OWASP France Meetup

Bordeaux - 22/02/2023

OWASP®

OWASP FOUNDATION

> whois

Tiana (pronounced *'Teen'/'Tine'*)
Find me > @razaina
Former smart card security evaluator
Mobile Security Analyst @eshard
OWASP MAS-related tests developer for our SAST/DAST tool
OWASP Mobile Top 10 volunteer

2015 → 2017 → 2021 → 2022

2017 — esDynamic by eShard — Chip Security Testing

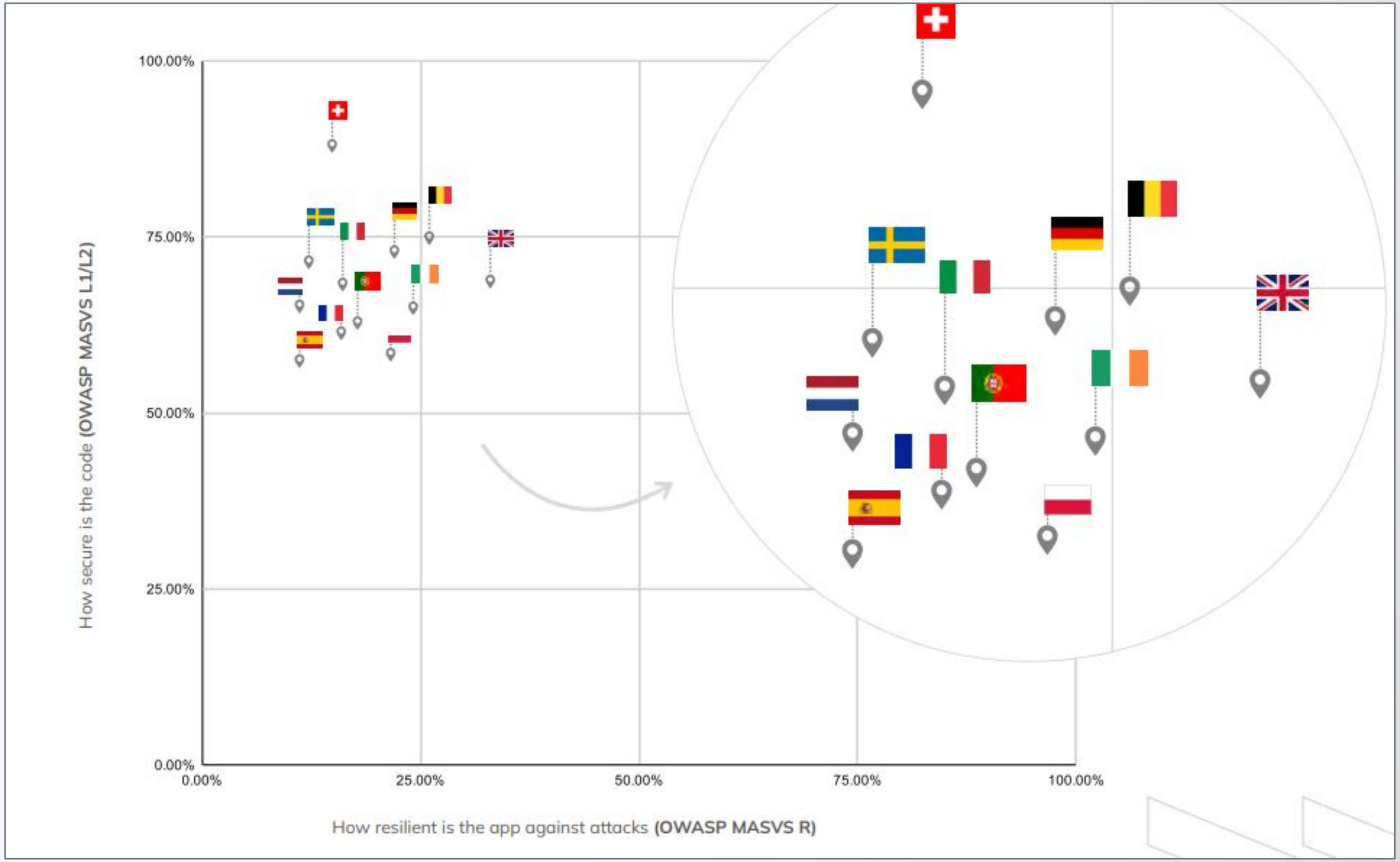
2021 — esChecker by eShard — Mobile App Security Testing

2022 — TETRANE — System Security Testing

How OWASP-compliant are Mobile Banking Apps in Europe?

⇒ 120 apps automatically tested
⇒ 0 are OWASP-compliant

Should we worry?



__src:__ White paper on “*European Mobile Banking Apps Security Benchmark*”, eShard.

Why should we care?

When was the last time you unlocked your phone?

The mobile app is an entry point
- ○ To remote servers
- ○ To the end-user’s device

The risks:
- ○ From the user perspective, e.g: personal data loss/leakage (bank account, password, etc.)
- ○ From the business perspective:
  - ■ Data leakage
  - ■ Intellectual property
  - ■ Business model impact:
    - ● Ads removal
    - ● Premium features enabled for everyone
    - ● Game cheats
  - ■ Overall reputation

them.

3. Approximately 90% of Users Use Mobile Banking Apps to View Their Account Balance

What many might find surprising is that people don't primarily use mobile banking apps to pay bills or transfer funds, as these actions come later. Instead, one of the

4. 97% of Millennials and 89% of Consumers Rely on Mobile Banking Apps

Mobile banking applications continue to evolve with the passage of time and

src: https://www.storyly.io/post/10-statistics-mobile-banking-finance-app

How can we limit those risks?

- ❑ Be **<u>at least</u>** compliant with existing standards, e.g: *OWASP Mobile Application Security Verification Standard* (MASVS)

- ❑ Pentesting the app is costly, but automated compliance processes can lower the overall costs

- ❑ Continuous Integration/Continuous Delivery (CI/CD) is already well known in DevOps

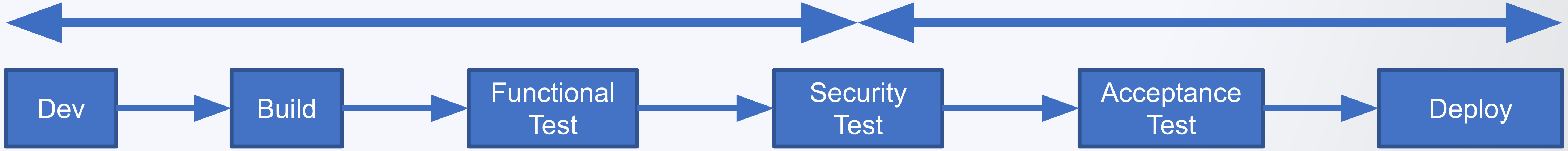
  ⇒ Why not include Mobile Application Security Testing (MAST) as well?

  ⇒ Foster DevSecOps culture to become more agile and respond more quickly to change and innovation

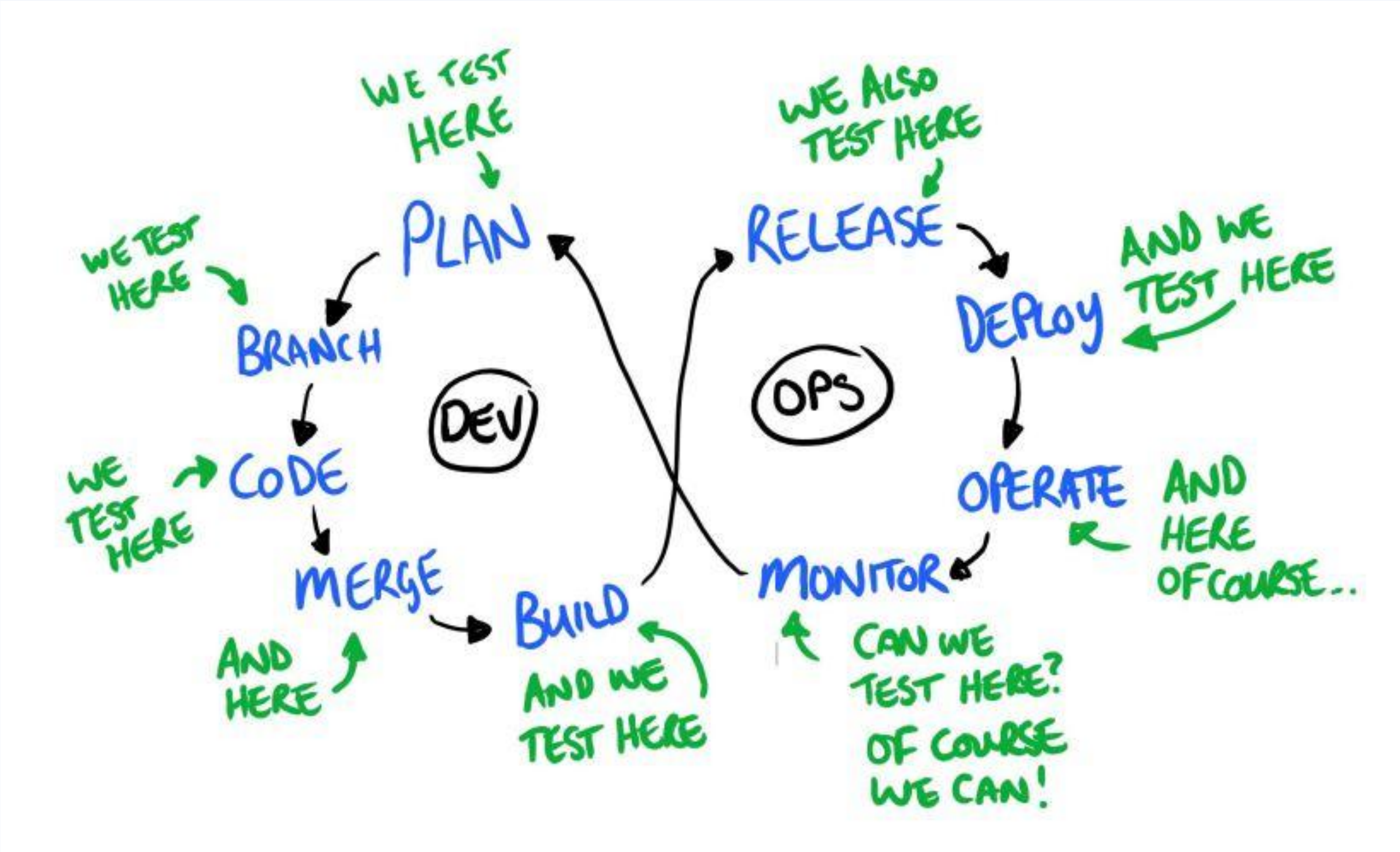
Introduction to CI/CD

Continuous Integration
Merge all code and automatically build the app throughout the day

Continuous Deployment/Delivery
Automatically deploy new app to prod



Introduction to Continuous Testing (CT)



Src: Dan Ashby

Why should you integrate MAST in your CI/CD?

Pentests are still very important and mandatory to assess:

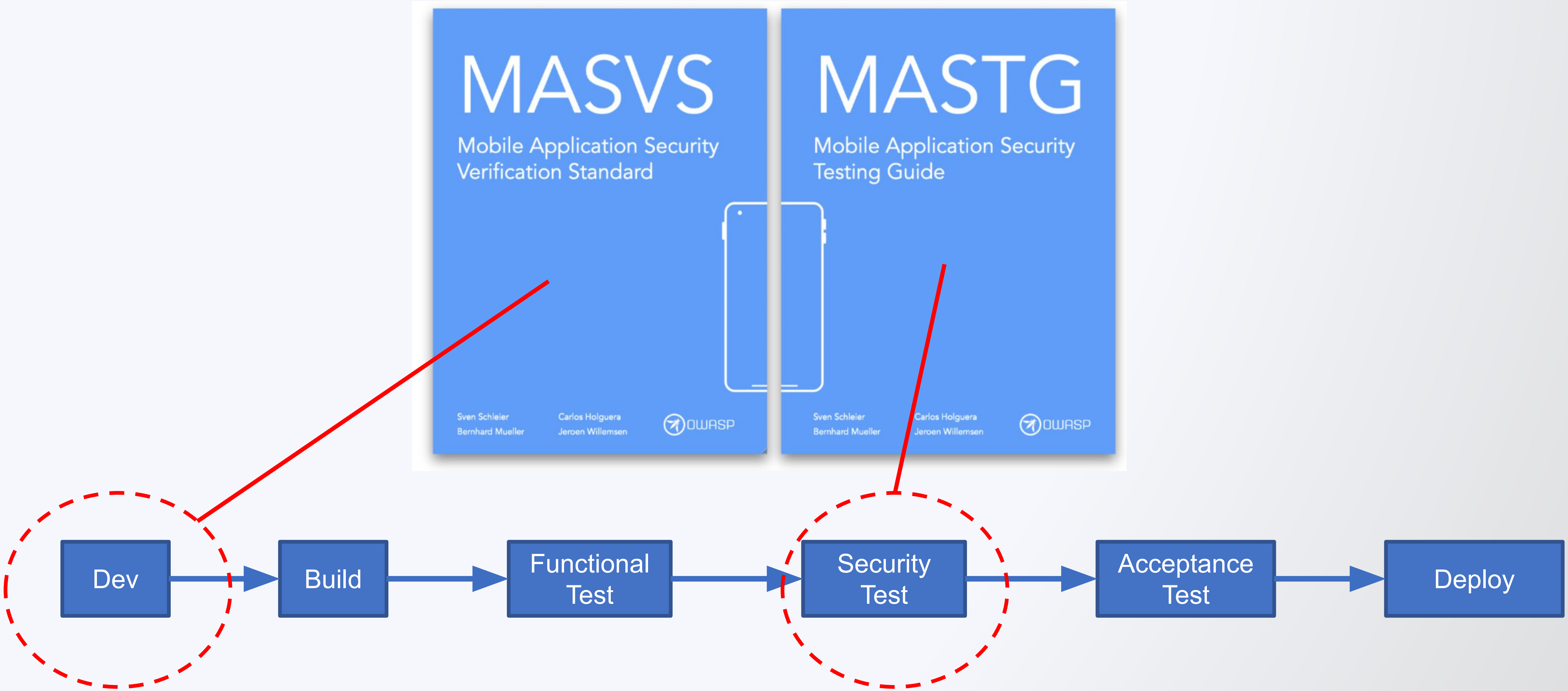
- Does the app embed the right protections?
- Are my protections triggered as expected?
- Are my sensitive assets protected enough?
- How long can my app withstand RE and/or attacks?

Paying for a pentest once or twice a year is definitely not enough!

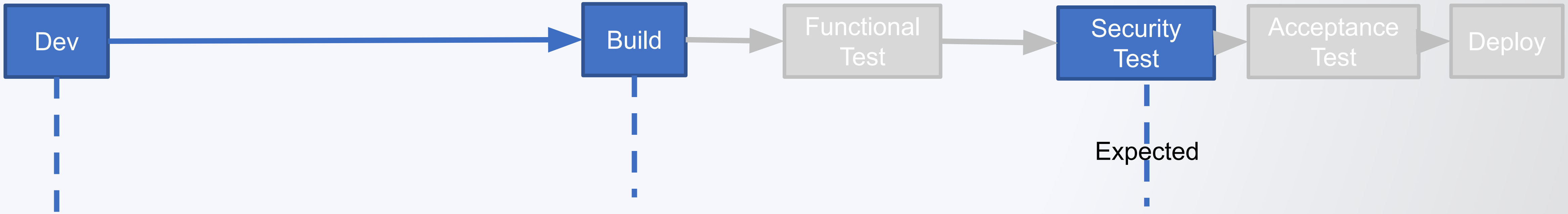
Mobile app releases frequency is increasing

Keep up by integrating automated security testing into the CI/CD toolchain

OWASP MASVS & MASTG



What it takes to protect and test a mobile app



- Implements security features
- Implements protections
- Leverages App Shielding and RASP tools to integrate protections
- Commit/push/merge

Leverages App Shielding and RASP tools to integrate protections

✔ Security features work
✔ Runtime protections are triggered
✔ Runtime and static protections reduce REing capabilities

Use case: a React-Native based Web3 Hot Wallet app

Client A has a banking application that can be used as a crypto wallet:

> "*We did not protect the application, we only rely on the security features provided by the mobile platform*"

Mobile platforms' security features:

- Application sandbox
- Biometric authentication
- Secure storage for storing cryptographic materials (KeyStore/Keychain)

The attack scenarios we proposed:
⇒ Your clients has been infected by a malware
⇒ one of your client got his device stolen

What can we do?

Attack scenario: malware infection

What does my malware need to attack the mobile application?

Mobile platform security features	Required malware features
Sandbox	Embedded root/jailbreak exploits
Biometric authentication	Bypass at runtime (app needs to be running)
KeyStore/Keychain	Crypto materials interception at runtime

Reverse Engineering & Code Tampering

Protecting the app logic is a recommendation in the OWASP Mobile Top 10

OWASP® *M8 – Code Tampering*

Mobile code runs within an environment that is not under the control of the organization producing the code. At the same time, there are plenty of different ways of altering the environment in which that code runs. These changes allow an adversary to tinker with the code and modify it at will.

OWASP® *M9 – Reverse Engineering*

Generally, most applications are susceptible to reverse engineering due to the inherent nature of code. Most languages used to write apps today are rich in metadata that greatly aides a programmer in debugging the app. This same capability also grealy aides an attacker in understanding how the app works.

Secure Local Storage & Cryptography

Insecure storage & Cryptography are even more important to consider

OWASP® *M2 – Insecure Data Storage*

Insecure data storage vulnerabilities occur when development teams assume that users or malware will not have access to a mobile device's filesystem and subsequent sensitive information in data-stores on the device. Filesystems are easily accessible. Organizations should expect a malicious user or malware to inspect sensitive data stores. Usage of poor encryption libraries is to be avoided. Rooting or jailbreaking a mobile device circumvents any encryption protections. When data is not protected properly, specialized tools are all that is needed to view application data.

OWASP® *M5 – Insufficient Cryptography*

In order to exploit this weakness, an adversary must successfully return encrypted code or sensitive data to its original unencrypted form due to weak encryption algorithms or flaws within the encryption process.

Insecure Authentication

OWASP® *M4 – Insecure Authentication*

Poor or missing authentication schemes allow an adversary to anonymously execute functionality within the mobile app or backend server used by the mobile app. Weaker authentication for mobile apps is fairly prevalent due to a mobile device's input form factor.

Reverse Engineering

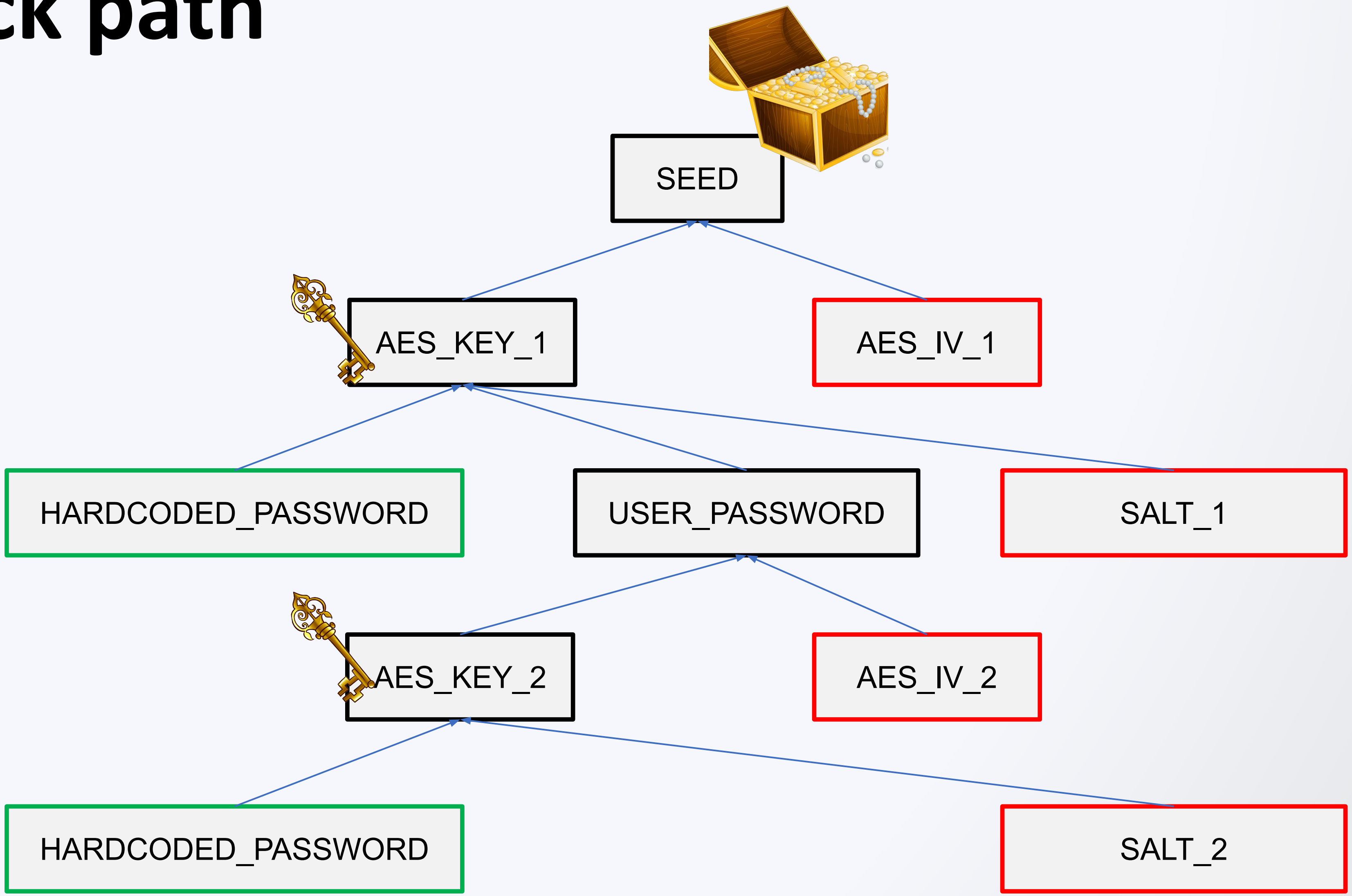
What can I do to learn about the application?

- Download the application from the application store
- Unzip the code and resources inside the application package
- Use open-source tools to reverse engineer the code

What did we learn from the reverse engineering?

- The app is a React-Native based application □ Hermes disabled □ Minified JavaScript code is in plain text
- Local database is not encrypted but some sensitive data are
- SEED is encrypted
  □ AES_KEY_1 = PBKDF2(HARDCODED_PASSWORD|USER_PASSWORD, SALT_1)
- USER_PASSWORD is encrypted
  □ AES_KEY_2 = PBKDF2(HARDCODED_PASSWORD, SALT_2)

Attack path


SEED
AES_KEY_1
AES_IV_1
HARDCODED_PASSWORD
USER_PASSWORD
SALT_1
AES_KEY_2
AES_IV_2
HARDCODED_PASSWORD
SALT_2

The risks

Mobile platform security features	Required malware features	Risks
Sandbox	Embedded root/jailbreak exploits	Exfiltrate database ⇒ decrypt SEED
Biometric authentication	Bypass at runtime (app needs to be running)	?
KeyStore/Keychain	Crypto materials interception at runtime	?

Code Tampering at Runtime

The best open-source tool for runtime code instrumentation ⇒ FRIDA

Problem ⇒ App is implemented using a framework for developing cross platform mobile apps

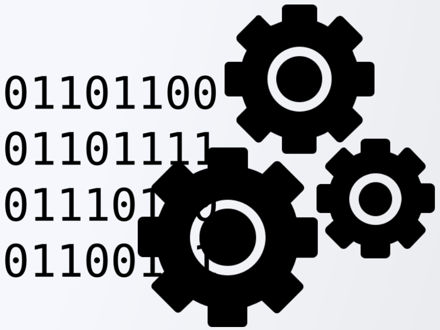
Pros/Cons	For developers	For the attacker
Pros	Only 1 programming language to learn for implementing Android and iOS apps	Vulnerabilities are very likely to be the same on Android and iOS
Cons	Third-party libraries is not as rich as native ones	Another layer to reverse engineer

Challenge ⇒ How to instrument with FRIDA this kind of application?

How does a mobile app development framework work?

Interpreter

Programming Language
(e.g JavaScript, DART, etc.)



Step1: program

Step 2: compile, bundle, package

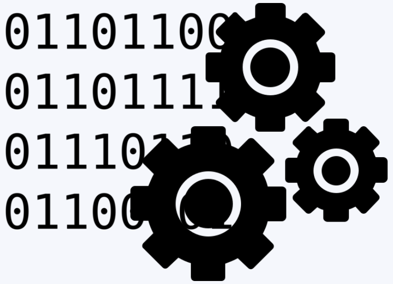
Step 3: load, interpret at runtime

How do we attack this?

- Reverse engineer the interpreter, and find a vulnerability?
- Or is there an easier way? (Yes there is)

Reverse Engineering of React-Native

Interpreter



```
jsi::Value JSCRuntime::evaluateJavaScript(
    const std::shared_ptr<const jsi::Buffer> &buffer,
    const std::string &sourceURL) {
  std::string tmp(
      reinterpret_cast<const char *>(buffer->data()), buffer->size());
  JSStringRef sourceRef = JSStringCreateWithUTF8CString(tmp.c_str());
  JSStringRef sourceURLRef = nullptr;
  if (!sourceURL.empty()) {
    sourceURLRef = JSStringCreateWithUTF8CString(sourceURL.c_str());
  }
  JSValueRef exc = nullptr;
  JSValueRef res =
      JSEvaluateScript(ctx_, sourceRef, nullptr, sourceURLRef, 0, &exc);
  JSStringRelease(sourceRef);
  if (sourceURLRef) {
    JSStringRelease(sourceURLRef);
  }
  checkException(res, exc);
  return createValue(res);
}
```

Convert buffer to string

Pass the code to the Interpreter

FRIDA script to tamper with React-Native code at runtime

```
function hookLibJSCFunctions(JSC) {
  //Set the address of the targeted function
  let JSStringCreateWithUTF8CString_addr = 0xBE18C

  //Hook the function
  Interceptor.attach(JSC.add(JSStringCreateWithUTF8CString_addr), {
    onEnter: function(args) {
      //Get the to-be-executed JavaScript code as a string
      let javascript_code = args[0]
      let c_string = Memory.readCString(ptr(javascript_code))

     //Inject your malicious code
      let new_str = c_string.replace(PATTERN, MALICIOUS_CODE_TO_INJECT)

      //Store the new JavaScript code somewhere in memory
      let new_ptr = Memory.allocUtf8String(new_str)
      this.keep_this_ptr_alive = new_ptr

      //Replace the pointer to the new JavaScript code
      args[0] = new_ptr

      }
    },
    onLeave: function(ret) {}
  })
}
```

Only ~ 11 lines of code is required to control a React-Native based application

Works for Android and iOS.

Code Instrumentation with FRIDA

```
function hookLibJSCFunctions(JSC) {
  //Set the address of the targeted function
  let JSStringCreateWithUTF8CString_addr = 0xBE18C
```

```
2023-01-25T16:42:07.891Z [INFO]  FRIDA ==> PASSWORD: E4m$7%qGKsK67Rv.C8Cu!Z9w8%!-K5PXDWu-AnakH$hHWYkU@5N3C?WDRUcPGRUu  SALT:
rse96s+Rq4Zw7A==
2023-01-25T16:42:07.891Z [INFO]  FRIDA ==> Generate key with {"password": "E4m$7%qGKsK67Rv.C8Cu!Z9w8%!-K5PXDWu-AnakH$hHWYkU
cPGRUu", "salt": "xd+8aIQJrse96s+Rq4Zw7A=="} key = [object Object]
2023-01-25T16:42:07.933Z [INFO]  FRIDA ==> CALLING [redacted] with arg: [object Object]
2023-01-25T16:42:07.934Z [INFO]  FRIDA ==> Get Encrypted mnemonic from AsyncStorage.
2023-01-25T16:42:07.935Z [INFO]  FRIDA ==> DECRYPT MNEMONIC !!!!
2023-01-25T16:42:07.935Z [INFO]  FRIDA ==> PASSWORD: E4m$7%qGKsK67Rv.C8Cu!Z9w8%!-K5PXDWu-AnakH$hHWYkU@5N3C?WDRUcPGRUu:[objec
  SALT: WTwmreBncOc1fJsx6RZjmA==
2023-01-25T16:42:07.935Z [INFO]  FRIDA ==> Generate key with {"password": "E4m$7%qGKsK67Rv.C8Cu!Z9w8%!-K5PXDWu-AnakH$hHWYkU
cPGRUu:[object Object]", "salt": "WTwmreBncOc1fJsx6RZjmA=="} key = [object Object]
2023-01-25T16:42:07.942Z [INFO]  FRIDA ==> decrypted_mnemonic: copy glow light build web dress pulse toast oyster wrestle c
rt
```

```
      }
    },
    onLeave: function(ret) {}
  })
}
```

Biometric Authentication Bypass and Fix

Bad KeyChain configuration leads to easy Biometric Authentication Bypass

```
await Keychain.setGenericPassword([redacted], encryptedPassword, {
  service: [redacted],
  accessible: Keychain.ACCESSIBLE.WHEN_UNLOCKED_THIS_DEVICE_ONLY,

  //eShard (20/01/2023): Partly fix bioAuth bypass
  accessControl: Keychain.ACCESS_CONTROL.BIOMETRY_CURRENT_SET_OR_DEVICE_PASSCODE,
  ///////////////////////////////////////////////////////////////////////////////
});
```

The risks

Mobile platform security features	Required malware features	Risks
Sandbox	Embedded root/jailbreak exploits	Exfiltrate database ⇒ decrypt SEED
Biometric authentication	Bypass at runtime (app needs to be running)	Sensitive operations (e.g: transactions) are not anymore protected behind bio. auth.
KeyStore/Keychain	Crypto materials interception at runtime	We can intercept any sensitive crypto materials at runtime.

Lessons learned

What are the issues:

- No rooted/jailbroken device detection
- No FRIDA (no runtime code tampering) detection
- No code integrity check
- Bad coding practice (logs the user password)
- Bad configuration of the keychain
- Insecure local storage
- Insecure cryptography
- No obfuscation
- Insecure React-Native configuration

Lessons learned

What are the issues:

- No rooted/jailbroken device detection
- No FRIDA (no runtime code tampering) detection
- No code integrity check
- Bad coding practice (logs the user password)
- Bad configuration of the keychain
- Insecure local storage
- Insecure cryptography
- No obfuscation
- Insecure React-Native configuration

No need to go through a full pentest to highlight those issues

Lessons learned

What are the issues:

- No rooted/jailbroken device detection
- No FRIDA (no runtime code tampering) detection
- No code integrity check
- Bad coding practice (logs the user password)

Dynamic checks (DAST)

- Bad configuration of the keychain
- Insecure local storage
- Insecure cryptography
- No obfuscation
- Insecure React-Native configuration

Static checks (SAST)

Conclusion

Keep up to date on

- ❑ OWASP Mobile Top 10,
- ❑ OWASP MASVS (if you are a developer, https://mas.owasp.org/MASVS/)
- ❑ OWASP MASTG (if you want to test your app, https://mas.owasp.org/MASTG/)

Foster DevSecOps ⇒ leverage MAST tools to automate your security testings and improve your CI/CD

/!\ MAST tools are complementary to pentests /!\

Mobile App Sec technical know-how should be integrated in the dev teams