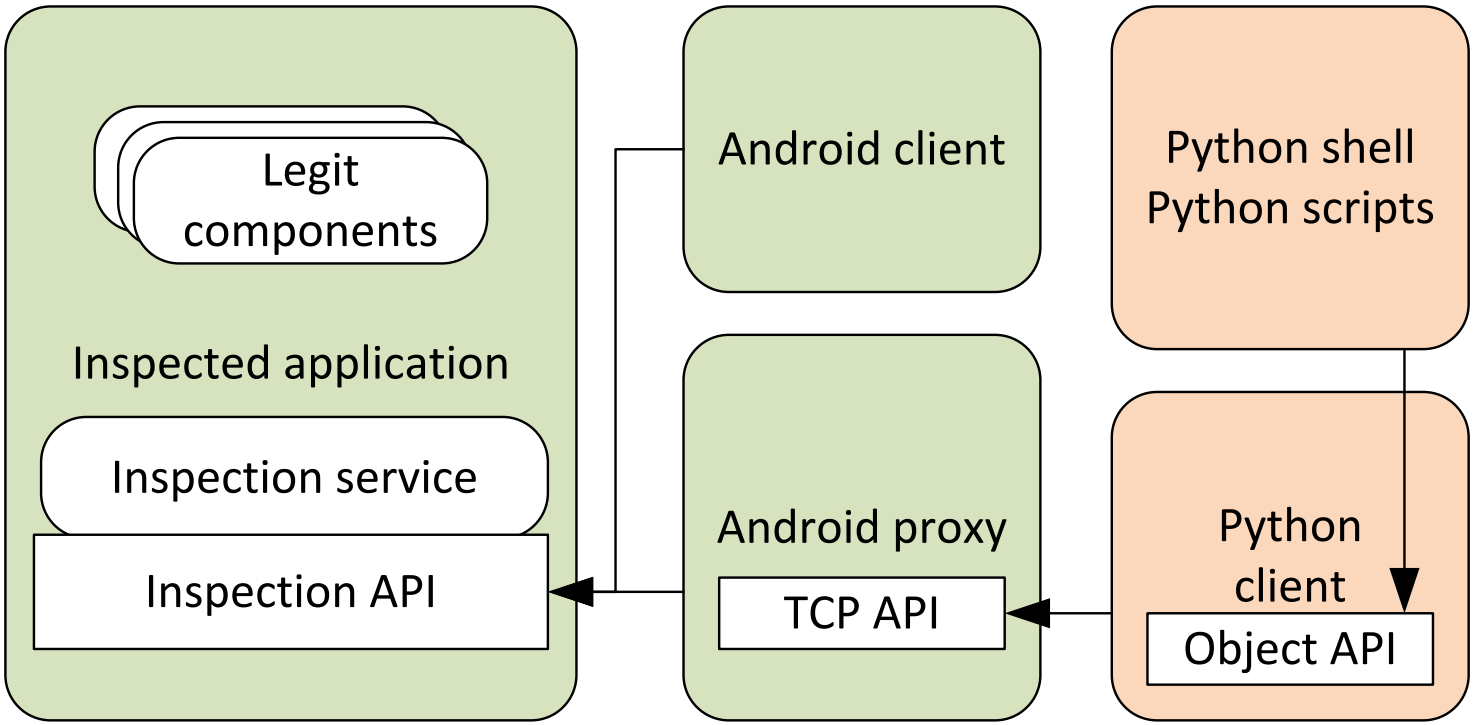
3 Minimal footprint techniques

4 Fino approach and implementation

5 Demo
- Demo 1
- Demo 2
- Demo 3
- Conclusion

Demo 1

Reminder



Demo 1

```
public class Obfu
{
  private static String s1 = "lbi8r6m5u66/vvqprqiztL0=";

  private static byte[] d(byte[] paramArrayOfByte)
  {
    for (int i = 0; ; i++)
    {
      if (i >= paramArrayOfByte.length)
        return paramArrayOfByte;
      paramArrayOfByte[i] = (byte)(0xDA ^ paramArrayOfByte[i]);
    }
  }

  public static String get()
  {
    return new String(d(Base64.decode(s1, 0)));
  }
}
```

Demo 2

```
public class LicenseManager
{
  public boolean CheckKey(String paramString)
  {
    boolean bool = false;
    try
    {
      MessageDigest localMessageDigest = MessageDigest.getInstance("MD5");
      localMessageDigest.update(paramString.getBytes(), 0, paramString.getBytes().le
      byte[] arrayOfByte = localMessageDigest.digest();
      StringBuffer localStringBuffer = new StringBuffer();
      for (int i = 0; ; i++)
      {
        if (i >= arrayOfByte.length)
        {
          bool = localStringBuffer.toString().equals("68435a9a7507710fafa909704b8de0
          break;
        }
        localStringBuffer.append(Integer.toString(256 + (0xFF & arrayOfByte[i]), 16)
      }
```

Demo 2

```
package com.sysdream.demo2;
import android.util.Log;

class LicenseManager {
    public boolean CheckKey(String key) {
        return false;
    }
}
class MyLicenseManager extends LicenseManager {
    public MyLicenseManager() {
        super();
    }
    public boolean CheckKey(String key) {
        return true;
    }
}
```

Demo 3


I DON'T ALWAYS PLAY MOBILE GAMES
BUT I MAKE TOP 1 WHEN I DO
memegenerator.net

Conclusion

Damien @virtualabs

Pierre @kaiyou_

Fino `http://github.com/sysdream/fino`

Gadget `http://github.com/sysdream/gadget`

Client `http://github.com/sysdream/gadget-client`

Questions?