VANTAGEPOINT

Fixing Mobile AppSec

The OWASP Mobile Security Testing Project

/usr/bin/whoami

Hi everyone my name is Sven.

•Principal Security Consultant at Vantage Point Security

•Based in Singapore, originally from Germany

•Unix nerd since 1999

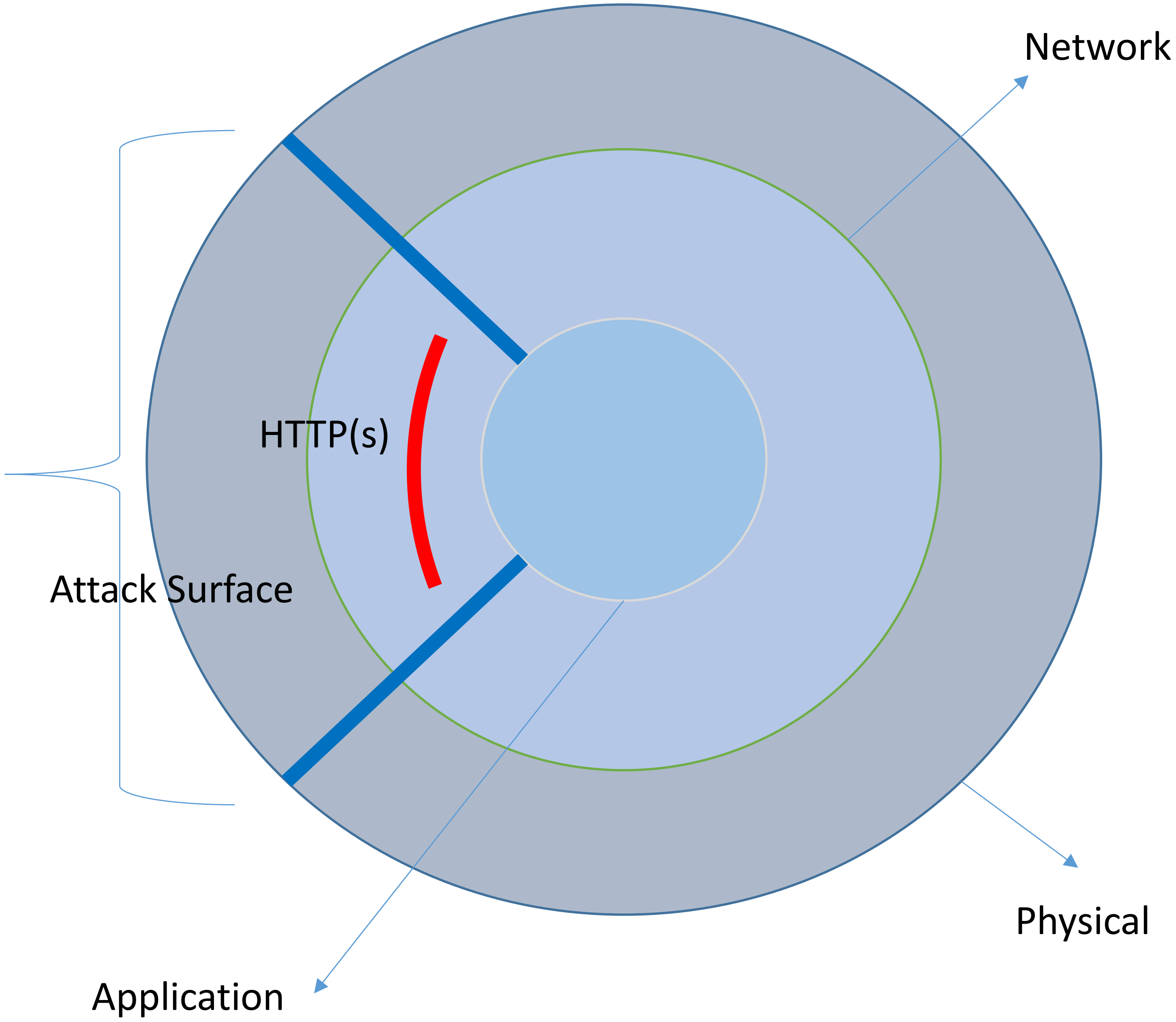
•Professional Penetration tester since 2010

•Security Architect for Web and Mobile Apps during SDLC

•One of the project leaders for the OWASP Mobile Security Testing Guide (MSTG) and Mobile AppSec Verification Standard (MASVS)

Why Mobile Application Security?

- It all started with Network & Physical Security
  - Protecting the perimeter
  - Ensuring endpoints are secure
- Network Security still plays an important part
- **But, different skills are required to support Mobile Application Security**



Common Situation

Key Pain Points	**Impact**
-Lack of security capabilities in development teams	-High number of security defects -Significant amount of re-work -Constant delays and increased cost
-Security addressed at the end of the development life cycle	-Team friction and stress -Missed deadlines -Delayed releases
-Insufficient supporting technologies	-Security bottlenecks are created -Low-level of visibility of security posture -Increased manual effort

OWASP Mobile Security Project – Our “Products”

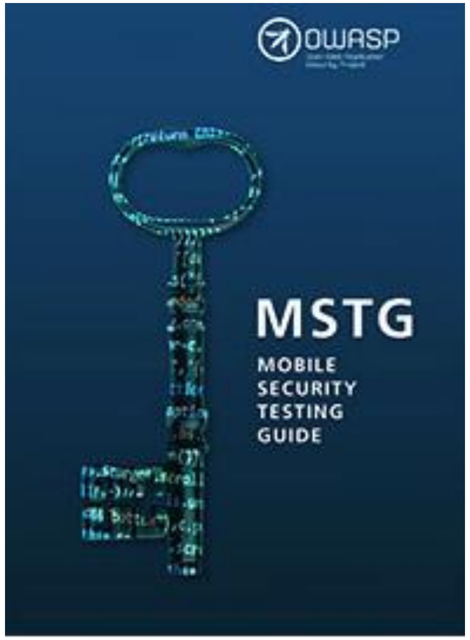
Mobile AppSec Verification Standard

PDF Download

https://github.com/OWASP/owasp-masvs/releases

Mobile AppSec Checklist

Excel ☹



Mobile Security Testing Guide

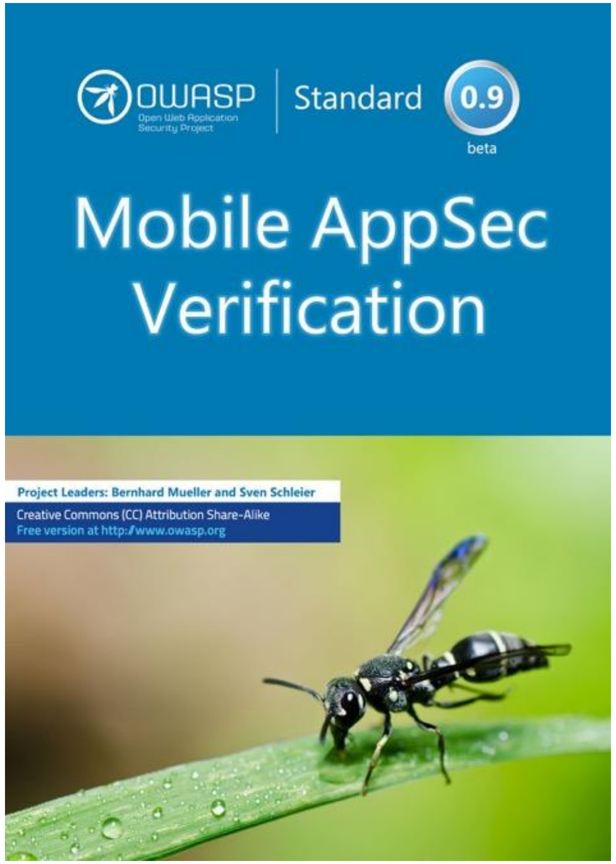
Target 700+ pages
~75% done
Free Ebook & Real,
Printed Book!

https://leanpub.com/mobile-security-testing-guide

OWASP Mobile Application Security Verification Standard (MASVS)

- Started as a fork of the ASVS (https://www.owasp.org/index.php/Category:OWASP_Application_Security_Verification_Standard_Project)
- Formalizes best practices
- Mobile-specific, high-level, OS-agnostic

#	Description	L1	L2
2.1	System credential storage facilities are used appropriately to store sensitive data, such as user credentials or cryptographic keys.	✓	✓
2.2	No sensitive data is written to application logs.	✓	✓
2.3	No sensitive data is shared with third parties unless it is a necessary part of the architecture.	✓	✓
2.4	The keyboard cache is disabled on text inputs that process sensitive data.	✓	✓
2.5	The clipboard is deactivated on text fields that may contain sensitive data.	✓	✓



Opinions, opinons, opinions…

Sample Question: Do we recommend using E2E encryption?

Request

Raw | Params | Headers | Hex

```
POST /middleware/Servlet HTTP/1.1
Content-Type: application/x-www-form-urlencoded;
charset=utf-8
Content-Length: 565
Host: uat.enterprise.com
Connection: close
Cookie: JSESSIONID=0000NSKFxjwj8hGmYQDpvsm0q05:18d379bnl
User-Agent: okhttp/3.4.1

platform=android&deviceId=KVJJyjk2aQ1uObOtkA1qLLuO4VvF1NYo
CT6lDCPMKW6i4bzUNy23sAcXXa3Y7W6w&&userId=%2BL8%2B%2BS
ZO4uCSTPHrmtY8sPYSjDbC%2BYw0XZ46tEXjBz4%3D&randomNumb
er=&serviceName=P2PLOGIN&unencyptedPINlength=6&channel=rc
&serviceID=login&encryptedPIN=398082b5048e1ea3826ee78b627
5d1945019dceeaa36df5b40e2bb3b16abc25482d258ca9fd13acf314
34f1aa51cc308d758ed2a1f244bcd6cc81a8e2a2ae60711b7fcf124a4
71f0446723d80baa814f8e76d67c1d461a59f9725a4a8a3c17891de
70ab0d2c3056e231a943dc5539632ed634c3d242771c9668c4b49c9
8f2fc7&ipAddress=192.168.1.138&appver=2.6.4
```

Opinions, opinons, opinions…

Sample Question: Do we recommend using E2E encryption?

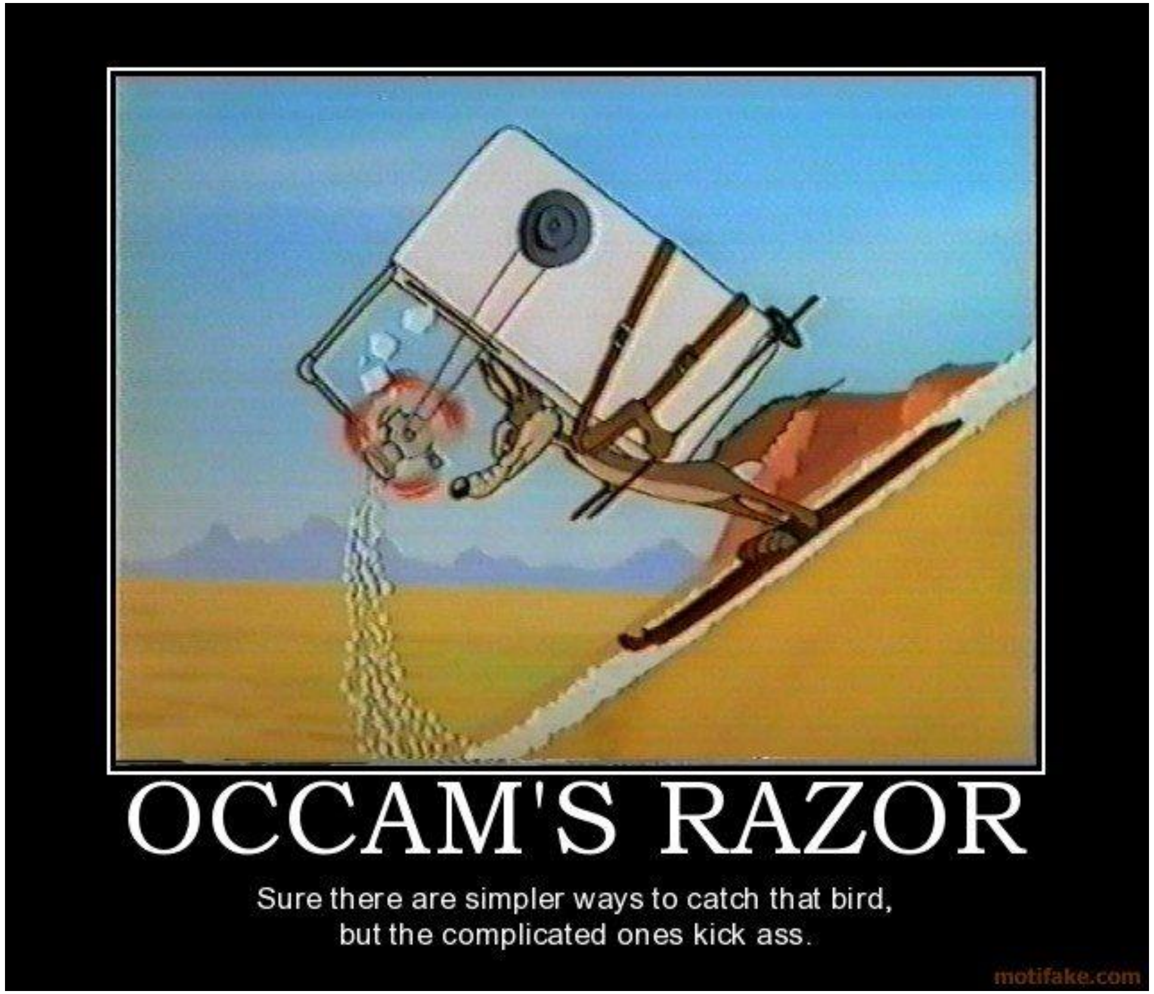
Pros

- Additional security layer
- Protects data in case TLS tunnel is compromised
- Protects data from exposure to intermediate systems

Cons

- Introduces additional complexity
- Implementation prone to errors
- Adds security by obscurity
  - Makes testing difficult
  - False sense of security
- Doesn’t add much security beyond what TLS already provides

Our Philosophy



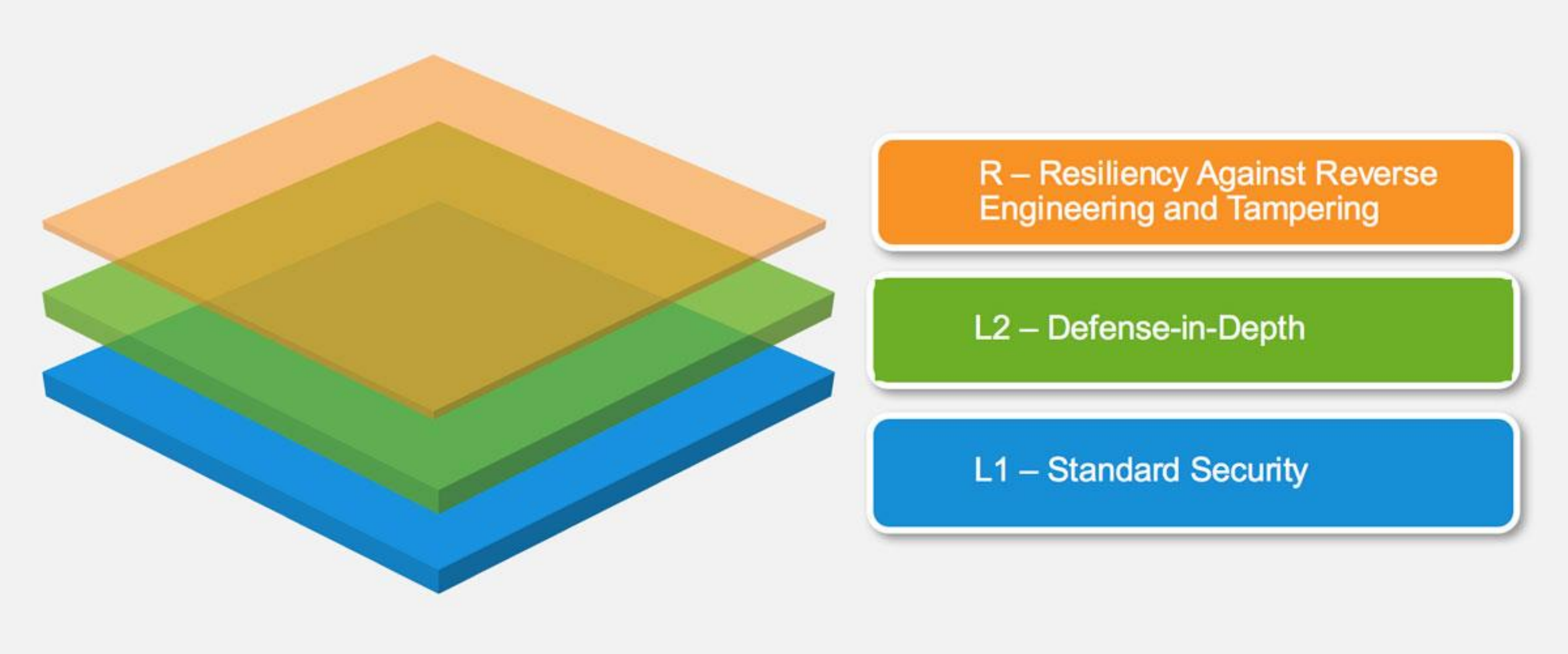
43 Security Requirements

19 Defense-in-Depth Measures

13 Anti-Reversing Controls

Covered in In 8 domains

Keeping Things Flexible: Requirement “Levels”



Keeping Things Flexible: Requirement "Levels"

MASVS-Level 1 (L1): Security best practices applicable to **all** mobile apps.

Example:

Security Verification Requirements

The vast majority of data disclosure issues can be prevented by following simple rules. Most of the controls listed in this chapter are mandatory for all verification levels.

#	Description	L1	L2
2.1	System credential storage facilities are used appropriately to store sensitive data, such as user credentials or cryptographic keys.	✓	✓
2.2	No sensitive data is written to application logs.	✓	✓
2.3	No sensitive data is shared with third parties unless it is a necessary part of the architecture.	✓	✓

Keeping Things Flexible: Requirement "Levels"

MASVS-Level 2 (L2): Defense-in-depth controls for sensitive apps (e.g. financial transactions)

Example:

Security Verification Requirements

#	Description	L1	L2
5.1	Data is encrypted on the network using TLS. The secure channel is used consistently throughout the app.	✓	✓
5.2	The TLS settings are in line with current best practices, or as close as possible if the mobile operating system does not support the recommended standards.	✓	✓
5.3	The app verifies the X.509 certificate of the remote endpoint when the secure channel is established. Only certificates signed by a trusted CA are accepted.	✓	✓
5.4	The app either uses its own certificate store, or pins the endpoint certificate or public key, and subsequently does not establish connections with endpoints that offer a different certificate or key, even if signed by a trusted CA.		✓

Keeping Things Flexible: Requirement "Levels"

MASVS- Resiliency Against Reverse Engineering and Tampering (R):
(Optional) Tamper-proofing to counter specific client-side threats.

Impede Dynamic Analysis and Tampering

#	Description	R
8.1	The app detects, and responds to, the presence of a rooted or jailbroken device either by alerting the user or terminating the app.	✓
8.2	The app prevents debugging and/or detects, and responds to, a debugger being attached. All available debugging protocols must be covered.	✓

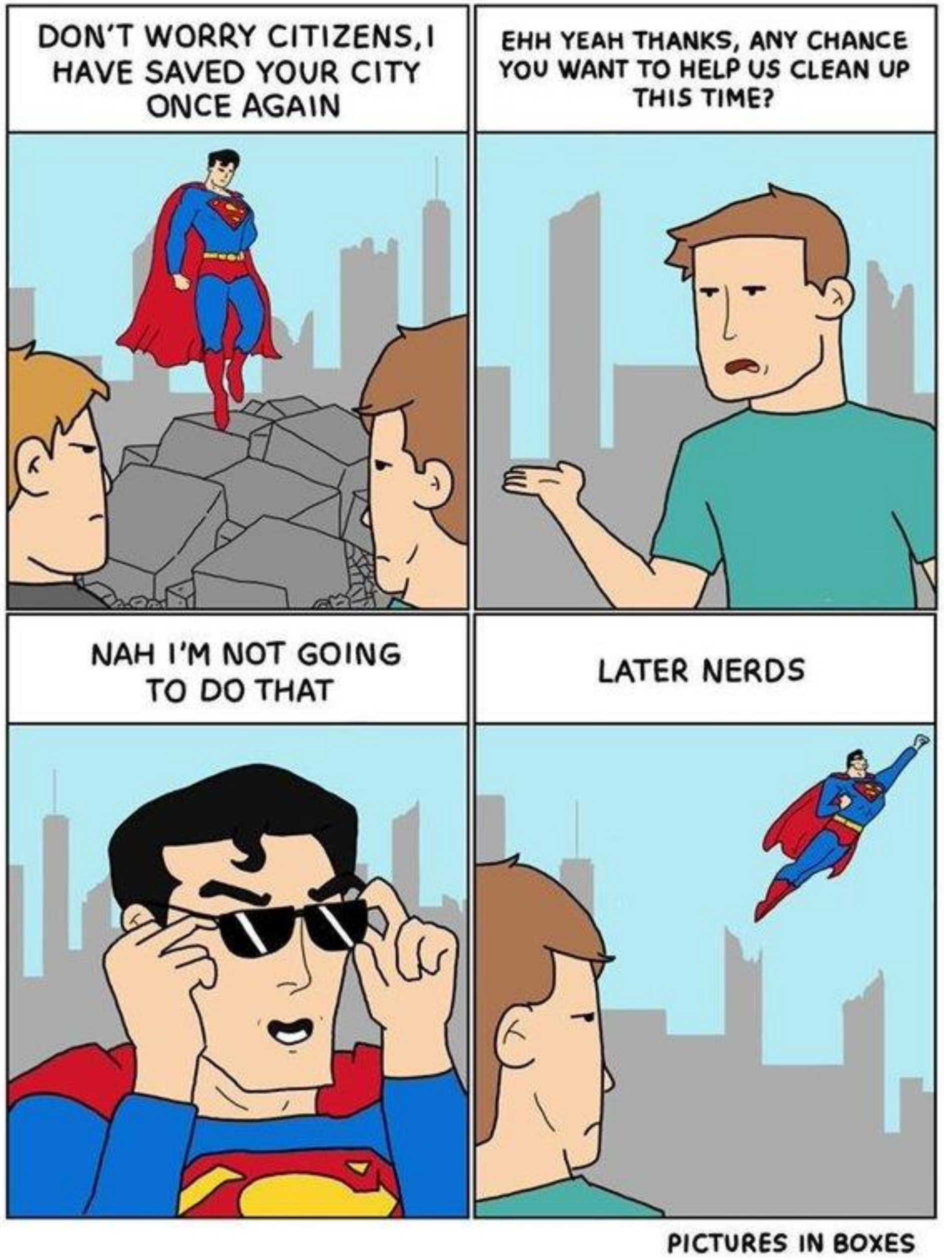
Level 1 vs. Level 2

#	Description	L1	L2
5.1	Data is encrypted on the network using TLS. The secure channel is used consistently throughout the app.	✓	✓
5.2	The TLS settings are in line with current best practices, or as close as possible if the mobile operating system does not support the recommended standards.	✓	✓
5.3	The app verifies the X.509 certificate of the remote endpoint when the secure channel is established. Only certificates signed by a valid CA are accepted.	✓	✓
5.4	The app either uses its own certificate store, or pins the endpoint certificate or public key, and subsequently does not establish connections with endpoints that offer a different certificate or key, even if signed by a trusted CA.		✓

Might be overkill for some apps!

Ok, so why are security requirements so important?

To avoid this:
Pentesters after turning a report in...

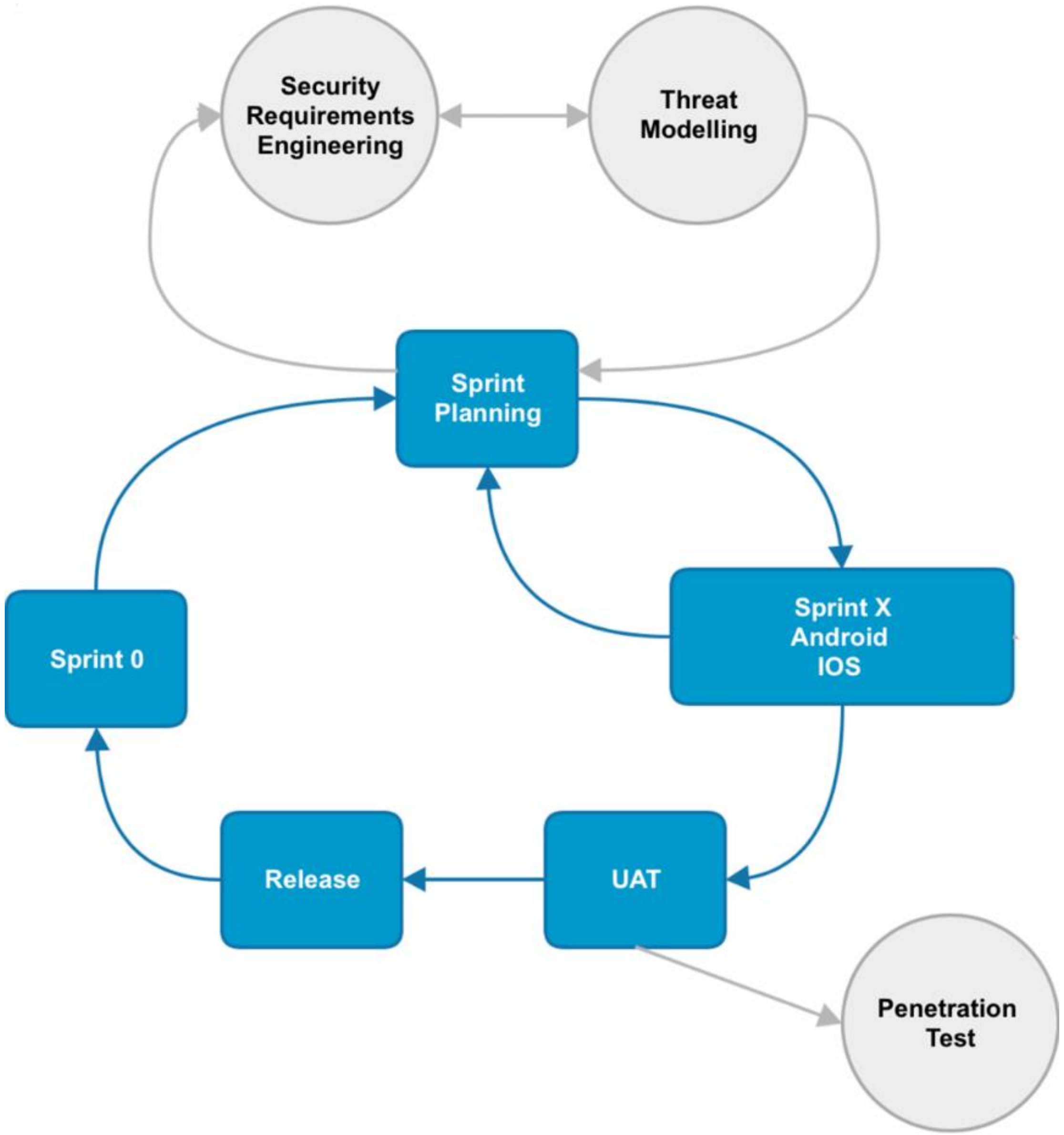


Ok, so why are security requirements so important?

- They enable you to build security into the app from the beginning
- They should be identified and defined already in the early stages of the SDLC
- Security requirements should be mapped to the user stories / journeys to address real problems

Ok, so why are security requirements so important?

Goal:
Build security in from the beginning!



How To Use the MASVS (as Developer)

Preparation during project kick-off (or Sprint 0):

- What MASVS level (L1, L2, R) and requirements are appropriate for the app?
- Use the MASVS as starting point and extend it with custom requirements as needed
- All involved parties need to agree on the decisions made
- This is the basis for all design decisions and security activities

Track the security requirements during development and implement them:

- Ideally in your issue tracking (e.g. Jira)
- Excel Checklist is available as an alternative

https://github.com/OWASP/owasp-mstg/tree/master/Checklists

How To Use the MASVS (as Security Tester)

Share the status of your security requirements with the Penetration Tester beforehand:

- This will allow him to focus on specifically these security controls
- Makes testing more efficient, as things like SSL Pinning might be out of scope according to your decision and then it won't be raised as vulnerability
- Makes testing consistent and tester and developers are on the same page

OWASP Mobile Security Testing Guide Standard (MSTG)

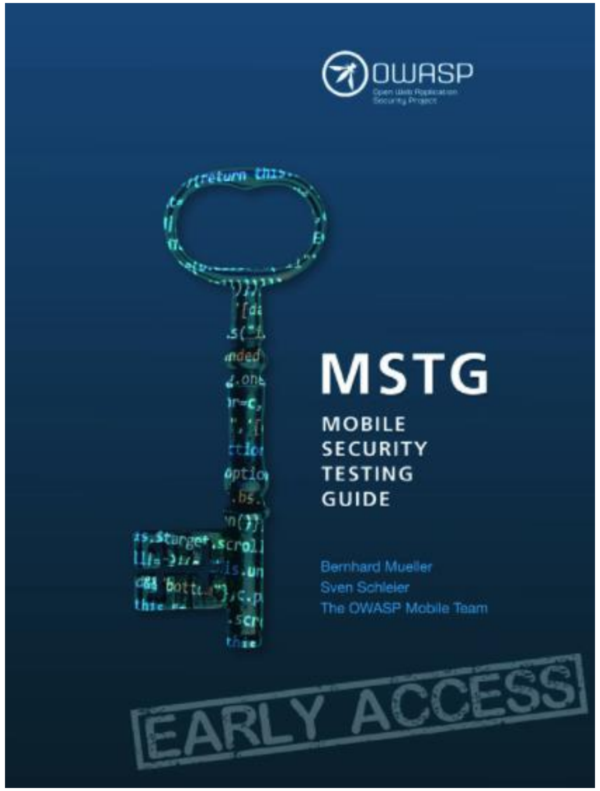
What is the Mobile Application Security Testing Guide?

- Manual for testing security maturity of mobile Apps
- Maps directly to the MASVS requirements
- Focusing on iOS and Android native applications
- Goal is to ensure completeness of mobile app security testing through a consistent testing methodology
- For security checks of the endpoint the OWASP Web Application Testing Guide should be used

OWASP Mobile Security Testing Guide Standard (MSTG)

Structure

- General Testing Guide
- Android Testing Guide
- iOS Testing Guide



- Platform Overview
- Security Testing Basics
- Test Cases
- Reverse Engineering

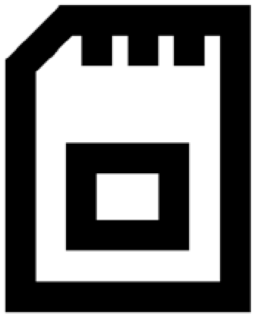
Gitbook: https://www.gitbook.com/book/b-mueller/the-owasp-mobile-security-testing-guide/details
PDF Download: https://leanpub.com/mobile-security-testing-guide

OWASP Mobile Security Testing Guide Standard (MSTG)

Example of some Key Topics

Testing Local Storage for sensitive information

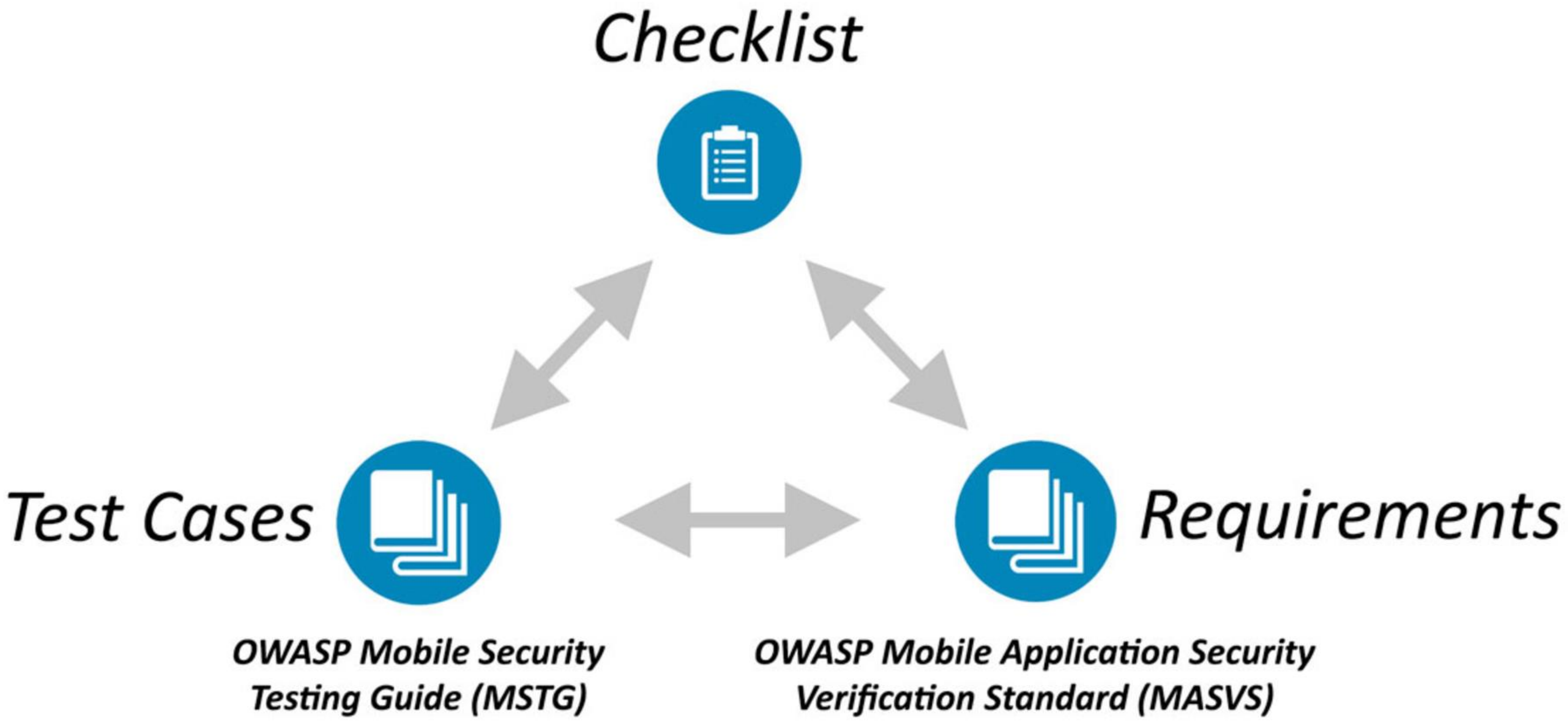
- Clarify how data can be stored on iOS and Android
- Check the usage of cryptographic functions



Testing Platform Interaction

- App permissions
- Verify usage of Interprocess communication (IPC)
- Check the implementation of WebViews
- Biometric Authentication (Touch ID)

OWASP Mobile Security Testing Guide Standard (MSTG)



Security Testers have no good way of dealing with software protection schemes

Developers and Pentesters are confused

Report lists "***lack of obfuscation***" as a critical security issue.
What are the developers supposed to do?

- MinifyEnabled = true?
- Maybe encrypt strings?
- Apply complex control flow obfuscation?
- Maybe use some whitebox crypto?

We want to develop a proper assessment methodology.

Skills needed for assessing ant-reversing schemes

1.Determine whether using software protections are used appropriately

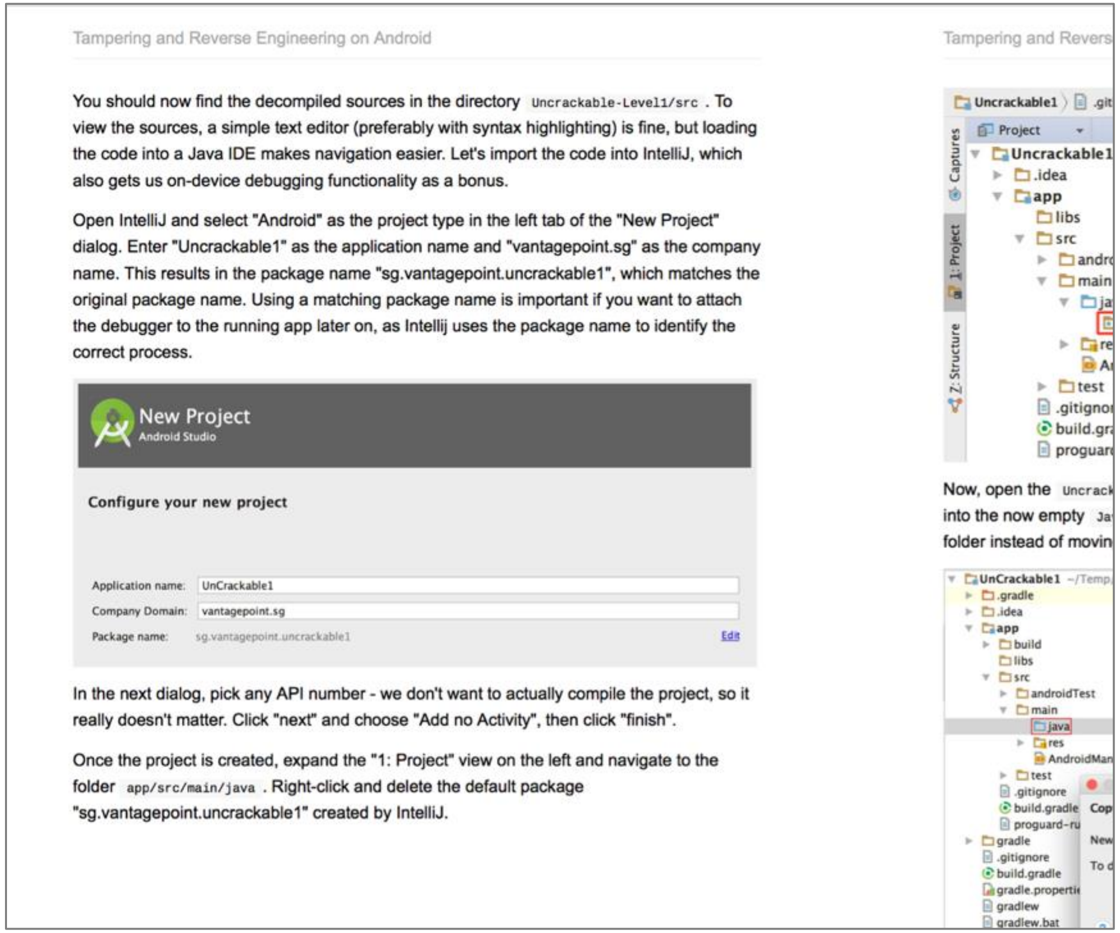
- Every software protection scheme can be defeated.
- Never to be used as replacement for security controls
- Viable uses: IP protection, Prevent modding / cheating in online games, hardening against code injection and instrumentation

2. Hands-on Reversing & Cracking

- Traditional the domain of malware reversers

MSTG – Reverse Engineering

- Building a reverse engineering requirements for free
- Static and dynamic analysis

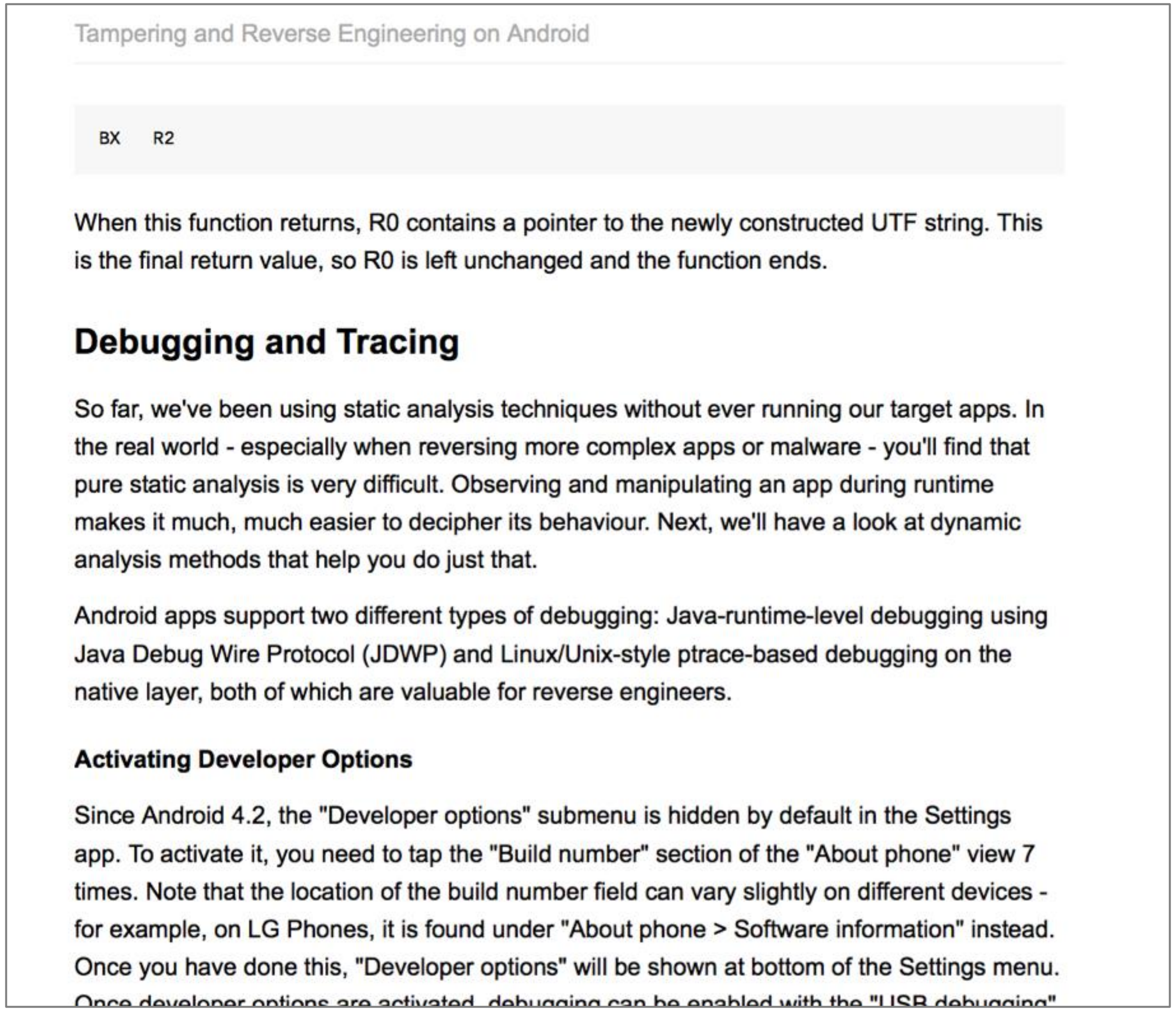


You should now find the decompiled sources in the directory `Uncrackable-Level1/src` . To view the sources, a simple text editor (preferably with syntax highlighting) is fine, but loading the code into a Java IDE makes navigation easier. Let's import the code into IntelliJ, which also gets us on-device debugging functionality as a bonus.

Open IntelliJ and select "Android" as the project type in the left tab of the "New Project" dialog. Enter "Uncrackable1" as the application name and "vantagepoint.sg" as the company name. This results in the package name "sg.vantagepoint.uncrackable1", which matches the original package name. Using a matching package name is important if you want to attach the debugger to the running app later on, as Intellij uses the package name to identify the correct process.

In the next dialog, pick any API number - we don't want to actually compile the project, so it really doesn't matter. Click "next" and choose "Add no Activity", then click "finish".

Once the project is created, expand the "1: Project" view on the left and navigate to the folder `app/src/main/java` . Right-click and delete the default package "sg.vantagepoint.uncrackable1" created by IntelliJ.




```
BX    R2
```

When this function returns, R0 contains a pointer to the newly constructed UTF string. This is the final return value, so R0 is left unchanged and the function ends.

Debugging and Tracing

So far, we've been using static analysis techniques without ever running our target apps. In the real world - especially when reversing more complex apps or malware - you'll find that pure static analysis is very difficult. Observing and manipulating an app during runtime makes it much, much easier to decipher its behaviour. Next, we'll have a look at dynamic analysis methods that help you do just that.

Android apps support two different types of debugging: Java-runtime-level debugging using Java Debug Wire Protocol (JDWP) and Linux/Unix-style ptrace-based debugging on the native layer, both of which are valuable for reverse engineers.

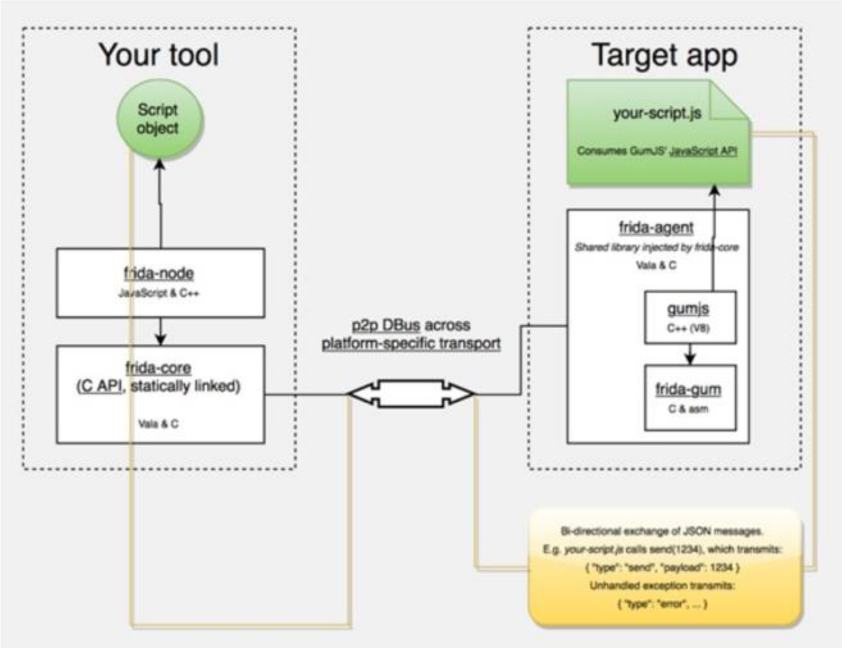
Activating Developer Options

Since Android 4.2, the "Developer options" submenu is hidden by default in the Settings app. To activate it, you need to tap the "Build number" section of the "About phone" view 7 times. Note that the location of the build number field can vary slightly on different devices - for example, on LG Phones, it is found under "About phone > Software information" instead. Once you have done this, "Developer options" will be shown at bottom of the Settings menu. Once developer options are activated, debugging can be enabled with the "USB debugging"

MSTG – Reverse Engineering

- Tampering, patching and runtime instrumentation

Frida injects a complete JavaScript runtime into the process, along with a powerful API that provides a wealth of useful functionality, including calling and hooking of native functions and injecting structured data into memory. It also supports interaction with the Android Java runtime, such as interacting with objects inside the VM.



FRIDA Architecture, source: http://www.frida.re/docs/hacking/

Here are some more APIs FRIDA offers on Android:

Tampering and Reverse Engineering on Android

Your Android device doesn't need to be rooted to get Frida running, but it's the easiest setup and we assume a rooted device here unless noted otherwise. Download the frida-server binary from the Frida releases page. Make sure that the server version (at least the major version number) matches the version of your local Frida installation. Usually, Pypi will install the latest version of Frida, but if you are not sure, you can check with the Frida command line tool:

```
$ frida --version
9.1.10
$ wget https://github.com/frida/frida/releases/download/9.1.10/frida-server-9.1.10-and
roid-arm.xz
```

Copy frida-server to the device and run it:

```
$ adb push frida-server /data/local/tmp/
$ adb shell "chmod 755 /data/local/tmp/frida-server"
$ adb shell "su -c /data/local/tmp/frida-server &"
```

With frida-server running, you should now be able to get a list of running processes with the following command:

```
$ frida-ps -U
  PID  Name
-----  ---------------------------------------------------------------
  276  adbd
  956  android.process.media
  198  bridgemgrd
 1191  com.android.nfc
```

MSTG – Reverse Engineering

- Advanced topics: Program analysis, writing kernel modules, customizing Android…

Installing Angr

Angr is written in Python 2 and available from PyPI. It is easy to install on *nix operating systems and Mac OS using pip:

```
$ pip install angr
```

It is recommended to create a dedicated virtual environment with Virtualenv as some of its dependencies contain forked versions Z3 and PyVEX that overwrite the original versions (you may skip this step if you don't use these libraries for anything else - on the other hand, using Virtualenv is generally a good idea).

Quite comprehensive documentation for angr is available on Gitbooks, including an installation guide, tutorials and usage examples [5]. A complete API reference is also available [6].

Using the Disassembler Backends

Symbolic Execution

Symbolic execution allows you to determine the conditions necessary to reach a specific target. It does this by translating the program's semantics into a logical formula, whereby some variables are represented as symbols with specific constraints. By resolving the constraints, you can find out the conditions necessary so that some branch of the program gets executed.

Amongst other things, this is useful in cases where we need to find the right inputs for reaching a certain block of code. In the following example, we'll use Angr to solve a simple Android crackme in an automated fashion. The crackme takes the form of a native ELF binary that can be downloaded here:

https://github.com/angr/angr-doc/tree/master/examples/android_arm_license_validation

Running the executable on any Android device should give you the following output:

So far, so good, but we really know nothing about how a valid license key might look like. Where do we start? Let's fire up IDA Pro to get a first good look at what is happening.

The main function is located at address 0x1874 in the disassembly (note that this is a PIE-enabled binary, and IDA Pro chooses 0x0 as the image base address). Function names have been stripped, but luckily we can see some references to debugging strings: It appears that the input string is base32-decoded (call to sub_1340). At the beginning of main, there's also a length check at loc_1898 that verifies that the length of the input string is exactly 16. So we're looking for a 16 character base32-encoded string! The decoded input is then passed to the function sub_1760, which verifies the validity of the license key.

The 16-character base32 input string decodes to 10 bytes, so we know that the validation function expects a 10 byte binary string. Next, we have a look at the core validation function at 0x1760:

Testing Anti-Reversing Defenses

- Root Detection
- Anti-Debugging
- Detecting Reverse Engineering Tools
- Emulator Detection / Anti-Emulation
- File and Memory Integrity Checks
- Device Binding
- Obfuscation

Some Original Research

- Android ART: Anti-JDWP debugging by manipulating JDWP-related vtables (JdwpSocketState / JdwpAdbState)
- Frida Detection
  - Frida server detection by local portscan
  - Memory scan to detect Frida agent/gadget artefacts
- Some variations of ptrace-based native anti-debugging

See chapter "Testing Anti-Reversing Defenses"

Also, see blog posts from Bernhard Mueller: http://goo.gl/hsU6bS

Practical Challenges!

Check out the « UnCrackable Mobile Apps »

https://github.com/OWASP/owasp-mstg/tree/master/Crackmes

Ongoing Work

- Obfuscation Metrics

https://github.com/b-mueller/obfuscation-metrics

- Assessment Methodology

https://github.com/OWASP/owasp-mstg/blob/master/Document/0x07d-Assessing-Anti-Reverse-Engineering-Schemes.md

Help is always needed!

MSTG - Contribution

65 Contributors according to GitHub

https://github.com/OWASP/owasp-mstg/graphs/contributors

Big Thanks to everybody that was already supporting the project!

MSTG - Contribution

We are still looking for people to support the project. So how to get started contributing

RTFM: https://github.com/OWASP/owasp-mstg/blob/master/README.md

Slack: https://owasp.slack.com/messages/project-mobile_omtg/details/

Issues: https://github.com/OWASP/owasp-mstg/issues

Resources

MASVS on GitHub
http://github.com/OWASP/owasp-masvs

MASVS releases
https://github.com/OWASP/owasp-masvs/releases

MSTG on Github
https://github.com/OWASP/owasp-mstg/

MSTG as GitBook
https://b-mueller.gitbooks.io/the-owasp-mobile-security-testing-guide/content/

MSTG for download (early access version)
https://leanpub.com/mobile-security-testing-guide

VANTAGEPOINT

Thank you. Any questions?

sven@vantagepoint.sg / sven.schleier@owasp.org

@bsd_daemon