OWASP
AppSec EU
Belfast

8th to 12th of May 2017

Waterfront Conference Center

OWASP
AppSec EU
Belfast

Fixing Mobile AppSec

Sven a.k.a. sushi2k
Bernhard a.k.a. bernhard

VANTAGEPOINT

Our ”Products”

Mobile AppSec Checklist

Excel ☹

Mobile AppSec Verification Standard

PDF Download
~90% done

OWASP
MSTG
MOBILE SECURITY TESTING GUIDE

Mobile Security Testing Guide

Target 700+ pages
~50% done
Free Ebook & Real,
Printed Book!

OWASP Mobile Application Security Verification Standard (MASVS)

- Started as a fork of the ASVS
- 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.	✓	✓

Questions, questions, questions…

Sample Question: Do we recommend using E2E encryption?

Pros

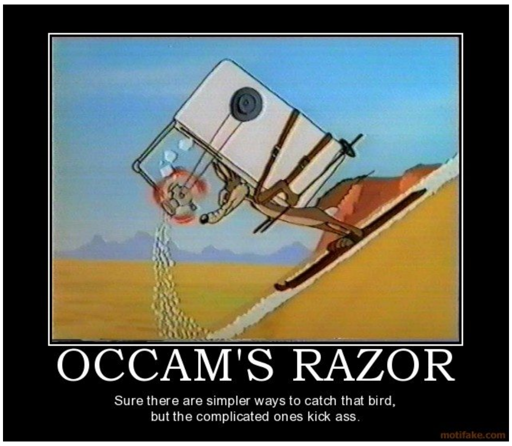
- Additional security layer
- Protects data in case TLS tunnel is compromised
- Protects data from exposure to intermediate systems

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

Check out the GitHub issues…

Our Philosophy


OCCAM'S RAZOR
Sure there are simpler ways to catch that bird,
but the complicated ones kick ass.
motifake.com

43 Security Requirements

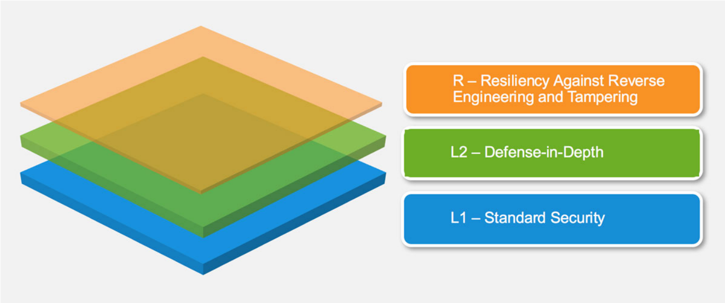
19 Defense-in-Depth Measures

13 Anti-Reversing Controls

OWASP AppSec EU Belfast

Keeping Things Flexible: Requirement "Levels"

- **MASVS-L1:** Security best practices applicable to **all** mobile apps.
- **MASVS-L2:** Defense-in-depth controls for sensitive apps (e.g. financial transactions)
- **MASVS-R:** Optional tamper-proofing to counter specific client-side threats



OWASP AppSec EU Belfast

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!

How To Use the MASVS

- During early stages of development:
  - As a basis for design decisions
  - To determine security requirements early on
  - Saves a ton of cost later

Example:

1.3	Security controls are never enforced only on the client side, but on the respective remote endpoints.	✓	✓

How To Use the MASVS

- In mobile app security testing (together with checklist and testing guide).

MASVS on GitHub

http://github.com/OWASP/owasp-masvs

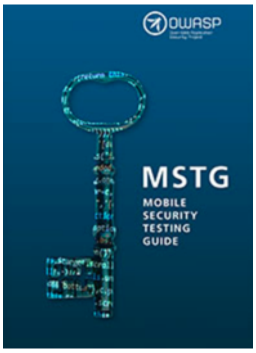
What is the Mobile Security Testing Guide (MSTG)?

What is the Mobile Security Testing Guide (MSTG)?

- 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

How does the document structure look like?

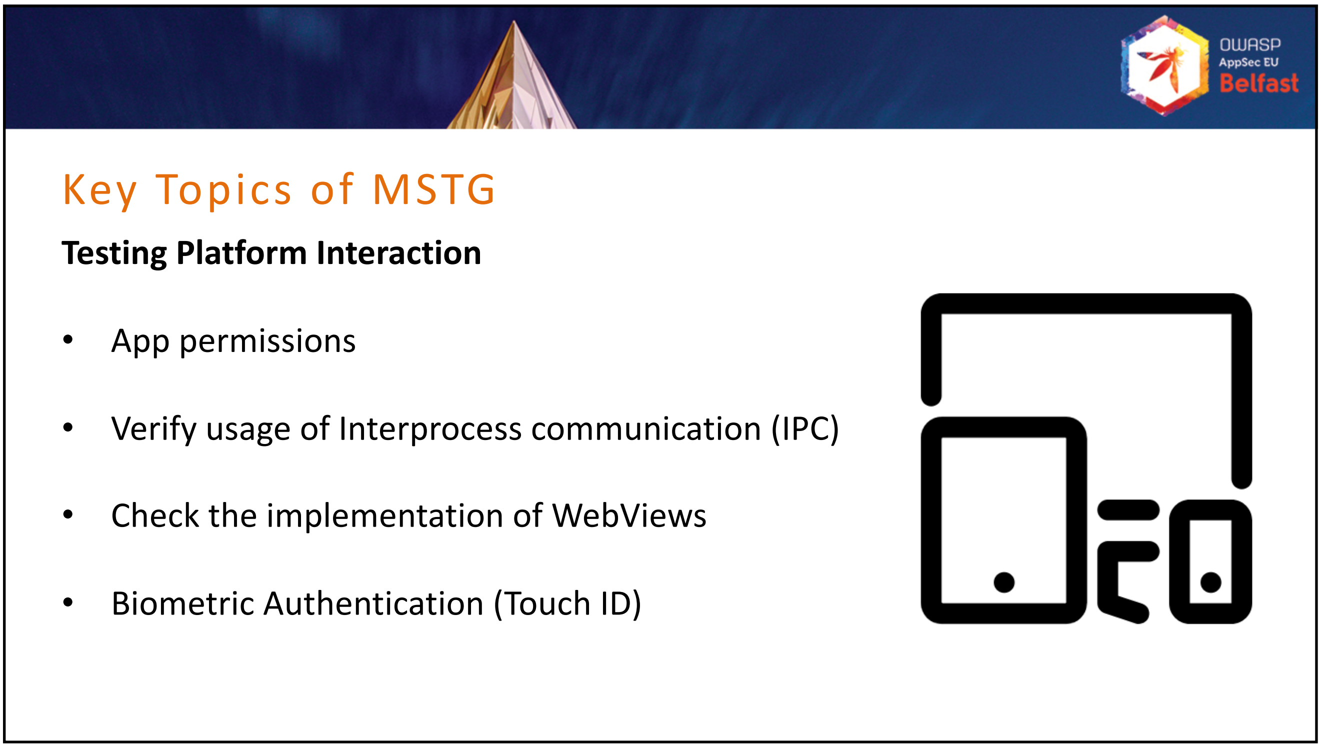
- Testing Processes and Techniques
- Platform Overview
- Security Testing Basics
- Test Cases
- Reverse Engineering and Cracking



Key Topics of MSTG

Testing Local Storage for sensitive information

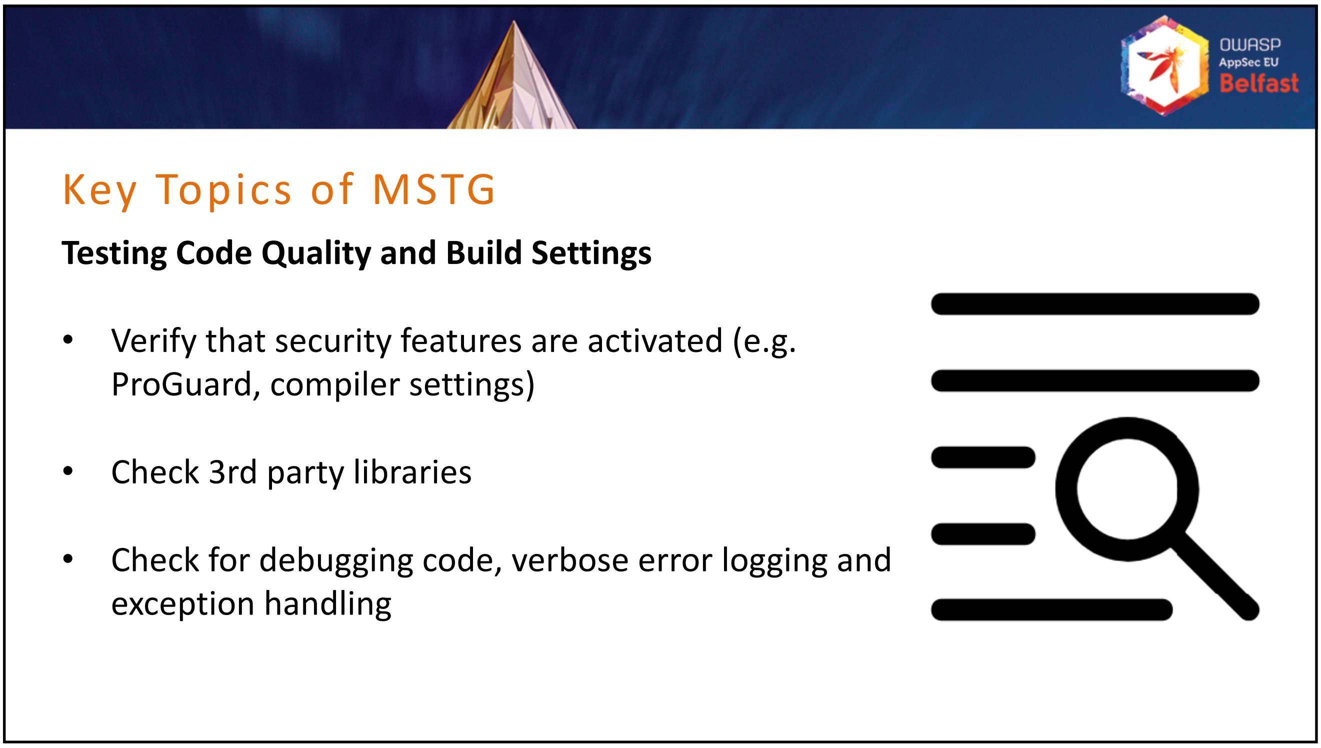
- Clarify how data can be stored on iOS and Android
- Check the usage of cryptographic functions
- Check backups for sensitive data



Key Topics of MSTG

Testing Platform Interaction

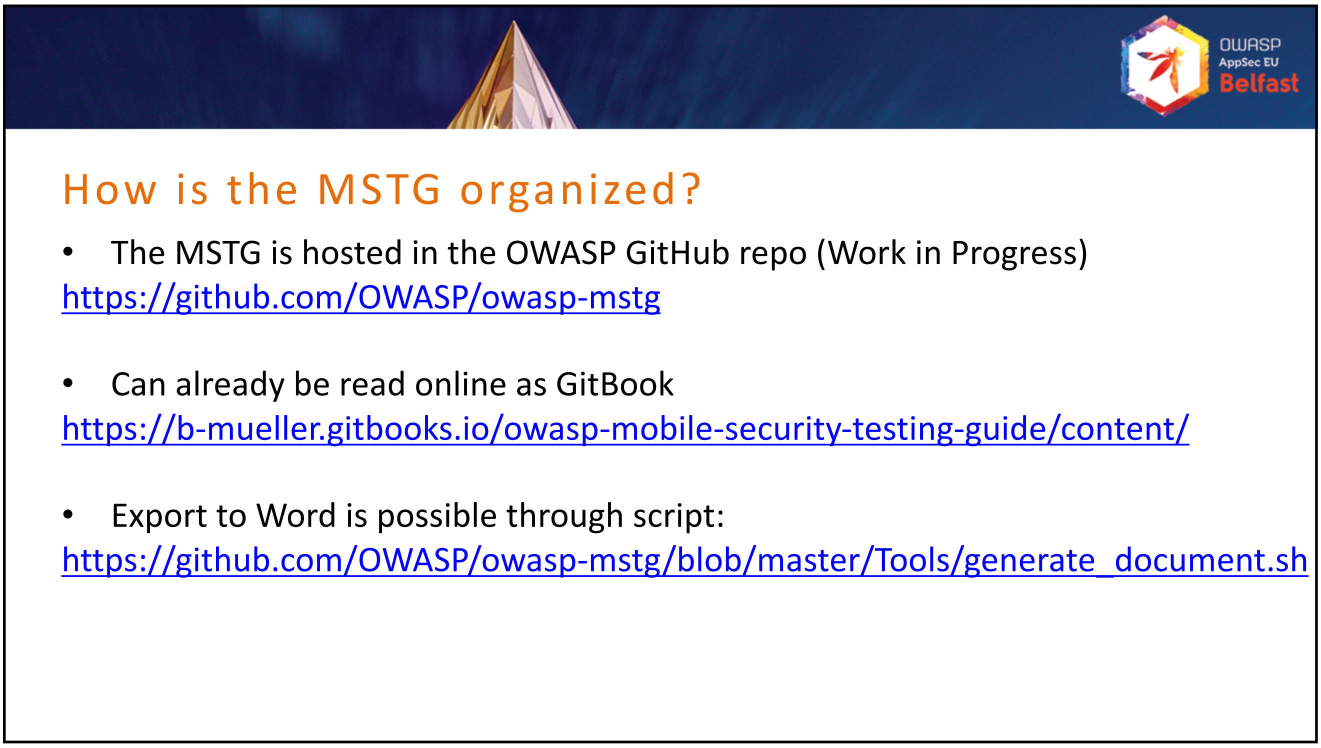
- App permissions
- Verify usage of Interprocess communication (IPC)
- Check the implementation of WebViews
- Biometric Authentication (Touch ID)



Key Topics of MSTG

Testing Code Quality and Build Settings

- Verify that security features are activated (e.g. ProGuard, compiler settings)
- Check 3rd party libraries
- Check for debugging code, verbose error logging and exception handling



How is the MSTG organized?

- The MSTG is hosted in the OWASP GitHub repo (Work in Progress) https://github.com/OWASP/owasp-mstg
- Can already be read online as GitBook https://b-mueller.gitbooks.io/owasp-mobile-security-testing-guide/content/
- Export to Word is possible through script: https://github.com/OWASP/owasp-mstg/blob/master/Tools/generate_document.sh

Reverse Engineering in the MSTG

Security Testers have no good way of dealing with mobile software protections

Pentesters are confused

Report with critical security issue: « Lack of Obfuscation »
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 Anti-Reversing Schemes

1. Determine whether using software protections are used appropriately

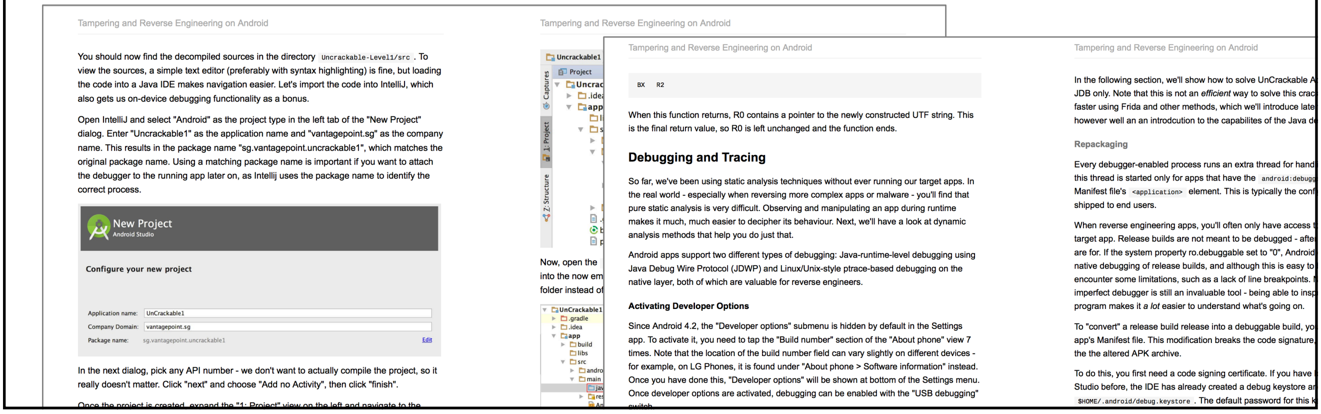
- Every software protection scheme can be defeated
- Never to be used as a replacement for security controls
- Viable uses: IP protection, DRM, preventing modding / cheating, hardening against code injection / instrumentation

2. Hands-on Reversing & Cracking Skills

- Traditional the domain of malware reversers

Reverse Engineering Content in the MSTG

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



Tampering and Reverse Engineering on Android

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.

Application name: UnCrackable1
Company Domain: vantagepoint.sg
Package name: sg.vantagepoint.uncrackable1

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

Tampering and Reverse Engineering on Android

Now, open the
into the now em
folder instead of

Tampering and Reverse Engineering on Android

```
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" switch.

Tampering and Reverse Engineering on Android

In the following section, we'll show how to solve UnCrackable A
JDB only. Note that this is not an *efficient* way to solve this crac
faster using Frida and other methods, which we'll introduce late
however well an an introdcution to the capabilites of the Java de

Repackaging

Every debugger-enabled process runs an extra thread for hand
this thread is started only for apps that have the `android:debugg`
Manifest file's `<application>` element. This is typically the conf
shipped to end users.

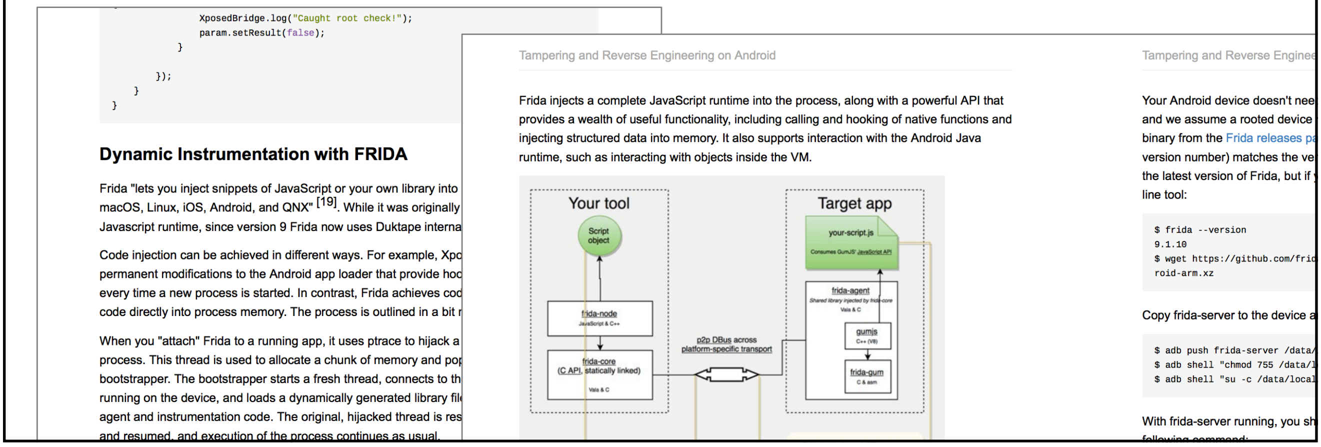
When reverse engineering apps, you'll often only have access t
target app. Release builds are not meant to be debugged - afte
are for. If the system property ro.debuggable set to "0", Android
native debugging of release builds, and although this is easy to
encounter some limitations, such as a lack of line breakpoints. N
imperfect debugger is still an invaluable tool - being able to insp
program makes it *a lot* easier to understand what's going on.

To "convert" a release build release into a debuggable build, yo
app's Manifest file. This modification breaks the code signature,
the the altered APK archive.

To do this, you first need a code signing certificate. If you have
Studio before, the IDE has already created a debug keystore ar
`$HOME/.android/debug.keystore` . The default password for this k

Reverse Engineering Content in the MSTG

- Tampering, patching and runtime instrumentation



```
                XposedBridge.log("Caught root check!");
                param.setResult(false);
            }

        });
    }
}
```

Dynamic Instrumentation with FRIDA

Frida "lets you inject snippets of JavaScript or your own library into
macOS, Linux, iOS, Android, and QNX" [19]. While it was originally
Javascript runtime, since version 9 Frida now uses Duktape interna

Code injection can be achieved in different ways. For example, Xpo
permanent modifications to the Android app loader that provide hoo
every time a new process is started. In contrast, Frida achieves cod
code directly into process memory. The process is outlined in a bit r

When you "attach" Frida to a running app, it uses ptrace to hijack a
process. This thread is used to allocate a chunk of memory and pop
bootstrapper. The bootstrapper starts a fresh thread, connects to th
running on the device, and loads a dynamically generated library fil
agent and instrumentation code. The original, hijacked thread is res
and resumed, and execution of the process continues as usual.

Tampering and Reverse Engineering on Android

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.

Tampering and Reverse Enginee

Your Android device doesn't nee
and we assume a rooted device
binary from the Frida releases pa
version number) matches the ve
the latest version of Frida, but if y
line tool:

```
$ frida --version
9.1.10
$ wget https://github.com/frid
roid-arm.xz
```

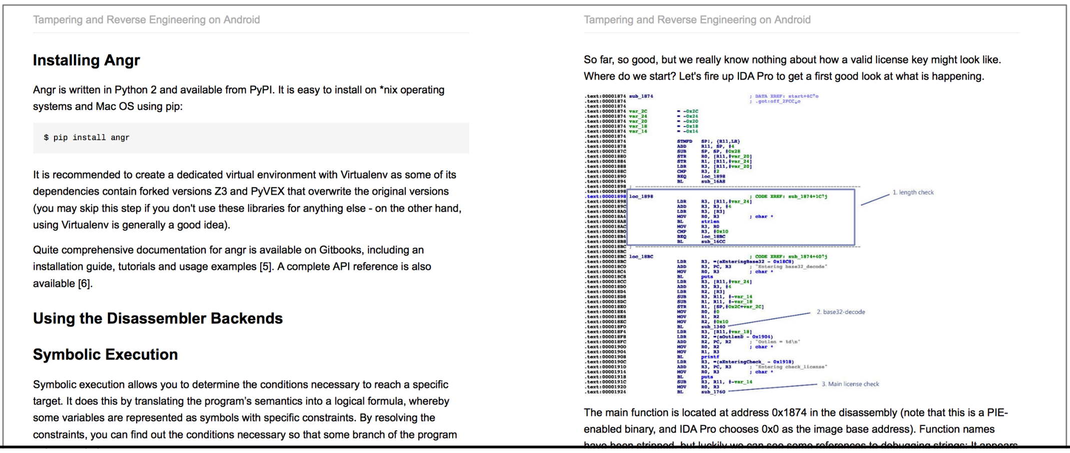
Copy frida-server to the device a

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

With frida-server running, you sh
following command:

Reverse Engineering Content in the MSTG

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



Tampering and Reverse Engineering on 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

Tampering and Reverse Engineering on Android

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.

```
.text:00001874 sub_1874                               ; DATA XREF: start+4C↑o
.text:00001874                                        ; .got:off_2FCC↓o
.text:00001874
.text:00001874 var_2C          = -0x2C
.text:00001874 var_24          = -0x24
.text:00001874 var_20          = -0x20
.text:00001874 var_18          = -0x18
.text:00001874 var_14          = -0x14
.text:00001874
.text:00001874                 STMFD   SP!, {R11,LR}
.text:00001878                 ADD     R11, SP, #4
.text:0000187C                 SUB     SP, SP, #0x28
.text:00001880                 STR     R0, [R11,#var_20]
.text:00001884                 STR     R1, [R11,#var_24]
.text:00001888                 LDR     R3, [R11,#var_20]
.text:0000188C                 CMP     R3, #2
.text:00001890                 BEQ     loc_1898
.text:00001894                 BL      sub_16A8
.text:00001898 ; ---------------------------------------------------------------
.text:00001898
.text:00001898 loc_1898                               ; CODE XREF: sub_1874+1C↑j      1. length check
.text:00001898                 LDR     R3, [R11,#var_24]
.text:0000189C                 ADD     R3, R3, #4
.text:000018A0                 LDR     R3, [R3]
.text:000018A4                 MOV     R0, R3          ; char *
.text:000018A8                 BL      strlen
.text:000018AC                 MOV     R3, R0
.text:000018B0                 CMP     R3, #0x10
.text:000018B4                 BEQ     loc_18BC
.text:000018B8                 BL      sub_16CC
.text:000018BC ; ---------------------------------------------------------------
.text:000018BC
.text:000018BC loc_18BC                               ; CODE XREF: sub_1874+40↑j
.text:000018BC                 LDR     R3, =(aEnteringBase32 - 0x18C8)
.text:000018C0                 ADD     R3, PC, R3      ; "Entering base32_decode"
.text:000018C4                 MOV     R0, R3          ; char *
.text:000018C8                 BL      puts
.text:000018CC                 LDR     R3, [R11,#var_24]
.text:000018D0                 ADD     R3, R3, #4
.text:000018D4                 LDR     R2, [R3]
.text:000018D8                 SUB     R3, R11, #-var_14
.text:000018DC                 SUB     R1, R11, #-var_18
.text:000018E0                 STR     R1, [SP,#0x2C+var_2C]
.text:000018E4                 MOV     R0, #0                          2. base32-decode
.text:000018E8                 MOV     R1, R2
.text:000018EC                 MOV     R2, #0x10
.text:000018F0                 BL      sub_1340
.text:000018F4                 LDR     R3, [R11,#var_18]
.text:000018F8                 LDR     R2, =(aOutlenD - 0x1904)
.text:000018FC                 ADD     R2, PC, R2      ; "Outlen = %d\n"
.text:00001900                 MOV     R0, R2          ; char *
.text:00001904                 MOV     R1, R3
.text:00001908                 BL      printf
.text:0000190C                 LDR     R3, =(aEnteringCheck_ - 0x1918)
.text:00001910                 ADD     R3, PC, R3      ; "Entering check_license"
.text:00001914                 MOV     R0, R3          ; char *
.text:00001918                 BL      puts
.text:0000191C                 SUB     R3, R11, #-var_14
.text:00001920                 MOV     R0, R3                          3. Main license check
.text:00001924                 BL      sub_1760
```

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

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"

Practical Challenges!

« UnCrackable Mobile Apps »

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

Ongoing Work

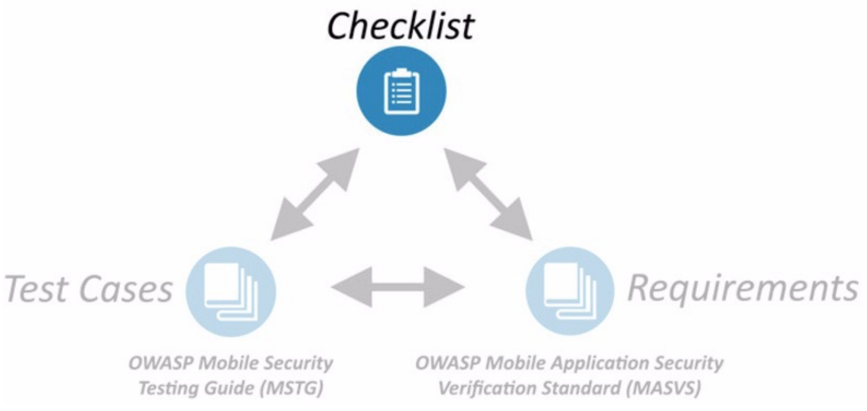
- Obfuscation Metrics

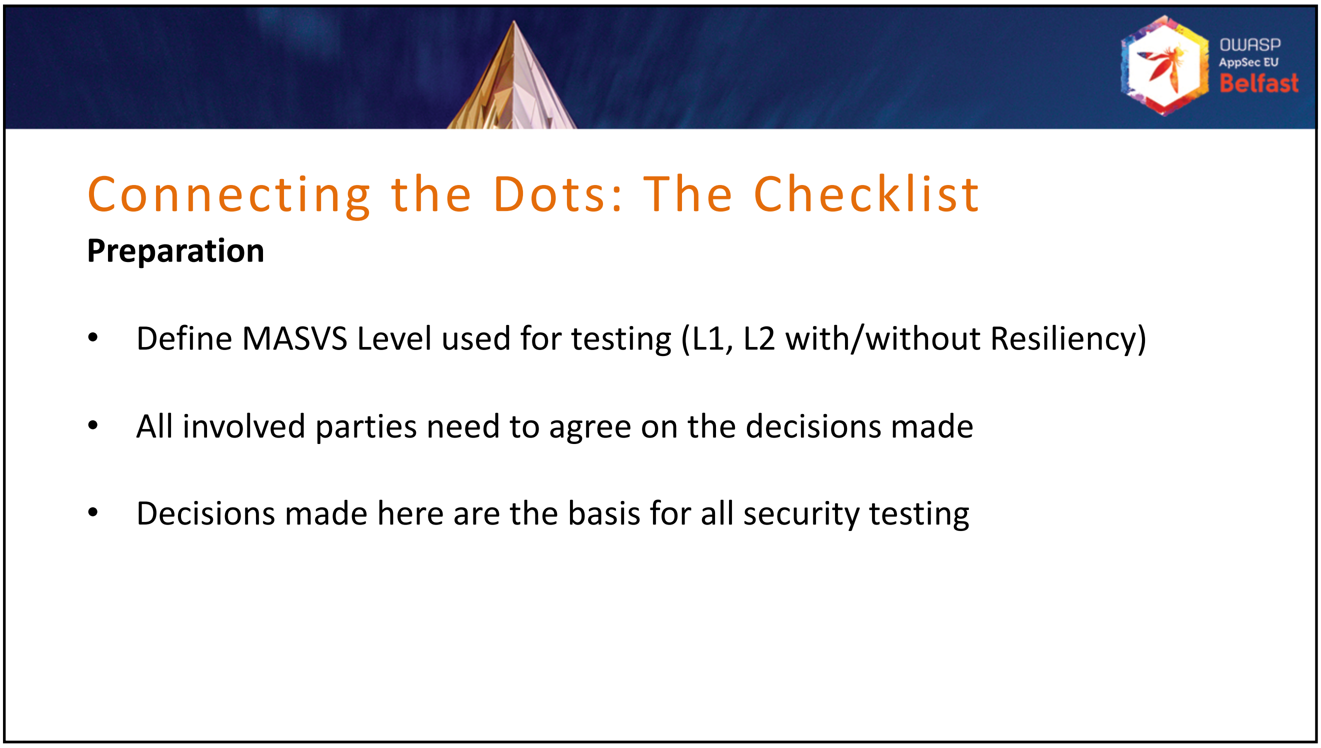
  https://github.com/b-mueller/obfuscation-metrics

- Assessment Methodology

  https://github.com/OWASP/owasp-mstg/blob/master/Document/0x07b-Assessing-Anti-Reverse-Engineering-Schemes.md

Help is always needed!

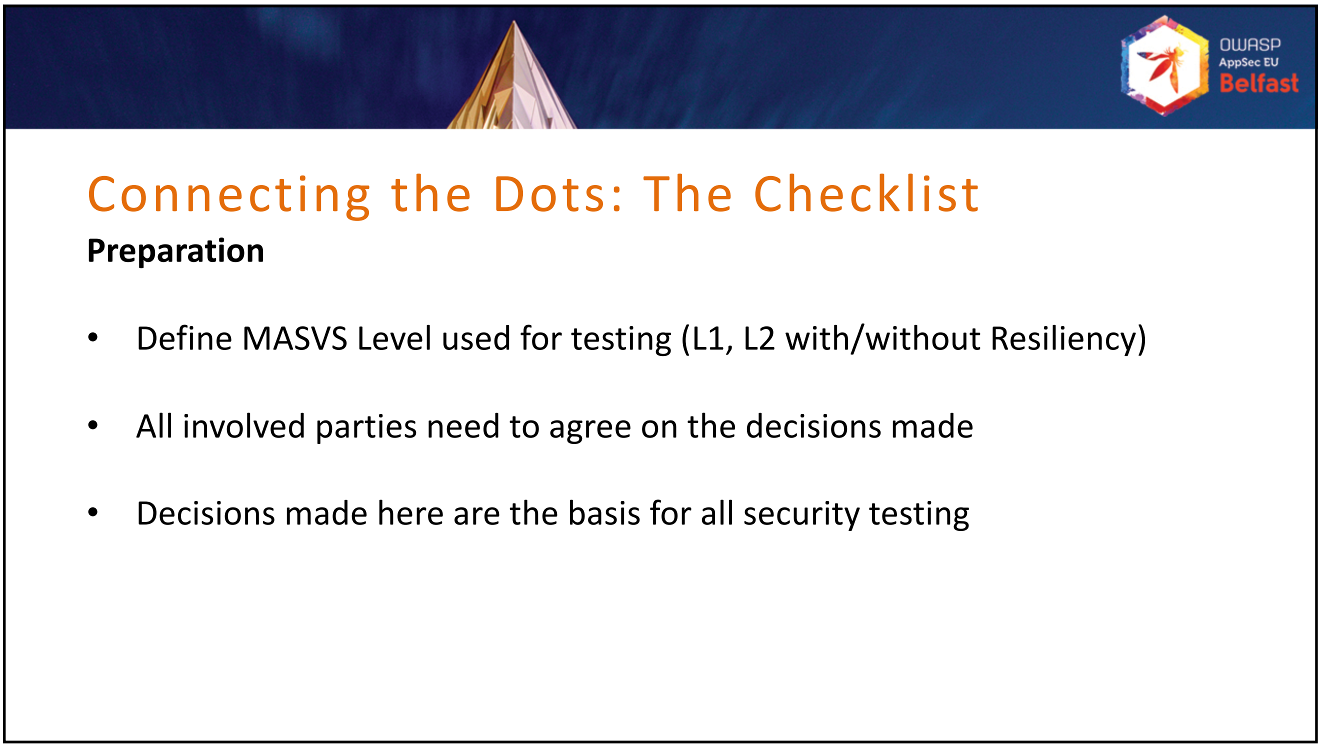

Checklist
Test Cases
Requirements
OWASP Mobile Security Testing Guide (MSTG)
OWASP Mobile Application Security Verification Standard (MASVS)



Connecting the Dots: The Checklist

Preparation

- Define MASVS Level used for testing (L1, L2 with/without Resiliency)
- All involved parties need to agree on the decisions made
- Decisions made here are the basis for all security testing



Connecting the Dots: The Checklist

Preparation

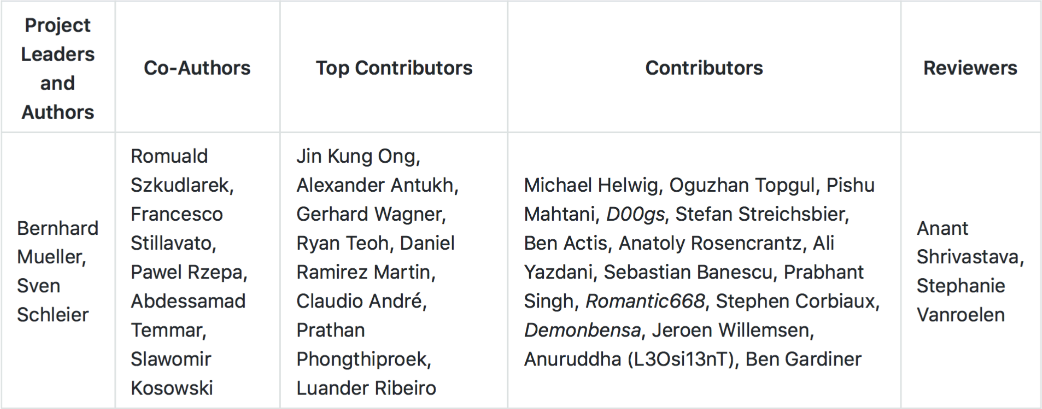
- Define MASVS Level used for testing (L1, L2 with/without Resiliency)
- All involved parties need to agree on the decisions made
- Decisions made here are the basis for all security testing

Connecting the Dots: The Checklist

Mobile App Security Testing

- Walk through the applicable requirements one-by-one
- Links are available to the respective test cases in the MSTG
- Covers iOS and Android Test cases including additional Resiliency Test Cases

32 Contributors according to GitHub!



Project Leaders and Authors	Co-Authors	Top Contributors	Contributors	Reviewers
Bernhard Mueller, Sven Schleier	Romuald Szkudlarek, Francesco Stillavato, Pawel Rzepa, Abdessamad Temmar, Slawomir Kosowski	Jin Kung Ong, Alexander Antukh, Gerhard Wagner, Ryan Teoh, Daniel Ramirez Martin, Claudio André, Prathan Phongthiproek, Luander Ribeiro	Michael Helwig, Oguzhan Topgul, Pishu Mahtani, *D00gs*, Stefan Streichsbier, Ben Actis, Anatoly Rosencrantz, Ali Yazdani, Sebastian Banescu, Prabhant Singh, *Romantic668*, Stephen Corbiaux, *Demonbensa*, Jeroen Willemsen, Anuruddha (L3Osi13nT), Ben Gardiner	Anant Shrivastava, Stephanie Vanroelen

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/

Slack Account Signup: http://owasp.herokuapp.com/

Project Dashboard: https://github.com/OWASP/owasp-mstg/projects/1

Thank you. Any questions?

bernhard.mueller@owasp.org
🐦 @muellerberndt

sven.schleier@owasp.org
🐦 @bsd_daemon

Pictures are partly from the https://thenounproject.com/