MASDG

Mobile Application Security Design Guide for iOS

OWASP®

Contents

April 1, 2023 Edition

LAC Co., Ltd.

Welcome to Mobile Application Security Design Guide iOS Edition.

The Mobile Application Security Design Guide iOS Edition is a document aimed at establishing a framework for designing, developing, and testing secure mobile applications on iOS, incorporating our own evaluation criteria (rulebook) and sample code into the OWASP Mobile Application Security Verification Standard (MASVS) and Mobile Application Security Testing Guide (MASTG) published by OWASP.

MASDG deals with best practices and samples that are specific to the design requirements for security, supporting the creation of security designs from security requirements considered based on MASVS, as well as evaluating the security design for any issues before conducting testing methods indicated in MASTG.

Our proprietary rulebook targets the MASVS L1 verification standard and aims to provide comprehensive security baselines when developing mobile applications. While new technologies will always bring risks and create privacy and safety issues, we have created this document to address the threats posed by mobile applications.

We have received feedback on MASVS and MASTG from various communities and industries, and we have developed and published MASDG as we believe it is essential to tackle the security risks associated with mobile applications that have become indispensable in our society. We welcome feedback from everyone.

Copyright and License

- The contents of this guide are current as of the time of writing. Please be aware of this if you use the sample code.
- LAC Co., Ltd. and the authors are not responsible for any consequences resulting from the use of this guide. Please use at your own risk.
- iOS is a trademark or registered trademark of Apple Inc. Company names, product names, and service names mentioned in this document are generally registered trademarks or trademarks of their respective companies. The ®, TM, and © symbols are not used throughout this document.
- Some of the content in this document is based on the materials provided by OWASP MASVS and OWASP MASTG, and has been replicated and revised.

Originator
Project Site - https://owasp.org/www-project-mobile-app-security/
Project Repository - https://github.com/OWASP/www-project-mobile-app-security
MAS Official Site - https://mas.owasp.org/
MAS Document Site - https://mas.owasp.org/MASVS/
MAS Document Site - https://mas.owasp.org/MASTG/
Document Site - https://mobile-security.gitbook.io/masvs
Document Repository - https://github.com/OWASP/owasp-masvs
Document Site - https://coky-t.gitbook.io/owasp-masvs-ja/
Document Repository - https://github.com/owasp-ja/owasp-masvs-ja
Document Site - https://mobile-security.gitbook.io/mobile-security-testing-guide
Document Repository - https://github.com/OWASP/owasp-mastg
Document Site - https://coky-t.gitbook.io/owasp-mastg-ja/
Document Repository - https://github.com/coky-t/owasp-mastg-ja

OWASP MASVS Authors

Project Lead	Lead Author	Contributors and Reviewers
Sven Schleier, Carlos Holguera	Bernhard Mueller, Sven Schleier, Jeroen Willemsen and Carlos Holguera	Alexander Antukh, Mesheryakov Aleksey, Elderov Ali, Bachevsky Artem, Jeroen Beckers, Jon-Anthoney de Boer, Damien Clochard, Ben Cheney, Will Chilcutt, Stephen Corbiaux, Manuel Delgado, Ratchenko Denis, Ryan Dewhurst, @empty_jack, Ben Gardiner, Anton Glezman, Josh Grossman, Sjoerd Langkemper, Vinícius Henrique Marangoni, Martin Marsicano, Roberto Martelloni, @PierrickV, Julia Potapenko, Andrew Orobator, Mehrad Rafii, Javier Ruiz, Abhinav Sejpal, Stefaan Seys, Yogesh Sharma, Prabhant Singh, Nikhil Soni, Anant Shrivastava, Francesco Stillavato, Abdessamad Temmar, Pauchard Thomas, Lukasz Wierzbicki

OWASP MASTG Authors
Bernhard Mueller
Sven Schleier
Jeroen Willemsen
Carlos Holguera
Romuald Szkudlarek
Jeroen Beckers
Vikas Gupta

OWASP MASVS ja Author
Koki Takeyama

OWASP MASTG ja Author
Koki Takeyama

1 Architecture, Design and Threat Modeling Requirements

1.1 MSTG-ARCH-1

All app components are identified and known to be needed.

1.1.1 Component

Get to know all the components of your application and remove unnecessary ones.

The main types of components are as follows

- Content
  - Charts
  - Image views
  - Text views
  - Web views
- Layout and Organization
  - Boxes
  - Collections
  - Column views
  - Disclosure Controls
  - Labels
  - Lists and tables
  - Lockups
  - Outline views
  - Split views
  - Tab views
- Menus and Actions
  - Activity views
  - Buttons
  - Context menus

  - Dock menus
  - Edit menus
  - Menus
  - Pop-up buttons
  - Pull-down buttons
  - Toolbars
- Navigation and Search
  - Navigation bars
  - Path controls
  - Search fields
  - Sidebars
  - Tab bars
  - Token fields
- Presentation
  - Action sheets
  - Alerts
  - Page controls
  - Panels
  - Popovers
  - Scroll views
  - Sheets
  - Windows
- Selection and Entry
  - Color wells
  - Combo boxes
  - Digit entry views
  - Image wells
  - Onscreen keyboards
  - Pickers
  - Segmented controls
  - Sliders
  - Steppers
  - Text fields
  - Toggles
- Condition
  - Activity rings
  - Gauges
  - Progress indicators
  - Rating indicators

- System
  - Complications
  - Home Screen quick actions
  - Live Activities
  - The menu bar
  - Notifications
  - Status bars
  - Top Shelf
  - Watch faces
  - Widgets

1.2 MSTG-ARCH-2

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

1.2.1 Falsification of Authentication/Authorization information

1.2.1.1 Appropriate authentication response

Perform the following steps when testing authentication and authorization.

- Identify the additional authentication factors the app uses.
- Locate all endpoints that provide critical functionality.
- Verify that the additional factors are strictly enforced on all server-side endpoints.

Authentication bypass vulnerabilities exist when authentication state is not consistently enforced on the server and when the client can tamper with the state. While the backend service is processing requests from the mobile client, it must consistently enforce authorization checks: verifying that the user is logged in and authorized every time a resource is requested.

Consider the following example from the OWASP Web Testing Guide. In the example, a web resource is accessed through a URL, and the authentication state is passed through a GET parameter:

```
http://www.site.com/page.asp?authenticated=no
```

The client can arbitrarily change the GET parameters sent with the request. Nothing prevents the client from simply changing the value of the authenticated parameter to "yes", effectively bypassing authentication.

Although this is a simplistic example that you probably won't find in the wild, programmers sometimes rely on "hidden" client-side parameters, such as cookies, to maintain authentication state. They assume that these parameters can't be tampered with. Consider, for example, the following classic vulnerability in Nortel Contact Center Manager. The administrative web application of Nortel's appliance relied on the cookie "isAdmin" to determine whether the logged-in user should be granted administrative privileges. Consequently, it was possible to get admin access by simply setting the cookie value as follows:

```
isAdmin=True
```

Security experts used to recommend using session-based authentication and maintaining session data on the server only. This prevents any form of client-side tampering with the session state. However, the whole point of using stateless authentication instead of session-based authentication is to not have session state on the server. Instead, state is stored in client-side tokens and transmitted with every request. In this case, seeing client-side parameters such as isAdmin is perfectly normal.

To prevent tampering cryptographic signatures are added to client-side tokens. Of course, things may go wrong, and popular implementations of stateless authentication have been vulnerable to attacks. For example, the signature verification of some JSON Web Token (JWT) implementations could be deactivated by setting the signature type to "None". We'll discuss this attack in more detail in the "Testing JSON Web Tokens" chapter.

Reference

- owasp-mastg Verifying that Appropriate Authentication is in Place (MSTG-ARCH-2 and MSTG-AUTH-1)

1.2.2 Injection Flaws

An injection flaw describes a class of security vulnerability occurring when user input is inserted into backend queries or commands. By injecting meta-characters, an attacker can execute malicious code that is inadvertently interpreted as part of the command or query. For example, by manipulating a SQL query, an attacker could retrieve arbitrary database records or manipulate the content of the backend database.

Vulnerabilities of this class are most prevalent in server-side web services. Exploitable instances also exist within mobile apps, but occurrences are less common, plus the attack surface is smaller.

For example, while an app might query a local SQLite database, such databases usually do not store sensitive data (assuming the developer followed basic security practices). This makes SQL injection a non-viable attack vector. Nevertheless, exploitable injection vulnerabilities sometimes occur, meaning proper input validation is a necessary best practice for programmers.

Reference

- owasp-mastg Injection Flaws (MSTG-ARCH-2 and MSTG-PLATFORM-2)

Rulebook

- *Enforce appropriate input validation (Required)*

1.2.2.1 SQL Injection

A SQL injection attack involves integrating SQL commands into input data, mimicking the syntax of a predefined SQL command. A successful SQL injection attack allows the attacker to read or write to the database and possibly execute administrative commands, depending on the permissions granted by the server.

Apps on both Android and iOS use SQLite databases as a means to control and organize local data storage. Assume an Android app handles local user authentication by storing the user credentials in a local database (a poor programming practice we'll overlook for the sake of this example). Upon login, the app queries the database to search for a record with the username and password entered by the user:

```
SQLiteDatabase db;

String sql = "SELECT * FROM users WHERE username = '" +  username + "' AND␣
↪password = '" + password +"'";

Cursor c = db.rawQuery( sql, null );

return c.getCount() != 0;
```

Let's further assume an attacker enters the following values into the "username" and "password" fields:

```
username = 1' or '1' = '1
password = 1' or '1' = '1
```

This results in the following query:

```
SELECT * FROM users WHERE username='1' OR '1' = '1' AND Password='1' OR '1' = '1'
```

Because the condition '1' = '1' always evaluates as true, this query return all records in the database, causing the login function to return true even though no valid user account was entered.

Ostorlab exploited the sort parameter of Yahoo's weather mobile application with adb using this SQL injection payload.

Another real-world instance of client-side SQL injection was discovered by Mark Woods within the "Qnotes" and "Qget" Android apps running on QNAP NAS storage appliances. These apps exported content providers vulnerable to SQL injection, allowing an attacker to retrieve the credentials for the NAS device. A detailed description of this issue can be found on the Nettitude Blog.

Reference

- owasp-mastg Injection Flaws (MSTG-ARCH-2 and MSTG-PLATFORM-2) SQL Injection

1.2.2.2 XML Injection

In a XML injection attack, the attacker injects XML meta-characters to structurally alter XML content. This can be used to either compromise the logic of an XML-based application or service, as well as possibly allow an attacker to exploit the operation of the XML parser processing the content.

A popular variant of this attack is XML eXternal Entity (XXE). Here, an attacker injects an external entity definition containing an URI into the input XML. During parsing, the XML parser expands the attacker-defined entity by accessing the resource specified by the URI.

The integrity of the parsing application ultimately determines capabilities afforded to the attacker, where the malicious user could do any (or all) of the following: access local files, trigger HTTP requests to arbitrary hosts and ports, launch a cross-site request forgery (CSRF) attack, and cause a denial-of-service condition. The OWASP web testing guide contains the following example for XXE:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
 <!DOCTYPE foo [
  <!ELEMENT foo ANY >
  <!ENTITY xxe SYSTEM "file:///dev/random" >]><foo>&xxe;</foo>
```

In this example, the local file /dev/random is opened where an endless stream of bytes is returned, potentially causing a denial-of-service.

The current trend in app development focuses mostly on REST/JSON-based services as XML is becoming less common. However, in the rare cases where user-supplied or otherwise untrusted content is used to construct XML queries, it could be interpreted by local XML parsers, such as NSXMLParser on iOS. As such, said input should always be validated and meta-characters should be escaped.

Reference

- owasp-mastg Injection Flaws (MSTG-ARCH-2 and MSTG-PLATFORM-2) XML Injection

Rulebook

- *Input should always be validated and meta-characters should be escaped (Required)*

1.2.2.3 Injection Attack Vectors

The attack surface of mobile apps is quite different from typical web and network applications. Mobile apps don't often expose services on the network, and viable attack vectors on an app's user interface are rare. Injection attacks against an app are most likely to occur through inter-process communication (IPC) interfaces, where a malicious app attacks another app running on the device.

Locating a potential vulnerability begins by either:

- Identifying possible entry points for untrusted input then tracing from those locations to see if the destination contains potentially vulnerable functions.
- Identifying known, dangerous library / API calls (e.g. SQL queries) and then checking whether unchecked input successfully interfaces with respective queries.

During a manual security review, you should employ a combination of both techniques. In general, untrusted inputs enter mobile apps through the following channels:

- IPC calls
- Custom URL schemes
- QR codes
- Input files received via Bluetooth, NFC, or other means
- Pasteboards
- User interface

Verify that the following best practices have been followed:

- Untrusted inputs are type-checked and/or validated using a list of acceptable values.
- Prepared statements with variable binding (i.e. parameterized queries) are used when performing database queries. If prepared statements are defined, user-supplied data and SQL code are automatically separated.
- When parsing XML data, ensure the parser application is configured to reject resolution of external entities in order to prevent XXE attack.
- When working with X.509 formatted certificate data, ensure that secure parsers are used. For instance Bouncy Castle below version 1.6 allows for Remote Code Execution by means of unsafe reflection.

We will cover details related to input sources and potentially vulnerable APIs for each mobile OS in the OS-specific testing guides.

Reference

- owasp-mastg Injection Flaws (MSTG-ARCH-2 and MSTG-PLATFORM-2) Injection Attack Vectors

Rulebook

- *Do not include potentially vulnerable functions in the destination (Required)*
- *Unchecked inputs are successfully linked to their respective queries (Required)*
- *Check for untrusted input (Required)*
- *Parser application is configured to refuse to resolve external entities (Required)*
- *Use a secure parser when using certificate data in X.509 format (Required)*

1.2.3 Rulebook

1. *Enforce appropriate input validation (Required)*
2. *Input should always be validated and meta-characters should be escaped (Required)*
3. *Do not include potentially vulnerable functions in the destination (Required)*
4. *Unchecked inputs are successfully linked to their respective queries (Required)*
5. *Check for untrusted input (Required)*
6. *Parser application is configured to refuse to resolve external entities (Required)*
7. *Use a secure parser when using certificate data in X.509 format (Required)*

1.2.3.1 Enforce appropriate input validation (Required)

Proper input validation is a necessary best practice for programmers, as injection vulnerabilities can be exploited.

Below is an example of input validation.

- Regular expression check
- Length/size check

If this is violated, the following may occur.

- An injection vulnerability may be exploited.

1.2.3.2 Input should always be validated and meta-characters should be escaped (Required)

If an XML query is created using user-supplied or untrusted content, XML metacharacters may be interpreted as XML content by the local XML parser.Therefore, input should always be validated and meta-characters should be escaped.

Below is an example of input validation.

- Regular expression check
- Length/size check

Below is an example of a meta-character.

Table 1.2.3.2.1 List of Meta-Characters

Character	Item Name	Entity Reference Notation
<	Right Greater Than	&lt;
>	Left Greater Than	&gt;
&	Ampersand	&amp;
“	Double Quotation	&quot;
‘	Single Quotation	&apos;

If this is violated, the following may occur.

- XML meta characters may be interpreted as XML content by the local XML parser.

1.2.3.3 Do not include potentially vulnerable functions in the destination (Required)

Identify potential entry points for untrusted inputs and trace from that location to see if the destination contains potentially vulnerable functions (third-party functions).

If this is violated, the following may occur.

- A malicious app could attack another app running on the device via the Interprocess Communication (IPC) interface.

1.2.3.4 Unchecked inputs are successfully linked to their respective queries (Required)

Identify known dangerous library /API calls (e.g., SQL queries) and verify that unchecked inputs work with the respective queries successfully. Also, check the reference of the library/API to be used and confirm that it is not deprecated.

iOS API Reference：https://developer.apple.com/documentation/

If this is violated, the following may occur.

- A malicious app could attack another app running on the device via the Interprocess Communication (IPC) interface.

1.2.3.5 Check for untrusted input (Required)

In general, untrusted inputs enter mobile apps through the following channels:

Below is an example of keywords that identify channel use.

- IPC calls：NSXPCConnection, XPC Services
- Custom URL schemes：deeplink, URL Schemes, identifier
- QR codes：qr, camera
- Input files received via Bluetooth, NFC, or other means：MCNearbyServiceBrowser, MCNearbyServiceAdvertiser, Core Bluetooth
- Pasteboards：UIPasteboard
- User interface：UITextField

If this is violated, the following may occur.

- A malicious app could attack another app running on the device via the Interprocess Communication (IPC) interface.

1.2.3.6 Parser application is configured to refuse to resolve external entities (Required)

To prevent XXE attacks, parser applications should be configured to refuse to resolve external entities.

The safest way to prevent XXE is always to disable DTDs (External Entities) completely. Depending on the parser, the method should be similar to the following:

```
factory.setFeature("http://apache.org/xml/features/disallow-doctype-decl", true);
```

Disabling DTD also makes the parser secure against denial of services (DOS) attacks such as Billion Laughs. If it is not possible to disable DTDs completely, then external entities and external document type declarations must be disabled in the way that' s specific to each parser.

If this is violated, the following may occur.

- Be vulnerable to XXE attacks.

Reference

- OWASP Cheat Sheet Series XML External Entity Prevention

1.2.3.7 Use a secure parser when using certificate data in X.509 format (Required)

Use a secure parser when handling X.509 format certificate data.

Below is an example of a safe parser.

- SecCertificateCreateWithData

If this is violated, the following may occur.

- Vulnerable to injection attacks, such as remote code execution via insecure reflection.

1.3 MSTG-ARCH-3

A high-level architecture for the mobile app and all connected remote services has been defined and security has been addressed in that architecture.

* Guide description is omitted in this document as this chapter is about support on the remote service side.

1.4 MSTG-ARCH-4

Data considered sensitive in the context of the mobile app is clearly identified.

Classifications of sensitive information differ by industry and country. In addition, organizations may take a restrictive view of sensitive data, and they may have a data classification policy that clearly defines sensitive information. If a data classification policies is not available, use the following list of information generally considered sensitive.

Reference

- owasp-mastg Mobile Application Security Testing Identifying Sensitive Datas

Rulebook

- *Identify sensitive data according to data classification policies (Required)*

1.4.1 User authentication information

User authentication information (credentials, PIN, etc.).

1.4.2 Personally Identifiable Information

Personally Identifiable Information (PII) that can be abused for identity theft: social security numbers, credit card numbers, bank account numbers, health information.

1.4.3 Device Identifier

Device identifiers that may identify a person

1.4.4 Sensitive data

Highly sensitive data whose compromise would lead to reputational harm and/or financial costs.

1.4.5 Data whose protection is a legal obligation

Data whose protection is a legal obligation.

1.4.6 Technical data

Technical data generated by the app (or its related systems) and used to protect other data or the system itself (e.g., encryption keys).

1.4.7 Rulebook

1. *Identify sensitive data according to data classification policies (Required)*

1.4.7.1 Identify sensitive data according to data classification policies (Required)

Identify sensitive data according to a data classification policy that is clearly defined by the industry, country, and organization. If a data classification policy is not available, use the following list of information generally considered confidential.

- User authentication information (credentials, PINs, etc.)
- Personally Identifiable Information (PII) that can be abused for identity theft: social security numbers, credit card numbers, bank account numbers, health information
- Device identifiers that may identify a person
- Highly sensitive data whose compromise would lead to reputational harm and/or financial costs
- Data whose protection is a legal obligation
- Technical data generated by the app (or its related systems) and used to protect other data or the system itself (e.g., encryption keys).

A definition of "sensitive data" must be decided before testing begins because detecting sensitive data leakage without a definition may be impossible.

If this is violated, the following may occur.

- Sensitive data that could be compromised based on the results of penetration testing may not be recognized as sensitive data, and may not be identified and addressed as a risk.

1.5 MSTG-ARCH-12

The app should comply with privacy laws and regulations.

1.5.1 General Privacy Laws and Regulations

Reference

- V6.1 Data Classification

1.5.1.1 Personal Information and Privacy

If personal information is handled, it must comply with GDPR and Act on the Protection of Personal Information.

1.5.1.2 Medical Data

If medical data is handled, it must be HIPAA and HITECH compliant.

1.5.1.3 Financial Information

Credit Card Information

If credit card information is handled, it must be PCI DSS compliant.

1.5.2 App Store privacy rules

Protecting user privacy is paramount in the Apple ecosystem, and you should use care when handling personal data to ensure you've complied with privacy best practices, applicable laws, and the terms of the Apple Developer Program License Agreement, not to mention customer expectations. More particularly:

Reference

- App Store Legal 5.1 Privacy

1.5.2.1 Data Collection and Storage

1. **Privacy Policies** App Store Connect metadata field and within the app in an easily accessible manner. The privacy policy must clearly and explicitly:
   - Identify what data, if any, the app/service collects, how it collects that data, and all uses of that data.
   - Confirm that any third party with whom an app shares user data (in compliance with these Guidelines)—such as analytics tools, advertising networks and third-party SDKs, as well as any parent, subsidiary or other related entities that will have access to user data—will provide the same or equal protection of user data as stated in the app's privacy policy and required by these Guidelines.
   - Explain its data retention/deletion policies and describe how a user can revoke consent and/or request deletion of the user's data.
2. **Permission** Apps that collect user or usage data must secure user consent for the collection, even if such data is considered to be anonymous at the time of or immediately following collection. Paid functionality must not be dependent on or require a user to grant access to this data. Apps must also provide the customer with an easily accessible and understandable way to withdraw consent. Ensure your purpose strings clearly and completely describe your use of the data. Apps that collect data for a legitimate interest without consent by relying on the terms of the European Union's General Data Protection Regulation ("GDPR") or similar statute must comply with all terms of that law. Learn more about Requesting Permission.
3. **Data Minimization** Apps should only request access to data relevant to the core functionality of the app and should only collect and use data that is required to accomplish the relevant task. Where possible, use the out-of-process picker or a share sheet rather than requesting full access to protected resources like Photos or Contacts.
4. **Access** Apps must respect the user's permission settings and not attempt to manipulate, trick, or force people to consent to unnecessary data access. For example, apps that include the ability to post photos to a social network must not also require microphone access before allowing the user to upload photos. Where possible, provide alternative solutions for users who don't grant consent. For example, if a user declines to share Location, offer the ability to manually enter an address.
5. **Account Sign-In** If your app doesn't include significant account-based features, let people use it without a login. If your app supports account creation, you must also offer account deletion within the app. Apps may not require users to enter personal information to function, except when directly relevant to the core functionality of the app or required by law. If your core app functionality is not related to a specific social network (e.g. Facebook, WeChat, Weibo, Twitter, etc.), you must provide access without a login or via another mechanism. Pulling basic profile information, sharing to the social network, or inviting friends to use the app are not considered core app functionality. The app must also include a mechanism to revoke social network credentials and disable data access between the app and social network from within the app. An app may not

store credentials or tokens to social networks off of the device and may only use such credentials or tokens to directly connect to the social network from the app itself while the app is in use.

6. Developers that use their apps to surreptitiously discover passwords or other private data will be removed from the Apple Developer Program.
7. SafariViewController must be used to visibly present information to users; the controller may not be hidden or obscured by other views or layers. Additionally, an app may not use SafariViewController to track users without their knowledge and consent.
8. Apps that compile personal information from any source that is not directly from the user or without the user's explicit consent, even public databases, are not permitted on the App Store.
9. Apps that provide services in highly regulated fields (such as banking and financial services, healthcare, gambling, legal cannabis use, and air travel) or that require sensitive user information should be submitted by a legal entity that provides the services, and not by an individual developer. Apps that facilitate the legal sale of cannabis must be geo-restricted to the corresponding legal jurisdiction.
10. Apps may request basic contact information (such as name and email address) so long as the request is optional for the user, features and services are not conditional on providing the information, and it complies with all other provisions of these guidelines, including limitations on collecting information from kids.

Reference

- App Store Legal 5.1.1 Data Collection and Storage

1.5.2.2 Data Use and Sharing

1. Unless otherwise permitted by law, you may not use, transmit, or share someone's personal data without first obtaining their permission. You must provide access to information about how and where the data will be used. Data collected from apps may only be shared with third parties to improve the app or serve advertising (in compliance with the Apple Developer Program License Agreement). You must receive explicit permission from users via the App Tracking Transparency APIs to track their activity. Learn more about tracking. Apps that share user data without user consent or otherwise complying with data privacy laws may be removed from sale and may result in your removal from the Apple Developer Program.
2. Data collected for one purpose may not be repurposed without further consent unless otherwise explicitly permitted by law.
3. Apps should not attempt to surreptitiously build a user profile based on collected data and may not attempt, facilitate, or encourage others to identify anonymous users or reconstruct user profiles based on data collected from Apple-provided APIs or any data that you say has been collected in an "anonymized," "aggregated," or otherwise non-identifiable way.
4. Do not use information from Contacts, Photos, or other APIs that access user data to build a contact database for your own use or for sale/distribution to third parties, and don't collect information about which other apps are installed on a user's device for the purposes of analytics or advertising/marketing.
5. Do not contact people using information collected via a user's Contacts or Photos, except at the explicit initiative of that user on an individualized basis; do not include a Select All option or default the selection of all contacts. You must provide the user with a clear description of how the message will appear to the recipient before sending it (e.g. What will the message say? Who will appear to be the sender?).
6. Data gathered from the HomeKit API, HealthKit, Clinical Health Records API, MovementDisorder APIs, ClassKit or from depth and/or facial mapping tools (e.g. ARKit, Camera APIs, or Photo APIs) may not be used for marketing, advertising or use-based data mining, including by third parties. Learn more about best practices for implementing CallKit, HealthKit, ClassKit,and ARKit.
7. Apps using Apple Pay may only share user data acquired via Apple Pay with third parties to facilitate or improve delivery of goods and services.

Reference

- App Store Legal 5.1.2 Data Use and Sharing

1.5.2.3 Health and Health Research

Health, fitness, and medical data are especially sensitive and apps in this space have some additional rules to make sure customer privacy is protected:

1. Apps may not use or disclose to third parties data gathered in the health, fitness, and medical research context —including from the Clinical Health Records API, HealthKit API, Motion and Fitness, MovementDisorder APIs, or health-related human subject research—for advertising, marketing, or other use-based data mining purposes other than improving health management, or for the purpose of health research, and then only with permission. Apps may, however, use a user's health or fitness data to provide a benefit directly to that user (such as a reduced insurance premium), provided that the app is submitted by the entity providing the benefit, and the data is not shared with a third party. You must disclose the specific health data that you are collecting from the device.
2. Apps must not write false or inaccurate data into HealthKit or any other medical research or health management apps, and may not store personal health information in iCloud.
3. Apps conducting health-related human subject research must obtain consent from participants or, in the case of minors, their parent or guardian. Such consent must include the (a) nature, purpose, and duration of the research; (b) procedures, risks, and benefits to the participant; (c) information about confidentiality and handling of data (including any sharing with third parties); (d) a point of contact for participant questions; and (e) the withdrawal process.
4. Apps conducting health-related human subject research must secure approval from an independent ethics review board. Proof of such approval must be provided upon request.

Reference

- App Store Legal 5.1.3 Health and Health Research

1.5.2.4 Kids

For many reasons, it is critical to use care when dealing with personal data from kids, and we encourage you to carefully review all the requirements for complying with laws like the Children's Online Privacy Protection Act ("COPPA"), the European Union's General Data Protection Regulation ("GDPR"), and any other applicable regulations or laws.

Apps may ask for birthdate and parental contact information only for the purpose of complying with these statutes, but must include some useful functionality or entertainment value regardless of a person's age.

Apps intended primarily for kids should not include third-party analytics or third-party advertising. This provides a safer experience for kids. In limited cases, third-party analytics and third-party advertising may be permitted provided that the services adhere to the same terms set forth in Guideline 1.3.

Moreover, apps in the Kids Category or those that collect, transmit, or have the capability to share personal information (e.g. name, address, email, location, photos, videos, drawings, the ability to chat, other personal data, or persistent identifiers used in combination with any of the above) from a minor must include a privacy policy and must comply with all applicable children's privacy statutes. For the sake of clarity, the parental gate requirement for the Kid's Category is generally not the same as securing parental consent to collect personal data under these privacy statutes.

As a reminder, Guideline 2.3.8 requires that use of terms like "For Kids" and "For Children" in app metadata is reserved for the Kids Category. Apps not in the Kids Category cannot include any terms in app name, subtitle, icon, screenshots or description that imply the main audience for the app is children.

Reference

- App Store Legal 5.1.4 Kids

1.5.2.5 Location Services

Use Location services in your app only when it is directly relevant to the features and services provided by the app. Location-based APIs shouldn't be used to provide emergency services or autonomous control over vehicles, aircraft, and other devices, except for small devices such as lightweight drones and toys, or remote control car alarm systems, etc. Ensure that you notify and obtain consent before collecting, transmitting, or using location data. If your app uses location services, be sure to explain the purpose in your app; refer to the Human Interface Guidelines for best practices for doing so.

Reference

- App Store Legal 5.1.5 Location Services

1.5.3 Intellectual Property App Store Rules

Make sure your app only includes content that you created or that you have a license to use. Your app may be removed if you've stepped over the line and used content without permission. Of course, this also means someone else's app may be removed if they've "borrowed" from your work. If you believe your intellectual property has been infringed by another developer on the App Store, submit a claim via our web form. Laws differ in different countries and regions, but at the very least, make sure to avoid the following common errors:

1. **Generally** Don't use protected third-party material such as trademarks, copyrighted works, or patented ideas in your app without permission, and don't include misleading, false, or copycat representations, names, or metadata in your app bundle or developer name. Apps should be submitted by the person or legal entity that owns or has licensed the intellectual property and other relevant rights.
2. **Third-Party Sites/Services** If your app uses, accesses, monetizes access to, or displays content from a third-party service, ensure that you are specifically permitted to do so under the service's terms of use. Authorization must be provided upon request.
3. **Audio/Video Downloading** Apps should not facilitate illegal file sharing or include the ability to save, convert, or download media from third-party sources (e.g. Apple Music, YouTube, SoundCloud, Vimeo, etc.) without explicit authorization from those sources. Streaming of audio/video content may also violate Terms of Use, so be sure to check before your app accesses those services. Documentation must be provided upon request.
4. **Apple Endorsements** Don't suggest or imply that Apple is a source or supplier of the App, or that Apple endorses any particular representation regarding quality or functionality. If your app is selected as an "Editor's Choice," Apple will apply the badge automatically.
5. **Apple Products** Don't create an app that appears confusingly similar to an existing Apple product, interface (e.g. Finder), app (such as the App Store, iTunes Store, or Messages) or advertising theme. Apps and extensions, including third-party keyboards and Sticker packs, may not include Apple emoji. Music from iTunes and Apple Music previews may not be used for their entertainment value (e.g. as the background music to a photo collage or the soundtrack to a game) or in any other unauthorized manner. If you provide music previews from iTunes or Apple Music, you must display a link to the corresponding music in iTunes or Apple Music. If your app displays Activity rings, they should not visualize Move, Exercise, or Stand data in a way that resembles the Activity control. The Human Interface Guidelines have more information on how to use Activity rings. If your app displays Apple Weather data, it should follow the attribution requirements provided in the WeatherKit documentation.

Reference

- App Store Legal 5.2 Intellectual Property

2 Data Storage and Privacy Requirements

2.1 MSTG-STORAGE-1

System credential storage facilities need to be used to store sensitive data, such as PII, user credentials or cryptographic keys.

As little sensitive data as possible should be saved in permanent local storage. However, in most practical scenarios, at least some user data must be stored. Fortunately, iOS offers secure storage APIs, which allow developers to use the cryptographic hardware available on every iOS device. If these APIs are used correctly, sensitive data and files can be secured via hardware-backed 256-bit AES encryption.

2.1.1 Data Protection API

App developers can leverage the iOS Data Protection APIs to implement fine-grained access control for user data stored in flash memory. The APIs are built on top of the Secure Enclave Processor (SEP), which was introduced with the iPhone 5S. The SEP is a coprocessor that provides cryptographic operations for data protection and key management. A device-specific hardware key-the device UID (Unique ID)-is embedded in the secure enclave, ensuring the integrity of data protection even when the operating system kernel is compromised.

The data protection architecture is based on a hierarchy of keys. The UID and the user passcode key (which is derived from the user' s passphrase via the PBKDF2 algorithm) sit at the top of this hierarchy. Together, they can be used to "unlock" so-called class keys, which are associated with different device states (e.g., device locked/unlocked).

Every file stored on the iOS file system is encrypted with its own per-file key, which is contained in the file metadata. The metadata is encrypted with the file system key and wrapped with the class key corresponding to the protection class the app selected when creating the file.

The following illustration shows the iOS Data Protection Key Hierarchy.

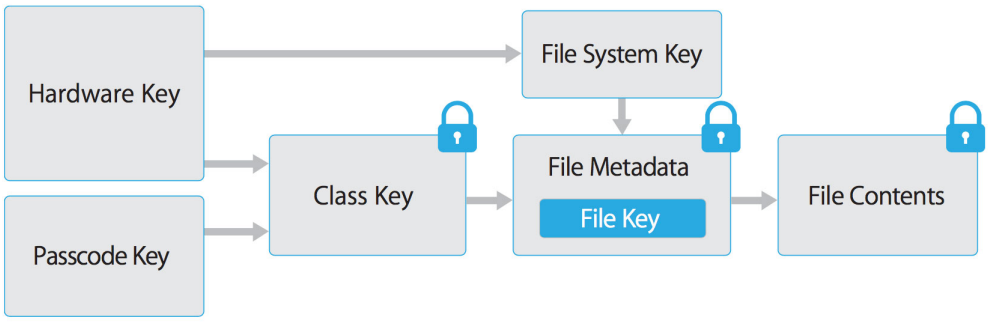


Fig 2.1.1.1 iOS Data Protection Key Hierarchy

Files can be assigned to one of four different protection classes, which are explained in more detail in the iOS Security Guide:

- Complete Protection (NSFileProtectionComplete): A key derived from the user passcode and the device UID protects this class key. The derived key is wiped from memory shortly after the device is locked, making the data inaccessible until the user unlocks the device.
- Protected Unless Open (NSFileProtectionCompleteUnlessOpen): This protection class is similar to Complete Protection, but, if the file is opened when unlocked, the app can continue to access the file even if the user locks the device. This protection class is used when, for example, a mail attachment is downloading in the background.
- Protected Until First User Authentication (NSFileProtectionCompleteUntilFirstUserAuthentication): The file can be accessed as soon as the user unlocks the device for the first time after booting. It can be accessed even if the user subsequently locks the device and the class key is not removed from memory.
- No Protection (NSFileProtectionNone): The key for this protection class is protected with the UID only. The class key is stored in "Effaceable Storage", which is a region of flash memory on the iOS device that allows the storage of small amounts of data. This protection class exists for fast remote wiping (immediate deletion of the class key, which makes the data inaccessible).

All class keys except NSFileProtectionNone are encrypted with a key derived from the device UID and the user's passcode. As a result, decryption can happen only on the device itself and requires the correct passcode.

Since iOS 7, the default data protection class is "Protected Until First User Authentication".

Reference

- owasp-mastg Testing Local Data Storage (MSTG-STORAGE-1 and MSTG-STORAGE-2) Data Protection API

RuleBook

- *Implement access control for user data stored in flash memory utilizing the iOS Data Protection API (Required)*

2.1.2 The Keychain

The iOS Keychain can be used to securely store short, sensitive bits of data, such as encryption keys and session tokens. It is implemented as an SQLite database that can be accessed through the Keychain APIs only.

On macOS, every user application can create as many Keychains as desired, and every login account has its own Keychain.

The structure of the Keychain on iOS is different: only one Keychain is available to all apps. Access to the items can be shared between apps signed by the same developer via the access groups feature of the attribute kSecAttrAccessGroup. Access to the Keychain is managed by the securityd daemon, which grants access according to the app's Keychain-access-groups, application-identifier, and application-group entitlements.

The Keychain API includes the following main operations:

- SecItemAdd
- SecItemUpdate
- SecItemCopyMatching
- SecItemDelete

Data stored in the Keychain is protected via a class structure that is similar to the class structure used for file encryption. Items added to the Keychain are encoded as a binary plist and encrypted with a 128-bit AES per-item key in Galois/Counter Mode (GCM). Note that larger blobs of data aren't meant to be saved directly in the Keychain-that's what the Data Protection API is for. You can configure data protection for Keychain items by setting the kSecAttrAccessible key in the call to SecItemAdd or SecItemUpdate. The following configurable accessibility values for kSecAttrAccessible are the Keychain Data Protection classes:

- kSecAttrAccessibleAlways: The data in the Keychain item can always be accessed, regardless of whether the device is locked.
- kSecAttrAccessibleAlwaysThisDeviceOnly: The data in the Keychain item can always be accessed, regardless of whether the device is locked. The data won't be included in an iCloud or local backup.
- kSecAttrAccessibleAfterFirstUnlock: The data in the Keychain item can't be accessed after a restart until the device has been unlocked once by the user.
- kSecAttrAccessibleAfterFirstUnlockThisDeviceOnly: The data in the Keychain item can't be accessed after a restart until the device has been unlocked once by the user. Items with this attribute do not migrate to a new device. Thus, after restoring from a backup of a different device, these items will not be present.
- kSecAttrAccessibleWhenUnlocked: The data in the Keychain item can be accessed only while the device is unlocked by the user.
- kSecAttrAccessibleWhenUnlockedThisDeviceOnly: The data in the Keychain item can be accessed only while the device is unlocked by the user. The data won't be included in an iCloud or local backup.
- kSecAttrAccessibleWhenPasscodeSetThisDeviceOnly: The data in the Keychain can be accessed only when the device is unlocked. This protection class is only available if a passcode is set on the device. The data won't be included in an iCloud or local backup.

AccessControlFlags define the mechanisms with which users can authenticate the key (SecAccessControlCreateFlags):

- kSecAccessControlDevicePasscode: Access the item via a passcode.
- kSecAccessControlBiometryAny: Access the item via one of the fingerprints registered to Touch ID. Adding or removing a fingerprint won't invalidate the item.
- kSecAccessControlBiometryCurrentSet: Access the item via one of the fingerprints registered to Touch ID. Adding or removing a fingerprint will invalidate the item.
- kSecAccessControlUserPresence: Access the item via either one of the registered fingerprints (using Touch ID) or default to the passcode.

Please note that keys secured by Touch ID (via kSecAccessControlBiometryAny or kSecAccessControlBiometryCurrentSet) are protected by the Secure Enclave: The Keychain holds a token only, not the actual key. The key resides in the Secure Enclave.

Starting with iOS 9, you can do ECC-based signing operations in the Secure Enclave. In that scenario, the private key and the cryptographic operations reside within the Secure Enclave. See the static analysis section for more info on creating the ECC keys. iOS 9 supports only 256-bit ECC. Furthermore, you need to store the public key in the Keychain because it can't be stored in the Secure Enclave. After the key is created, you can use the kSecAttrKeyType to indicate the type of algorithm you want to use the key with.

In case you want to use these mechanisms, it is recommended to test whether the passcode has been set. In iOS 8, you will need to check whether you can read/write from an item in the Keychain protected by the kSecAttrAccessibleWhenPasscodeSetThisDeviceOnly attribute. From iOS 9 onward you can check whether a lock screen is set, using LAContext:

Swift:

```
public func devicePasscodeEnabled() -> Bool {
    return LAContext().canEvaluatePolicy(.deviceOwnerAuthentication, error: nil)
}
```

Objective-C:

```
-(BOOL)devicePasscodeEnabled:(LAContex)context{
  if ([context canEvaluatePolicy:LAPolicyDeviceOwnerAuthentication error:nil]) {
        return true;
    } else {
        return false;
    }
}
```

Keychain Data Persistence

On iOS, when an application is uninstalled, the Keychain data used by the application is retained by the device, unlike the data stored by the application sandbox which is wiped. In the event that a user sells their device without performing a factory reset, the buyer of the device may be able to gain access to the previous user's application accounts and data by reinstalling the same applications used by the previous user. This would require no technical ability to perform.

When assessing an iOS application, you should look for Keychain data persistence. This is normally done by using the application to generate sample data that may be stored in the Keychain, uninstalling the application, then reinstalling the application to see whether the data was retained between application installations. Use objection runtime mobile exploration toolkit to dump the keychain data. The following objection command demonstrates this procedure:

```
...itudehacks.DVIAswiftv2.develop on (iPhone: 13.2.3) [usb] # ios keychain dump
Note: You may be asked to authenticate using the devices passcode or TouchID
Save the output by adding `--json keychain.json` to this command
Dumping the iOS keychain...
Created                   Accessible                      ACL    Type      ␣
↪Account                    Service                                         ␣
↪          Data
------------------------  ------------------------------  -----  --------  -------
↪--------------------  ------------------------------------------------------------
↪  ------------------------------------
2020-02-11 13:26:52 +0000  WhenUnlocked                    None   Password ␣
↪keychainValue              com.highaltitudehacks.DVIAswiftv2.develop            ␣
↪          mysecretpass123
```

There's no iOS API that developers can use to force wipe data when an application is uninstalled. Instead, developers should take the following steps to prevent Keychain data from persisting between application installations:

- When an application is first launched after installation, wipe all Keychain data associated with the application. This will prevent a device's second user from accidentally gaining access to the previous user's accounts. The following Swift example is a basic demonstration of this wiping procedure:

```
let userDefaults = UserDefaults.standard

if userDefaults.bool(forKey: "hasRunBefore") == false {
    // Remove Keychain items here

    // Update the flag indicator
    userDefaults.set(true, forKey: "hasRunBefore")
    userDefaults.synchronize() // Forces the app to update UserDefaults
}
```

- When developing logout functionality for an iOS application, make sure that the Keychain data is wiped as part of account logout. This will allow users to clear their accounts before uninstalling an application.

Static Analysis

When you have access to the source code of an iOS app, identify sensitive data that's saved and processed throughout the app. This includes passwords, secret keys, and personally identifiable information (PII), but it may as well include other data identified as sensitive by industry regulations, laws, and company policies. Look for this data being saved via any of the local storage APIs listed below.

Make sure that sensitive data is never stored without appropriate protection. For example, authentication tokens should not be saved in NSUserDefaults without additional encryption. Also avoid storing encryption keys in .plist files, hardcoded as strings in code, or generated using a predictable obfuscation function or key derivation function based on stable attributes.

Sensitive data should be stored by using the Keychain API (that stores them inside the Secure Enclave), or stored encrypted using envelope encryption. Envelope encryption, or key wrapping, is a cryptographic construct that uses symmetric encryption to encapsulate key material. Data encryption keys (DEK) can be encrypted with key encryption keys (KEK) which must be securely stored in the Keychain. Encrypted DEK can be stored in NSUserDefaults or written in files. When required, application reads KEK, then decrypts DEK. Refer to OWASP Cryptographic Storage Cheat Sheet to learn more about encrypting cryptographic keys.

Keychain

The encryption must be implemented so that the secret key is stored in the Keychain with secure settings, ideally kSecAttrAccessibleWhenPasscodeSetThisDeviceOnly. This ensures the usage of hardware-backed storage mechanisms. Make sure that the AccessControlFlags are set according to the security policy of the keys in the Keychain.

Generic examples of using the Keychain to store, update, and delete data can be found in the official Apple documentation. The official Apple documentation also includes an example of using Touch ID and passcode protected keys.

Here is sample Swift code you can use to create keys (Notice the kSecAttrTokenID as String: kSecAttrTokenIDSecureEnclave: this indicates that we want to use the Secure Enclave directly.):

```
// private key parameters
let privateKeyParams = [
    kSecAttrLabel as String: "privateLabel",
    kSecAttrIsPermanent as String: true,
    kSecAttrApplicationTag as String: "applicationTag",
] as CFDictionary

// public key parameters
let publicKeyParams = [
    kSecAttrLabel as String: "publicLabel",
    kSecAttrIsPermanent as String: false,
    kSecAttrApplicationTag as String: "applicationTag",
] as CFDictionary

// global parameters
let parameters = [
    kSecAttrKeyType as String: kSecAttrKeyTypeEC, // Note that kSecAttrKeyTypeEC␣
↪is now deprecated.
    kSecAttrKeySizeInBits as String: 256,
    kSecAttrTokenID as String: kSecAttrTokenIDSecureEnclave,
    kSecPublicKeyAttrs as String: publicKeyParams,
    kSecPrivateKeyAttrs as String: privateKeyParams,
] as CFDictionary

var pubKey, privKey: SecKey?
let status = SecKeyGeneratePair(parameters, &pubKey, &privKey) // Note that␣
↪SecKeyGeneratePair is now deprecated.

if status != errSecSuccess {
    // Keys created successfully
}
```

When checking an iOS app for insecure data storage, consider the following ways to store data because none of them encrypt data by default:

Reference

- owasp-mastg Testing Local Data Storage (MSTG-STORAGE-1 and MSTG-STORAGE-2) The Keychain
- owasp-mastg Testing Local Data Storage (MSTG-STORAGE-1 and MSTG-STORAGE-2) Static Analysis

RuleBook

- *Securely store values using the Keychain Services API (Required)*

2.1.3 NSUserDefaults

The NSUserDefaults class provides a programmatic interface for interacting with the default system. The default system allows an application to customize its behavior according to user preferences. Data saved by NSUserDefaults can be viewed in the application bundle. This class stores data in a plist file, but it's meant to be used with small amounts of data.

Reference

- owasp-mastg Testing Local Data Storage (MSTG-STORAGE-1 and MSTG-STORAGE-2) NSUserDefaults

Rulebook

- *Selecting a storage destination based on the amount of data (Recommended)*

2.1.4 File system

2.1.4.1 NSData

creates static data objects, while NSMutableData creates dynamic data objects. NSData and NSMutableData are typically used for data storage, but they are also useful for distributed objects applications, in which data contained in data objects can be copied or moved between applications. The following are methods used to write NSData objects:

- NSDataWritingWithoutOverwriting
- NSDataWritingFileProtectionNone
- NSDataWritingFileProtectionComplete
- NSDataWritingFileProtectionCompleteUnlessOpen
- NSDataWritingFileProtectionCompleteUntilFirstUserAuthentication

Reference

- owasp-mastg Testing Local Data Storage (MSTG-STORAGE-1 and MSTG-STORAGE-2) File system

RuleBook

- *Implement access control for user data stored in flash memory utilizing the iOS Data Protection API (Required)*

2.1.4.2 writeToFile

stores data as part of the NSData class

Reference

- owasp-mastg Testing Local Data Storage (MSTG-STORAGE-1 and MSTG-STORAGE-2) File system

2.1.4.3 used to manage file paths

NSSearchPathForDirectoriesInDomains, NSTemporaryDirectory: used to manage file paths

Reference

- owasp-mastg Testing Local Data Storage (MSTG-STORAGE-1 and MSTG-STORAGE-2) File system

2.1.4.4 NSFileManager

lets you examine and change the contents of the file system. You can use createFileAtPath to create a file and write to it.

The following example shows how to create a complete encrypted file using the FileManager class. You can find more information in the Apple Developer Documentation "Encrypting Your App's Files".

Swift:

```
FileManager.default.createFile(
    atPath: filePath,
    contents: "secret text".data(using: .utf8),
    attributes: [FileAttributeKey.protectionKey: FileProtectionType.complete]
)
```

Objective-C:

```
[[NSFileManager defaultManager] createFileAtPath:[self filePath]
  contents:[@"secret text" dataUsingEncoding:NSUTF8StringEncoding]
  attributes:[NSDictionary dictionaryWithObject:NSFileProtectionComplete
  forKey:NSFileProtectionKey]];
```

Reference

- owasp-mastg Testing Local Data Storage (MSTG-STORAGE-1 and MSTG-STORAGE-2) File system

Rulebook

- *How to use encryption in the FileManager class (Required)*

2.1.5 CoreData/SQLite Databases

2.1.5.1 Core Data

Core Data is a framework for managing the model layer of objects in your application. It provides general and automated solutions to common tasks associated with object life cycles and object graph management, including persistence. Core Data can use SQLite as its persistent store, but the framework itself is not a database.

CoreData does not encrypt it's data by default. As part of a research project (iMAS) from the MITRE Corporation, that was focused on open source iOS security controls, an additional encryption layer can be added to CoreData. See the GitHub Repo for more details.

Reference

- owasp-mastg Testing Local Data Storage (MSTG-STORAGE-1 and MSTG-STORAGE-2) Core Data

2.1.5.2 SQLite Databases

The SQLite 3 library must be added to an app if the app is to use SQLite. This library is a C++ wrapper that provides an API for the SQLite commands.

Reference

- owasp-mastg Testing Local Data Storage (MSTG-STORAGE-1 and MSTG-STORAGE-2) SQLite Databases

2.1.5.3 Firebase Real-time Databases

Firebase is a development platform with more than 15 products, and one of them is Firebase Real-time Database. It can be leveraged by application developers to store and sync data with a NoSQL cloud-hosted database. The data is stored as JSON and is synchronized in real-time to every connected client and also remains available even when the application goes offline.

A misconfigured Firebase instance can be identified by making the following network call:

```
https://\<firebaseProjectName\>.firebaseio.com/.json
```

The firebaseProjectName can be retrieved from the property list(.plist) file. For example, PROJECT_ID key stores the corresponding Firebase project name in GoogleService-Info.plist file.

Alternatively, the analysts can use Firebase Scanner, a python script that automates the task above as shown below:

```
python FirebaseScanner.py -f <commaSeparatedFirebaseProjectNames>
```

Reference

- owasp-mastg Testing Local Data Storage (MSTG-STORAGE-1 and MSTG-STORAGE-2) Firebase Real-time Databases

2.1.5.4 Realm databases

Realm Objective-C and Realm Swift aren't supplied by Apple, but they are still worth noting. They store everything unencrypted, unless the configuration has encryption enabled.

The following example demonstrates how to use encryption with a Realm database:

```
// Open the encrypted Realm file where getKey() is a method to obtain a key from␣
↪the Keychain or a server
let config = Realm.Configuration(encryptionKey: getKey())
do {
 let realm = try Realm(configuration: config)
 // Use the Realm as normal
} catch let error as NSError {
 // If the encryption key is wrong, `error` will say that it's an invalid database
 fatalError("Error opening realm: \(error)")
}
```

Reference

- owasp-mastg Testing Local Data Storage (MSTG-STORAGE-1 and MSTG-STORAGE-2) Real Databases

2.1.5.5 Couchbase Lite Databases

Couchbase Lite is a lightweight, embedded, document-oriented (NoSQL) database engine that can be synced. It compiles natively for iOS and macOS. Reference

- owasp-mastg Testing Local Data Storage (MSTG-STORAGE-1 and MSTG-STORAGE-2) Couchbase Lite Databases

2.1.5.6 YapDatabase

YapDatabase is a key/value store built on top of SQLite.

Reference

- owasp-mastg Testing Local Data Storage (MSTG-STORAGE-1 and MSTG-STORAGE-2) YapDatabase

Dynamic Analysis

One way to determine whether sensitive information (like credentials and keys) is stored insecurely without leveraging native iOS functions is to analyze the app's data directory. Triggering all app functionality before the data is analyzed is important because the app may store sensitive data only after specific functionality has been triggered. You can then perform static analysis for the data dump according to generic keywords and app-specific data.

The following steps can be used to determine how the application stores data locally on a jailbroken iOS device:

1. Trigger the functionality that stores potentially sensitive data.
2. Connect to the iOS device and navigate to its Bundle directory (this applies to iOS versions 8.0 and above): /var/mobile/Containers/Data/Application/$APP_ID/
3. Execute grep with the data that you've stored, for example: grep -iRn "USERID".
4. If the sensitive data is stored in plaintext, the app fails this test.

You can analyze the app's data directory on a non-jailbroken iOS device by using third-party applications, such as iMazing.

1. Trigger the functionality that stores potentially sensitive data.
2. Connect the iOS device to your host computer and launch iMazing.
3. Select "Apps", right-click the desired iOS application, and select "Extract App".
4. Navigate to the output directory and locate $APP_NAME.imazing. Rename it to $APP_NAME.zip.
5. Unpack the ZIP file. You can then analyze the application data.

Note that tools like iMazing don't copy data directly from the device. They try to extract data from the backups they create. Therefore, getting all the app data that's stored on the iOS device is impossible: not all folders are included in backups. Use a jailbroken device or repackage the app with Frida and use a tool like objection to access all the data and files.

If you added the Frida library to the app and repackaged it as described in "Dynamic Analysis on Non-Jailbroken Devices" (from the "Tampering and Reverse Engineering on iOS" chapter), you can use objection to transfer files directly from the app's data directory or read files in objection as explained in the chapter "Basic Security Testing on iOS", section "Host-Device Data Transfer".

The Keychain contents can be dumped during dynamic analysis. On a jailbroken device, you can use Keychain dumper as described in the chapter "Basic Security Testing on iOS".

The path to the Keychain file is

```
/private/var/Keychains/keychain-2.db
```

On a non-jailbroken device, you can use objection to dump the Keychain items created and stored by the app.

Reference

- owasp-mastg Testing Local Data Storage (MSTG-STORAGE-1 and MSTG-STORAGE-2) Dynamic Analysis

2.1.6 Dynamic Analysis with Xcode and iOS simulator

This test is only available on macOS, as Xcode and the iOS simulator is needed.

For testing the local storage and verifying what data is stored within it, it's not mandatory to have an iOS device. With access to the source code and Xcode the app can be build and deployed in the iOS simulator. The file system of the current device of the iOS simulator is available in ~/Library/Developer/CoreSimulator/Devices.

Once the app is running in the iOS simulator, you can navigate to the directory of the latest simulator started with the following command:

```
$ cd ~/Library/Developer/CoreSimulator/Devices/$(
ls -alht ~/Library/Developer/CoreSimulator/Devices | head -n 2 |
awk '{print $9}' | sed -n '1!p')/data/Containers/Data/Application
```

The command above will automatically find the UUID of the latest simulator started. Now you still need to grep for your app name or a keyword in your app. This will show you the UUID of the app.

```
grep -iRn keyword .
```

Then you can monitor and verify the changes in the filesystem of the app and investigate if any sensitive information is stored within the files while using the app.

Reference

- owasp-mastg Testing Local Data Storage (MSTG-STORAGE-1 and MSTG-STORAGE-2) Dynamic Analysis with Xcode and iOS simulator

2.1.7 Dynamic Analysis with Objection

You can use the Objection runtime mobile exploration toolkit to find vulnerabilities caused by the application's data storage mechanism. Objection can be used without a Jailbroken device, but it will require patching the iOS Application.

Reference

- owasp-mastg Testing Local Data Storage (MSTG-STORAGE-1 and MSTG-STORAGE-2) Dynamic Analysis with Objection

2.1.7.1 Reading the Keychain

To use Objection to read the Keychain, execute the following command:

```
...itudehacks.DVIAswiftv2.develop on (iPhone: 13.2.3) [usb] # ios keychain dump
Note: You may be asked to authenticate using the devices passcode or TouchID
Save the output by adding `--json keychain.json` to this command
Dumping the iOS keychain...
Created                   Accessible                      ACL    Type      ␣
↪Account                    Service                                         ␣
↪         Data
------------------------  ------------------------------  -----  --------  -------
↪------------------  ------------------------------------------------------------
↪  ----------------------------------
2020-02-11 13:26:52 +0000  WhenUnlocked                    None   Password ␣
↪keychainValue              com.highaltitudehacks.DVIAswiftv2.develop           ␣
↪         mysecretpass123
```

Reference

- owasp-mastg Testing Local Data Storage (MSTG-STORAGE-1 and MSTG-STORAGE-2) Dynamic Analysis Reading the Keychain

2.1.7.2 Searching for Binary Cookies

iOS applications often store binary cookie files in the application sandbox. Cookies are binary files containing cookie data for application WebViews. You can use objection to convert these files to a JSON format and inspect the data.

```
...itudehacks.DVIAswiftv2.develop on (iPhone: 13.2.3) [usb] # ios cookies get --
↪json
[
    {
        "domain": "highaltitudehacks.com",
        "expiresDate": "2051-09-15 07:46:43 +0000",
        "isHTTPOnly": "false",
        "isSecure": "false",
        "name": "username",
        "path": "/",
        "value": "admin123",
        "version": "0"
    }
]
```

Reference

- owasp-mastg Testing Local Data Storage (MSTG-STORAGE-1 and MSTG-STORAGE-2) Dynamic Analysis Searching for Binary Cookies

2.1.7.3 Searching for Property List Files

iOS applications often store data in property list (plist) files that are stored in both the application sandbox and the IPA package. Sometimes these files contain sensitive information, such as usernames and passwords; therefore, the contents of these files should be inspected during iOS assessments. Use the ios plist cat plistFileName.plist command to inspect the plist file.

To find the file userInfo.plist, use the env command. It will print out the locations of the applications Library, Caches and Documents directories:

```
...itudehacks.DVIAswiftv2.develop on (iPhone: 13.2.3) [usb] # env
Name               Path
-----------------  ------------------------------------------------------------
↪---------------------------
BundlePath         /private/var/containers/Bundle/Application/B2C8E457-1F0C-4DB1-
↪8C39-04ACBFFEE7C8/DVIA-v2.app
CachesDirectory    /var/mobile/Containers/Data/Application/264C23B8-07B5-4B5D-8701-
↪C020C301C151/Library/Caches
DocumentDirectory  /var/mobile/Containers/Data/Application/264C23B8-07B5-4B5D-8701-
↪C020C301C151/Documents
LibraryDirectory   /var/mobile/Containers/Data/Application/264C23B8-07B5-4B5D-8701-
↪C020C301C151/Library
```

Reference

- owasp-mastg Testing Local Data Storage (MSTG-STORAGE-1 and MSTG-STORAGE-2) Dynamic Analysis Searching for Property List Files

Go to the Documents directory and list all files using ls.

```
...itudehacks.DVIAswiftv2.develop on (iPhone: 13.2.3) [usb] # ls
NSFileType      Perms  NSFileProtection                     Read    Write    ␣
↪Owner          Group         Size      Creation                   Name
------------  -------  -----------------------------------  ------  -------  -----
↪-------  ------------  --------  -------------------------  --------------------
↪---
Directory         493  n/a                                  True    True     ␣
↪mobile (501)  mobile (501)  192.0 B   2020-02-12 07:03:51 +0000  default.realm.
```

(continues on next page)

(continued from previous page)

```
↪management
Regular           420  CompleteUntilFirstUserAuthentication  True    True    ␣
↪mobile (501)  mobile (501)  16.0 KiB  2020-02-12 07:03:51 +0000  default.realm
Regular           420  CompleteUntilFirstUserAuthentication  True    True    ␣
↪mobile (501)  mobile (501)  1.2 KiB   2020-02-12 07:03:51 +0000  default.realm.
↪lock
Regular           420  CompleteUntilFirstUserAuthentication  True    True    ␣
↪mobile (501)  mobile (501)  284.0 B   2020-05-29 18:15:23 +0000  userInfo.plist
Unknown           384  n/a                                   True    True    ␣
↪mobile (501)  mobile (501)  0.0 B     2020-02-12 07:03:51 +0000  default.realm.
↪note

Readable: True  Writable: True
```

Execute the ios plist cat command to inspect the content of userInfo.plist file.

```
...itudehacks.DVIAswiftv2.develop on (iPhone: 13.2.3) [usb] # ios plist cat␣
↪userInfo.plist
{
        password = password123;
        username = userName;
}
```

2.1.7.4 Searching for SQLite Databases

iOS applications typically use SQLite databases to store data required by the application. Testers should check the data protection values of these files and their contents for sensitive data. Objection contains a module to interact with SQLite databases. It allows to dump the schema, their tables and query the records.

```
...itudehacks.DVIAswiftv2.develop on (iPhone: 13.2.3) [usb] # sqlite connect Model.
↪sqlite
Caching local copy of database file...
Downloading /var/mobile/Containers/Data/Application/264C23B8-07B5-4B5D-8701-
↪C020C301C151/Library/Application Support/Model.sqlite to /var/folders/4m/dsg0mq_
↪17g39g473z0996r7m0000gq/T/tmpdr_7rvxi.sqlite
Streaming file from device...
Writing bytes to destination...
Successfully downloaded /var/mobile/Containers/Data/Application/264C23B8-07B5-4B5D-
↪8701-C020C301C151/Library/Application Support/Model.sqlite to /var/folders/4m/
↪dsg0mq_17g39g473z0996r7m0000gq/T/tmpdr_7rvxi.sqlite
Validating SQLite database format
Connected to SQLite database at: Model.sqlite

SQLite @ Model.sqlite > .tables
+--------------+
| name         |
+--------------+
| ZUSER        |
| Z_METADATA   |
| Z_MODELCACHE |
| Z_PRIMARYKEY |
+--------------+
Time: 0.013s

SQLite @ Model.sqlite > select * from Z_PRIMARYKEY
+-------+--------+---------+-------+
| Z_ENT | Z_NAME | Z_SUPER | Z_MAX |
+-------+--------+---------+-------+
| 1     | User   | 0       | 0     |
+-------+--------+---------+-------+
```

(continues on next page)

(continued from previous page)

```
1 row in set
Time: 0.013s
```

Reference

- owasp-mastg Testing Local Data Storage (MSTG-STORAGE-1 and MSTG-STORAGE-2) Dynamic Analysis Searching for SQLite Databases

2.1.7.5 Searching for Cache Databases

By default NSURLSession stores data, such as HTTP requests and responses in the Cache.db database. This database can contain sensitive data, if tokens, usernames or any other sensitive information has been cached. To find the cached information open the data directory of the app (/var/mobile/Containers/Data/Application/) and go to /Library/Caches/. The WebKit cache is also being stored in the Cache.db file. Objection can open and interact with the database with the command sqlite connect Cache.db, as it is a normal SQLite database.

It is recommended to disable Caching this data, as it may contain sensitive information in the request or response. The following list below shows different ways of achieving this:

1. It is recommended to remove Cached responses after logout. This can be done with the provided method by Apple called removeAllCachedResponses You can call this method as follows:

   ```
   URLCache.shared.removeAllCachedResponses()
   ```

   This method will remove all cached requests and responses from Cache.db file.

2. If you don't need to use the advantage of cookies it would be recommended to just use the .ephemeral configuration property of URLSession, which will disable saving cookies and Caches.

   Apple documentation :

   An ephemeral session configuration object is similar to a default session configuration (see default), except that the corresponding session object doesn't store caches, credential stores, or any session-related data to disk. Instead, session-related data is stored in RAM. The only time an ephemeral session writes data to disk is when you tell it to write the contents of a URL to a file.

3. Cache can be also disabled by setting the Cache Policy to .notAllowed. It will disable storing Cache in any fashion, either in memory or on disk.

Reference

- owasp-mastg Testing Local Data Storage (MSTG-STORAGE-1 and MSTG-STORAGE-2) Dynamic Analysis Searching for Cache Databases

Rulebook

- *Review cache specifications and data stored in Cache.db (Required)*

2.1.8 RuleBook

1. *Implement access control for user data stored in flash memory utilizing the iOS Data Protection API (Required)*
2. *Erase keychain data on first startup after installation (Required)*
3. *Clear keychain data at account logout (Required)*
4. *Securely store values using the Keychain Services API (Required)*
5. *Selecting a storage destination based on the amount of data (Recommended)*
6. *How to use encryption in the FileManager class (Required)*
7. *Review cache specifications and data stored in Cache.db (Required)*

2.1.8.1 Implement access control for user data stored in flash memory utilizing the iOS Data Protection API (Required)

The iOS Data Protection API is used for files with large data. iOS has a file protection mechanism called Data Protection. The following data protection attributes are defined in iOS.

- NSFileProtectionNone
- NSFileProtectionComplete
- NSFileProtectionCompleteUnlessOpen
- NSFileProtectionCompleteUntilFirstUserAuthentication

Which attribute is applied by default when creating a file depends on the framework and API. For example, in Data#write(to: options: error:), NSFileProtectionComplete is applied if not specified, but for Core Data database files persisted with NSPersistentStoreCoordinator NSFileProtectionCompleteUtilFirstUserAuthentication is applied.

The following is an example of Data class generation.

```
import UIKit

class GetDataBitSample {
    func getData() -> Data? {

        let text: String = "Hello."

        // Convert string to Data type.
        guard let data = text.data(using: .utf8) else {
            return nil
        }

        return data
    }
}
```

Protection Attribute Complete Protection（NSFileProtectionComplete）

Option to allow access to files only while the device is unlocked.

The system stores files in encrypted format and apps can only read or write files while the device is unlocked. Otherwise, apps will fail when attempting to read or write files.

The sample code below is an example of using the option "Protection Attribute Complete Protection".

```
import UIKit

class ViewController: UIViewController {

    func writeFileDataCompleteFileProtection(data: Data, fileURL: URL) {

        do {
            // Write the contents of the data buffer in the URL location.
            // Option to allow access to files only while the device is unlocked.
            try data.write(to: fileURL, options: .completeFileProtection)

        } catch {
            // exception write failed.
        }

    }
}
```

Protection Attribute Protected Unless Open（NSFileProtectionCompleteUnlessOpen）

Option to allow access to a file when the device is unlocked or the file is already open.

If the device is locked, the app cannot open and read/write files, but can create new files. If a file is open when the device is locked, the app can read and write to the open file.

The sample code below is an example of using the option "Protection Attribute Protected Unless Open".

```
import UIKit

class ViewController: UIViewController {

    func writeFileDataCompleteFileProtectionUnlessOpen(data: Data, fileURL: URL) {

        do {

            // Write the contents of the data buffer in the URL location.
            // Option to allow access to a file when the device is unlocked or␣
↪the file is already open.
            try data.write(to: fileURL, options: .completeFileProtectionUnlessOpen)

        } catch {
            // exception write failed.
        }

    }
}
```

Protection Attribute Protected Until First User Authentication（NSFileProtectionCompleteUntilFirstUserAuthentication）

Option to allow the user to access files after first unlocking the device.

While the device is unlocked, the app can read or write files, but files opened by the app will be in the same state of protection as Complete Protection.

The sample code below is an example of using the option "Protection Attribute Protected Until First User Authentication".

```
import UIKit

class ViewController: UIViewController {

    func writeFileDataCompleteFileProtectionUntilFirstUserAuthentication(data:␣
↪Data, fileURL: URL) {

        do {

            // Write the contents of the data buffer in the URL location.
            // Option to allow users to access files after first unlocking the␣
↪device
            try data.write(to: fileURL, options: .
↪completeFileProtectionUntilFirstUserAuthentication)

        } catch {
            // exception write failed.
        }

    }
}
```

Protection Attribute No Protection（NSFileProtectionNone）

An insecure (unprotected) option that creates the protection class key only with UID when writing out the file.

The system encrypts the file but is consequently insecure because the protection class key is created only with the UID, and the app always has access to this file.

The sample code below is an example of using the option "Protection Attribute No Protection".

```
import UIKit

class ViewController: UIViewController {

    func writeFileDataNoFileProtection(data: Data, fileURL: URL) {

        do {

            // Write the contents of the data buffer in the URL location.
            // Option to not encrypt files when writing them out.
            try data.write(to: fileURL, options: .noFileProtection)

        } catch {
            // exception write failed.
        }
    }
}
```

Setting the appropriate level of protection

Depending on the protection level of the file, it may not be possible to read or write the contents of the file if the user subsequently locks the device. To ensure that the app has access to the file, the following should be followed

- Assign a "Complete Protection" level to the file you are accessing only when the app is in the foreground. (Complete Protection)
- If the app supports background features such as location update processing, assign a different protection level to the file so that it can be accessed while in the background. (Protected Unless Open)

Files containing users' personal information and files created directly by users always require the highest level of protection.

Reference

- Encrypting Your App's Files

If this is violated, the following may occur.

- Unintended access to files may be implemented if the appropriate level of protection is not set.

2.1.8.2 Erase keychain data on first startup after installation (Required)

When the application is launched for the first time after installation, all Keychain data associated with the application is erased. This prevents the second user of the device from accidentally accessing the previous user's account.

The following Swift example is a basic demonstration of this erasure procedure.

```
let userDefaults = UserDefaults.standard

if userDefaults.bool(forKey: "hasRunBefore") == false {
    // Remove Keychain items here

    // Update the flag indicator
    userDefaults.set(true, forKey: "hasRunBefore")
    userDefaults.synchronize() // Forces the app to update UserDefaults
}
```

If this is violated, the following may occur.

- The second user of the device may accidentally access the previous user's account.

2.1.8.3 Clear keychain data at account logout (Required)

When developing a logout feature for an iOS application, ensure that Keychain data is erased as part of the account logout. This allows users to clear their accounts before uninstalling the application.

If this is violated, the following may occur.

- Keychain data may be used by another user.

2.1.8.4 Securely store values using the Keychain Services API (Required)

The Keychain Service API can store data in an encrypted database called Keychain. This is suitable for storing sensitive data (passwords, credit card information, encryption keys and certificates managed by Certificate, Key, and Trust Services, etc.).

Keychain items are encrypted using two different AES-256-GCM keys: a table key (metadata) and a per-row key (secret key). Keychain metadata (all attributes except kSecValue) is encrypted with the metadata key to increase search speed, and the secret value (kSecValueData) is encrypted with the corresponding secret key. The metadata key is protected by Secure Enclave, but is cached in the application processor for speed. The private key must always be exchanged through the Secure Enclave.

Hardware support is required to use Secure Enclave. (iOS devices with A7 or later processors)

Keychains are implemented in SQLite database format and stored on the file system. There is only one database, and the securityd daemon determines which Keychain items each process or app can access.

Sharing of Keychain items is only possible between apps by the same developer. To share Keychain items, third-party apps use access groups based on a prefix assigned through the Apple Developer Program for the application group. The prefix requirement and application group uniqueness is accomplished through code signing, provisioning profiles, and the Apple Developer Program.

Keychain data is protected by a class structure similar to that used in file data protection and behaves in the same manner as each class of file data protection.

When available	File Data Protection	Keychain Data Protection
When unlocked	NSFileProtectionComplete	kSecAttrAccessibleWhenUnlocked
Locked	NSFileProtectionCompleteUnlessOpen	not allowed
After initial unlocking	NSFileProtectionCompleteUntilFirstUserAuthentication	kSecAttrAccessibleAfterFirstUnlock
always on	NSFileProtectionNone	kSecAttrAccessibleAlways (This item is currently deprecated.)
When the passcode is valid	not allowed	kSecAttrAccessibleWhenPasscodeSetThisDeviceOnly

Apps using the background update service use kSecAttrAccessibleAfterFirstUnlock for keychain items that need to be accessed. SecAttrAccessibleWhenPasscodeSetThisDeviceOnly works the same as kSecAttrAccessibleWhenUnlocked, but is only available when a passcode is configured on the device.

The sample code below is an example of the process of retrieving data from keychain and saving it to keychain if it does not exist.

```
import Foundation

class Keychain {

    let group: String = "group_1" // group
    var id: String = "id"

    let backgroundQueue = DispatchQueue.global(qos: .userInitiated)

    func addKeychain(data: Data) {
```

(continues on next page)

(continued from previous page)

```
        // Argument settings when executing the API
        let dic: [String: Any] = [kSecClass as String: kSecClassGenericPassword, //
↪ Class:Password Class
            kSecAttrGeneric as String: group,                // optional item
            kSecAttrAccount as String: id,                  // Account (Login ID)
            kSecValueData as String: data,                 // Password and other␣
↪stored information
            kSecAttrService as String: "key"]             // service name

        // Dictionary for search
        let search: [String: Any] = [kSecClass as String: kSecClassGenericPassword,
                                     kSecAttrService as String: "key",
                                     kSecReturnAttributes as String:␣
↪kCFBooleanTrue as Any,
                                     kSecMatchLimit as String:␣
↪kSecMatchLimitOne]as [String : Any]

        // Search data from keychain
        findKeychainItem(attributes: search as CFDictionary, { status, item in
            switch status {
                case errSecItemNotFound: // No data exists in keychain
                    // Save to keychain
                    _ = SecItemAdd(dic as CFDictionary, nil)
                default:
                    break
            }
        })

    }

    /// Search for the existence of an item from Keychain
    /// - Parameters:
    ///   - attrs: Data for search
    ///   - completion: search results
    func findKeychainItem(attributes attrs: CFDictionary, _ completion: @escaping␣
↪(OSStatus, CFTypeRef?) -> Void) {

        // The application UI may hang when called from the main thread because␣
↪the calling thread is blocked.
        // Recommended to run in a separate thread.
        backgroundQueue.async {
            var item: CFTypeRef?
            let result = SecItemCopyMatching(attrs, &item)
            completion(result, item)
        }
    }

}
```

Save to keychain SecItemAdd

SecItemAdd adds one or more items to the Keychain. To add multiple items to the keychain at once, use a dictionary key with an array of dictionaries as values. (Only password items are excluded.)

As a performance consideration, the app UI may hang when called from the main thread because it blocks threads. Execution outside the main thread is recommended.

The sample code below is an example of the process of saving to keychain.

```
import Foundation

class KeyChainSample {
```

(continues on next page)

(continued from previous page)

```
    let queue = DispatchQueue(label: "queuename", attributes: .concurrent)

    func addKeychainItem(query: CFDictionary, _ completion: @escaping (OSStatus) ->
↪ Void) {
        queue.async {
            let result = SecItemAdd(query, nil)
            completion(result)
        }
    }
}
```

keychain updates SecItemUpdate

SecItemUpdate updates items matching the search query. Update the relevant item by passing a dictionary containing the attribute whose value is to be changed and the new value.

See SecItemCopyMatching function for information on how to create a search query.

As a performance consideration, the app UI may hang when called from the main thread because it blocks threads. Running off the main thread is recommended.

The sample code below is an example of a keychain update process.

```
import Foundation

class KeyChainSample {
    let queue = DispatchQueue(label: "queuename", attributes: .concurrent)

    func updateKeychainItem(key: String, loginId: String, data: String, update␣
↪updateAttrs: CFDictionary, _ completion: @escaping (OSStatus) -> Void) {
        queue.async {
            let attrs: CFDictionary =
            [kSecClass as String: kSecClassGenericPassword,
             kSecAttrGeneric as String: key,           // Free items (group)
             kSecAttrAccount as String: loginId,       // Account (e.g. login ID)
             kSecValueData as String: data            // Preserved Information
            ] as CFDictionary

            // Matching execution of keychain
            let matchingStatus = SecItemCopyMatching(attrs, nil)

            // Status of Matching Results
            switch matchingStatus {
            case errSecSuccess: // Search Success
                // update
                let result = SecItemUpdate(attrs, updateAttrs)
                completion(result)
            default:
                debugPrint("failed updateKeychainItem status. (\(matchingStatus)")
                completion(matchingStatus)
            }

        }
    }
}
```

keychain data search SecItemCopyMatching

SecItemCopyMatching returns one or more Keychain items matching the search query or copies the attributes of a particular Keychain item.

By default, this function searches for items in the Keychain. (Same as specifying kSecReturnData) To limit the Keychain search to a specific Keychain or multiple Keychains, specify a search key and pass a dictionary with objects

containing items of item type as their values.

Change the designation of how search results are returned

- kSecReturnData : Returns the item's data in object format (1 item). (Default)
- kSecReturnAttributes : Returns the item's data in dictionary format (multiple items).
- kSecReturnRef : Returns a reference to the item.
- kSecReturnPersistentRef : Returns a reference to an item. Unlike normal references, persistent references return references that are stored on disk or used between processes.

As a performance consideration, the app UI may hang when called from the main thread because it blocks threads. Execution outside the main thread is recommended.

The sample code below is an example of keychain data retrieval process.

```
import Foundation

class KeyChainSample {
    let backgroundQueue = DispatchQueue(label: "queuename", attributes: .
↪concurrent)

    func findKeychainItem(loginId: String,_ completion: @escaping (OSStatus,␣
↪CFTypeRef?) -> Void) {

        let query : CFDictionary = [
                kSecClass as String       : kSecClassGenericPassword,
                kSecAttrAccount as String : loginId,
                kSecReturnData as String  : true,  // default value␣
↪(kSecReturnData: true)
                kSecMatchLimit as String  : kSecMatchLimitOne ] as CFDictionary

        backgroundQueue.async {
            var item: CFTypeRef?
            let result = SecItemCopyMatching(query, &item)
            completion(result, item)
        }
    }
}
```

keychain data deletion SecItemDelete

SecItemDelete deletes items matching the search query.

As a performance consideration, the app UI may hang when called from the main thread because it blocks threads. Execution outside the main thread is recommended.

The sample code below is an example of keychain data deletion process.

```
import Foundation

class KeyChainSample {
    let queue = DispatchQueue(label: "queuename", attributes: .concurrent)

    private func deleteKeychainItem(searchAttributes attrs: CFDictionary, _␣
↪completion: @escaping (OSStatus) -> Void) {
        queue.async {
            let result = SecItemDelete(attrs)
            completion(result)
        }
    }
}
```

Setting conditions for access to keychain

It is possible to set when the app can access the data of Keychain items.

By default, Keychain items can only be accessed when the device is unlocked. However, to account for devices that do not have passcodes set, it may be necessary to allow access to items only from devices protected by passcodes. Alternatively, access restrictions can be relaxed to allow items to be accessed from background processes when the device is locked.

The Keychain service provides a way to manage the accessibility of individual Keychain items according to the state of the device in combination with input from the user.

Control your app' s access to Keychain items related to device state by setting the item' s attributes when the item is created. In terms of setting accessibility values, a query dictionary that uses the default accessibility should specify the following

The sample code below is an example of how to specify a query dictionary that uses default accessibility.

```
import Foundation

class KeyChainSample {

    func querySample() {

        let account = "test string"
        let server = "test string"
        let password = "test string"

        // Configure KeyChain Item
        var query: [String: Any] = [kSecClass as String: kSecClassInternetPassword,
                                    kSecAttrAccount as String: account,
                                    kSecAttrServer as String: server,
                                    kSecAttrAccessible as String:␣
↪kSecAttrAccessibleWhenUnlocked, // accessible
                                    kSecValueData as String: password]

        SecItemAdd(query as CFDictionary, nil)

    }
}
```

Possible values for accessibility are as follows

kSecAttrAccessibleAlways

Data on Keychain items is always accessible, regardless of whether the device is locked or not. Using an encrypted backup will migrate items with this attribute to the new device. Note that this item is currently deprecated.

kSecAttrAccessibleAlwaysThisDeviceOnly

Keychain item data is always accessible, regardless of whether the device is locked or not. Items with this attribute will not be migrated to a new device. Therefore, these items will no longer exist when restoring from a backup on another device. Note that this item is currently deprecated.

kSecAttrAccessibleAfterFirstUnlock

Data on Keychain items cannot be accessed after a reboot until the user has unlocked the device once. After the first unlock, the data can be accessed until the next reboot. This is recommended if the item needs to be accessed from a background application. Using an encrypted backup will migrate items with this attribute to the new device.

kSecAttrAccessibleAfterFirstUnlockThisDeviceOnly

Data on Keychain items cannot be accessed after a reboot until the user has unlocked the device once. After the first unlock, the data can be accessed until the next reboot. This is recommended if the item needs to be accessed from a background application. Items with this attribute are not migrated to the new device. Therefore, these items will no longer be present when restoring from a backup on another device.

kSecAttrAccessibleWhenUnlocked

Data on Keychain items can only be accessed while the user is unlocking the device. This is recommended for items that need to be accessible only while the application is in the foreground. Using an encrypted backup, items with this attribute will be migrated to the new device.

This is the default value for Keychain items added without explicitly setting an accessibility constant.

kSecAttrAccessibleWhenUnlockedThisDeviceOnly

Data on Keychain items can only be accessed while the user is unlocking the device. This is recommended for items that need to be accessible only while the application is in the foreground. Items with this attribute will not be migrated to a new device. Therefore, in a restore from a backup on another device, these items will no longer exist.

kSecAttrAccessibleWhenPasscodeSetThisDeviceOnly

Data in the Keychain can only be accessed if the device is unlocked. It can only be used if a passcode has been set on the device. This is recommended for items that need to be accessible only while the application is in the foreground. Items with this attribute will not be migrated to a new device. If a backup is restored to a new device, these items will be lost. Devices without a passcode cannot store items in this class. Disabling the device passcode will delete all items in this class.

Always use the most restrictive option appropriate for your app so that the system can protect that item as much as possible. For very sensitive data that you do not want to store in iCloud, use kSecAttrAccessibleWhenPasscodeSet-ThisDeviceOnly.

In addition, the following rules must be followed when using Keychain.

- When using Keychain, the AccessControlFlags setting must follow the security policy for the keys in the Keychain.
- When using Keychain, the recommended accessibility value to set is kSecAtrAccessibilityWhenPasscodeSet-ThisDeviceOnly

Accessibility restricted access by device state, but allowing access only when the device is unlocked may not be secure enough in all cases. If the app can control the bank account directly, it should again verify the authorized user just before retrieving login credentials from the Keychain. This will protect the account even if the user passes the device to someone else in an unlocked state.

This restriction can be added by specifying an instance (SecAccessControlCreateFlags) as the value of the attribute when creating the keychain item. Access control is used in that case.

The sample code below is an example of how to specify access control flags in the access control.

```
import Foundation

class KeyChainSample {

    func flagsSample() {
        // flags (kSecAccessControlBiometryAny |␣
↪kSecAccessControlApplicationPassword)
        // (using OptionSet, a Swift bit set type)
        let flags: SecAccessControlCreateFlags = [.biometryAny, .
↪applicationPassword]
        let access = SecAccessControlCreateWithFlags(nil,
                                                     ␣
↪kSecAttrAccessibleWhenPasscodeSetThisDeviceOnly,
                                                      flags, //Can also be␣
↪specified directly as ".biometryAny"
                                                     nil)
    }
}
```

The sample code below is an example of how to specify a query dictionary using access control.

```
import UIKit
```

(continues on next page)

(continued from previous page)

```
class KeyChainSample {
    func keySample(tag: String) {
        let account = "test string"
        let server = "test string"
        let password = "test string"

        // Ccreate SecAccessControl Flags
        guard let access = SecAccessControlCreateWithFlags(nil,  // Use the␣
↪default allocator.
                                                            ␣
↪kSecAttrAccessibleWhenUnlocked, // accessible
                                                             .userPresence,
                                                             nil
        ) else {
            // Abnormal handling
            return
        }

        // kSecAttrAccessible and kSecAttrAccessControl cannot coexist. Either␣
↪one should be specified.
         var query: [String: Any] = [kSecClass as String:␣
↪kSecClassInternetPassword,
                                     kSecAttrAccount as String: account,
                                     kSecAttrServer as String: server,
                                     kSecAttrAccessControl as String: access,
                                     kSecValueData as String: password]

        SecItemAdd(query as CFDictionary, nil)
    }
}
```

The options that can be set for access control are as follows

kSecAccessControlDevicePasscode

Constraints for accessing items with a passcode.

kSecAccessControlBiometryAny

Constraints for accessing items with Touch ID or Face ID of the registered finger. Touch ID must be available and registered with at least one finger. Or Face ID must be available and registered. Items can continue to be accessed with Touch ID if a finger is added or removed, or with Face ID if the user is re-registered.

kSecAccessControlBiometryCurrentSet

Constraints to access items using Touch ID on the currently registered finger or from Face ID of the currently registered user. Touch ID must be available and registered with at least one finger, or Face ID must be available and registered. If a finger is added or removed from Touch ID, or if the user re-enrolls in Face ID, the item is disabled.

kSecAccessControlUserPresence

Constraints on access to items using either biometry or passcodes. Biometry need not be available or registered. Items can be accessed by Touch ID even if a finger is added or removed, or by Face ID if the user is re-enrolled.

Furthermore, in addition to requesting specific device status and user presence, an application-specific password can be requested. This is different from a passcode that unlocks the device and thus can explicitly protect a specific Keychain item.

To do this, include the flag when defining the access control flags in the access control.

The sample code below is an example of how to include flags and specify query dictionary when defining access control flags in access control.

```
import UIKit

class KeyChainSample {
    func keySample(tag: String) {
        let account = "test string"
        let server = "test string"
        let password = "test string"

        // Ccreate SecAccessControl Flags

        // flags (kSecAccessControlBiometryAny |␣
↪kSecAccessControlApplicationPassword)
        // (Using OptionSet, a Swift bit-set type)
        let flags: SecAccessControlCreateFlags = [.biometryAny, .
↪applicationPassword]
        guard let access = SecAccessControlCreateWithFlags(nil,
                                                           ␣
↪kSecAttrAccessibleWhenPasscodeSetThisDeviceOnly,
                                                            flags,
                                                            nil
        ) else {
            //Abnormal handling
            return
        }
        // kSecAttrAccessible and kSecAttrAccessControl cannot coexist. Specify␣
↪either.
         var query: [String: Any] = [kSecClass as String:␣
↪kSecClassInternetPassword,
                                     kSecAttrAccount as String: account,
                                     kSecAttrServer as String: server,
                                     kSecAttrAccessControl as String: access,
                                     kSecValueData as String: password]

        SecItemAdd(query as CFDictionary, nil)
    }
}
```

Adding this flag will prompt the user for a password when the item is created and again before the item can be retrieved. Items can only be retrieved if the user successfully enters the password, regardless of whether other conditions are met.

In addition, when multiple access controls are set, it is possible to add a condition that requires all constraints to be passed (kSecAccessControlAnd) or a condition that can be satisfied by passing one of the constraints (kSecAccess-ControlOr). Add to the access control flags by bitwise operation.

The sample code below is an example of how to specify bits for access control.

```
import Foundation

class KeyChainTest {

    func flagsTest() {
        // Access control bit specification (using OptionSet, a Swift bit set␣
↪type)
        let flags: SecAccessControlCreateFlags = [.biometryAny, .devicePasscode, .
↪or]

        let access = SecAccessControlCreateWithFlags(nil,
                                                     ␣
↪kSecAttrAccessibleWhenUnlocked,
                                                     flags,
                                                     nil) // Ignore any error.
```

(continues on next page)

(continued from previous page)

```
    }
}
```

kSecAccessControlApplicationPassword

This may be specified in addition to any constraints.

kSecAccessControlPrivateKeyUsage

This option can be combined with any other access control flags.

You typically use this constraint when you create a key pair and store the private key inside a device' s Secure Enclave (by specifying the kSecAttrTokenID attribute with a value of kSecAttrTokenIDSecureEnclave). This makes the private key available for use in signing and verification tasks that happen inside the Secure Enclave with calls to the SecKeyRawSign and SecKeyRawVerify functions. An attempt to use this constraint while generating a key pair outside the Secure Enclave fails. Similarly, an attempt to sign a block with a private key generated without this constraint inside the Secure Enclave fails.

Specify the algorithm for creating your own encryption key with kSecAttrKeyType

An encryption key is a string of bytes combined with other data in special mathematical operations to enhance security. In most cases, the key is obtained from an ID, certificate, or Keychain. However, you may need to create your own key.

Creating Asymmetric Key Pairs

An asymmetric encryption key pair consists of a public key and a private key that are generated together. The public key can be freely distributed, while the private key is kept private. The private key generated can be stored in Keychain.

The sample code below is an example of how to create an asymmetric key pair by creating an attribute dictionary.

```
import UIKit

class KeyChainSample {

    func keySample(tag: String) {
        let tag = "com.example.keys.mykey".data(using: .utf8)!
        let attributes: [String: Any] = [kSecAttrKeyType as String:␣
↪kSecAttrKeyTypeRSA, // RSA algorithm specification
                                         kSecAttrKeySizeInBits as String: 2048,   ␣
↪  // 2048bit
                                         kSecPrivateKeyAttrs as String: [
                                            kSecAttrIsPermanent as String:true,
                                            kSecAttrApplicationTag as String: tag]]

        // Private Key Creation
        var error: Unmanaged<CFError>?
        guard let privateKey = SecKeyCreateRandomKey(attributes as CFDictionary, &
↪error) else {
            // error handling
            return
        }

        // Public Key Creation
        let generatedPublicKey = SecKeyCopyPublicKey(privateKey)

    }
}
```

The above example shows a 2048-bit RSA key, but other options are available. (The service will decide which algorithm to specify accordingly)

Algorithm Key Type Value

- kSecAttrKeyTypeDSA
  - DSA algorithm.
- kSecAttrKeyTypeAES
  - AES algorithm.
- kSecAttrKeyTypeCAST
  - CAST algorithm.
- kSecAttrKeyTypeECSECPrimeRandom
  - Elliptic curve algorithm.

The tag data specified in the attribute dictionary is created from a string using reverse DNS notation, but can be any unique tag. Care should be taken not to generate multiple keys with the same tag. Unless they differ in other searchable characteristics, it will be impossible to distinguish them during the search. Instead, a unique tag should be used for each key generation operation, or an old key using a particular tag should be deleted and a new key created using that tag.

To store a private key in Keychain, create a private key with kSecAttrIsPermanent set to true.

The sample code below is an example of how to create a secret key by specifying true for kSecAttrIsPermanent.

```
import UIKit

class KeyChainSample {
    func keySample(tag: String) {
        let attributes: [String: Any] = [kSecAttrKeyType as String:␣
↪kSecAttrKeyTypeRSA,
                                         kSecAttrKeySizeInBits as String: 2048,
                                         kSecPrivateKeyAttrs as String: [
                                             kSecAttrIsPermanent as String:␣
↪true, // Flag to store privateKey in Keychain
                                         kSecAttrApplicationTag as String:␣
↪tag]]


    }
}
```

Objective-C is responsible for freeing the associated memory once these key references are complete.

The sample code below is an example of a key-related memory release process.

```
#import <Foundation/Foundation.h>
#import <Security/Security.h>

@interface MySecurity : NSObject  {}

- (void)execute;
@end


@implementation MySecurity {}

-(void)execute {

    //a tag to read/write keychain storage
    NSString *tag = @"my_pubKey";
    NSData *tagData = [NSData dataWithBytes:[tag UTF8String] length:[tag length]];

    //kSecClassKey
    NSMutableDictionary *publicKey = [[NSMutableDictionary alloc] init];
```

(continues on next page)

(continued from previous page)

```
    [publicKey setObject:(__bridge id) kSecClassKey forKey:(__bridge id)kSecClass];
    [publicKey setObject:(__bridge id) kSecAttrKeyTypeRSA forKey:(__bridge␣
↪id)kSecAttrKeyType];
    [publicKey setObject:tagData forKey:(__bridge id)kSecAttrApplicationTag];
    SecItemDelete((__bridge CFDictionaryRef)publicKey);

    // Add persistent version of the key to system keychain
    NSData *data = [[NSData alloc] initWithBase64EncodedString:@"sssssssss"␣
↪options:NSDataBase64DecodingIgnoreUnknownCharacters];
    [publicKey setObject:data forKey:(__bridge id)kSecValueData];
    [publicKey setObject:(__bridge id) kSecAttrKeyClassPublic forKey:(__bridge id)
     kSecAttrKeyClass];
    [publicKey setObject:[NSNumber numberWithBool:YES] forKey:(__bridge id)
     kSecReturnPersistentRef];

    CFTypeRef persistKey = nil;
    OSStatus status = SecItemAdd((__bridge CFDictionaryRef)publicKey, &persistKey);



    // C pointer type CFTypeRef release
    if (persistKey != nil){
        CFRelease(persistKey);
    }


    if(status != noErr){
        return;
    }


}

@end
```

Reference

- KeychainServices
- Keychain Data Protection
- Secure Enclave
- SecItemAdd
- SecItemUpdate
- SecItemCopyMatching
- SecItemDelete
- Restricting Keychain Item Accessibility
- Generate a new encryption key
- Saving Keys to Keychain

If this is violated, the following may occur.

- Confidential data may be compromised.

2.1.8.5 Selecting a storage destination based on the amount of data (Recommended)

When storing data, it is important to design the storage location according to the amount of data. For example, data stored by NSUserDefaults can be viewed in the application bundle. This class stores data in plist files, but is intended for small amounts of data.

In iOS 13, a limit of 4194304 bytes was added to the size that can be saved in NSUserDefaults, and a warning was issued when saving more than that (there is no official notice from Apple). Depending on the future OS version, it may not be possible to save, so it is recommended to change the save location according to the amount of data to be used.

For large volumes

- server
- Local database

for small quantities

- NSUserDefaults

If this is not noted, the following may occur.

- If the save destination is not properly designed, it may not be possible to save and retrieve data.

2.1.8.6 How to use encryption in the FileManager class (Required)

By specifying the protection attribute when creating a file with the FileManager class, the file can be saved to disk in an encrypted format. It is necessary to set appropriate protection attributes. The following sample code is an example of specifying the protection attribute complete.

Swift:

```
FileManager.default.createFile(
    atPath: filePath,
    contents: "secret text".data(using: .utf8),
    attributes: [FileAttributeKey.protectionKey: FileProtectionType.complete]
)
```

Objective-C:

```
[[NSFileManager defaultManager] createFileAtPath:[self filePath]
  contents:[@"secret text" dataUsingEncoding:NSUTF8StringEncoding]
  attributes:[NSDictionary dictionaryWithObject:NSFileProtectionComplete
  forKey:NSFileProtectionKey]];
```

See the rulebook below for protected attributes.

- *Implement access control for user data stored in flash memory utilizing the iOS Data Protection API (Required)*

If this is violated, the following may occur.

- Unencrypted data is easily revealed when files are leaked.

2.1.8.7 Review cache specifications and data stored in Cache.db (Required)

When the iOS application communicates with the web server, if cache control is not performed on the server side, the response returned from the web server is saved as a cache in the Cache.db file. Therefore, confidential information will be saved if cache control is not implemented in iOS applications. Files stored as cache in Cache.db remain in plaintext, so if a cache containing sensitive information is stored, an attacker can steal the cache.

There are several ways to do cache control. Appropriate cache control is required.

1. If you are using caching, it is recommended to remove Cached responses after logout. This can be done with the provided method by Apple called removeAllCachedResponses. Note that this method will remove all cached requests and responses from Cache.db file.

2. If you don't need to use the advantage of cookies it would be recommended to just use the .ephemeral configuration property of URLSession, which will disable saving cookies and Caches.
3. Cache can be also disabled by setting the Cache Policy to .notAllowed. It will disable storing Cache in any fashion, either in memory or on disk.

If this is violated, the following may occur.

- If you do not have appropriate cache control, your personal information may be leaked and abused.

2.2 MSTG-STORAGE-2

No sensitive data should be stored outside of the app container or system credential storage facilities.

* The same contents are described in *MSTG-STORAGE-1*, so the description in this chapter is omitted.

2.3 MSTG-STORAGE-3

No sensitive data is written to application logs.

2.3.1 Log Outputs

There are many legitimate reasons for creating log files on a mobile device, including keeping track of crashes or errors that are stored locally while the device is offline (so that they can be sent to the app's developer once online), and storing usage statistics. However, logging sensitive data, such as credit card numbers and session information, may expose the data to attackers or malicious applications. Log files can be created in several ways. The following list shows the methods available on iOS:

- NSLog Method
- printf-like function
- NSAssert-like function
- Macro

Use the following keywords to check the app's source code for predefined and custom logging statements:

- For predefined and built-in functions:
  - NSLog
  - NSAssert
  - NSCAssert
  - fprintf
- For custom functions:
  - Logging
  - Logfile

Rulebook

- *Prevent sensitive data from being exposed via application logs (Required)*

2.3.2 Debug log disabled by define

A generalized approach to this issue is to use a define to enable NSLog statements for development and debugging, then disable them before shipping the software. You can do this by adding the following code to the appropriate PREFIX_HEADER (*.pch) file:

```
#ifdef DEBUG
#   define NSLog (...) NSLog(__VA_ARGS__)
#else
#   define NSLog (...)
#endif
```

Reference

- owasp-mastg Checking Logs for Sensitive Data (MSTG-STORAGE-3)
- owasp-mastg Checking Logs for Sensitive Data (MSTG-STORAGE-3) Static Analysis

Rulebook

- *Prevent sensitive data from being exposed via application logs (Required)*

2.3.3 Dynamic Analysis

In the section "Monitoring System Logs" of the chapter "iOS Basic Security Testing" various methods for checking the device logs are explained. Navigate to a screen that displays input fields that take sensitive user information.

After starting one of the methods, fill in the input fields. If sensitive data is displayed in the output, the app fails this test.

Reference

- owasp-mastg Checking Logs for Sensitive Data (MSTG-STORAGE-3) Dynamic Analysis

2.3.4 RuleBook

1. *Prevent sensitive data from being exposed via application logs (Required)*

2.3.4.1 Prevent sensitive data from being exposed via application logs (Required)

When outputting logs, it is necessary to confirm that confidential information is not included in the output contents.

Following are the general class for log output.

log

- print
- NSLog

print A function that writes the textual representation of the item to standard output. Display on the console in Xcode when Xcode (Run) is executed.

```
import Foundation

func examplePrint(message: String) {

    print(message)
}
```

NSLog A function that logs an error message to the Apple system logging facility. Display on the console in Xcode when Xcode (Run) is executed. Output log messages written with NSLog to Console.app as well.

```
import Foundation

func exampleLog(message: String) {

    NSLog(message)
}
```

In addition, defines can be used to enable NSLog statements during development and debugging, but they must be disabled before the software can be shipped. You can do this by adding the following code to the appropriate PREFIX_HEADER (*.pch) file:

```
#ifdef DEBUG
#   define NSLog (...) NSLog(__VA_ARGS__)
#else
#   define NSLog (...)
#endif
```

If this is violated, the following may occur.

- Logging sensitive data may expose the data to attackers or malicious applications.

2.4 MSTG-STORAGE-4

Sensitive information can be leaked to third parties through several means. in iOS, this is usually through third-party services built into the app.

2.4.1 Sharing data to third-party services

Sensitive information might be leaked to third parties by several means. On iOS typically via third-party services embedded in the app.

The downside is that developers don't usually know the details of the code executed via third-party libraries. Consequently, no more information than is necessary should be sent to a service, and no sensitive information should be disclosed.

Most third-party services are implemented in two ways:

- with a standalone library
- with a full SDK

Static Analysis To determine whether API calls and functions provided by the third-party library are used according to best practices, review their source code, requested permissions and check for any known vulnerabilities (see *Third Party Library*).

All data that's sent to third-party services should be anonymized to prevent exposure of PII (Personal Identifiable Information) that would allow the third party to identify the user account. No other data (such as IDs that can be mapped to a user account or session) should be sent to a third party.

Dynamic Analysis Check all requests to external services for embedded sensitive information. To intercept traffic between the client and server, you can perform dynamic analysis by launching a man-in-the-middle (MITM) attack with Burp Suite Professional or OWASP ZAP. Once you route the traffic through the interception proxy, you can try to sniff the traffic that passes between the app and server. All app requests that aren't sent directly to the server on which the main function is hosted should be checked for sensitive information, such as PII in a tracker or ad service.

Reference

- owasp-mastg Determining Whether Sensitive Data Is Shared with Third Parties (MSTG-STORAGE-4)

Rulebook

- *Do not share unnecessarily confidential information to third-party libraries (Required)*

- *Check for known vulnerabilities in the third-party libraries you use (Required)*
- *Anonymize all data sent to third-party services (Required)*

2.4.2 RuleBook

1. *Do not share unnecessarily confidential information to third-party libraries (Required)*
2. *Check for known vulnerabilities in the third-party libraries you use (Required)*
3. *Anonymize all data sent to third-party services (Required)*

2.4.2.1 Do not share unnecessarily confidential information to third-party libraries (Required)

The downside is that developers don't usually know the details of the code executed via third-party libraries. Consequently, no more information than is necessary should be sent to a service, and no sensitive information should be disclosed.

Check all requests to external services for embedded sensitive information. To intercept traffic between the client and server, you can perform dynamic analysis by launching a man-in-the-middle (MITM) attack with Burp Suite Professional or OWASP ZAP.

If this is violated, the following may occur.

- May jeopardize intellectual property (IP).

2.4.2.2 Check for known vulnerabilities in the third-party libraries you use (Required)

Third-party libraries may contain vulnerabilities, incompatible licenses, or malicious content.

In order to ensure that the libraries used by the apps are not carrying vulnerabilities, one can best check the dependencies installed by CocoaPods or Carthage.

- Carthage is open source and can be used for Swift and Objective-C packages. It is written in Swift, decentralized and uses the Cartfile file to document and manage project dependencies.
- CocoaPods is open source and can be used for Swift and Objective-C packages. It is written in Ruby, utilizes a centralized package registry for public and private packages and uses the Podfile file to document and manage project dependencies.

If this is violated, the following may occur.

- The application contains malicious or vulnerable code that can be exploited.
- Licenses included in third-party libraries may require deployment of the app's source code.

2.4.2.3 Anonymize all data sent to third-party services (Required)

All data that's sent to third-party services should be anonymized to prevent exposure of PII (Personal Identifiable Information) that would allow the third party to identify the user account.

Personally identifiable information (PII) is any data that can be used to identify an individual. Any information directly or indirectly linked to an individual is considered PII. Names, e-mail addresses, telephone numbers, bank account numbers, and government-issued identification numbers are all examples of PII.

If this is violated, the following may occur.

- They may be victims of identity theft or other attacks.

2.5 MSTG-STORAGE-5

The keyboard cache is disabled on text inputs that process sensitive data.

2.5.1 Keyboard Predictive Conversion Input

Several options for simplifying keyboard input are available to users. These options include autocorrection and spell checking. Most keyboard input is cached by default, in /private/var/mobile/Library/Keyboard/dynamic-text.dat.

The UITextInputTraits protocal is used for keyboard caching. The UITextField, UITextView, and UISearchBar classes automatically support this protocol and it offers the following properties:

- var autocorrectionType: UITextAutocorrectionType determines whether autocorrection is enabled during typing. When autocorrection is enabled, the text object tracks unknown words and suggests suitable replacements, replacing the typed text automatically unless the user overrides the replacement. The default value of this property is UITextAutocorrectionTypeDefault, which for most input methods enables autocorrection.
- var secureTextEntry: BOOL determines whether text copying and text caching are disabled and hides the text being entered for UITextField. The default value of this property is NO.

Static Analysis

- Search through the source code for similar implementations, such as

```
  textObject.autocorrectionType = UITextAutocorrectionTypeNo;
  textObject.secureTextEntry = YES;
```

- Open xib and storyboard files in the Interface Builder of Xcode and verify the states of Secure Text Entry and Correction in the Attributes Inspector for the appropriate object. The application must prevent the caching of sensitive information entered into text fields. You can prevent caching by disabling it programmatically, using the textObject.autocorrectionType = UITextAutocorrectionTypeNo directive in the desired UITextFields, UITextViews, and UISearchBars. For data that should be masked, such as PINs and passwords, set textObject.secureTextEntry to YES.

```
UITextField *textField = [ [ UITextField alloc ] initWithFrame: frame ];
textField.autocorrectionType = UITextAutocorrectionTypeNo;
```

Dynamic Analysis

If a jailbroken iPhone is available, execute the following steps:

1. Reset your iOS device keyboard cache by navigating to Settings > General > Reset > Reset Keyboard Dictionary.
2. Use the application and identify the functionalities that allow users to enter sensitive data.
3. Dump the keyboard cache file dynamic-text.dat into the following directory (which might be different for iOS versions before 8.0): /private/var/mobile/Library/Keyboard/
4. Look for sensitive data, such as username, passwords, email addresses, and credit card numbers. If the sensitive data can be obtained via the keyboard cache file, the app fails this test.

```
UITextField *textField = [ [ UITextField alloc ] initWithFrame: frame ];
textField.autocorrectionType = UITextAutocorrectionTypeNo;
```

If you must use a non-jailbroken iPhone:

1. Reset the keyboard cache.
2. Key in all sensitive data.
3. Use the app again and determine whether autocorrect suggests previously entered sensitive information.

Reference

- owasp-mastg Finding Sensitive Data in the Keyboard Cache (MSTG-STORAGE-5)

Rulebook

- *Do not cache sensitive information entered in text fields (Required)*

2.5.2 RuleBook

1. *Do not cache sensitive information entered in text fields (Required)*

2.5.2.1 Do not cache sensitive information entered in text fields (Required)

The UITextInputTraits protocal is used for keyboard caching. The UITextField, UITextView, and UISearchBar classes automatically support this protocol and it offers the following properties:

- var autocorrectionType: UITextAutocorrectionType determines whether autocorrection is enabled during typing. When autocorrection is enabled, the text object tracks unknown words and suggests suitable replacements, replacing the typed text automatically unless the user overrides the replacement. The default value of this property is UITextAutocorrectionTypeDefault, which for most input methods enables autocorrection.
- var secureTextEntry: BOOL determines whether text copying and text caching are disabled and hides the text being entered for UITextField. The default value of this property is NO.
- Open xib and storyboard files in the Interface Builder of Xcode and verify the states of Secure Text Entry and Correction in the Attributes Inspector for the appropriate object. The application must prevent the caching of sensitive information entered into text fields. You can prevent caching by disabling it programmatically, using the textObject.autocorrectionType = UITextAutocorrectionTypeNo directive in the desired UITextFields, UITextViews, and UISearchBars. For data that should be masked, such as PINs and passwords, set textObject.secureTextEntry to YES.

```
UITextField *textField = [ [ UITextField alloc ] initWithFrame: frame ];
textField.autocorrectionType = UITextAutocorrectionTypeNo;
```

If this is violated, the following may occur.

- There is a risk of confidential information being stolen and misused.

2.6 MSTG-STORAGE-6

No sensitive data is exposed via IPC mechanisms.

2.6.1 XPC Services

XPC is a structured, asynchronous library that provides basic interprocess communication. It is managed by launchd. It is the most secure and flexible implementation of IPC on iOS and should be the preferred method. It runs in the most restricted environment possible: sandboxed with no root privilege escalation and minimal file system access and network access. Two different APIs are used with XPC Services:

- NSXPCConnection API
- XPC Services API

Static Analysis

The following section summarizes keywords that you should look for to identify IPC implementations within iOS source code.

XPC Services

Several classes may be used to implement the NSXPCConnection API:

- NSXPCConnection
- NSXPCInterface
- NSXPCListener
- NSXPCListenerEndpoint

You can set security attributes for the connection. The attributes should be verified.

Check for the following two files in the Xcode project for the XPC Services API (which is C-based):

- xpc.h
- connection.h

Reference

- owasp-mastg Determining Whether Sensitive Data Is Exposed via IPC Mechanisms (MSTG-STORAGE-6)
- owasp-mastg Determining Whether Sensitive Data Is Exposed via IPC Mechanisms (MSTG-STORAGE-6) XPC Services

2.6.2 Mach Ports

All IPC communication ultimately relies on the Mach Kernel API. Mach Ports allow local communication (intra-device communication) only. They can be implemented either natively or via Core Foundation (CFMachPort) and Foundation (NSMachPort) wrappers.

Static Analysis

The following section summarizes keywords that you should look for to identify IPC implementations within iOS source code.

Several classes may be used to implement the NSXPCConnection API:

- mach_port_t
- mach_msg_*

Keywords to look for in high-level implementations (Core Foundation and Foundation wrappers):Keywords to look for in high-level implementations (Core Foundation and Foundation wrappers):

- CFMachPort
- CFMessagePort
- NSMachPort
- NSMessagePort

Reference

- owasp-mastg Determining Whether Sensitive Data Is Exposed via IPC Mechanisms (MSTG-STORAGE-6)
- owasp-mastg Determining Whether Sensitive Data Is Exposed via IPC Mechanisms (MSTG-STORAGE-6) Mach Ports

2.6.3 NSFileCoordinator

The class NSFileCoordinator can be used to manage and send data to and from apps via files that are available on the local file system to various processes. NSFileCoordinator methods run synchronously, so your code will be blocked until they stop executing. That's convenient because you don't have to wait for an asynchronous block callback, but it also means that the methods block the running thread.

Static Analysis

The following section summarizes keywords that you should look for to identify IPC implementations within iOS source code.

Keywords to look for:

- NSFileCoordinator

Reference

- owasp-mastg Determining Whether Sensitive Data Is Exposed via IPC Mechanisms (MSTG-STORAGE-6)
- owasp-mastg Determining Whether Sensitive Data Is Exposed via IPC Mechanisms (MSTG-STORAGE-6) NSFileCoordinator

2.7 MSTG-STORAGE-7

No sensitive data, such as passwords or pins, is exposed through the user interface.

2.7.1 Exposure of sensitive data in the user interface

Entering sensitive information when, for example, registering an account or making payments, is an essential part of using many apps. This data may be financial information such as credit card data or user account passwords. The data may be exposed if the app doesn't properly mask it while it is being typed.

In order to prevent disclosure and mitigate risks such as shoulder surfing you should verify that no sensitive data is exposed via the user interface unless explicitly required (e.g. a password being entered). For the data required to be present it should be properly masked, typically by showing asterisks or dots instead of clear text.

Carefully review all UI components that either show such information or take it as input. Search for any traces of sensitive information and evaluate if it should be masked or completely removed.

Reference

- owasp-mastg Checking for Sensitive Data Disclosed Through the User Interface (MSTG-STORAGE-7)

2.7.2 input text

A text field that masks its input can be configured in two ways:

Storyboard

In the iOS project's storyboard, navigate to the configuration options for the text field that takes sensitive data. Make sure that the option "Secure Text Entry" is selected. If this option is activated, dots are shown in the text field in place of the text input.

Source Code

Source Code If the text field is defined in the source code, make sure that the option isSecureTextEntry is set to "true". This option obscures the text input by showing dots.

```
sensitiveTextField.isSecureTextEntry = true
```

To determine whether the application leaks any sensitive information to the user interface, run the application and identify components that either show such information or take it as input.

If the information is masked by, for example, asterisks or dots, the app isn't leaking data to the user interface.

Reference

- owasp-mastg Checking for Sensitive Data Disclosed Through the User Interface (MSTG-STORAGE-7) Static Analysis
- owasp-mastg Checking for Sensitive Data Disclosed Through the User Interface (MSTG-STORAGE-7) Dynamic Analysis

RuleBook

- *Mask password entry (Required)*

2.7.3 RuleBook

1. *Mask password entry (Required)*

2.7.3.1 Mask password entry (Required)

In the iOS UITextField, there is a property Secure Text Entry that hides (masks) text for input of confidential information and prohibits copying of text objects.

There are two ways to set the property: by checking the checkbox in the GUI on the Storyboard or by setting the property in the source code.

Storyboard

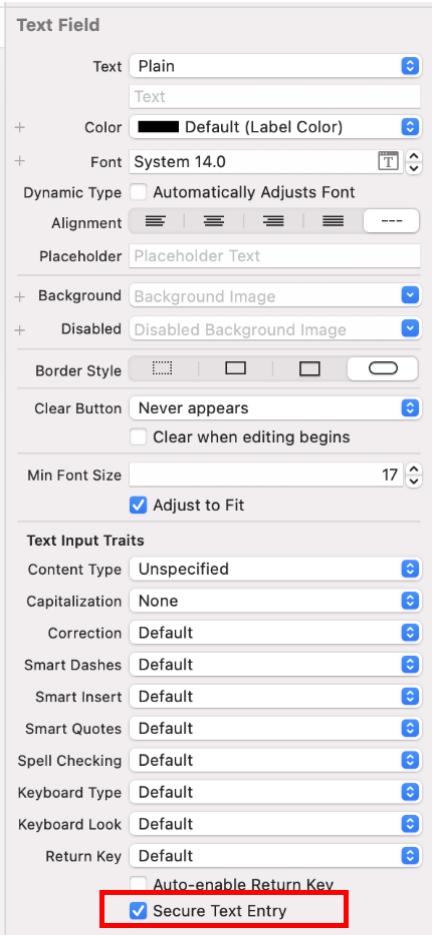


Fig 2.7.3.1.1 Storyboard Secure Text Entry Settings

SourceCode

```
import UIKit

class ViewController: UIViewController {

    @IBOutlet weak var passwordTextFiled: UITextField!

    override func viewDidLoad() {
        super.viewDidLoad()

        passwordTextFiled.isSecureTextEntry = true

    }

}
```

If this is violated, the following may occur.

- Third parties will be able to read confidential information.

2.8 MSTG-STORAGE-12

The app educates the user about the types of personally identifiable information processed, as well as security best practices the user should follow in using the app.

2.8.1 Testing User Education on Data Privacy on the App Marketplace

At this point, we're only interested in knowing which privacy-related information is being disclosed by the developers and trying to evaluate if it seems reasonable (similarly as you'd do when testing for permissions).

It's possible that the developers are not declaring certain information that is indeed being collected and or shared, but that's a topic for a different test extending this one here. As part of this test, you are not supposed to provide privacy violation assurance.

Reference

- owasp-mastg Testing User Education (MSTG-STORAGE-12) Testing User Education on Data Privacy on the App Marketplace

2.8.2 Static Analysis

You can follow these steps:

1. Search for the app in the corresponding app marketplace (e.g. Google Play, App Store).
2. Go to the section "Privacy Details" (App Store) or "Safety Section" (Google Play).
3. Verify if there's any information available at all.

The test passes if the developer has compiled with the app marketplace guidelines and included the required labels and explanations. Store and provide the information you got from the app marketplace as evidence, so that you can later use it to evaluate potential violations of privacy or data protection.

Reference

- owasp-mastg Testing User Education (MSTG-STORAGE-12) Static Analysis

2.8.3 Dynamic analysis

As an optional step, you can also provide some kind of evidence as part of this test. For instance, if you're testing an iOS app you can easily enable app activity recording and export a Privacy Report containing detailed app access to different resources such as photos, contacts, camera, microphone, network connections, etc.

Doing this has actually many advantages for testing other MASVS categories. It provides very useful information that you can use to test network communication in MASVS-NETWORK or when testing app permissions in MASVS-PLATFORM. While testing these other categories you might have taken similar measurements using other testing tools. You can also provide this as evidence for this test.

Ideally, the information available should be compared against what the app is actually meant to do. However, that's far from a trivial task that could take from several days to weeks to complete depending on your resources and support from automated tooling. It also heavily depends on the app functionality and context and should be ideally performed on a white box setup working very closely with the app developers.

Reference

- owasp-mastg Testing User Education (MSTG-STORAGE-12) Dynamic analysis

2.8.4 Testing User Education on Security Best Practices

Testing this might be especially challenging if you intend to automate it. We recommend using the app extensively and try to answer the following questions whenever applicable:

- Fingerprint usage: when fingerprints are used for authentication providing access to high-risk transactions/information,

  does the app inform the user about potential issues when having multiple fingerprints of other people registered to the device as well?

- Rooting/Jailbreaking: when root or jailbreak detection is implemented,

  does the app inform the user of the fact that certain high-risk actions will carry additional risk due to the jailbroken/rooted status of the device?

- Specific credentials: when a user gets a recovery code, a password or a pin from the application (or sets one),

  does the app instruct the user to never share this with anyone else and that only the app will request it?

- Application distribution: in case of a high-risk application and in order to prevent users from downloading compromised versions of the application,

  does the app manufacturer properly communicate the official way of distributing the app (e.g. from Google Play or the App Store)?

- Prominent Disclosure: in any case,

  does the app display prominent disclosure of data access, collection, use, and sharing? e.g. does the app use the App Tracking Transparency Framework to ask for the permission on iOS?

Reference

- owasp-mastg Testing User Education (MSTG-STORAGE-12) Testing User Education on Security Best Practices

RuleBook

- *Use the app extensively to answer questions about security best practices (Recommended)*

2.8.5 RuleBook

1. *Use the app extensively to answer questions about security best practices (Recommended)*

2.8.5.1 Use the app extensively to answer questions about security best practices (Recommended)

It is recommended that the app be used extensively to answer the following questions regarding security best practices

- Use of fingerprints : Fingerprints are used for authentication to provide access to high-risk transactions/information. Does the application inform the user about possible problems that may occur when the device has multiple fingerprints of other people on it?
- Rooting /Jailbreak : Rooting or Jailbreak detection is implemented. Does the application inform the user of the fact that certain high-risk actions carry additional risk due to the Jailbreak/root status of the device?
- Specific credentials : user obtains (or sets) recovery codes, passwords, and pins from the application. If you obtain (or set) a recovery code, password, or PIN from the application, does the application instruct you to never share this with other users and only request it from the application?
- Application distribution : For high-risk applications, to prevent users from downloading unsafe versions of the application. Does the application manufacturer adequately communicate the official method of distributing the application (e.g., Google Play, App Store, etc.)?

- Prominent Disclosure : All cases. Does the application prominently disclose access, collection, use, and sharing of data? For example, does the application use the App Tracking Transparency Framework to ask for permission on iOS?

If this is not noted, the following may occur.

- Confidential information is used in an unintended process.
- Third parties will be able to read confidential information.

3 Cryptography Requirements

3.1 MSTG-CRYPTO-1

The app does not rely on symmetric cryptography with hardcoded keys as a sole method of encryption.

3.1.1 Problematic Encryption Configuration

3.1.1.1 Insufficient Key Length

Even the most secure encryption algorithm becomes vulnerable to brute-force attacks when that algorithm uses an insufficient key size.

Ensure that the key length fulfills accepted industry standards. In Japan, check the List of Cipher Specifications on the "e-Government Recommended Ciphers List" .

Reference

- owasp-mastg Common Configuration Issues (MSTG-CRYPTO-1, MSTG-CRYPTO-2 and MSTG-CRYPTO-3) Insufficient Key Length

Rulebook

- *Set key lengths that meet industry standards (Required)*

3.1.1.2 Symmetric Encryption with Hard-Coded Cryptographic Keys

The security of symmetric encryption and keyed hashes (MACs) depends on the secrecy of the key. If the key is disclosed, the security gained by encryption is lost. To prevent this, never store secret keys in the same place as the encrypted data they helped create. A common mistake is encrypting locally stored data with a static, hardcoded encryption key and compiling that key into the app. This makes the key accessible to anyone who can use a disassembler.

Hardcoded encryption key means that a key is:

- part of application resources
- value which can be derived from known values
- hardcoded in code

First, ensure that no keys or passwords are stored within the source code. This means you should check Objective-C/Swift in iOS. Note that hard-coded keys are problematic even if the source code is obfuscated since obfuscation is easily bypassed by dynamic instrumentation.

If the app is using two-way TLS (both server and client certificates are validated), make sure that:

- The password to the client certificate isn't stored locally or is locked in the device Keychain.
- The client certificate isn't shared among all installations.

If the app relies on an additional encrypted container stored in app data, check how the encryption key is used. If a key-wrapping scheme is used, ensure that the master secret is initialized for each user or the container is re-encrypted with new key. If you can use the master secret or previous password to decrypt the container, check how password changes are handled.

Secret keys must be stored in secure device storage whenever symmetric cryptography is used in mobile apps. For more information on the platform-specific APIs, see the "Data Storage in iOS" chapters.

Reference

- owasp-mastg Common Configuration Issues (MSTG-CRYPTO-1, MSTG-CRYPTO-2 and MSTG-CRYPTO-3) Symmetric Encryption with Hard-Coded Cryptographic Keys

Rulebook

- *Do not store keys or passwords in source code (Required)*
- *Do not store client certificate passwords locally. Lock passwords in your device's keychain if you want to save them (Required)*
- *Client certificates are not shared among all installations (Required)*
- *If container dependent, verify how encryption keys are used (Required)*
- *Store private keys in secure device storage whenever symmetric encryption is used in mobile apps (Required)*

3.1.1.3 Weak Key Generation Functions

Cryptographic algorithms (such as symmetric encryption or some MACs) expect a secret input of a given size. For example, AES uses a key of exactly 16 bytes. A native implementation might use the user-supplied password directly as an input key. Using a user-supplied password as an input key has the following problems:

- If the password is smaller than the key, the full key space isn't used. The remaining space is padded (spaces are sometimes used for padding).
- A user-supplied password will realistically consist mostly of displayable and pronounceable characters. Therefore, only some of the possible 256 ASCII characters are used and entropy is decreased by approximately a factor of four.

Ensure that passwords aren't directly passed into an encryption function. Instead, the user-supplied password should be passed into a KDF to create a cryptographic key. Choose an appropriate iteration count when using password derivation functions. For example, NIST recommends an iteration count of at least 10,000 for PBKDF2 and for critical keys where user-perceived performance is not critical at least 10,000,000. For critical keys, it is recommended to consider implementation of algorithms recognized by Password Hashing Competition (PHC) like Argon2.

Reference

- owasp-mastg Common Configuration Issues (MSTG-CRYPTO-1, MSTG-CRYPTO-2 and MSTG-CRYPTO-3) Weak Key Generation Functions

Rulebook

- *If using an encryption algorithm (such as symmetric encryption or some MACs), use a secret input of the specific size assumed (Required)*
- *User-supplied passwords are passed to KDF to create encryption keys (Required)*
- *If using a password derivation function, select the appropriate number of iterations (Required)*

3.1.1.4 Weak Random Number Generators

It is fundamentally impossible to produce truly random numbers on any deterministic device. Pseudo-random number generators (RNG) compensate for this by producing a stream of pseudo-random numbers - a stream of numbers that appear as if they were randomly generated. The quality of the generated numbers varies with the type of algorithm used. Cryptographically secure RNGs generate random numbers that pass statistical randomness tests, and are resilient against prediction attacks (e.g. it is statistically infeasible to predict the next number produced).

Mobile SDKs offer standard implementations of RNG algorithms that produce numbers with sufficient artificial randomness. We'll introduce the available APIs in the iOS specific sections.

Reference

- owasp-mastg Common Configuration Issues (MSTG-CRYPTO-1, MSTG-CRYPTO-2 and MSTG-CRYPTO-3) Weak Random Number Generators

Rulebook

- *Identify a standard implementation of the RNG algorithm that generates numbers with sufficient artificial randomness (Required)*

3.1.1.5 Custom Implementations of Cryptography

Inventing proprietary cryptographic functions is time consuming, difficult, and likely to fail. Instead, we can use well-known algorithms that are widely regarded as secure. Mobile operating systems offer standard cryptographic APIs that implement those algorithms.

Carefully inspect all the cryptographic methods used within the source code, especially those that are directly applied to sensitive data. All cryptographic operations should use standard cryptographic APIs for iOS (we'll write about those in more detail in the platform-specific chapters). Any cryptographic operations that don't invoke standard routines from known providers should be closely inspected. Pay close attention to standard algorithms that have been modified. Remember that encoding isn't the same as encryption! Always investigate further when you find bit manipulation operators like XOR (exclusive OR).

At all implementations of cryptography, you need to ensure that the following always takes place:

- Worker keys (like intermediary/derived keys in AES/DES/Rijndael) are properly removed from memory after consumption or in case of error.
- The inner state of a cipher should be removed from memory as soon as possible.

Reference

- owasp-mastg Common Configuration Issues (MSTG-CRYPTO-1, MSTG-CRYPTO-2 and MSTG-CRYPTO-3) Custom Implementations of Cryptography

Rulebook

- *All cryptography-related implementations properly manage memory state (Required)*
- *Use industry-standard cryptographic APIs provided by the OS (Required)*

3.1.1.6 Incorrect AES Configuration

Advanced Encryption Standard (AES) is the widely accepted standard for symmetric encryption in mobile apps. It's an iterative block cipher that is based on a series of linked mathematical operations. AES performs a variable number of rounds on the input, each of which involve substitution and permutation of the bytes in the input block. Each round uses a 128-bit round key which is derived from the original AES key.

As of this writing, no efficient cryptanalytic attacks against AES have been discovered. However, implementation details and configurable parameters such as the block cipher mode leave some margin for error.

Weak Block Cipher Mode

Block-based encryption is performed upon discrete input blocks (for example, AES has 128-bit blocks). If the plaintext is larger than the block size, the plaintext is internally split up into blocks of the given input size and encryption is

performed on each block. A block cipher mode of operation (or block mode) determines if the result of encrypting the previous block impacts subsequent blocks.

ECB (Electronic Codebook) divides the input into fixed-size blocks that are encrypted separately using the same key. If multiple divided blocks contain the same plaintext, they will be encrypted into identical ciphertext blocks which makes patterns in data easier to identify. In some situations, an attacker might also be able to replay the encrypted data.

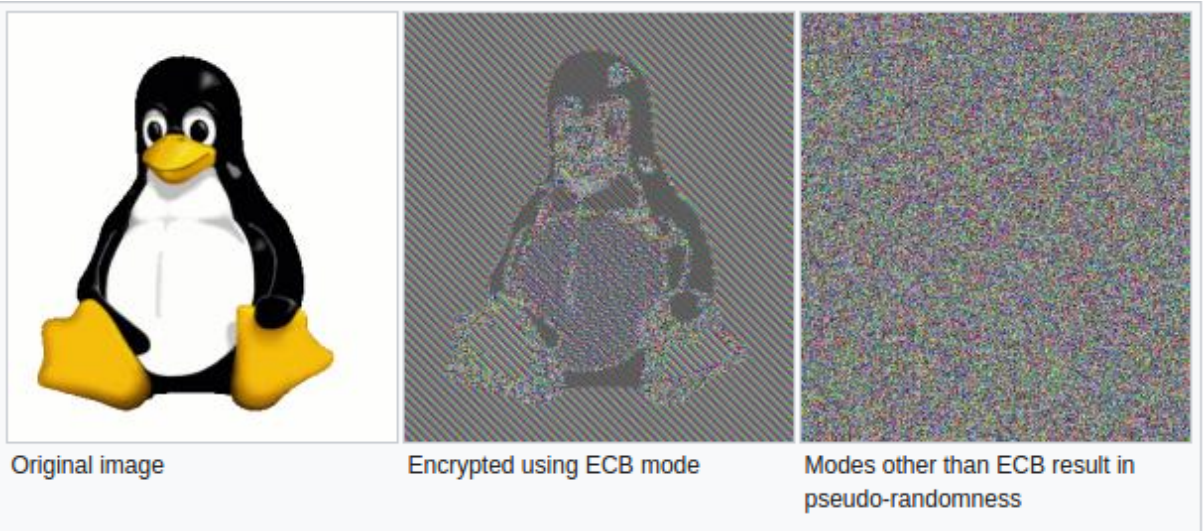


Fig 3.1.1.6.1 ECB Encryption Example

Verify that Cipher Block Chaining (CBC) mode is used instead of ECB. In CBC mode, plaintext blocks are XORed with the previous ciphertext block. This ensures that each encrypted block is unique and randomized even if blocks contain the same information. Please note that it is best to combine CBC with an HMAC and/or ensure that no errors are given such as "Padding error", "MAC error", "decryption failed" in order to be more resistant to a padding oracle attack.

When storing encrypted data, we recommend using a block mode that also protects the integrity of the stored data, such as Galois/Counter Mode (GCM). The latter has the additional benefit that the algorithm is mandatory for each TLSv1.2 implementation, and thus is available on all modern platforms.

For more information on effective block modes, see the NIST guidelines on block mode selection.

Predictable Initialization Vector

CBC, OFB, CFB, PCBC, GCM mode require an initialization vector (IV) as an initial input to the cipher. The IV doesn't have to be kept secret, but it shouldn't be predictable: it should be random and unique/non-repeatable for each encrypted message. Make sure that IVs are generated using a cryptographically secure random number generator. For more information on IVs, see Crypto Fail's initialization vectors article.

Pay attention to cryptographic libraries used in the code: many open source libraries provide examples in their documentations that might follow bad practices (e.g. using a hardcoded IV). A popular mistake is copy-pasting example code without changing the IV value.

Initialization Vectors in stateful operation modes

Please note that the usage of IVs is different when using CTR and GCM mode in which the initialization vector is often a counter (in CTR combined with a nonce). So here using a predictable IV with its own stateful model is exactly what is needed. In CTR you have a new nonce plus counter as an input to every new block operation. For example: for a 5120 bit long plaintext: you have 20 blocks, so you need 20 input vectors consisting of a nonce and counter. Whereas in GCM you have a single IV per cryptographic operation, which should not be repeated with the same key. See section 8 of the documentation from NIST on GCM for more details and recommendations of the IV.

Reference

- owasp-mastg Common Configuration Issues (MSTG-CRYPTO-1, MSTG-CRYPTO-2 and MSTG-CRYPTO-3) Inadequate AES Configuration
- owasp-mastg Common Configuration Issues (MSTG-CRYPTO-1, MSTG-CRYPTO-2 and MSTG-CRYPTO-3) Weak Block Cipher Mode
- owasp-mastg Common Configuration Issues (MSTG-CRYPTO-1, MSTG-CRYPTO-2 and MSTG-CRYPTO-3) Predictable Initialization Vector
- owasp-mastg Common Configuration Issues (MSTG-CRYPTO-1, MSTG-CRYPTO-2 and MSTG-CRYPTO-3) Initialization Vectors in stateful operation modes

Rulebook

- *To counter padding oracle attacks, CBC should not be combined with HMAC or generate errors such as padding errors, MAC errors, decryption failures, etc. (Required)*
- *When storing encrypted data, use a block mode such as Galois/Counter Mode (GCM) that also protects the integrity of the stored data (Recommended)*
- *IV is generated using a cryptographically secure random number generator (Required)*
- *Note that IV is used differently when using CTR and GCM modes, where the initialization vector is often a counter (Required)*

3.1.1.7 Padding Oracle Attacks due to Weaker Padding or Block Operation Implementations

In the old days, PKCS1.5 padding (in code: PKCS1Padding) was used as a padding mechanism when doing asymmetric encryption. This mechanism is vulnerable to the padding oracle attack. Therefore, it is best to use OAEP (Optimal Asymmetric Encryption Padding) captured in PKCS#1 v2.0 (in code: OAEPPadding, OAEPwithSHA-256andMGF1Padding, OAEPwithSHA-224andMGF1Padding, OAEPwithSHA-384andMGF1Padding, OAEPwithSHA-512andMGF1Padding). Note that, even when using OAEP, you can still run into an issue known best as the Mangers attack as described in the blog at Kudelskisecurity.

Note: AES-CBC with PKCS #5 has shown to be vulnerable to padding oracle attacks as well, given that the implementation gives warnings, such as "Padding error" , "MAC error" , or "decryption failed" . See The Padding Oracle Attack and The CBC Padding Oracle Problem for an example. Next, it is best to ensure that you add an HMAC after you encrypt the plaintext: after all a ciphertext with a failing MAC will not have to be decrypted and can be discarded.

Reference

- owasp-mastg Common Configuration Issues (MSTG-CRYPTO-1, MSTG-CRYPTO-2 and MSTG-CRYPTO-3) Padding Oracle Attacks due to Weaker Padding or Block Operation Implementations

Rulebook

- *Use OAEP incorporated in PKCS#1 v2.0 as a padding mechanism for asymmetric encryption (Required)*

3.1.1.8 Protecting Keys in Storage and in Memory

When memory dumping is part of your threat model, then keys can be accessed the moment they are actively used. Memory dumping either requires root-access (e.g. a rooted device or jailbroken device) or it requires a patched application with Frida (so you can use tools like Fridump). Therefore it is best to consider the following, if keys are still needed at the device:

- Keys in a Remote Server: you can use remote Key vaults such as Amazon KMS or Azure Key Vault. For some use cases, developing an orchestration layer between the app and the remote resource might be a suitable option. For instance, a serverless function running on a Function as a Service (FaaS) system (e.g. AWS Lambda or Google Cloud Functions) which forwards requests to retrieve an API key or secret. There are other alternatives such as Amazon Cognito, Google Identity Platform or Azure Active Directory.

- Keys inside Secure Hardware-backed Storage: make sure that all cryptographic actions and the key itself remain in the Secure Enclave (e.g. use the Keychain). Refer to the iOS Data Storage chapters for more information.
- Keys protected by Envelope Encryption: If keys are stored outside of the TEE / SE, consider using multi-layered encryption: an envelope encryption approach (see OWASP Cryptographic Storage Cheat Sheet, Google Cloud Key management guide, AWS Well-Architected Framework guide), or a HPKE approach to encrypt data encryption keys with key encryption keys.
- Keys in Memory: make sure that keys live in memory for the shortest time possible and consider zeroing out and nullifying keys after successful cryptographic operations, and in case of error. For general cryptocoding guidelines, refer to Clean memory of secret data. For more detailed information refer to sections Testing Memory for Sensitive Data respectively.

Note: given the ease of memory dumping, never share the same key among accounts and/or devices, other than public keys used for signature verification or encryption.

Reference

- owasp-mastg Common Configuration Issues (MSTG-CRYPTO-1, MSTG-CRYPTO-2 and MSTG-CRYPTO-3) Protecting Keys in Storage and in Memory

Rulebook

- *Use key considering memory dump (Required)*
- *Do not share the same key across accounts or devices (Required)*

3.1.1.9 Key handling during transfer

When keys need to be transported from one device to another, or from the app to a backend, make sure that proper key protection is in place, by means of a transport keypair or another mechanism. Often, keys are shared with obfuscation methods which can be easily reversed. Instead, make sure asymmetric cryptography or wrapping keys are used. For example, a symmetric key can be encrypted with the public key from an asymmetric key pair.

Reference

- owasp-mastg Common Configuration Issues (MSTG-CRYPTO-1, MSTG-CRYPTO-2 and MSTG-CRYPTO-3) Protecting Keys in Transport

Rulebook

- *Appropriate key protection by means of transport symmetric keys or other mechanisms (Required)*

3.1.2 Rulebook

1. *Set key lengths that meet industry standards (Required)*
2. *Do not store keys or passwords in source code (Required)*
3. *Do not store client certificate passwords locally. Lock passwords in your device' s keychain if you want to save them (Required)*
4. *Client certificates are not shared among all installations (Required)*
5. *If container dependent, verify how encryption keys are used (Required)*
6. *Store private keys in secure device storage whenever symmetric encryption is used in mobile apps (Required)*
7. *If using an encryption algorithm (such as symmetric encryption or some MACs), use a secret input of the specific size assumed (Required)*
8. *User-supplied passwords are passed to KDF to create encryption keys (Required)*
9. *If using a password derivation function, select the appropriate number of iterations (Required)*

10. *Identify a standard implementation of the RNG algorithm that generates numbers with sufficient artificial randomness (Required)*
11. *All cryptography-related implementations properly manage memory state (Required)*
12. *Use industry-standard cryptographic APIs provided by the OS (Required)*
13. *To counter padding oracle attacks, CBC should not be combined with HMAC or generate errors such as padding errors, MAC errors, decryption failures, etc. (Required)*
14. *When storing encrypted data, use a block mode such as Galois/Counter Mode (GCM) that also protects the integrity of the stored data (Recommended)*
15. *IV is generated using a cryptographically secure random number generator (Required)*
16. *Note that IV is used differently when using CTR and GCM modes, where the initialization vector is often a counter (Required)*
17. *Use OAEP incorporated in PKCS#1 v2.0 as a padding mechanism for asymmetric encryption (Required)*
18. *Use key considering memory dump (Required)*
19. *Do not share the same key across accounts or devices (Required)*
20. *Appropriate key protection by means of transport symmetric keys or other mechanisms (Required)*

3.1.2.1 Set key lengths that meet industry standards (Required)

Ensure that the key length fulfills accepted industry standards. In Japan, check the List of Cipher Specifications on the "e-Government Recommended Ciphers List" . Even the most secure encryption algorithms are vulnerable to brute force attacks if insufficient key sizes are used.

* No sample code due to conceptual rule.

If this is violated, the following may occur.

- Become vulnerable to brute force attacks.

3.1.2.2 Do not store keys or passwords in source code (Required)

Since obfuscation is easily bypassed by dynamic instrumentation, hardcoded keys are problematic even if the source code is obfuscated. Therefore, do not store keys or passwords within the source code (Objective-C/Swift code).

* No sample code due to deprecated rules.

If this is violated, the following may occur.

- Obfuscation is bypassed by dynamic instrumentation and keys and passwords are compromised.

3.1.2.3 Do not store client certificate passwords locally. Lock passwords in your device' s keychain if you want to save them (Required)

If the app uses bi-directional TLS (both server and client certificates are verified), do not store the client certificate password locally. Or lock it in the device' s Keychain.

See the rulebook below for sample code.

Rulebook

- *Securely store values using the Keychain Services API (Required)*

If this is violated, the following may occur.

- The password is read and abused by other applications.

3.1.2.4 Client certificates are not shared among all installations (Required)

If the app uses bi-directional TLS (both server and client certificates are verified), client certificates are not shared among all installations.

* No sample code due to deprecated rules.

If this is violated, the following may occur.

- The client is spoofed by an attacker.

3.1.2.5 If container dependent, verify how encryption keys are used (Required)

If the app relies on an encrypted container stored within the app' s data, identify how the encryption key is used.

When using the key wrap method

Confirm the following

- The master secret of each user must be initialized
- That the container is re-encrypted with the new key

If the container can be decrypted using the master secret or a previous password

Check how password changes are handled.

* No sample code due to conceptual rules.

If this is violated, the following may occur.

- The password or master secret is used for purposes other than those for which it was intended.

3.1.2.6 Store private keys in secure device storage whenever symmetric encryption is used in mobile apps (Required)

Whenever symmetric encryption is used in a mobile app, the private key must be stored in secure device storage. See "*Data Protection API*" for information on how to store private keys on the iOS platform.

Rulebook

- *Implement access control for user data stored in flash memory utilizing the iOS Data Protection API (Required)*

If this is violated, the following may occur.

- The private key is read by another application or third party.

3.1.2.7 If using an encryption algorithm (such as symmetric encryption or some MACs), use a secret input of the specific size assumed (Required)

When using encryption algorithms (such as symmetric encryption and some MACs), it is necessary to use a secret input of the specific size expected. For example, AES uses a key of exactly 16 bytes.

Native implementations may use user-supplied passwords directly as input keys. When using user-supplied passwords as input keys, the following problems exist.

- If the password is smaller than the key, the full key space is not used. The remaining spaces are padded (sometimes spaces are used for padding).
- User-supplied passwords, in reality, consist mostly of characters that can be displayed and pronounced. Thus, only a fraction of the 256 ASCII characters are used, reducing entropy by about a factor of four.

* No sample code due to conceptual rules.

If this is violated, the following may occur.

- A vulnerable key is generated.

3.1.2.8 User-supplied passwords are passed to KDF to create encryption keys (Required)

If the encryption function is used,User-supplied passwords should be passed to KDF to create the encryption key. The password should not be passed directly to the encryption function. Instead, the user-supplied password should be passed to the KDF to create the encryption key. When using the password derivation function, select an appropriate number of iterations.

* No sample code due to conceptual rules.

If this is violated, the following may occur.

- If the password is smaller than the key, the full keyspace is not used. The remaining space is padded.
- Entropy is reduced by about a factor of four.

3.1.2.9 If using a password derivation function, select the appropriate number of iterations (Required)

When using a password derivation function, an appropriate number of iterations should be selected. For example, NIST recommends an iteration count of at least 10,000 for PBKDF2 and for critical keys where user-perceived performance is not critical at least 10,000,000. For critical keys, it is recommended to consider implementation of algorithms recognized by Password Hashing Competition (PHC) like Argon2.

* No sample code because of server-side rules.

If this is violated, the following may occur.

- A vulnerable key is generated.

3.1.2.10 Identify a standard implementation of the RNG algorithm that generates numbers with sufficient artificial randomness (Required)

Cryptographically secure RNGs generate random numbers that pass statistical randomness tests and are resistant to predictive attacks. Using random numbers generated by an RNG algorithm that does not meet the safe level increases the likelihood of a successful prediction attack. Therefore, it is necessary to privately use an RNG algorithm that generates numbers with sufficient artificial randomness.

Refer to the following for APIs that generate highly secure random numbers in the iOS standard.

Rulebook

- *Generate safe random numbers using Randomization Services API (Recommended)*

If this is violated, the following may occur.

- Increased likelihood of a successful predictive attack.

3.1.2.11 All cryptography-related implementations properly manage memory state (Required)

All cryptographic implementations require that worker keys (like intermediate/derived keys in AES/DES/Rijndael) be properly removed from memory after consumption or in the event of an error. The internal state of the cipher also needs to be removed from memory as soon as possible.

AES implementation and post-execution release :

```
import Foundation
import CryptoSwift

class CryptoAES {
    var aes: AES? = nil

    // cryptographic process
    func encrypt(key: String, iv:String, text:String) -> String {
```

(continues on next page)

(continued from previous page)

```
        do {
            // Execute at the end of function
            defer {
                aes = nil
            }
            // AES Instantiation
            aes = try AES(key: key, iv: iv)
            guard let encrypt = try aes?.encrypt(Array(text.utf8)) else {
                return ""
            }


            // Data Type Conversion
            let data = Data( encrypt )
            // base64 conversion
            let base64Data = data.base64EncodedData()
            // UTF-8 conversion nil Not possible
            guard let base64String =
                String(data: base64Data as Data, encoding: String.Encoding.utf8)␣
↪else {
                    return ""
                }
            // base64 character string
            return base64String

        } catch {
            return ""
        }
    }

    // composite process
    func decrypt(key: String, iv:String, base64:String) -> String {

        do {
            // Execute at the end of function
            defer {
                aes = nil
            }
            // AES Instantiation
            aes = try AES(key: key, iv:iv)

            // From base64 to Data type
            let byteData = base64.data(using: String.Encoding.utf8)! as Data
            // base64 daecode
            guard let data = Data(base64Encoded: byteData) else {
                return ""
            }

            // Creating a UInt8 array
            let aBuffer = Array<UInt8>(data)
            // AES Composite
            guard let decrypted = try aes?.decrypt(aBuffer) else {
                return ""
            }
            // UTF-8 conversion
            guard let text = String(data: Data(decrypted), encoding: .utf8)else {
                return ""
            }

            return text
        } catch {
```

(continues on next page)

(continued from previous page)

```
            return ""
        }
    }
}
```

Example of CryptoSwift Pods Usage :

```
target 'MyApp' do
  use_frameworks!
  # Add the CryptoSwift library
  pod 'CryptoSwift'
end
```

If this is violated, the following may occur.

- Encrypted information left in memory is used in an unintended process.

3.1.2.12 Use industry-standard cryptographic APIs provided by the OS (Required)

Developing one' s own cryptographic functions is time consuming, difficult, and likely to fail. Instead, well-known algorithms that are widely recognized as secure can be used. Mobile operating systems provide standard cryptographic APIs that implement these algorithms and should be used for secure encryption.

See the rulebook below for sample code.

Rulebook

- *Implementation of the iOS encryption algorithm Apple CryptoKit (Recommended)*

If this is violated, the following may occur.

- May result in an implementation containing vulnerabilities.

3.1.2.13 To counter padding oracle attacks, CBC should not be combined with HMAC or generate errors such as padding errors, MAC errors, decryption failures, etc. (Required)

In CBC mode, the plaintext block is XORed with the immediately preceding ciphertext block. This ensures that each encrypted block is unique and random, even if the blocks contain the same information.

Example of CBC mode implementation:

```
import Foundation
import CryptoSwift

class CryptoCBC {
    // Example of plain text to be encrypted
    let cleartext = "Hello CryptoSwift"

    // cryptographic process
    func encrypt( text:String) -> (String, String) {
        // Appropriate 256-bit length key (string)
        let key = "BDC171111B7285F67F035497EE9A081D"

        // encode
        let byteText = text.data(using: .utf8)!.bytes
        let byteKey = key.data(using: .utf8)!.bytes

        // IV (initialization vector), type iv is [UInt8].
        let iv = AES.randomIV(AES.blockSize)
        do {
            // Creation of AES-256-CBC instance
```

(continues on next page)

(continued from previous page)

```
            let aes = try AES(key: byteKey, blockMode: CBC(iv: iv))

            // Encrypt plain text
            // Default padding is PKC7
            let encrypted = try aes.encrypt(byteText)

            // Output IV, encrypted Encode to Base64 string
            let strIV = NSData(bytes: iv, length: iv.count).
↪base64EncodedString(options: .lineLength64Characters)
            print("IV: " + strIV)  // output -> IV: lBMiK2GWEwrPgNdGfrJEig==
            let strEnc = NSData(bytes: encrypted, length: encrypted.count).
↪base64EncodedString(options: .lineLength64Characters)
            print("Encrypted: " + strEnc)  // output -> Encrypted: MHf5ZeUL/
↪gjviiZitpZKJFuqppdTgEe+IklDgg3N1fQ=
            return (strIV, strEnc)
        } catch {
          print("Error")
        }
        return ("", "")
    }
}
```

If this is violated, the following may occur.

- Become vulnerable to padding oracle attacks.

3.1.2.14 When storing encrypted data, use a block mode such as Galois/Counter Mode (GCM) that also protects the integrity of the stored data (Recommended)

When storing encrypted data, it is recommended to use a block mode that also protects the integrity of the stored data, such as Galois/Counter Mode (GCM). The latter has the advantage that this algorithm is mandatory for each TLSv1.2 implementation and can therefore be used on all modern platforms.

Example of GCM mode implementation:

```
import Foundation
import CryptoKit

class CryptoGCM {

    // Key Generation
    let symmetricKey: SymmetricKey = SymmetricKey(size: .bits256)
    /// encryption
    /// - Parameter data: Data to be encrypted
    func encrypt(data: Data) -> Data? {
        do {
            // GCM encrypt
            let sealedBox = try AES.GCM.seal(data, using: symmetricKey)
            guard let data = sealedBox.combined else {
                return nil
            }
            return data
        } catch _ {
            return nil
        }
    }

    /// decoding
    /// - Parameter data: Data to be decrypted
    private func decrypt(data: Data) -> Data? {
        do {
```

(continues on next page)

(continued from previous page)

```
            // GCM decrypt
            let sealedBox = try AES.GCM.SealedBox(combined: data)
            return try AES.GCM.open(sealedBox, using: symmetricKey)
        } catch _ {
            return nil
        }
    }
}
```

If this is violated, the following may occur.

- Easily identifiable patterns in the data.

3.1.2.15 IV is generated using a cryptographically secure random number generator (Required)

In CBC, OFB, CFB, PCBC, and GCM modes, the initialization vector (IV) is required as the initial input to the cipher. The IV need not be secret, but it must not be predictable. It must be random, unique, and non-reproducible for each encrypted message. Therefore, the IV must be generated using a cryptographically secure random number generator. For more information on IVs, see Crypto Fail' s article on initialization vectors.

See the rulebook below for sample code.

Rulebook

- *Generate safe random numbers using Randomization Services API (Recommended)*

If this is violated, the following may occur.

- A predictable initialization vector is generated.

3.1.2.16 Note that IV is used differently when using CTR and GCM modes, where the initialization vector is often a counter (Required)

Note that IVs are used differently when using CTR and GCM modes, where the initialization vector is often a counter (a combination of CTR and nonce). Therefore, it is necessary to use a predictable IV with its own stateful model. The CTR uses a new nonce and counter as input for each new block operation. Example : In the case of a plaintext of 5120 bits in length, there are 20 blocks, so 20 input vectors consisting of a nonce and a counter are needed. GCM, on the other hand, has only one IV per encryption operation and does not repeat with the same key. For details and recommendations on IVs, see section 8 of NIST document on GCM.

* No sample code due to conceptual rules.

If this is violated, the following may occur.

- Failure to meet the initialization vector requirements for each mode.

3.1.2.17 Use OAEP incorporated in PKCS#1 v2.0 as a padding mechanism for asymmetric encryption (Required)

Previously, PKCS1.5 padding (code: PKCS1Padding) was used as a padding mechanism for asymmetric encryption. This mechanism is vulnerable to the padding Oracle attack. Therefore, PKCS#1 v2.0 (codes: OAEPwithSHA-256andMGF1Padding , OAEPwithSHA-224andMGF1Padding , and OAEPwithSHA-384andMGF1Padding , OAEPwithSHA-512andMGF1Padding). OAEP is the most appropriate method to use. Note that even if you use OAEP, you may encounter the well-known problem known as the Mangers attack described in Kudelskisecurity' s blog.

The sample code below shows how OAEP is used.

```
let plainData = "TEST TEXT".data(using: .utf8)!

// Generate RsaOAEPPadding class with main digest SHA256, MGF1 digest SHA1
let padding = RsaOAEPPadding(mainDigest: OAEPDigest.SHA256, mgf1Digest: OAEPDigest.
```

(continues on next page)

(continued from previous page)

```
↪SHA1)

// Calculate OAEP Padding (RSA key length is assumed to be 2048bit = 256byte)
let padded = try! padding.pad(plain: plainData, blockSize: 256);

// Encrypted in raw
guard let cipherData = SecKeyCreateEncryptedData(publicKey, SecKeyAlgorithm.
↪rsaEncryptionRaw, padded as CFData, &error) else {
    // Error Handling
}
```

If this is violated, the following may occur.

- Become vulnerable to padding oracle attacks.

3.1.2.18 Use key considering memory dump (Required)

If memory dumps are part of the threat model, keys can be accessed at the moment they are actively used. Memory dumps require root access (e.g. rooted or jailbroken devices) or an application patched with Frida (so that tools such as Fridump can be used). Therefore, if a key is still needed on a device, it is best to consider the following

- Remote server keys : remote key vaults such as Amazon KMS or Azure Key Vault may be used. For some use cases, developing an orchestration layer between the app and the remote resource may be an appropriate option. For example, a serverless function running on a Function as a Service (FaaS) system (such as AWS Lambda or Google Cloud Functions) might forward requests to retrieve an API key or secret. Other options include Amazon Cognito, Google Identity Platform, and Azure Active Directory.
- Keys in hardware-protected secure storage: all encryption actions and the key itself remain in the Secure Enclave (e.g. using Keychain). For more information, see the chapter iOS Data Storage.
- Key protected by envelope encryption : If the key is stored outside of the TEE/SE, consider using multi-layered encryption. Envelope Encryption Approach (OWASP Cryptographic Storage Cheat Sheet、Google Cloud Key management guide、AWS Well-Architected Framework guide see also), Or use the HPKE approach, which key-encrypts the data encryption key.
- Keys in memory : Ensure that keys remain in memory for as short a time as possible, and consider zeroing and deactivating keys after a successful encryption operation or in the event of an error. For general encryption guidelines, see "Clearing Memory of Sensitive Data" . For more detailed information, see "Testing Memory for Sensitive Data" .

The sample code below is a process to prevent leakage of keys in memory in an application.

```
data.resetBytes(in: NSRange(location:0, length:data.length))
```

Also, by not storing the key at all, it is guaranteed that the key material will not be dumped. This can be accomplished by using a password key derivation function such as PKBDF-2. See example below.

```
func pbkdf2SHA1(password: String, salt: Data, keyByteCount: Int, rounds: Int) ->␣
↪Data? {
    return pbkdf2(hash: CCPBKDFAlgorithm(kCCPRFHmacAlgSHA1), password: password,␣
↪salt: salt, keyByteCount: keyByteCount, rounds: rounds)
}

func pbkdf2SHA256(password: String, salt: Data, keyByteCount: Int, rounds: Int) ->␣
↪Data? {
    return pbkdf2(hash: CCPBKDFAlgorithm(kCCPRFHmacAlgSHA256), password: password,␣
↪salt: salt, keyByteCount: keyByteCount, rounds: rounds)
}

func pbkdf2SHA512(password: String, salt: Data, keyByteCount: Int, rounds: Int) ->␣
↪Data? {
```

(continues on next page)

(continued from previous page)

```
    return pbkdf2(hash: CCPBKDFAlgorithm(kCCPRFHmacAlgSHA512), password: password,␣
↪salt: salt, keyByteCount: keyByteCount, rounds: rounds)
}

func pbkdf2(hash: CCPBKDFAlgorithm, password: String, salt: Data, keyByteCount:␣
↪Int, rounds: Int) -> Data? {
    let passwordData = password.data(using: String.Encoding.utf8)!
    var derivedKeyData = Data(repeating: 0, count: keyByteCount)
    let derivedKeyDataLength = derivedKeyData.count
    let derivationStatus = derivedKeyData.withUnsafeMutableBytes { derivedKeyBytes␣
↪in
        salt.withUnsafeBytes { saltBytes in

            CCKeyDerivationPBKDF(
                CCPBKDFAlgorithm(kCCPBKDF2),
                password, passwordData.count,
                saltBytes, salt.count,
                hash,
                UInt32(rounds),
                derivedKeyBytes, derivedKeyDataLength
            )
        }
    }
    if derivationStatus != 0 {
        // Error
        return nil
    }

    return derivedKeyData
}

func testKeyDerivation() {
    let password = "password"
    let salt = Data([0x73, 0x61, 0x6C, 0x74, 0x44, 0x61, 0x74, 0x61])
    let keyByteCount = 16
    let rounds = 100_000

    let derivedKey = pbkdf2SHA1(password: password, salt: salt, keyByteCount:␣
↪keyByteCount, rounds: rounds)
}
```

- source : https://stackoverflow.com/questions/8569555/pbkdf2-using-commoncrypto-on-ios (Tested with the Arcane library test suite)

If this is violated, the following may occur.

- If the memory dump is part of the threat model, the key is accessible the moment it is actively used.

3.1.2.19 Do not share the same key across accounts or devices (Required)

To facilitate memory dumps, the same key is not shared among accounts or devices, except for the public key used for signature verification and encryption.

* No sample code due to deprecated rules.

If this is violated, the following may occur.

- Facilitate a memory dump of the key.

3.1.2.20 Appropriate key protection by means of transport symmetric keys or other mechanisms (Required)

If keys need to be transferred between devices or from the app to the backend, ensure that proper key protection is in place through transport symmetric keys or other mechanisms. In many cases, keys are shared in an obfuscated state and can be easily undone. Instead, ensure that asymmetric encryption or wrapping keys are used. For example, a symmetric key can be encrypted with an asymmetric public key.

Keychain is used to properly protect keys. See the rulebook below for key storage with Keychain.

Rulebook

- *Securely store values using the Keychain Services API (Required)*

If this is violated, the following may occur.

- The key is undone and read.

3.2 MSTG-CRYPTO-2

The app uses proven implementations of cryptographic primitives.

3.2.1 Problematic Encryption Configuration

* Check the contents of MSTG-CRYPTO-1 3.1.1. problematic encryption configuration.

3.2.2 Composition of Encryption Standard Algorithm

Apple provides libraries that include implementations of most common cryptographic algorithms. Apple's Cryptographic Services Guide is a great reference. It contains generalized documentation of how to use standard libraries to initialize and use cryptographic primitives, information that is useful for source code analysis.

Reference

- owasp-mastg Testing Key Management (MSTG-CRYPTO-1 and MSTG-CRYPTO-5)

3.2.2.1 CryptoKit

Apple CryptoKit was released with iOS 13 and is built on top of Apple's native cryptographic library corecrypto which is FIPS 140-2 validated. The Swift framework provides a strongly typed API interface, has effective memory management, conforms to equatable, and supports generics. CryptoKit contains secure algorithms for hashing, symmetric-key cryptography, and public-key cryptography. The framework can also utilize the hardware based key manager from the Secure Enclave.

Apple CryptoKit contains the following algorithms:

Hashes:

- MD5 (Insecure Module)
- SHA1 (Insecure Module)
- SHA-2 256-bit digest
- SHA-2 384-bit digest
- SHA-2 512-bit digest

Symmetric-Key:

- Message Authentication Codes (HMAC)
- Authenticated Encryption

  - AES-GCM
  - ChaCha20-Poly1305

Public-Key:

- Key Agreement
  - Curve25519
  - NIST P-256
  - NIST P-384
  - NIST P-512

Examples:

Generating and releasing a symmetric key:

```
let encryptionKey = SymmetricKey(size: .bits256)
```

Calculating a SHA-2 512-bit digest:

```
let rawString = "OWASP MTSG"
let rawData = Data(rawString.utf8)
let hash = SHA512.hash(data: rawData) // Compute the digest
let textHash = String(describing: hash)
print(textHash) // Print hash text
```

For more information about Apple CryptoKit, please visit the following resources:

- Apple CryptoKit | Apple Developer Documentation
- Performing Common Cryptographic Operations | Apple Developer Documentation
- WWDC 2019 session 709 | Cryptography and Your Apps
- How to calculate the SHA hash of a String or Data instance | Hacking with Swift

Reference

- owasp-mastg Verifying the Configuration of Cryptographic Standard Algorithms (MSTG-CRYPTO-2 and MSTG-CRYPTO-3) CryptoKit

Rulebook

- *Implementation of the iOS encryption algorithm Apple CryptoKit (Recommended)*

3.2.2.2 CommonCrypto, SecKey and Wrapper libraries

The most commonly used Class for cryptographic operations is the CommonCrypto, which is packed with the iOS runtime. The functionality offered by the CommonCrypto object can best be dissected by having a look at the source code of the header file:

- The Commoncryptor.h gives the parameters for the symmetric cryptographic operations.
- The CommonDigest.h gives the parameters for the hashing Algorithms.
- The CommonHMAC.h gives the parameters for the supported HMAC operations.
- The CommonKeyDerivation.h gives the parameters for supported KDF functions.
- The CommonSymmetricKeywrap.h gives the function used for wrapping a symmetric key with a Key Encryption Key.

Unfortunately, CommonCryptor lacks a few types of operations in its public APIs, such as: GCM mode is only available in its private APIs See its source code. For this, an additional binding header is necessary or other wrapper libraries can be used.

Next, for asymmetric operations, Apple provides SecKey. Apple provides a nice guide in its Developer Documentation on how to use this.

As noted before: some wrapper-libraries exist for both in order to provide convenience. Typical libraries that are used are, for instance:

- IDZSwiftCommonCrypto
- Heimdall
- SwiftyRSA
- RNCryptor
- Arcane

Reference

- owasp-mastg Verifying the Configuration of Cryptographic Standard Algorithms (MSTG-CRYPTO-2 and MSTG-CRYPTO-3) CommonCrypto, SecKey and Wrapper libraries

Rulebook

- *Implementation of the iOS encryption algorithm Apple CryptoKit (Recommended)*

3.2.2.3 Third party libraries

There are various third party libraries available, such as:

- CJOSE: With the rise of JWE, and the lack of public support for AES GCM, other libraries have found their way, such as CJOSE. CJOSE still requires a higher level wrapping as they only provide a C/C++ implementation.
- CryptoSwift: A library in Swift, which can be found at GitHub. The library supports various hash-functions, MAC-functions, CRC-functions, symmetric ciphers, and password-based key derivation functions. It is not a wrapper, but a fully self-implemented version of each of the ciphers. It is important to verify the effective implementation of a function.
- OpenSSL: OpenSSL is the toolkit library used for TLS, written in C. Most of its cryptographic functions can be used to do the various cryptographic actions necessary, such as creating (H)MACs, signatures, symmetric- & asymmetric ciphers, hashing, etc.. There are various wrappers, such as OpenSSL and MIHCrypto.
- LibSodium: Sodium is a modern, easy-to-use software library for encryption, decryption, signatures, password hashing and more. It is a portable, cross-compilable, installable, packageable fork of NaCl, with a compatible API, and an extended API to improve usability even further. See LibSodiums documentation for more details. There are some wrapper libraries, such as Swift-sodium, NAChloride, and libsodium-ios.
- Tink: A new cryptography library by Google. Google explains its reasoning behind the library on its security blog. The sources can be found at Tinks GitHub repository..
- Themis: a Crypto library for storage and messaging for Swift, Obj-C, Android/Java, C ++, JS, Python, Ruby, PHP, Go. Themis uses LibreSSL/OpenSSL engine libcrypto as a dependency. It supports Objective-C and Swift for key generation, secure messaging (e.g. payload encryption and signing), secure storage and setting up a secure session. See their wiki for more details.
- Others: There are many other libraries, such as CocoaSecurity, Objective-C-RSA, and aerogear-ios-crypto. Some of these are no longer maintained and might never have been security reviewed. Like always, it is recommended to look for supported and maintained libraries.
- DIY: An increasing amount of developers have created their own implementation of a cipher or a cryptographic function. This practice is highly discouraged and should be vetted very thoroughly by a cryptography expert if used.

Reference

- owasp-mastg Verifying the Configuration of Cryptographic Standard Algorithms (MSTG-CRYPTO-2 and MSTG-CRYPTO-3) Third party libraries

3.2.2.4 Static Analysis

A lot has been said about deprecated algorithms and cryptographic configurations in section Cryptography for Mobile Apps. Obviously, these should be verified for each of the mentioned libraries in this chapter. Pay attention to how-to-be-removed key-holding datastructures and plain-text data structures are defined. If the keyword let is used, then you create an immutable structure which is harder to wipe from memory. Make sure that it is part of a parent structure which can be easily removed from memory (e.g. a struct that lives temporally).

CommonCryptor If the app uses standard cryptographic implementations provided by Apple, the easiest way to determine the status of the related algorithm is to check for calls to functions from CommonCryptor, such as CCCrypt and CCCryptorCreate. The source code contains the signatures of all functions of CommonCryptor.h. For instance, CCCryptorCreate has following signature:

```
CCCryptorStatus CCCryptorCreate(
    CCOperation op,             /* kCCEncrypt, etc. */
    CCAlgorithm alg,            /* kCCAlgorithmDES, etc. */
    CCOptions options,          /* kCCOptionPKCS7Padding, etc. */
    const void *key,            /* raw key material */
    size_t keyLength,
    const void *iv,             /* optional initialization vector */
    CCCryptorRef *cryptorRef);  /* RETURNED */
```

You can then compare all the enum types to determine which algorithm, padding, and key material is used. Pay attention to the keying material: the key should be generated securely - either using a key derivation function or a random-number generation function. Note that functions which are noted in chapter "Cryptography for Mobile Apps" as deprecated, are still programmatically supported. They should not be used.

Third party libraries Given the continuous evolution of all third party libraries, this should not be the place to evaluate each library in terms of static analysis. Still there are some points of attention:

- Find the library being used: This can be done using the following methods:
  - Check the cartfile if Carthage is used.
  - Check the podfile if Cocoapods is used.
  - Check the linked libraries: Open the xcodeproj file and check the project properties. Go to the Build Phases tab and check the entries in Link Binary With Libraries for any of the libraries. See earlier sections on how to obtain similar information using MobSF.
  - In the case of copy-pasted sources: search the headerfiles (in case of using Objective-C) and otherwise the Swift files for known methodnames for known libraries.
- Determine the version being used: Always check the version of the library being used and check whether there is a new version available in which possible vulnerabilities or shortcomings are patched. Even without a newer version of a library, it can be the case that cryptographic functions have not been reviewed yet. Therefore we always recommend using a library that has been validated or ensure that you have the ability, knowledge and experience to do validation yourself.
- By hand?: We recommend not to roll your own crypto, nor to implement known cryptographic functions yourself.

Reference

- owasp-mastg Verifying the Configuration of Cryptographic Standard Algorithms (MSTG-CRYPTO-2 and MSTG-CRYPTO-3) Static Analysis

3.2.3 Rulebook

1. *Implementation of the iOS encryption algorithm Apple CryptoKit (Recommended)*

3.2.3.1 Implementation of the iOS encryption algorithm Apple CryptoKit (Recommended)

Use Apple CryptoKit to perform the following common encryption operations.

- Compute and compare cryptographically secure digests.
- Uses public key cryptography to create and evaluate digital signatures and perform key exchange. In addition to manipulating keys stored in memory, private keys stored and managed in Secure Enclave can also be used.
- Symmetric keys are generated and used in operations such as message authentication and encryption.

Use of CryptoKit is recommended over low-level interfaces. CryptoKit automatically handles tasks that make apps more secure, such as freeing apps from managing raw pointers and overwriting sensitive data during memory deallocation.

Hash value generation using CryptoKit

CryptoKit can be used to generate the following hashes

Cryptographically secure hashes

- struct SHA512
  - Secure Hashing Algorithm 2 (SHA-2) hash implementation with 512-bit digests.
- struct SHA384
  - Secure Hashing Algorithm 2 (SHA-2) hash implementation with 384-bit digests.
- struct SHA256
  - Secure Hashing Algorithm 2 (SHA-2) hash implementation with 256-bit digests.

Cryptographically insecure hashes

- Insecure. struct MD5
  - MD5 hash implementation.
- Insecure. struct SHA1
  - SHA1 hash implementation.

```
import CryptoKit
import UIKit

func sha256Hash(str: String) -> String? {
    let data = Data(str.utf8)
    let hashed = SHA256.hash(data: data)

    return hashed.compactMap { String(format: "%02x", $0) }.joined()
}

func md5Hash(str: String) -> String? {

    let data = Data(str.utf8)
    // Insecure Container of old, cryptographically insecure algorithms
    let hashed = Insecure.MD5.hash(data: data)

    return hashed.compactMap { String(format: "%02x", $0) }.joined()
}

func createSha256Hashing() {
    //  sha256 strings generated
```

(continues on next page)

(continued from previous page)

```
    let hash = sha256Hash(str: "test") //␣
↪9f86d081884c7d659a2feaa0c55ad015a3bf4f1b2b0b822cd15d6c15b0f00a08
}

func createMd5hash() {
    // md5 strings generated
    let hash = md5Hash(str: "test") //098f6bcd4621d373cade4e832627b4f6

}
```

Digital signatures using CryptoKit (Cryptographic Signature)

CryptoKit can be used to sign and verify with the following public key ciphers.

public key encryption

- enum Curve25519
  - X25519 Elliptic curve allowing key agreement and ed25519 signature.
- enum P521
  - NIST P-521 Elliptic curves that allow signature and key agreement.
- enum P384
  - NIST P-384 Elliptic curves that allow signature and key agreement.
- enum P256
  - NIST P-256 Elliptic curve to allow signature and key agreement.

```
import UIKit
import CryptoKit

// CryptoSigningProtocol
struct CryptoSignature {
    var signature: Data
    var signedData: Data
}


struct CryptoSigning {

    var rawPrivateKey: Data = Data(base64Encoded: "EDpGUyQuE0Xtjt3/
↪j8KmxtBdaKQNP+7uTU3nJg7pzsg=") ?? Data()

    func createKey() -> Data? {
        guard let privateKey = try? Curve25519.Signing.
↪PrivateKey(rawRepresentation: rawPrivateKey) else { return nil }
        return privateKey.publicKey.rawRepresentation
    }

    func sign(str: String) -> CryptoSignature? {
        guard let data = str.data(using: .utf8),
        let privateKey = try? Curve25519.Signing.PrivateKey(rawRepresentation:␣
↪rawPrivateKey),
        let signature = try? privateKey.signature(for: data) else { return nil }

        return CryptoSignature(signature: signature, signedData: data)
    }

    func isValid(rawPublicKey: Data, signature: CryptoSignature) -> Bool {
        guard let signingPublicKey = try? Curve25519.Signing.
```

(continues on next page)

(continued from previous page)

```
↪PublicKey(rawRepresentation: rawPublicKey) else { return false }

        return signingPublicKey.isValidSignature(signature.signature, for:␣
↪signature.signedData)
    }
}

class CryptoSigningSample {

    func testSuccessCaseForCryptoSigning() {

        let cryptoSigning = CryptoSigning()
        let rawPublicKey = cryptoSigning.createKey()!
        let signedSignature = cryptoSigning.sign(str: "ABC")!


        if cryptoSigning.isValid(rawPublicKey: rawPublicKey, signature:␣
↪signedSignature) {
            // Verification OK
        }
    }
}
```

Target Key Cryptography with CryptoKit (Symmetric Encryption)

CryptoKit allows you to operate with the following symmetric key encryption schemes.

cipher

- enum AES
  - Advanced Encryption Standard (AES) cipher container.
  - To use GCM mode Use GCM in the AES container.
- enum ChaChaPoly
  - ChaCha20-Poly1305 cipher implementation.

```
import UIKit
import CryptoKit

struct ChaChaPolyEncryption {

    var cryptoKey: SymmetricKey = SymmetricKey(size: .bits256)

    func encrypt(str: String) -> Data? {
        let data = Data(str.utf8)
        guard let sealedBox = try? ChaChaPoly.seal(data, using: cryptoKey) else {␣
↪return nil }

        return sealedBox.combined
    }

    func decrypt(data: Data) -> String? {
        guard let sealedBox = try? ChaChaPoly.SealedBox(combined: data) else {␣
↪return nil }
        guard let decryptedData = try? ChaChaPoly.open(sealedBox, using:␣
↪cryptoKey) else { return nil }

        return String(data: decryptedData, encoding: .utf8)
    }
}
```

(continues on next page)

(continued from previous page)

```
class ChaChaPolyEncryptionSample {

    func execChaChaPolyEncryption() {

        let encryption = ChaChaPolyEncryption()

        // Encrypt
        guard let signature = encryption.encrypt(str: "ABC") else {
            return
        }

        // Decrypt
        guard let decryptText =  encryption.decrypt(data: signature)  else {
            return
        }

    }
}
```

Implementation of encryption with CommonCrypto and SecKey

Before CryptoKit was created, CommonCrypto and SecKey were used. Currently, Apple recommends the use of CryptoKit, but if you cannot use CryptoKit due to OS version or other reasons, you can use this standard OS API.

The following sample code shows how CommonCrypto is used.

```
#ifndef SwiftAES_Bridging_Header_h
#define SwiftAES_Bridging_Header_h


#endif /* SwiftAES_Bridging_Header_h */

#import <CommonCrypto/CommonCrypto.h>
```

```
import UIKit
import CryptoKit
import CommonCrypto

public class Chiper {

    enum AESError : Error {
        case encryptFailed(String, Any)
        case decryptFailed(String, Any)
        case otherFailed(String, Any)
    }

    /// Convert binary Data to hexadecimal string
     /// - Parameter binaryData: Data containing binary
     /// - Returns: hexadecimal string
     public static func convetHexString(frombinary data: Data) -> String {

         return data.reduce("") { (a : String, v : UInt8) -> String in
             return a + String(format: "%02x", v)
         }

     }

    public class AES {
        /// cipher
        public static func encrypt(plainString: String, sharedKey: String, iv:␣
```

(continues on next page)

(continued from previous page)

```
↪String) throws -> Data {
            guard let initialzeVector = (iv.data(using: .utf8)) else {
                throw Chiper.AESError.otherFailed("Encrypt iv failed", iv)
            }
            guard let keyData = sharedKey.data(using: .utf8) else {
                throw Chiper.AESError.otherFailed("Encrypt sharedkey failed",␣
↪sharedKey)
            }
            guard let data = plainString.data(using: .utf8) else {
                throw Chiper.AESError.otherFailed("Encrypt plainString failed",␣
↪plainString)
            }

            // Calculate the size of the data after encryption
            let cryptLength = size_t(Int(ceil(Double(data.count /␣
↪kCCBlockSizeAES128)) + 1.0) * kCCBlockSizeAES128)

            var cryptData = Data(count:cryptLength)
            var numBytesEncrypted: size_t = 0

            // encryption
            let cryptStatus = cryptData.withUnsafeMutableBytes {cryptBytes in
                initialzeVector.withUnsafeBytes {ivBytes in
                    data.withUnsafeBytes {dataBytes in
                        keyData.withUnsafeBytes {keyBytes in
                            CCCrypt(CCOperation(kCCEncrypt),
                                    CCAlgorithm(kCCAlgorithmAES),
                                    CCOptions(kCCOptionPKCS7Padding),
                                    keyBytes, keyData.count,
                                    ivBytes,
                                    dataBytes, data.count,
                                    cryptBytes, cryptLength,
                                    &numBytesEncrypted)
                        }
                    }
                }
            }

            if UInt32(cryptStatus) != UInt32(kCCSuccess) {
                throw Chiper.AESError.encryptFailed("Encrypt Failed", kCCSuccess)
            }
            return cryptData
        }

        /// decoding
        public static func decrypt(encryptedData: Data, sharedKey: String, iv:␣
↪String) throws -> String {
            guard let initialzeVector = (iv.data(using: .utf8)) else {
                throw Chiper.AESError.otherFailed("Encrypt iv failed", iv)
            }
            guard let keyData = sharedKey.data(using: .utf8) else {
                throw Chiper.AESError.otherFailed("Encrypt sharedKey failed",␣
↪sharedKey)
            }

            let clearLength = size_t(encryptedData.count + kCCBlockSizeAES128)
            var clearData   = Data(count:clearLength)

            var numBytesEncrypted :size_t = 0

            // decoding
```

(continues on next page)

(continued from previous page)

```
            let cryptStatus = clearData.withUnsafeMutableBytes {clearBytes in
                initialzeVector.withUnsafeBytes {ivBytes in
                    encryptedData.withUnsafeBytes {dataBytes in
                        keyData.withUnsafeBytes {keyBytes in
                            CCCrypt(CCOperation(kCCDecrypt),
                                    CCAlgorithm(kCCAlgorithmAES),
                                    CCOptions(kCCOptionPKCS7Padding),
                                    keyBytes, keyData.count,
                                    ivBytes,
                                    dataBytes, encryptedData.count,
                                    clearBytes, clearLength,
                                    &numBytesEncrypted)
                        }
                    }
                }
            }

            if UInt32(cryptStatus) != UInt32(kCCSuccess) {
                throw Chiper.AESError.decryptFailed("Decrypt Failed", kCCSuccess)
            }

            // Perform string conversion by discarding data for the number of␣
↪characters that were padded.
            guard let decryptedStr = String(data: clearData.
↪prefix(numBytesEncrypted), encoding: .utf8) else {
                throw Chiper.AESError.decryptFailed("PKSC Unpad Failed", clearData)
            }
            return decryptedStr
        }

        /// Random IV generation
        public static func generateRandamIV() throws -> String {
            // Obtain random numbers from CSPRNG
            var randData = Data(count: 8)
            let result = randData.withUnsafeMutableBytes {mutableBytes in
                SecRandomCopyBytes(kSecRandomDefault, 16, mutableBytes)
            }
            if result != errSecSuccess {
                // SecRandomCopyBytes failed (not originally possible)
                throw Chiper.AESError.otherFailed("SecRandomCopyBytes Failed␣
↪GenerateRandam IV", result)
            }
            // Hexadecimal Stringing
            let ivStr = Chiper.convetHexString(frombinary: randData)
            return ivStr
        }
    }
}
```

The following sample code shows how to use SecKey.

```
import Foundation

public class SecKeyHelper {

   /// Convert a key string in base64pem format to SecKey format for use in iOS.
    ///
    /// - Parameters:
    ///   - argBase64Key: public or private key in base64pem format
    ///   - keyType: kSecAttrKeyClassPublic|kSecAttrKeyClassPrivate
    /// - Returns: Key data in SecKey format
```

(continues on next page)

(continued from previous page)

```
    /// - Throws: RSAError
    static func convertSecKeyFromBase64Key(_ argBase64Key: String, _ keyType:␣
↪CFString) throws -> SecKey {

        var keyData = Data(base64Encoded: argBase64Key, options: [.
↪ignoreUnknownCharacters])!
        let keyClass = keyType

        let sizeInBits = keyData.count * 8
        let keyDict: [CFString: Any] = [
            kSecAttrKeyType: kSecAttrKeyTypeRSA,
            kSecAttrKeyClass: keyClass,
            kSecAttrKeySizeInBits: NSNumber(value: sizeInBits),
            kSecReturnPersistentRef: true
        ]
        var error: Unmanaged<CFError>?
        guard let key = SecKeyCreateWithData(keyData as CFData, keyDict as␣
↪CFDictionary, &error) else {
            throw RSAError.keyCreateFailed(status: 0)
        }
        return key

    }

    /// Encryption with public key
    ///
    /// - Parameters:
    ///   - argBody: target string
    ///   - argBase64PublicKey: Public key string (base64)
    /// - Returns: coded data
    static func encrypt(_ argBody: String, _ argBase64PublicKey: String) -> Data {

        do {
            let pubKey = try self.convertSecKeyFromBase64Key(argBase64PublicKey,␣
↪kSecAttrKeyClassPublic)
            let plainBuffer = [UInt8](argBody.utf8)
            var cipherBufferSize = Int(SecKeyGetBlockSize(pubKey))
            var cipherBuffer = [UInt8](repeating:0, count:Int(cipherBufferSize))
            // Encrypto  should less than key length
            let status = SecKeyEncrypt(pubKey, SecPadding.PKCS1, plainBuffer,␣
↪plainBuffer.count, &cipherBuffer, &cipherBufferSize)
            if (status != errSecSuccess) {
                print("Failed Encryption")
            }
            return Data(bytes: cipherBuffer)
        }
        catch {
            // error handling
        }

        return Data()

    }
}
```

If this is not noted, the following may occur.

- Potentially vulnerable encryption implementations.

3.3 MSTG-CRYPTO-3

The app uses cryptographic primitives that are appropriate for the particular use-case, configured with parameters that adhere to industry best practices.

3.3.1 Problematic Encryption Configuration

* Check the contents of MSTG-CRYPTO-1 3.1.1. Problematic Encryption Configuration.

3.3.2 Composition of Encryption Standard Algorithm

* Confirm the contents of MSTG-CRYPTO-2 3.2.2. Composition of Encryption Standard Algorithm.

3.4 MSTG-CRYPTO-4

The app does not use cryptographic protocols or algorithms that are widely considered deprecated for security purposes.

3.4.1 Insecure or deprecated encryption algorithms

When assessing a mobile app, you should make sure that it does not use cryptographic algorithms and protocols that have significant known weaknesses or are otherwise insufficient for modern security requirements. Algorithms that were considered secure in the past may become insecure over time; therefore, it's important to periodically check current best practices and adjust configurations accordingly.

Verify that cryptographic algorithms are up to date and in-line with industry standards. Vulnerable algorithms include outdated block ciphers (such as DES and 3DES), stream ciphers (such as RC4), hash functions (such as MD5 and SHA1), and broken random number generators (such as Dual_EC_DRBG and SHA1PRNG). Note that even algorithms that are certified (for example, by NIST) can become insecure over time. A certification does not replace periodic verification of an algorithm's soundness. Algorithms with known weaknesses should be replaced with more secure alternatives. Additionally, algorithms used for encryption must be standardized and open to verification. Encrypting data using any unknown, or proprietary algorithms may expose the application to different cryptographic attacks which may result in recovery of the plaintext.

Inspect the app's source code to identify instances of cryptographic algorithms that are known to be weak, such as:

- DES, 3DES
- RC2
- RC4
- BLOWFISH
- MD4
- MD5
- SHA1

The names of cryptographic APIs depend on the particular mobile platform.

Please make sure that:

- Cryptographic algorithms are up to date and in-line with industry standards. This includes, but is not limited to outdated block ciphers (e.g. DES), stream ciphers (e.g. RC4), as well as hash functions (e.g. MD5) and broken random number generators like Dual_EC_DRBG (even if they are NIST certified). All of these should be marked as insecure and should not be used and removed from the application and server.

- Key lengths are in-line with industry standards and provide protection for sufficient amount of time. A comparison of different key lengths and protection they provide taking into account Moore's law is available online.
- Cryptographic means are not mixed with each other: e.g. you do not sign with a public key, or try to reuse a key pair used for a signature to do encryption.
- Cryptographic parameters are well defined within reasonable range. This includes, but is not limited to: cryptographic salt, which should be at least the same length as hash function output, reasonable choice of password derivation function and iteration count (e.g. PBKDF2, scrypt or bcrypt), IVs being random and unique, fit-for-purpose block encryption modes (e.g. ECB should not be used, except specific cases), key management being done properly (e.g. 3DES should have three independent keys) and so on.

Reference

- owasp-mastg Identifying Insecure and/or Deprecated Cryptographic Algorithms (MSTG-CRYPTO-4)

Rulebook

- *Do not use unsecured or deprecated encryption algorithms (Required)*

3.4.2 Rulebook

1. *Do not use unsecured or deprecated encryption algorithms (Required)*

3.4.2.1 Do not use unsecured or deprecated encryption algorithms (Required)

Implement with the latest encryption algorithms compliant with industry standards.

Specific aspects of implementation should be in accordance with the following contents.

- The encryption algorithm is up-to-date and conforms to industry standards. See "*Set key lengths that meet industry standards (Required)*" for industry standards.
- Key lengths comply with industry standards and provide sufficient time protection. A comparison of various key lengths and their protection performance considering Moore's Law can be found online.
- Do not mix encryption methods with each other: for example, do not sign with the public key or reuse the symmetric key used for signing for encryption.
- Define cryptographic parameters as reasonably appropriate. These include cipher salt that must be at least as long as the hash function output, appropriate choice of password derivation function and number of iterations (e.g. PBKDF2 , scrypt, bcrypt), IV must be random and unique, block cipher mode appropriate for the purpose (e.g. ECB should not be used except in certain cases), and key management must be appropriate (e.g. 3DES should have three independent keys).), that the IV is random and unique, that the block encryption mode is appropriate for the purpose (e.g. ECB should not be used except in certain cases), and that key management is properly implemented (e.g. 3DES should have three independent keys).

See the rulebook below for sample code.

Rulebook

- *Implementation of the iOS encryption algorithm Apple CryptoKit (Recommended)*

If this is violated, the following may occur.

- May result in a weak encryption process.

3.5 MSTG-CRYPTO-5

The app doesn't re-use the same cryptographic key for multiple purposes.

Reference

- owasp-mastg Testing Key Management (MSTG-CRYPTO-1 and MSTG-CRYPTO-5)

3.5.1 Key management validation

There are various methods on how to store the key on the device. Not storing a key at all will ensure that no key material can be dumped. This can be achieved by using a Password Key Derivation function, such as PKBDF-2. See the example below:

```
func pbkdf2SHA1(password: String, salt: Data, keyByteCount: Int, rounds: Int) ->␣
↪Data? {
    return pbkdf2(hash: CCPBKDFAlgorithm(kCCPRFHmacAlgSHA1), password: password,␣
↪salt: salt, keyByteCount: keyByteCount, rounds: rounds)
}

func pbkdf2SHA256(password: String, salt: Data, keyByteCount: Int, rounds: Int) ->␣
↪Data? {
    return pbkdf2(hash: CCPBKDFAlgorithm(kCCPRFHmacAlgSHA256), password: password,␣
↪salt: salt, keyByteCount: keyByteCount, rounds: rounds)
}

func pbkdf2SHA512(password: String, salt: Data, keyByteCount: Int, rounds: Int) ->␣
↪Data? {
    return pbkdf2(hash: CCPBKDFAlgorithm(kCCPRFHmacAlgSHA512), password: password,␣
↪salt: salt, keyByteCount: keyByteCount, rounds: rounds)
}

func pbkdf2(hash: CCPBKDFAlgorithm, password: String, salt: Data, keyByteCount:␣
↪Int, rounds: Int) -> Data? {
    let passwordData = password.data(using: String.Encoding.utf8)!
    var derivedKeyData = Data(repeating: 0, count: keyByteCount)
    let derivedKeyDataLength = derivedKeyData.count
    let derivationStatus = derivedKeyData.withUnsafeMutableBytes { derivedKeyBytes␣
↪in
        salt.withUnsafeBytes { saltBytes in

            CCKeyDerivationPBKDF(
                CCPBKDFAlgorithm(kCCPBKDF2),
                password, passwordData.count,
                saltBytes, salt.count,
                hash,
                UInt32(rounds),
                derivedKeyBytes, derivedKeyDataLength
            )
        }
    }
    if derivationStatus != 0 {
        // Error
        return nil
    }

    return derivedKeyData
}

func testKeyDerivation() {
    let password = "password"
```

(continues on next page)

(continued from previous page)

```
    let salt = Data([0x73, 0x61, 0x6C, 0x74, 0x44, 0x61, 0x74, 0x61])
    let keyByteCount = 16
    let rounds = 100_000

    let derivedKey = pbkdf2SHA1(password: password, salt: salt, keyByteCount:␣
↪keyByteCount, rounds: rounds)
}
```

- Source: https://stackoverflow.com/questions/8569555/pbkdf2-using-commoncrypto-on-ios, tested in the test suite of the Arcane library

When you need to store the key, it is recommended to use the Keychain as long as the protection class chosen is not kSecAttrAccessibleAlways. Storing keys in any other location, such as the NSUserDefaults, property list files or by any other sink from Core Data or Realm, is usually less secure than using the KeyChain. Even when the sync of Core Data or Realm is protected by using NSFileProtectionComplete data protection class, we still recommend using the KeyChain. See the chapter "Data Storage on iOS" for more details.

The KeyChain supports two type of storage mechanisms: a key is either secured by an encryption key stored in the secure enclave or the key itself is within the secure enclave. The latter only holds when you use an ECDH signing key. See the Apple Documentation for more details on its implementation.

The last three options consist of using hardcoded encryption keys in the source code, having a predictable key derivation function based on stable attributes, and storing generated keys in places that are shared with other applications. Using hardcoded encryption keys is obviously not the way to go, as this would mean that every instance of the application uses the same encryption key. An attacker needs only to do the work once in order to extract the key from the source code (whether stored natively or in Objective-C/Swift). Consequently, the attacker can decrypt any other data that was encrypted by the application. Next, when you have a predictable key derivation function based on identifiers which are accessible to other applications, the attacker only needs to find the KDF and apply it to the device in order to find the key. Lastly, storing symmetric encryption keys publicly also is highly discouraged.

Two more notions you should never forget when it comes to cryptography:

- Always encrypt/verify with the public key and always decrypt/sign with the private key.
- Never reuse the key(pair) for another purpose: this might allow leaking information about the key: have a separate key pair for signing and a separate key(pair) for encryption.

Rulebook

- *Use key considering memory dump (Required)*
- *Do not store keys or passwords in source code (Required)*
- *Observance of not-to-be-forgotten concepts regarding cryptography (Required)*

Data Storage and Privacy Requirements Rulebook

- *Securely store values using the Keychain Services API (Required)*

3.5.1.1 Static Analysis

There are various keywords to look for: check the libraries mentioned in the overview and static analysis of the section "Problematic Encryption Configuration" for which keywords you can best check on how keys are stored.

Always make sure that:

- keys are not synchronized over devices if it is used to protect high-risk data.
- keys are not stored without additional protection.
- keys are not hardcoded.
- keys are not derived from stable features of the device.
- keys are not hidden by use of lower level languages (e.g. C/C++).
- keys are not imported from unsafe locations.

Most of the recommendations for static analysis can already be found in chapter "Testing Data Storage for iOS". Next, you can read up on it at the following pages:

- Apple Developer Documentation: Certificates and keys
- Apple Developer Documentation: Generating new keys
- Apple Developer Documentation: Key generation attributes

Rulebook

- *Do not store keys or passwords in source code (Required)*
- *Do not store client certificate passwords locally. Lock passwords in your device's keychain if you want to save them (Required)*
- *Client certificates are not shared among all installations (Required)*
- *Store private keys in secure device storage whenever symmetric encryption is used in mobile apps (Required)*

Data Storage and Privacy Requirements Rulebook

- *Securely store values using the Keychain Services API (Required)*

3.5.1.2 Dynamic Analysis

Hook cryptographic methods and analyze the keys that are being used. Monitor file system access while cryptographic operations are being performed to assess where key material is written to or read from.

3.5.2 Rulebook

1. *Observance of not-to-be-forgotten concepts regarding cryptography (Required)*

3.5.2.1 Observance of not-to-be-forgotten concepts regarding cryptography (Required)

There are two more concepts that should not be forgotten regarding cryptography. The following concepts must be observed

- Always encrypt and verify with the public key and decrypt and sign with the private key.
- Do not reuse symmetric keys for other purposes. This could result in the leakage of information about the key. Provide separate symmetric keys for signing and encryption.

If this is violated, the following may occur.

- Information about the key may be compromised.

3.6 MSTG-CRYPTO-6

All random values are generated using a sufficiently secure random number generator.

Reference

- owasp-mastg Testing Random Number Generation (MSTG-CRYPTO-6)

3.6.1 Random Number Generator Selection

Apple provides a Randomization Services API, which generates cryptographically secure random numbers.

The Randomization Services API uses the SecRandomCopyBytes function to generate numbers. This is a wrapper function for the /dev/random device file, which provides cryptographically secure pseudorandom values from 0 to 255. Make sure that all random numbers are generated with this API. There is no reason for developers to use a different one.

3.6.1.1 Static Analysis

In Swift, the SecRandomCopyBytes API is defined as follows:

```
func SecRandomCopyBytes(_ rnd: SecRandomRef?,
                        _ count: Int,
                        _ bytes: UnsafeMutablePointer<UInt8>) -> Int32
```

The Objective-C version is

```
int SecRandomCopyBytes(SecRandomRef rnd, size_t count, uint8_t *bytes);
```

The following is an example of the APIs usage:

```
int result = SecRandomCopyBytes(kSecRandomDefault, 16, randomBytes);
```

Note: if other mechanisms are used for random numbers in the code, verify that these are either wrappers around the APIs mentioned above or review them for their secure-randomness. Often this is too hard, which means you can best stick with the implementation above.

Rulebook

- *Generate safe random numbers using Randomization Services API (Recommended)*

3.6.1.2 Dynamic Analysis

If you want to test for randomness, you can try to capture a large set of numbers and check with Burp's sequencer plugin to see how good the quality of the randomness is.

3.6.2 Rulebook

1. *Generate safe random numbers using Randomization Services API (Recommended)*

3.6.2.1 Generate safe random numbers using Randomization Services API (Recommended)

Strengths such as key generation, password string generation, etc. where the characters in the string are completely random (and hidden), leave the attacker no choice but to try all possible combinations one by one in a brute force attack. For sufficiently long strings, parsing becomes impractical.

However, it depends on the quality of the randomization. True randomization is not possible in deterministic systems where software instructions from a bounded set are executed according to clearly defined rules. As a service to sufficiently randomize, Apple uses the Randomization Services API to generate cryptographically secure random numbers.

```
import UIKit

class GenerateBitSample {
    func generate10bitRandom() -> [Int8?] {

        var bytes = [Int8](repeating: 0, count: 10)
        let status = SecRandomCopyBytes(kSecRandomDefault, bytes.count, &bytes)
```

(continues on next page)

(continued from previous page)

```
        if status == errSecSuccess { // Always test the status.
            return bytes
        }

        return []
    }
}
```

If this is not noted, the following may occur.

- Insecure random numbers are generated.

4 Authentication and Session Management Requirements

4.1 MSTG-AUTH-1

If the app provides users access to a remote service, some form of authentication, such as username/password authentication, is performed at the remote endpoint.

* Refer to "*1.2.1.1. Appropriate authentication response*" for appropriate authentication support in this chapter.

Reference

- owasp-mastg Verifying that Appropriate Authentication is in Place
- owasp-mastg Testing OAuth 2.0 Flows

4.2 MSTG-AUTH-2

If stateful session management is used, the remote endpoint uses randomly generated session identifiers to authenticate client requests without sending the user' s credentials.

4.2.1 Session Information Management

Stateful (or "session-based") authentication is characterized by authentication records on both the client and server. The authentication flow is as follows:

1. The app sends a request with the user' s credentials to the backend server.
2. The server verifies the credentials If the credentials are valid, the server creates a new session along with a random session ID.
3. The server sends to the client a response that includes the session ID.
4. The client sends the session ID with all subsequent requests. The server validates the session ID and retrieves the associated session record.
5. After the user logs out, the server-side session record is destroyed and the client discards the session ID.

When sessions are improperly managed, they are vulnerable to a variety of attacks that may compromise the session of a legitimate user, allowing the attacker to impersonate the user. This may result in lost data, compromised confidentiality, and illegitimate actions.

Reference

- owasp-mastg Testing Stateful Session Management

4.2.1.1 Session Management Best Practices

Locate any server-side endpoints that provide sensitive information or functions and verify the consistent enforcement of authorization. The backend service must verify the user's session ID or token and make sure that the user has sufficient privileges to access the resource. If the session ID or token is missing or invalid, the request must be rejected.

Make sure that:

- Session IDs are randomly generated on the server side.
- The IDs can't be guessed easily (use proper length and entropy).
- Session IDs are always exchanged over secure connections (e.g. HTTPS).
- The mobile app doesn't save session IDs in permanent storage.
- The server verifies the session whenever a user tries to access privileged application elements, (a session ID must be valid and must correspond to the proper authorization level).
- The session is terminated on the server side and session information deleted within the mobile app after it times out or the user logs out.

Authentication shouldn't be implemented from scratch but built on top of proven frameworks. Many popular frameworks provide ready-made authentication and session management functionality. If the app uses framework APIs for authentication, check the framework security documentation for best practices. Security guides for common frameworks are available at the following links:

- Spring (Java)
- Struts (Java)
- Laravel (PHP)
- Ruby on Rails

A great resource for testing server-side authentication is the OWASP Web Testing Guide, specifically the Testing Authentication and Testing Session Management chapters.

Reference

- owasp-mastg Testing Stateful Session Management Session Management Best Practices

Rulebook

- *Ensure mobile apps do not store session IDs in persistent storage (Required)*

4.2.2 Rulebook

1. *Ensure mobile apps do not store session IDs in persistent storage (Required)*

4.2.2.1 Ensure mobile apps do not store session IDs in persistent storage (Required)

Storing session IDs in persistent storage may be read/written by users or used by third parties. Therefore, session IDs should not be stored in such storage.

* No sample code due to deprecated rules.

If this is violated, the following may occur.

- There is a risk that the session ID can be read/written by the user or used by a third party.

4.3 MSTG-AUTH-3

If stateless token-based authentication is used, the server provides a token that has been signed using a secure algorithm.

4.3.1 Token management

Token-based authentication is implemented by sending a signed token (verified by the server) with each HTTP request. The most commonly used token format is the JSON Web Token, defined in RFC7519. A JWT may encode the complete session state as a JSON object. Therefore, the server doesn't have to store any session data or authentication information.

JWT tokens consist of three Base64Url-encoded parts separated by dots. The Token structure is as follows:

```
base64UrlEncode(header).base64UrlEncode(payload).base64UrlEncode(signature)
```

The following example shows a Base64Url-encoded JSON Web Token:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6Ikpva
G4gRG9lIiwiYWRtaW4iOnRydWV9.TJVA95OrM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ
```

The header typically consists of two parts: the token type, which is JWT, and the hashing algorithm being used to compute the signature. In the example above, the header decodes as follows:

```
{"alg":"HS256","typ":"JWT"}
```

The second part of the token is the payload, which contains so-called claims. Claims are statements about an entity (typically, the user) and additional metadata. For example:

```
{"sub":"1234567890","name":"John Doe","admin":true}
```

The signature is created by applying the algorithm specified in the JWT header to the encoded header, encoded payload, and a secret value. For example, when using the HMAC SHA256 algorithm the signature is created in the following way:

```
HMACSHA256(base64UrlEncode(header) + "." + base64UrlEncode(payload), secret)
```

Note that the secret is shared between the authentication server and the backend service - the client does not know it. This proves that the token was obtained from a legitimate authentication service. It also prevents the client from tampering with the claims contained in the token.

Reference

- owasp-mastg Testing Stateless (Token-Based) Authentication

4.3.2 Static Analysis

Identify the JWT library that the server and client use. Find out whether the JWT libraries in use have any known vulnerabilities.

Verify that the implementation adheres to JWT best practices:

- Verify that the HMAC is checked for all incoming requests containing a token;
- Verify the location of the private signing key or HMAC secret key. The key should remain on the server and should never be shared with the client. It should be available for the issuer and verifier only.
- Verify that no sensitive data, such as personal identifiable information, is embedded in the JWT. If, for some reason, the architecture requires transmission of such information in the token, make sure that payload encryption is being applied. See the sample Java implementation on the OWASP JWT Cheat Sheet.

- Make sure that replay attacks are addressed with the jti (JWT ID) claim, which gives the JWT a unique identifier.
- Make sure that cross service relay attacks are addressed with the aud (audience) claim, which defines for which application the token is entitled.
- Verify that tokens are stored securely on the mobile phone, with, for example, KeyChain (iOS).

Reference

- owasp-mastg Testing Stateless (Token-Based) Authentication Static Analysis

Rulebook

- *Identify JWT libraries in use and whether they have known vulnerabilities (Required)*
- *Ensure tokens are securely stored on mobile devices by Keychain (iOS) (Required)*

4.3.2.1 Enforcing the Hashing Algorithm

An attacker executes this by altering the token and, using the 'none' keyword, changing the signing algorithm to indicate that the integrity of the token has already been verified. Some libraries might treat tokens signed with the 'none' algorithm as if they were valid tokens with verified signatures, so the application will trust altered token claims.

Reference

- owasp-mastg Testing Stateless (Token-Based) Authentication (MSTG-AUTH-3) Enforcing the Hashing Algorithm

Rulebook

- *Need to request the expected hash algorithm (Required)*

4.3.2.2 Token Expiration

Once signed, a stateless authentication token is valid forever unless the signing key changes. A common way to limit token validity is to set an expiration date. Make sure that the tokens include an "exp" expiration claim and the backend doesn't process expired tokens.

A common method of granting tokens combines access tokens and refresh tokens. When the user logs in, the backend service issues a short-lived access token and a long-lived refresh token. The application can then use the refresh token to obtain a new access token, if the access token expires.

For apps that handle sensitive data, make sure that the refresh token expires after a reasonable period of time. The following example code shows a refresh token API that checks the refresh token's issue date. If the token is not older than 14 days, a new access token is issued. Otherwise, access is denied and the user is prompted to login again.

```
app.post('/renew_access_token', function (req, res) {
    // verify the existing refresh token
    var profile = jwt.verify(req.body.token, secret);

    // if refresh token is more than 14 days old, force login
    if (profile.original_iat - new Date() > 14) { // iat == issued at
      return res.send(401); // re-login
    }

    // check if the user still exists or if authorization hasn't been revoked
    if (!valid) return res.send(401); // re-logging

    // issue a new access token
    var renewed_access_token = jwt.sign(profile, secret, { expiresInMinutes: 60*5 }
↪);
    res.json({ token: renewed_access_token });
  });
```

Reference

- owasp-mastg Testing Stateless (Token-Based) Authentication Token Expiration

Rulebook

- *Set an appropriate period for refresh token expiration (Required)*

4.3.3 Dynamic Analysis

Investigate the following JWT vulnerabilities while performing dynamic analysis:

- Token Storage on the client:
  - The token storage location should be verified for mobile apps that use JWT.
- Cracking the signing key:
  - Token signatures are created via a private key on the server. After you obtain a JWT, choose a tool for brute forcing the secret key offline.
- Information Disclosure:
  - Decode the Base64Url-encoded JWT and find out what kind of data it transmits and whether that data is encrypted.
- Tampering with the Hashing Algorithm:
  - Usage of asymmetric algorithms. JWT offers several asymmetric algorithms as RSA or ECDSA. When these algorithms are used, tokens are signed with the private key and the public key is used for verification. If a server is expecting a token to be signed with an asymmetric algorithm and receives a token signed with HMAC, it will treat the public key as an HMAC secret key. The public key can then be misused, employed as an HMAC secret key to sign the tokens.
  - Modify the alg attribute in the token header, then delete HS256, set it to none, and use an empty signature (e.g., signature = ""). Use this token and replay it in a request. Some libraries treat tokens signed with the none algorithm as a valid token with a verified signature. This allows attackers to create their own "signed" tokens.

There are two different Burp Plugins that can help you for testing the vulnerabilities listed above:

- JSON Web Token Attacker
- JSON Web Tokens

Also, make sure to check out the OWASP JWT Cheat Sheet for additional information.

Reference

- owasp-mastg Testing Stateless (Token-Based) Authentication Dynamic Analysis

4.3.4 Testing OAuth 2.0 Flows

OAuth 2.0 defines a delegation protocol for conveying authorization decisions across APIs and a network of web-enabled applications. It is used in a variety of applications, including user authentication applications.

Common uses for OAuth2 include:

- Getting permission from the user to access an online service using their account.
- Authenticating to an online service on behalf of the user.
- Handling authentication errors.

According to OAuth 2.0, a mobile client seeking access to a user's resources must first ask the user to authenticate against an authentication server. With the users' approval, the authorization server then issues a token that allows the app to act on behalf of the user. Note that the OAuth2 specification doesn't define any particular kind of authentication or access token format.

OAuth 2.0 defines four roles:

- Resource Owner: the account owner
- Client: the application that wants to access the user' s account with the access tokens
- Resource Server: hosts the user accounts
- Authorization Server: verifies user identity and issues access tokens to the application

Note: The API fulfills both the Resource Owner and Authorization Server roles. Therefore, we will refer to both as the API.

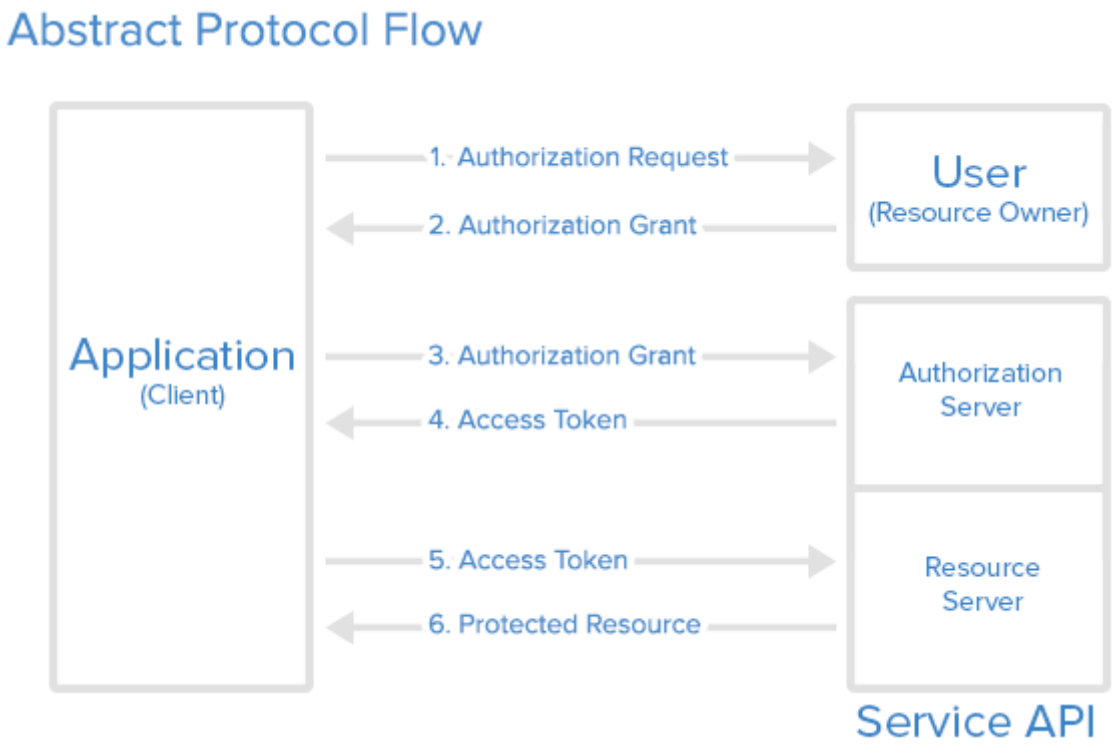


Fig 4.3.4.1 Abstract Protocol Flow

Here is a more detailed explanation of the steps in the diagram:

1. The application requests user authorization to access service resources.
2. If the user authorizes the request, the application receives an authorization grant. The authorization grant may take several forms (explicit, implicit, etc.).
3. The application requests an access token from the authorization server (API) by presenting authentication of its own identity along with the authorization grant.
4. If the application identity is authenticated and the authorization grant is valid, the authorization server (API) issues an access token to the application, completing the authorization process. The access token may have a companion refresh token.
5. The application requests the resource from the resource server (API) and presents the access token for authentication. The access token may be used in several ways (e.g., as a bearer token).
6. If the access token is valid, the resource server (API) serves the resource to the application.

Reference

- owasp-mastg Testing Stateless (Token-Based) Authentication

4.3.5 OAuth 2.0 Best Practices

Verify that the following best practices are followed:

User agent:

- The user should have a way to visually verify trust (e.g., Transport Layer Security (TLS) confirmation, website mechanisms).
- To prevent man-in-the-middle attacks, the client should validate the server's fully qualified domain name with the public key the server presented when the connection was established.

Type of grant:

- On native apps, code grant should be used instead of implicit grant.
- When using code grant, PKCE (Proof Key for Code Exchange) should be implemented to protect the code grant. Make sure that the server also implements it.
- The auth "code" should be short-lived and used immediately after it is received. Verify that auth codes only reside on transient memory and aren't stored or logged.

Client secrets:

- Shared secrets should not be used to prove the client's identity because the client could be impersonated ("client_id" already serves as proof). If they do use client secrets, be sure that they are stored in secure local storage.

End-User credentials:

- Secure the transmission of end-user credentials with a transport-layer method, such as TLS.

Tokens:

- Keep access tokens in transient memory.
- Access tokens must be transmitted over an encrypted connection.
- Reduce the scope and duration of access tokens when end-to-end confidentiality can't be guaranteed or the token provides access to sensitive information or transactions.
- Remember that an attacker who has stolen tokens can access their scope and all resources associated with them if the app uses access tokens as bearer tokens with no other way to identify the client.
- Store refresh tokens in secure local storage; they are long-term credentials.

Reference

- owasp-mastg OAUTH 2.0 Best Practices

Rulebook

- *Adherence to OAuth 2.0 best practices (Required)*

4.3.5.1 External User Agent vs. Embedded User Agent

OAuth2 authentication can be performed either through an external user agent (e.g. Chrome or Safari) or in the app itself (e.g. through a WebView embedded into the app or an authentication library). None of the two modes is intrinsically "better" - instead, what mode to choose depends on the context.

Using an external user agent is the method of choice for apps that need to interact with social media accounts (Facebook, Twitter, etc.). Advantages of this method include:

- The user's credentials are never directly exposed to the app. This guarantees that the app cannot obtain the credentials during the login process ("credential phishing").
- Almost no authentication logic must be added to the app itself, preventing coding errors.

On the negative side, there is no way to control the behavior of the browser (e.g. to activate certificate pinning).

For apps that operate within a closed ecosystem, embedded authentication is the better choice. For example, consider a banking app that uses OAuth2 to retrieve an access token from the bank's authentication server, which is then used to access a number of micro services. In that case, credential phishing is not a viable scenario. It is likely preferable to keep the authentication process in the (hopefully) carefully secured banking app, instead of placing trust on external components.

Reference

- owasp-mastg Testing OAuth 2.0 Flows (MSTG-AUTH-1 and MSTG-AUTH-3) External User Agent vs. Embedded User Agent

Rulebook

- *Main recommended libraries used for OAuth2 authentication (Recommended)*
- *When using an external user agent for OAuth2 authentication, understand the advantages and disadvantages and use it (Recommended)*

4.3.6 Other OAuth 2.0 Best Practices

For additional best practices and detailed information please refer to the following source documents:

- RFC6749 - The OAuth 2.0 Authorization Framework (October 2012)
- RFC8252 - OAuth 2.0 for Native Apps (October 2017)
- RFC6819 - OAuth 2.0 Threat Model and Security Considerations (January 2013)

Reference

- owasp-mastg Other OAuth2 Best Practices

4.3.7 Rulebook

1. *Identify JWT libraries in use and whether they have known vulnerabilities (Required)*
2. *Ensure tokens are securely stored on mobile devices by Keychain (iOS) (Required)*
3. *Need to request the expected hash algorithm (Required)*
4. *Set an appropriate period for refresh token expiration (Required)*
5. *Adherence to OAuth 2.0 best practices (Required)*
6. *Main recommended libraries used for OAuth2 authentication (Recommended)*
7. *When using an external user agent for OAuth2 authentication, understand the advantages and disadvantages and use it (Recommended)*

4.3.7.1 Identify JWT libraries in use and whether they have known vulnerabilities (Required)

The following information should be reviewed to determine if there are any vulnerabilities.

- Identify the JWT libraries you are using and find out whether the JWT libraries in use have any known vulnerabilities.
- Ensure that the HMAC is verified for all incoming requests containing tokens.
- Verify the location of the private signing key or HMAC private key. Keys should be stored on the server and never shared with clients. It should only be available to the issuer and verifier.
- Ensure that no sensitive data, such as personally identifiable information, is embedded in the JWT. For architectures that need to transmit such information in the token for any reason, ensure that payload encryption is applied. See the sample Java implementation in OWASP JWT Cheat Sheet.

- Ensure that replay attacks are addressed with jti (JWT ID) claims that give the JWT a unique identifier.
- Ensure that cross-service relay attacks are addressed with aud (audience) claims that define which application the token is for.
- Ensure that the token is securely stored on the mobile device by means of Keychain (iOS).

The sample code below is an example of processing to retrieve a token from Keychain and save it to Keychain if the token does not exist in Keychain.

```
import Foundation

class Keychain {

    let group: String = "group_1" // group
    var id: String = "id"

    let backgroundQueue = DispatchQueue.global(qos: .userInitiated)

    func addKeychain(token: Data) {

        // Argument settings when executing the API
        let dic: [String: Any] = [kSecClass as String: kSecClassGenericPassword, //
↪ Class:Password Class
            kSecAttrGeneric as String: group,                // optional item
            kSecAttrAccount as String: id,                  // Account (Login ID)
            kSecValueData as String: token,                 // Password and other␣
↪stored information
            kSecAttrService as String: "key"]             // service name

        // Dictionary for search
        let search: [String: Any] = [kSecClass as String: kSecClassGenericPassword,
                                     kSecAttrService as String: "key",
                                     kSecReturnAttributes as String:␣
↪kCFBooleanTrue as Any,
                                     kSecMatchLimit as String:␣
↪kSecMatchLimitOne]as [String : Any]

        // Search data from keychain
        findKeychainItem(attributes: search as CFDictionary, { status, item in
            switch status {
                case errSecItemNotFound: // No data exists in keychain
                    // Save to keychain
                    _ = SecItemAdd(dic as CFDictionary, nil)
                default:
                    break
            }
        })

    }

    /// Search for the existence of an item from Keychain
    /// - Parameters:
    ///   - attrs: Data for search
    ///   - completion: search results
    func findKeychainItem(attributes attrs: CFDictionary, _ completion: @escaping␣
↪(OSStatus, CFTypeRef?) -> Void) {

        // The application UI may hang when called from the main thread because␣
↪the calling thread is blocked.
        // Recommended to run in a separate thread.
        backgroundQueue.async {
            var item: CFTypeRef?
```

(continues on next page)

(continued from previous page)

```
            let result = SecItemCopyMatching(attrs, &item)
            completion(result, item)
        }
    }

}
```

If this is violated, the following may occur.

- A vulnerability related to the JWT library used is exploited.

4.3.7.2 Ensure tokens are securely stored on mobile devices by Keychain (iOS) (Required)

Tokens must be securely stored on the mobile device by Keychain. Failure to do so may allow server authentication using tokens by a third party.

Refer to the following rules for how to use Keychain.

Data Storage and_Privacy Requirements RuleBook

- *Securely store values using the Keychain Services API (Required)*

If this is violated, the following may occur.

- Server authentication using a token by a third party becomes possible.

4.3.7.3 Need to request the expected hash algorithm (Required)

An attacker may modify the token and change the signature algorithm using the "none" keyword to indicate that the integrity of the token has already been verified. The following is an implementation of JWT decoding using JWTDecode.swift.

```
import JWTDecode

class JWTSample {
    func jwtSample() {
        // JWT
        let jwtText = "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
↪eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.
↪SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c"

        // Extract header and body values
        guard let jwt = try? decode(jwt: jwtText),
              let algorithm = jwt.header["alg"],
              let type = jwt.header["typ"],
              let subject = jwt.body["sub"] as? String,
              let name = jwt.body["name"] as? String,
              let issuedAt = jwt.body["iat"] as? Int else {
            return
        }
    }
}
```

Podfile:

```
# Uncomment the next line to define a global platform for your project
# platform :ios, '9.0'

target 'SampleApp' do
  # Comment the next line if you don't want to use dynamic frameworks
  use_frameworks!
```

(continues on next page)

(continued from previous page)

```
  # Pods for SampleApp

  target 'SampleAppTests' do
    # Pods for testing
  end

  pod 'JWTDecode'

end
```

If this is violated, the following may occur.

- Treat tokens signed with the ” none ” algorithm as if they were valid tokens with verified signatures, and applications may trust requests for modified tokens.

4.3.7.4 Set an appropriate period for refresh token expiration (Required)

It is necessary to ensure that the refresh tokens are valid for the appropriate period of time.If the refresh token is within the expiration date, a new access token should be issued; if it is outside the expiration date, there is a need to deny access and prompt the user to log in again.

If this is violated, the following may occur.

- If the expiration date is inappropriate, there is a possibility that the login user can be used by a third party.

4.3.7.5 Adherence to OAuth 2.0 best practices (Required)

Adhere to the following OAuth 2.0 best practices

User agent

- Users should have a way to visually confirm trust (e.g. Transport Layer Security (TLS) confirmation, website mechanisms).
- To prevent man-in-the-middle attacks, the client must verify the fully qualified domain name of the server with the public key presented by the server when establishing the connection.

Type of grant

- Native apps should use code assignment rather than implicit assignment.
- If code assignment is used, PKCE (Proof Key for Code Exchange) must be implemented to protect the code assignment. Make sure it is also implemented on the server side.
- Authentication “codes” should have a short expiration date and be used immediately upon receipt. Ensure that the authentication code exists only in temporary memory and is not stored or logged.

Client secret

- The client should not use the shared secret to prove the client’ s identity, since it can be spoofed (“client_id” already serves as proof). If the client secret is used, make sure it is stored in secure local storage.

End User Authentication Information

- Transport layer methods such as TLS protect the transmission of end-user authentication information.

token

- The access token is stored in temporary memory.
- The access token must be sent over an encrypted connection.
- Reduce the scope and duration of the access token if end-to-end confidentiality cannot be guaranteed or if the token provides access to sensitive information or transactions.
- Note that if the application uses the access token as a bearer token and there is no other way to identify the client, an attacker who steals the token will have access to its scope and all resources associated with it.

- Refresh tokens should be stored in secure local storage.

If this is violated, the following may occur.

- Potentially a vulnerable OAuth 2.0 implementation.

4.3.7.6 Main recommended libraries used for OAuth2 authentication (Recommended)

It is important not to develop your own OAuth2 authentication implementation.
The main libraries used to implement OAuth2 authentication are

- GTMAppAuth (Google)

gtm-oauth2 (Google) is now deprecated. As an alternative, use GTMAppAuth (Google).

Podfile Specification:

```
platform :ios, '10.0'

target 'helloworld' do
  use_frameworks!
  project 'helloworld'

  # In production, you would use:
  pod 'GTMAppAuth', '~> 2.0'

  pod 'AppAuth', '~> 1.0'
  pod 'GTMSessionFetcher/Core', '>= 1.0', '< 4.0'
end
```

Authentication Flow

To initiate the authorization request, use the authStateByPresentingAuthorizationRequest function. Within this function, an OAuth token exchange is automatically performed, all of which is PKCE-protected behavior (if server support is provided). The return value of this function is an OIDExternalUserAgentSession instance.

In the authentication flow, a redirect URI (authentication response URL) for authorization comes across in a custom URL schema. It is necessary to set the authorization response URL to the OIDExternalUserAgentSession instance obtained in the authentication flow.

The sample code below is an example of OAuth2 authentication with gtm-oauth2.

```
- (IBAction)authWithAutoCodeExchange:(nullable id)sender {
  NSURL *issuer = [NSURL URLWithString:kIssuer];
  NSURL *redirectURI = [NSURL URLWithString:kRedirectURI];

  // discovers endpoints
  [OIDAuthorizationService discoverServiceConfigurationForIssuer:issuer
      completion:^(OIDServiceConfiguration *_Nullable configuration, NSError *_
↪Nullable error) {

    if (!configuration) {
      [self setGtmAuthorization:nil];
      return;
    }

    // builds authentication request
    OIDAuthorizationRequest *request =
        [[OIDAuthorizationRequest alloc] initWithConfiguration:configuration
                                                      clientId:kClientID
                                                        scopes:@[OIDScopeOpenID,␣
↪OIDScopeProfile]
                                                   redirectURL:redirectURI
                                                  responseType:OIDResponseTypeCode
```

(continues on next page)

(continued from previous page)

```
                                                additionalParameters:nil];
    // performs authentication request
    AppDelegate *appDelegate = (AppDelegate *)[UIApplication sharedApplication].
↪delegate;

    appDelegate.currentAuthorizationFlow =
        [OIDAuthState authStateByPresentingAuthorizationRequest:request
            presentingViewController:self
                            callback:^(OIDAuthState *_Nullable authState,
                                       NSError *_Nullable error) {
      if (authState) {
        GTMAppAuthFetcherAuthorization *authorization =
          [[GTMAppAuthFetcherAuthorization alloc] initWithAuthState:authState];

        [self setGtmAuthorization:authorization];
        [self logMessage:@"Got authorization tokens. Access token: %@",
                         authState.lastTokenResponse.accessToken];
      } else {
        [self setGtmAuthorization:nil];
        [self logMessage:@"Authorization error: %@", [error localizedDescription]];
      }
    }];
  }];
}


// AppDelegate
- (BOOL)application:(UIApplication *)app
            openURL:(NSURL *)url
            options:(NSDictionary<NSString *, id> *)options {
  // Sends the URL to the current authorization flow (if any) which will process␣
↪it if it relates to
  // an authorization response.
  if ([_currentAuthorizationFlow resumeExternalUserAgentFlowWithURL:url]) {
    _currentAuthorizationFlow = nil;
    return YES;
  }

  // Your additional URL handling (if any) goes here.

  return NO;
}
```

Keychain storage for authorization instances

GTMAppAuthFetcherAuthorization Easily save an instance of OIDExternalUserAgentSession to the Keychain using the included functionality.
Saved instances can also be easily restored.

Saving and Deleting Keychains:

```
- (void)saveState {

    if (_authorization.canAuthorize) {
        // save
        [GTMAppAuthFetcherAuthorization saveAuthorization:_authorization␣
↪toKeychainForName:kAuthorizerKey];

    } else {
        // remove
        [GTMAppAuthFetcherAuthorization␣
```

(continues on next page)

(continued from previous page)

```
↪removeAuthorizationFromKeychainForName:kAuthorizerKey];

    }

}
```

Read from Keychain:

```
- (void)loadState {

    // Restore from Keychain
    GTMAppAuthFetcherAuthorization *authorization =␣
↪[GTMAppAuthFetcherAuthorization authorizationFromKeychainForName:kAuthorizerKey];
    [self setGtmAuthorization:authorization];
}
```

Example of overall implementation

AppDelegate.h:

```
#import <UIKit/UIKit.h>

@protocol OIDExternalUserAgentSession;

@interface AppDelegate : UIResponder <UIApplicationDelegate>

@property(nonatomic, strong, nullable) id<OIDExternalUserAgentSession>␣
↪currentAuthorizationFlow;

@end
```

AppDelegate.m:

```
#import "AppDelegate.h"

@import AppAuth;

#import "GTMAppAuthExampleViewController.h"

@implementation AppDelegate

@synthesize window = _window;

- (BOOL)application:(UIApplication *)application
    didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {
  UIWindow *window = [[UIWindow alloc] initWithFrame:[UIScreen mainScreen].bounds];
  UIViewController *mainViewController =
      [[GTMAppAuthExampleViewController alloc] init];
  window.rootViewController = mainViewController;

  _window = window;
  [_window makeKeyAndVisible];

  return YES;
}

- (BOOL)application:(UIApplication *)app
            openURL:(NSURL *)url
            options:(NSDictionary<NSString *, id> *)options {
  // Sends the URL to the current authorization flow (if any) which will process␣
↪it if it relates to
```

(continues on next page)

(continued from previous page)

```
  // an authorization response.
  if ([_currentAuthorizationFlow resumeExternalUserAgentFlowWithURL:url]) {
    _currentAuthorizationFlow = nil;
    return YES;
  }

  // Your additional URL handling (if any) goes here.

  return NO;
}

- (BOOL)application:(UIApplication *)application
            openURL:(NSURL *)url
  sourceApplication:(NSString *)sourceApplication
         annotation:(id)annotation {

  return [self application:application
                   openURL:url
                   options:@{}];
}

@end
```

GTMAppAuthExampleViewController.h:

```
#import <UIKit/UIKit.h>

@class OIDAuthState;
@class GTMAppAuthFetcherAuthorization;
@class OIDServiceConfiguration;

NS_ASSUME_NONNULL_BEGIN

/*! @brief The example application's view controller.
 */
@interface GTMAppAuthExampleViewController : UIViewController

/*! @brief The authorization state.
 */
@property(nonatomic, nullable) GTMAppAuthFetcherAuthorization *authorization;

- (IBAction)authWithAutoCodeExchange:(nullable id)sender;

- (IBAction)userinfo:(nullable id)sender;

- (IBAction)clearAuthState:(nullable id)sender;


@end

NS_ASSUME_NONNULL_END
```

GTMAppAuthExampleViewController.m

```
#import "GTMAppAuthExampleViewController.h"
#import <QuartzCore/QuartzCore.h>

@import AppAuth;
@import GTMSessionFetcher;
@import GTMAppAuth;
```

(continues on next page)

(continued from previous page)

```
#import "AppDelegate.h"

static NSString *const kIssuer = @"https://accounts.google.com";

static NSString *const kClientID = @"xxxxx.apps.googleusercontent.com";


static NSString *const kRedirectURI =
    @"com.googleusercontent.apps.YOUR_CLIENT:/oauthredirect";

static NSString *const kAuthorizerKey = @"authorizationkey";

@interface GTMAppAuthExampleViewController () <OIDAuthStateChangeDelegate,
                                               OIDAuthStateErrorDelegate>
@end

@implementation GTMAppAuthExampleViewController

- (void)viewDidLoad {
  [super viewDidLoad];

  [self loadState];
}


- (void)saveState {

  if (_authorization.canAuthorize) {
      [GTMAppAuthFetcherAuthorization saveAuthorization:_authorization␣
↪toKeychainForName:kAuthorizerKey];

  } else {
      [GTMAppAuthFetcherAuthorization␣
↪removeAuthorizationFromKeychainForName:kAuthorizerKey];

  }

}

- (void)loadState {

    // Restore from Keychain
    GTMAppAuthFetcherAuthorization *authorization =␣
↪[GTMAppAuthFetcherAuthorization authorizationFromKeychainForName:kAuthorizerKey];
    [self setGtmAuthorization:authorization];
}

- (void)setGtmAuthorization:(GTMAppAuthFetcherAuthorization*)authorization {
    if ([_authorization isEqual:authorization]) {
        return;
    }

    _authorization = authorization;
    [self stateChanged];
}

- (void)stateChanged {
  [self saveState];
}

- (void)didChangeState:(OIDAuthState *)state {
```

(continues on next page)

(continued from previous page)

```
  [self stateChanged];
}

- (void)authState:(OIDAuthState *)state didEncounterAuthorizationError:(NSError␣
↪*)error {
    // error
}

- (IBAction)authWithAutoCodeExchange:(nullable id)sender {
  NSURL *issuer = [NSURL URLWithString:kIssuer];
  NSURL *redirectURI = [NSURL URLWithString:kRedirectURI];

  // discovers endpoints
  [OIDAuthorizationService discoverServiceConfigurationForIssuer:issuer
      completion:^(OIDServiceConfiguration *_Nullable configuration, NSError *_
↪Nullable error) {

    if (!configuration) {
      [self setGtmAuthorization:nil];
      return;
    }

    // builds authentication request
    OIDAuthorizationRequest *request =
        [[OIDAuthorizationRequest alloc] initWithConfiguration:configuration
                                                      clientId:kClientID
                                                        scopes:@[OIDScopeOpenID,␣
↪OIDScopeProfile]
                                                   redirectURL:redirectURI
                                                  responseType:OIDResponseTypeCode
                                            additionalParameters:nil];
    // performs authentication request
    AppDelegate *appDelegate = (AppDelegate *)[UIApplication sharedApplication].
↪delegate;

    appDelegate.currentAuthorizationFlow =
        [OIDAuthState authStateByPresentingAuthorizationRequest:request
            presentingViewController:self
                            callback:^(OIDAuthState *_Nullable authState,
                                       NSError *_Nullable error) {
      if (authState) {
        GTMAppAuthFetcherAuthorization *authorization =
          [[GTMAppAuthFetcherAuthorization alloc] initWithAuthState:authState];

        [self setGtmAuthorization:authorization];
        [self logMessage:@"Got authorization tokens. Access token: %@",
                         authState.lastTokenResponse.accessToken];
      } else {
        [self setGtmAuthorization:nil];
        [self logMessage:@"Authorization error: %@", [error localizedDescription]];
      }
    }];
  }];
}

- (IBAction)clearAuthState:(nullable id)sender {
  [self setGtmAuthorization:nil];
}

- (IBAction)userinfo:(nullable id)sender {
  [self logMessage:@"Performing userinfo request"];
```

(continues on next page)

(continued from previous page)

```
  GTMSessionFetcherService *fetcherService = [[GTMSessionFetcherService alloc]␣
↪init];
  fetcherService.authorizer = self.authorization;

  // Creates a fetcher for the API call.
  NSURL *userinfoEndpoint = [NSURL URLWithString:@"https://www.googleapis.com/
↪oauth2/v3/userinfo"];
  GTMSessionFetcher *fetcher = [fetcherService fetcherWithURL:userinfoEndpoint];
  [fetcher beginFetchWithCompletionHandler:^(NSData *data, NSError *error) {

    // Checks for an error.
    if (error) {

      // OIDOAuthTokenErrorDomain indicates an issue with the authorization.
      if ([error.domain isEqual:OIDOAuthTokenErrorDomain]) {
        [self setGtmAuthorization:nil];
        [self logMessage:@"Authorization error during token refresh, clearing␣
↪state. %@", error];
      // Other errors are assumed transient.
      } else {
        [self logMessage:@"Transient error during token refresh. %@", error];
      }
      return;
    }

    // Parses the JSON response.
    NSError *jsonError = nil;
    id jsonDictionaryOrArray =
        [NSJSONSerialization JSONObjectWithData:data options:0 error:&jsonError];

    // JSON error.
    if (jsonError) {
      [self logMessage:@"JSON decoding error %@", jsonError];
      return;
    }

    // Success response!
    [self logMessage:@"Success: %@", jsonDictionaryOrArray];
  }];
}


- (void)logMessage:(NSString *)format, ... NS_FORMAT_FUNCTION(1,2) {
  // gets message as string
  va_list argp;
  va_start(argp, format);
  NSString *log = [[NSString alloc] initWithFormat:format arguments:argp];
  va_end(argp);

  // outputs to stdout
  NSLog(@"%@", log);
}

@end
```

If this is not noted, the following may occur.

- Potentially a vulnerable OAuth 2.0 implementation.

4.3.7.7 When using an external user agent for OAuth2 authentication, understand the advantages and disadvantages and use it (Recommended)

OAuth2 authentication can be performed via an external user agent (e.g. Chrome or Safari). The advantages and disadvantages of using an external user agent are as follows

Advantages

- The user's credentials are never disclosed directly to the app. For this reason, it is not possible for an app to obtain authentication information when logging in ("authentication information phishing").
- Since little or no authentication logic needs to be added to the application itself, coding errors can be avoided.

Disadvantage

- There is no way to control browser behavior (e.g., enable certificate pinning).

If this is not noted, the following may occur.

- Not available when certificate pinning needs to be enabled.

4.4 MSTG-AUTH-4

The remote endpoint terminates the existing session when the user logs out.

4.4.1 Destroying Remote Session Information

The purpose of this test case is verifying logout functionality and determining whether it effectively terminates the session on both client and server and invalidates a stateless token.

Failing to destroy the server-side session is one of the most common logout functionality implementation errors. This error keeps the session or token alive, even after the user logs out of the application. An attacker who gets valid authentication information can continue to use it and hijack a user's account.

Many mobile apps don't automatically log users out. There can be various reasons, such as: because it is inconvenient for customers, or because of decisions made when implementing stateless authentication. The application should still have a logout function, and it should be implemented according to best practices, destroying all locally stored tokens or session identifiers. If session information is stored on the server, it should also be destroyed by sending a logout request to that server. In case of a high-risk application, tokens should be invalidated. Not removing tokens or session identifiers can result in unauthorized access to the application in case the tokens are leaked. Note that other sensitive types of information should be removed as well, as any information that is not properly cleared may be leaked later, for example during a device backup.

4.4.2 Static Analysis

If server code is available, make sure logout functionality terminates the session correctly. This verification will depend on the technology. Here are different examples of session termination for proper server-side logout:

- Spring (Java)
- Ruby on Rails
- PHP

If access and refresh tokens are used with stateless authentication, they should be deleted from the mobile device. The refresh token should be invalidated on the server.

4.4.3 Dynamic Analysis

Use an interception proxy for dynamic application analysis and execute the following steps to check whether the logout is implemented properly:

1. Log in to the application.
2. Access a resource that requires authentication, typically a request for private information belonging to your account.
3. Log out of the application.
4. Try to access the data again by resending the request from step 2.

If the logout is correctly implemented on the server, an error message or redirect to the login page will be sent back to the client. On the other hand, if you receive the same response you got in step 2, the token or session ID is still valid and hasn't been correctly terminated on the server. The OWASP Web Testing Guide (WSTG-SESS-06) includes a detailed explanation and more test cases.

Reference

- owasp-mastg Testing User Logout

Rulebook

- *Best practices for discarding remote session information when login functionality is present in the app (Recommended)*

4.4.4 Rulebook

1. *Best practices for discarding remote session information when login functionality is present in the app (Recommended)*

4.4.4.1 Best practices for discarding remote session information when login functionality is present in the app (Recommended)

If login capability exists in the app, destroy remote session information according to the following best practices.

- Provide a logout function in the application.
- Discard all locally stored tokens and session identifiers upon logout.
- If session information is stored on the server, send a logout request to the server and destroy it.
- Disable tokens for high-risk applications.

Failure to destroy tokens and session identifiers could result in unauthorized access to the application if the tokens are leaked. Information that is not destroyed should be destroyed in the same manner as other sensitive information, as it could be leaked later, such as when backing up a device.

If this is not noted, the following may occur.

- Login information remains in memory and can be used by third parties.

4.5 MSTG-AUTH-5

A password policy exists and is enforced at the remote endpoint.

4.5.1 Password Policy Compliance

Password strength is a key concern when passwords are used for authentication. The password policy defines requirements to which end users should adhere. A password policy typically specifies password length, password complexity, and password topologies. A "strong" password policy makes manual or automated password cracking difficult or impossible. The following sections will cover various areas regarding password best practices. For further information please consult the OWASP Authentication Cheat Sheet.

Reference

- owasp-mastg Testing Best Practices for Passwords

4.5.2 Static Analysis

Confirm the existence of a password policy and verify the implemented password complexity requirements according to the OWASP Authentication Cheat Sheet which focuses on length and an unlimited character set. Identify all password-related functions in the source code and make sure that a verification check is performed in each of them. Review the password verification function and make sure that it rejects passwords that violate the password policy.

Reference

- owasp-mastg Testing Best Practices for Passwords Static Analysis

Rulebook

- *A "strong" password policy (Recommended)*

4.5.2.1 zxcvbn

zxcvbn is a common library that can be used for estimating password strength, inspired by password crackers. It is available in JavaScript but also for many other programming languages on the server side. There are different methods of installation, please check the Github repo for your preferred method. Once installed, zxcvbn can be used to calculate the complexity and the amount of guesses to crack the password.

After adding the zxcvbn JavaScript library to the HTML page, you can execute the command zxcvbn in the browser console, to get back detailed information about how likely it is to crack the password including a score.

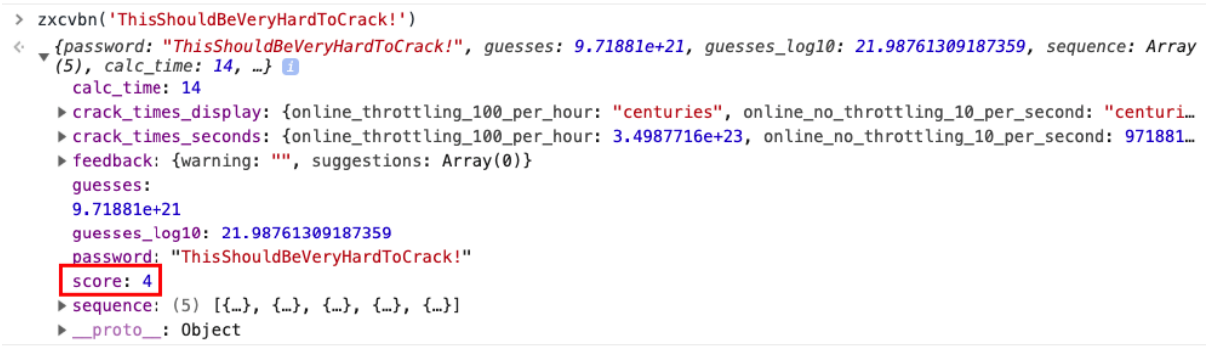


Fig 4.5.2.1.1 zxcvbn Command Example

The score is defined as follows and can be used for a password strength bar for example:

```
0 # too guessable: risky password. (guesses < 10^3)

1 # very guessable: protection from throttled online attacks. (guesses < 10^6)

2 # somewhat guessable: protection from unthrottled online attacks. (guesses < 10^
↪8)

3 # safely unguessable: moderate protection from offline slow-hash scenario.␣
↪(guesses < 10^10)

4 # very unguessable: strong protection from offline slow-hash scenario. (guesses >
↪= 10^10)
```

Note that zxcvbn can be implemented by the app-developer as well using the Java (or other) implementation in order to guide the user into creating a strong password.

Reference

- owasp-mastg Testing Best Practices for Passwords (MSTG-AUTH-5 and MSTG-AUTH-6) zxcvbn

4.5.3 Have I Been Pwned: PwnedPasswords

In order to further reduce the likelihood of a successful dictionary attack against a single factor authentication scheme (e.g. password only), you can verify whether a password has been compromised in a data breach. This can be done using services based on the Pwned Passwords API by Troy Hunt (available at api.pwnedpasswords.com). For example, the "Have I been pwned?" companion website. Based on the SHA-1 hash of a possible password candidate, the API returns the number of times the hash of the given password has been found in the various breaches collected by the service. The workflow takes the following steps:

- Encode the user input to UTF-8 (e.g.: the password test).
- Take the SHA-1 hash of the result of step 1 (e.g.: the hash of test is A94A8FE5CC⋯).
- Copy the first 5 characters (the hash prefix) and use them for a range-search by using the following API: http GET https://api.pwnedpasswords.com/range/A94A8
- Iterate through the result and look for the rest of the hash (e.g. is FE5CC⋯ part of the returned list?). If it is not part of the returned list, then the password for the given hash has not been found. Otherwise, as in case of FE5CC⋯, it will return a counter showing how many times it has been found in breaches (e.g.: FE5CC⋯:76479).

Further documentation on the Pwned Passwords API can be found online.

Note that this API is best used by the app-developer when the user needs to register and enter a password to check whether it is a recommended password or not.

Reference

- owasp-mastg Testing Best Practices for Passwords Have I Been Pwned: PwnedPasswords

Rulebook

- *Check if the password is recommended (Recommended)*

4.5.3.1 Login limit

Check the source code for a throttling procedure: a counter for logins attempted in a short period of time with a given user name and a method to prevent login attempts after the maximum number of attempts has been reached. After an authorized login attempt, the error counter should be reset.

Observe the following best practices when implementing anti-brute-force controls:

- After a few unsuccessful login attempts, targeted accounts should be locked (temporarily or permanently), and additional login attempts should be rejected.
- A five-minute account lock is commonly used for temporary account locking.
- The controls must be implemented on the server because client-side controls are easily bypassed.
- Unauthorized login attempts must be tallied with respect to the targeted account, not a particular session.

Additional brute force mitigation techniques are described on the OWASP page Blocking Brute Force Attacks.

Reference

- owasp-mastg Testing Best Practices for Passwords (MSTG-AUTH-5 and MSTG-AUTH-6) Login Throttling

Rulebook

- *Adhere to the following best practices for brute force protection (Required)*

4.5.4 Rulebook

1. *A "strong" password policy (Recommended)*
2. *Check if the password is recommended (Recommended)*
3. *Adhere to the following best practices for brute force protection (Required)*

4.5.4.1 A "strong" password policy (Recommended)

The following is an example of a policy for setting a "strong" password.

- Password Length
  - Minimum: Passwords of less than 8 characters are considered weak.
  - Maximum: The typical maximum length is 64 characters. It is necessary to set the maximum length to prevent long password Denial of Service attacks.
- Do not truncate passwords without notifying the user.
- Password configuration rulesAllow the use of all characters, including Unicode and whitespace.
- Credential RotationCredential rotation should be performed when a password compromise occurs or is identified.
- Password Strength MeterPrepare a password strength meter to block users from creating complex passwords and setting passwords that have been identified in the past.

The sample code below is an example of the maximum input limit and validation check process that should be supported in an iOS application.

Maximum input limit

```
import UIKit

class ClassSample {
    func textField(_ textField: UITextField, shouldChangeCharactersIn range:␣
↪NSRange, replacementString string: String) -> Bool {

        if let text = textField.text,
```

(continues on next page)

(continued from previous page)

```
            let textRange = Range(range, in: text) {
             let updatedText = text.replacingCharacters(in: textRange,with: string)
             if updatedText.count > 64{
                 return false
             }
         }
         return true
     }
}
```

Validation Check

```
import Foundation

class ValidateSample {
    func validate(text: String, with regex: String) -> Bool {
            // Length Check
            if text.count <= 64 || text.count >= 8 {
                return true
            }
            return false
    }
}
```

Reference

- OWASP CheatSheetSeries Implement Proper Password Strength Controls

If this is not noted, the following may occur.

- Predicted to be a weak password.

4.5.4.2 Check if the password is recommended (Recommended)

The Pwned Passwords API can be used to check if a password is recommended when a user registers or enters a password.

If this is not noted, the following may occur.

- Predicted to be a weak password.

4.5.4.3 Adhere to the following best practices for brute force protection (Required)

The following best practices should be followed as a brute force measure.

- After several failed login attempts, the target account should be locked (temporarily or permanently) and subsequent login attempts should be denied.
- A 5-minute account lock is commonly used for temporary account locks.
- Control should be on the server, as client-side controls are easily bypassed.
- Unauthorized login attempts should be aggregated for the target account, not for a specific session.

Additional brute force mitigation techniques are described on the OWASP page Blocking Brute Force Attacks.

* Sample code is not available, as these are server-side rules.

If this is violated, the following may occur.

- Become vulnerable to brute force attacks.

4.6 MSTG-AUTH-6

The remote endpoint implements a mechanism to protect against the submission of credentials an excessive number of times.

* This guide is omitted in this document because it is a chapter about support on the Remote Service side.

Reference

- owasp-mastg Dynamic Testing (MSTG-AUTH-6)

4.7 MSTG-AUTH-7

Sessions are invalidated at the remote endpoint after a predefined period of inactivity and access tokens expire.

* This guide is omitted in this document because it is a chapter about support on the Remote Service side.

Reference

- owasp-mastg Testing Session Timeout

4.8 MSTG-AUTH-12

Authorization models should be defined and enforced at the remote endpoint.

* This guide is omitted in this document because it is a chapter about support on the Remote Service side.

5 Network Communication Requirements

5.1 MSTG-NETWORK-1

Data is encrypted on the network using TLS. The secure channel is used consistently throughout the app.

5.1.1 Configuring secure network communication

All the presented cases must be carefully analyzed as a whole. For example, even if the app does not permit cleartext traffic in its Info.plist, it might actually still be sending HTTP traffic. That could be the case if it' s using a low-level API (for which ATS is ignored) or a badly configured cross-platform framework.

IMPORTANT: You should apply these tests to the app main code but also to any app extensions, frameworks or Watch apps embedded within the app as well.

For more information refer to the article "Preventing Insecure Network Connections" and "Fine-tune your App Transport Security settings" in the Apple Developer Documentation.

Reference

- owasp-mastg Testing Data Encryption on the Network (MSTG-NETWORK-1)

Rulebook

- *Use App Transport Security (ATS) (Required)*

5.1.2 Static Analysis

5.1.2.1 Verification of network requests with secure protocols

First, you should identify all network requests in the source code and ensure that no plain HTTP URLs are used. Make sure that sensitive information is sent over secure channels by using URLSession (which uses the standard URL Loading System from iOS) or Network (for socket-level communication using TLS and access to TCP and UDP).

Reference

- owasp-mastg Testing Data Encryption on the Network (MSTG-NETWORK-1) Testing Network Requests over Secure Protocols

Rulebook

- *Using URLSession (Recommended)*
- *Using the Network Framework (Recommended)*

5.1.2.2 Check for Low-Level Networking API Usage

Identify the network APIs used by the app and see if it uses any low-level networking APIs.

Apple Recommendation: Prefer High-Level Frameworks in Your App: "ATS doesn't apply to calls your app makes to lower-level networking interfaces like the Network framework or CFNetwork. In these cases, you take responsibility for ensuring the security of the connection. You can construct a secure connection this way, but mistakes are both easy to make and costly. It's typically safest to rely on the URL Loading System instead" (see source).

If the app uses any low-level APIs such as Network or CFNetwork, you should carefully investigate if they are being used securely. For apps using cross-platform frameworks (e.g. Flutter, Xamarin, ⋯) and third party frameworks (e.g. Alamofire) you should analyze if they're being configured and used securely according to their best practices.

Make sure that the app:

- verifies the challenge type and the host name and credentials when performing server trust evaluation.
- doesn't ignore TLS errors.
- doesn't use any insecure TLS configurations (see "Testing the TLS Settings (MSTG-NETWORK-2)")

These checks are orientative, we cannot name specific APIs since every app might use a different framework. Please use this information as a reference when inspecting the code.

Reference

- owasp-mastg Testing Data Encryption on the Network (MSTG-NETWORK-1) Check for Low-Level Networking API Usage

Rulebook

- *Use App Transport Security (ATS) (Required)*
- *Use CFNetwork (Deprecated)*
- *Using the Network Framework (Recommended)*
- *Use low-level networking APIs safely according to best practices (Required)*

5.1.2.3 Testing for Cleartext Traffic

Ensure that the app is not allowing cleartext HTTP traffic. Since iOS 9.0 cleartext HTTP traffic is blocked by default (due to App Transport Security (ATS)) but there are multiple ways in which an application can still send it:

- Configuring ATS to enable cleartext traffic by setting the NSAllowsArbitraryLoads attribute to true (or YES) on NSAppTransportSecurity in the app's Info.plist.
- Retrieve the Info.plist
- Check that NSAllowsArbitraryLoads is not set to true globally of for any domain.
- If the application opens third party web sites in WebViews, then from iOS 10 onwards NSAllowsArbitraryLoadsInWebContent can be used to disable ATS restrictions for the content loaded in web views.

Apple warns: Disabling ATS means that unsecured HTTP connections are allowed. HTTPS connections are also allowed, and are still subject to default server trust evaluation. However, extended security checks—like requiring a minimum Transport Layer Security (TLS) protocol version—are disabled. Without ATS, you're also free to loosen the default server trust requirements, as described in "Performing Manual Server Trust Authentication".

The following snippet shows a vulnerable example of an app disabling ATS restrictions globally.

```
<key>NSAppTransportSecurity</key>
<dict>
    <key>NSAllowsArbitraryLoads</key>
    <true/>
</dict>
```

ATS should be examined taking the application's context into consideration. The application may have to define ATS exceptions to fulfill its intended purpose. For example, the Firefox iOS application has ATS disabled globally. This exception is acceptable because otherwise the application would not be able to connect to any HTTP website that does not have all the ATS requirements. In some cases, apps might disable ATS globally but enable it for certain domains to e.g. securely load metadata or still allow secure login.

ATS should include a justification string for this (e.g. "The app must connect to a server managed by another entity that doesn't support secure connections.").

Reference

- owasp-mastg Testing Data Encryption on the Network (MSTG-NETWORK-1) Testing for Cleartext Traffic

Rulebook

- *Use App Transport Security (ATS) (Required)*

5.1.3 Dynamic Analysis

Intercept the tested app's incoming and outgoing network traffic and make sure that this traffic is encrypted. You can intercept network traffic in any of the following ways:

- Capture all HTTP(S) and Websocket traffic with an interception proxy like OWASP ZAP or Burp Suite and make sure all requests are made via HTTPS instead of HTTP.
- Interception proxies like Burp and OWASP ZAP will show HTTP(S) traffic only. You can, however, use a Burp plugin such as Burp-non-HTTP-Extension or the tool mitm-relay to decode and visualize communication via XMPP and other protocols.

Some applications may not work with proxies like Burp and OWASP ZAP because of Certificate Pinning. In such a scenario, please check "Testing Custom Certificate Stores and Certificate Pinning".

For more details refer to:

- "Intercepting Traffic on the Network Layer" from chapter "Testing Network Communication"
- "Setting up a Network Testing Environment" from chapter iOS Basic Security Testing

Reference

- owasp-mastg Testing Data Encryption on the Network (MSTG-NETWORK-1) Dynamic Analysis

Rulebook

- *Use App Transport Security (ATS) (Required)*

5.1.4 Rulebook

1. *Use App Transport Security (ATS) (Required)*
2. *Using URLSession (Recommended)*
3. *Use CFNetwork (Deprecated)*
4. *Using the Network Framework (Recommended)*
5. *Use low-level networking APIs safely according to best practices (Required)*

5.1.4.1 Use App Transport Security (ATS) (Required)

The ATS requires the use of HTTPS for all HTTP connections. In addition, it imposes extended security checks that complement the default server trust rating specified by the Transport Layer Security (TLS) protocol; ATS blocks connections that do not meet minimum security specifications.

In order to ensure that the data used within an application is as secure as possible, it is important to first understand if the connection is currently unsecured.

To check, set the Allow Arbitrary Loads attribute to "NO" in the App Transport Security Settings of the Info.plist to disable all active ATS exceptions. If an application makes an unsecured connection, a runtime error will occur on Xcode for each connection.

ATS is effective when:

- App Transport Security Settings is not specified in Info.plist.
- In Info.plist, "NO" is specified for "Allow Arbitrary Loads" attribute, "Allows Arbitrary Loads for Media" attribute, "Allows Arbitrary Loads in Web Content" attribute, and "NSExceptionAllowsInsecureHTTPLoads" attribute, and "NSExceptionRequiresForwardSecrecy" attribute is "YES" is specified and 1.2 or higher is specified in the "NSExceptionMinimumTLSVersion" attribute.

info.plist setting on Xcode:

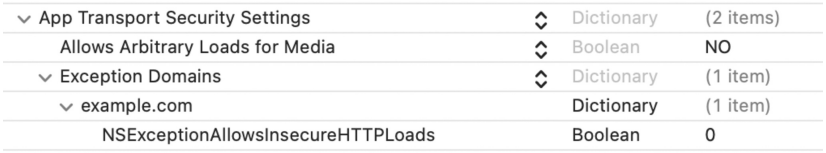


Fig 5.1.4.1.1 Example of exclusion by ATS

If this is violated, the following may occur.

- May implement connections that do not meet security specifications.

5.1.4.2 Using URLSession (Recommended)

To manipulate URLs and communicate with servers using standard Internet protocols, iOS provides a URL Loading System. URLSession is used for this purpose.

```
import UIKit

class Fetch {
    func getDataAPI(completion: @escaping (Any) -> Void) {
        let requestUrl = URL(string: "http://xxxxx")!
            // Use URLSession's datatask to acquire data.
        let task = URLSession.shared.dataTask(with: requestUrl) { data, response,␣
↪error in
                // Process when an error is returned.
            if let error = error {
                completion(error)
            } else if let data = data {
                completion(data)
            }
        }
        task.resume()
    }
}
```

If this is not noted, the following may occur.

- Sensitive information may be transmitted over insecure channels.

5.1.4.3 Use CFNetwork (Deprecated)

CFNetwork is an API for BSD socket handling, HTTP and FTP server communication provided in iOS 2.0.

Most of them are now deprecated. URLSession (using the standard iOS URL Loading System) is recommended for ATS to function effectively.

The reasons for this deprecation are as follows.

- Sensitive information may be transmitted over insecure channels.

5.1.4.4 Using the Network Framework (Recommended)

Use when direct access to TLS, TCP, UDP or proprietary application protocols is required. Use URLSession as before for HTTP(S) and URL-based resource loading.

```
import Foundation
import Network

class NetworkUDP {
    // Constant
    let networkType = "_myApp._udp."
    let networkDomain = "local"

    private func startListener(name: String) {
        // Create listener with udp for using
        guard let listener = try? NWListener(using: .udp, on: 11111) else {␣
↪fatalError() }

        listener.service = NWListener.Service(name: name, type: networkType)

        let listnerQueue = DispatchQueue(label: "com.myapp.queue.listener")

        // Processing new connection visits
        listener.newConnectionHandler = { [unowned self] (connection:␣
↪NWConnection) in
            connection.start(queue: listnerQueue)
            self.receive(on: connection)
        }

        // Listener start
        listener.start(queue: listnerQueue)

    }

    private func receive(on connection: NWConnection) {
        /* Data reception */
        connection.receive(minimumIncompleteLength: 0,
                           maximumLength: 65535,
                           completion:{(data, context, flag, error) in
            if let error = error {
                NSLog("\(#function), \(error)")
            } else {
                if data != nil {
                    /* Deserialization of incoming data */

                }
                else {
                    NSLog("receiveMessage data nil")
                }
            }
        })
    }
```

(continues on next page)

(continued from previous page)

```
}
```

If this is not noted, the following may occur.

- Sensitive information may be transmitted over insecure channels.

5.1.4.5 Use low-level networking APIs safely according to best practices (Required)

If your app uses low-level APIs, you need to carefully investigate whether they are being used safely. If the app uses cross-platform frameworks (e.g. Flutter, Xamarin) or third-party frameworks (e.g. Alamofire), it should be analyzed to ensure that they are being configured and used safely according to best practices.

Ensure that the app complies with the following: * Verify that the app is compliant with the

- Validate challenge type and hostname and credentials when performing server reliability assessments.
- Do not ignore TLS errors.
- Not using insecure TLS settings.（" See Verify TLS settings (MSTG-NETWORK-2)"）

These checks are for reference only and cannot name specific APIs, as each application may use a different framework. They should be used as reference information when examining the code.

If this is violated, the following may occur.

- The app may conduct insecure communication.

5.2 MSTG-NETWORK-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.

5.2.1 Recommended TLS Settings

Ensuring proper TLS configuration on the server side is also important. The SSL protocol is deprecated and should no longer be used. Also TLS v1.0 and TLS v1.1 have known vulnerabilities and their usage is deprecated in all major browsers by 2020. TLS v1.2 and TLS v1.3 are considered best practice for secure transmission of data.

When both the client and server are controlled by the same organization and used only for communicating with one another, you can increase security by hardening the configuration.

If a mobile application connects to a specific server, its networking stack can be tuned to ensure the highest possible security level for the server's configuration. Lack of support in the underlying operating system may force the mobile application to use a weaker configuration.

Remember to inspect the corresponding justifications to discard that it might be part of the app intended purpose.

Some examples of justifications eligible for consideration are:

- The app must connect to a server managed by another entity that doesn't support secure connections.
- The app must support connecting to devices that cannot be upgraded to use secure connections, and that must be accessed using public host names.
- The app must display embedded web content from a variety of sources, but can't use a class supported by the web content exception.
- The app loads media content that is encrypted and that contains no personalized information.

When submitting your app to the App Store, provide sufficient information for the App Store to determine why your app cannot make secure connections by default.

It is possible to verify which ATS settings can be used when communicating to a certain endpoint. On macOS the command line utility nscurl can be used. A permutation of different settings will be executed and verified against the specified endpoint. If the default ATS secure connection test is passing, ATS can be used in its default secure configuration. If there are any fails in the nscurl output, please change the server side configuration of TLS to make the server side more secure, rather than weakening the configuration in ATS on the client. See the article "Identifying the Source of Blocked Connections" in the Apple Developer Documentation for more details.

Refer to section "Verifying the TLS Settings" in chapter Testing Network Communication for details.

Reference

- owasp-mastg Verifying the TLS Settings (MSTG-NETWORK-2) Recommended TLS Settings
- owasp-mastg Testing the TLS Settings (MSTG-NETWORK-2)

Rulebook

- *Secure communication protocol (Required)*
- *Use App Transport Security (ATS) (Required)*

5.2.2 Recommended Cipher Suites

Cipher suites have the following structure:

```
Protocol_KeyExchangeAlgorithm_WITH_BlockCipher_IntegrityCheckAlgorithm
```

This structure includes:

- A Protocol used by the cipher
- A Key Exchange Algorithm used by the server and the client to authenticate during the TLS handshake
- A Block Cipher used to encrypt the message stream
- A Integrity Check Algorithm used to authenticate messages

Example: TLS_RSA_WITH_3DES_EDE_CBC_SHA

In the example above the cipher suites uses:

- TLS as protocol
- RSA Asymmetric encryption for Authentication
- 3DES for Symmetric encryption with EDE_CBC mode
- SHA Hash algorithm for integrity

Note that in TLSv1.3 the Key Exchange Algorithm is not part of the cipher suite, instead it is determined during the TLS handshake.

In the following listing, we' ll present the different algorithms of each part of the cipher suite.

Protocols:

- SSLv1
- SSLv2 - RFC 6176
- SSLv3 - RFC 6101
- TLSv1.0 - RFC 2246
- TLSv1.1 - RFC 4346
- TLSv1.2 - RFC 5246
- TLSv1.3 - RFC 8446

Key Exchange Algorithms:

- DSA - RFC 6979
- ECDSA - RFC 6979
- RSA - RFC 8017
- DHE - RFC 2631 - RFC 7919
- ECDHE - RFC 4492
- PSK - RFC 4279
- DSS - FIPS186-4
- DH_anon - RFC 2631 - RFC 7919
- DHE_RSA - RFC 2631 - RFC 7919
- DHE_DSS - RFC 2631 - RFC 7919
- ECDHE_ECDSA - RFC 8422
- ECDHE_PSK - RFC 8422 - RFC 5489
- ECDHE_RSA - RFC 8422

Block Ciphers:

- DES - RFC 4772
- DES_CBC - RFC 1829
- 3DES - RFC 2420
- 3DES_EDE_CBC - RFC 2420
- AES_128_CBC - RFC 3268
- AES_128_GCM - RFC 5288
- AES_256_CBC - RFC 3268
- AES_256_GCM - RFC 5288
- RC4_40 - RFC 7465
- RC4_128 - RFC 7465
- CHACHA20_POLY1305 - RFC 7905 - RFC 7539

Integrity Check Algorithms:

- MD5 - RFC 6151
- SHA - RFC 6234
- SHA256 - RFC 6234
- SHA384 - RFC 6234

Note that the efficiency of a cipher suite depends on the efficiency of its algorithms.

The following resources contain the latest recommended cipher suites to use with TLS:

- IANA recommended cipher suites can be found in TLS Cipher Suites.
- OWASP recommended cipher suites can be found in the TLS Cipher String Cheat Sheet.

Some iOS versions do not support some of the recommended cipher suites, so for compatibility purposes you can check the supported cipher suites for iOS versions and choose the top supported cipher suites.

If you want to verify whether your server supports the right cipher suites, there are various tools you can use:

- nscurl - see iOS Network Communication for more details.

- testssl.sh which "is a free command line tool which checks a server' s service on any port for the support of TLS/SSL ciphers, protocols as well as some cryptographic flaws" .

Finally, verify that the server or termination proxy at which the HTTPS connection terminates is configured according to best practices. See also the OWASP Transport Layer Protection cheat sheet and the Qualys SSL/TLS Deployment Best Practices.

Reference

- owasp-mastg Verifying the TLS Settings (MSTG-NETWORK-2) Cipher Suites Terminology

Rulebook

- *Recommended cipher suites for TLS (Recommended)*

5.2.3 Rulebook

1. *Secure communication protocol (Required)*
2. *Recommended cipher suites for TLS (Recommended)*

5.2.3.1 Secure communication protocol (Required)

Ensuring proper TLS configuration on the server side is also important. The SSL protocol is deprecated and should no longer be used.

Deprecated Protocols

- SSL
- TLS v1.0
- TLS v1.1

TLS v1.0 and TLS v1.1 have been deprecated in all major browsers by 2020.

Recommended Protocols

- TLS v1.2
- TLS v1.3

If this is violated, the following may occur.

- Vulnerable to security exploits.

5.2.3.2 Recommended cipher suites for TLS (Recommended)

The following is an example of a recommended cipher suite. (Lists the cipher suites defined by iOS among those recommended by TLS Cipher Suites.)

- TLS_DHE_RSA_WITH_AES_128_GCM_SHA256
- TLS_DHE_RSA_WITH_AES_256_GCM_SHA384
- TLS_DHE_PSK_WITH_AES_128_GCM_SHA256
- TLS_DHE_PSK_WITH_AES_256_GCM_SHA384
- TLS_AES_128_GCM_SHA256
- TLS_AES_256_GCM_SHA384
- TLS_CHACHA20_POLY1305_SHA256
- TLS_AES_128_CCM_SHA256
- TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256
- TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384

- TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
- TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
- TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256
- TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305_SHA256

If this is not noted, the following may occur.

- Potential use of vulnerable cipher suites.

5.3 MSTG-NETWORK-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.

* The description of certificate validation is summarized in "*Configuring secure network communication*" and is omitted from this chapter.

6 Platform Interaction Requirements

6.1 MSTG-PLATFORM-1

The app only requests the minimum set of permissions necessary.

In contrast to Android, where each app runs on its own user ID, iOS makes all third-party apps run under the non-privileged mobile user. Each app has a unique home directory and is sandboxed, so that they cannot access protected system resources or files stored by the system or by other apps. These restrictions are implemented via sandbox policies (aka. profiles), which are enforced by the Trusted BSD (MAC) Mandatory Access Control Framework via a kernel extension. iOS applies a generic sandbox profile to all third-party apps called container. Access to protected resources or data (some also known as app capabilities) is possible, but it's strictly controlled via special permissions known as entitlements.

Some permissions can be configured by the app's developers (e.g. Data Protection or Keychain Sharing) and will directly take effect after the installation. However, for others, the user will be explicitly asked the first time the app attempts to access a protected resource, for example:

- Bluetooth peripherals
- Calendar data
- Camera
- Contacts
- Health sharing
- Health updating
- HomeKit
- Location
- Microphone
- Motion
- Music and the media library
- Photos
- Reminders
- Siri
- Speech recognition
- the TV provider

Even though Apple urges to protect the privacy of the user and to be very clear on how to ask permissions, it can still be the case that an app requests too many of them for non-obvious reasons.

Some permissions such as Camera, Photos, Calendar Data, Motion, Contacts or Speech Recognition should be pretty straightforward to verify as it should be obvious if the app requires them to fulfill its tasks. Let's consider the following examples ragarding the Photos permission, which, if granted, gives the app access to all user photos in the "Camera Roll" (the iOS default system-wide location for storing photos):

- The typical QR Code scanning app obviously requires the camera to function but might be requesting the photos permission as well. If storage is explicitly required, and depending on the sensitivity of the pictures being taken, these apps might better opt to use the app sandbox storage to avoid other apps (having the photos permission) to access them. See the chapter "Data Storage on iOS" for more information reagarding storage of sensitive data.
- Some apps require photo uploads (e.g. for profile pictures). Recent versions of iOS introduce new APIs such as UIImagePickerController (iOS 11+) and its modern replacement PHPickerViewController (iOS 14+). These APIs run on a separate process from your app and by using them, the app gets read-only access exclusively to the images selected by the user instead of to the whole "Camera Roll". This is considered a best practice to avoid requesting unnecessary permissions.

Other permissions like Bluetooth or Location require deeper verification steps. They may be required for the app to properly function but the data being handled by those tasks might not be properly protected. For more information and some examples please refer to the "*Source Code Inspection*)" in the "Static Analysis" section below and to the "Dynamic Analysis" section.

When collecting or simply handling (e.g. caching) sensitive data, an app should provide proper mechanisms to give the user control over it, e.g. to be able to revoke access or to delete it. However, sensitive data might not only be stored or cached but also sent over the network. In both cases, it has to be ensured that the app properly follows the appropriate best practices, which in this case involve implementing proper data protection and transport security. More information on how to protect this kind of data can be found in the chapter "Network APIs".

As you can see, using app capabilities and permissions mostly involve handling personal data, therefore being a matter of protecting the user's privacy. See the articles "Protecting the User's Privacy" and "Accessing Protected Resources" in Apple Developer Documentation for more details.

Reference

- owasp-mastg Testing App Permissions (MSTG-PLATFORM-1)

Rulebook

- *How to request permission (Required)*
- *Settings for accessing photos (Required)*
- *Implement appropriate data protection and transfer security (Required)*

6.1.1 Device Capabilities

Device capabilities are used by the App Store to ensure that only compatible devices are listed and therefore are allowed to download the app. They are specified in the Info.plist file of the app under the UIRequiredDeviceCapabilities key.

```
<key>UIRequiredDeviceCapabilities</key>
<array>
    <string>armv7</string>
</array>
```

Typically you'll find the armv7 capability, meaning that the app is compiled only for the armv7 instruction set, or if it's a 32/64-bit universal app.

For example, an app might be completely dependent on NFC to work (e.g. a "NFC Tag Reader" app). According to the archived iOS Device Compatibility Reference, NFC is only available starting on the iPhone 7 (and iOS 11). A developer might want to exclude all incompatible devices by setting the nfc device capability.

Regarding testing, you can consider UIRequiredDeviceCapabilities as a mere indication that the app is using some specific resources. Unlike the entitlements related to app capabilities, device capabilities do not confer any right or access to protected resources. Additional configuration steps might be required for that, which are very specific to each capability.

For example, if BLE is a core feature of the app, Apple's Core Bluetooth Programming Guide explains the different things to be considered:

- The bluetooth-le device capability can be set in order to restrict non-BLE capable devices from downloading their app.
- App capabilities like bluetooth-peripheral or bluetooth-central (both UIBackgroundModes) should be added if BLE background processing is required.

However, this is not yet enough for the app to get access to the Bluetooth peripheral, the NSBluetoothPeripheralUsageDescription key has to be included in the Info.plist file, meaning that the user has to actively give permission. See "Purpose Strings in the Info.plist File" below for more information.

Reference

- owasp-mastg Testing App Permissions (MSTG-PLATFORM-1) Overview Device Capabilities

6.1.2 Entitlements

According to Apple's iOS Security Guide:

Entitlements are key value pairs that are signed in to an app and allow authentication beyond runtime factors, like UNIX user ID. Since entitlements are digitally signed, they can't be changed. Entitlements are used extensively by system apps and daemons to perform specific privileged operations that would otherwise require the process to run as root. This greatly reduces the potential for privilege escalation by a compromised system app or daemon.

Many entitlements can be set using the "Summary" tab of the Xcode target editor. Other entitlements require editing a target's entitlements property list file or are inherited from the iOS provisioning profile used to run the app.

Entitlement Sources:

1. Entitlements embedded in a provisioning profile that is used to code sign the app, which are composed of:
   - Capabilities defined on the Xcode project's target Capabilities tab, and/or:
   - Enabled Services on the app's App ID which are configured on the Identifiers section of the Certificates, ID's and Profiles website.
   - Other entitlements that are injected by the profile generation service.
2. Entitlements from a code signing entitlements file.

Entitlement Destinations:

1. The app's signature.
2. The app's embedded provisioning profile.

The Apple Developer Documentation also explains:

- During code signing, the entitlements corresponding to the app's enabled Capabilities/Services are transferred to the app's signature from the provisioning profile Xcode chose to sign the app.
- The provisioning profile is embedded into the app bundle during the build (embedded.mobileprovision).
- Entitlements from the "Code Signing Entitlements" section in Xcode's "Build Settings" tab are transferred to the app's signature.

For example, if you want to set the "Default Data Protection" capability, you would need to go to the Capabilities tab in Xcode and enable Data Protection. This is directly written by Xcode to the <appname>.entitlements file as the com.apple.developer.default-data-protection entitlement with default value NSFileProtectionComplete. In the IPA we might find this in the embedded.mobileprovision as:

```
<key>Entitlements</key>
<dict>
    ...
    <key>com.apple.developer.default-data-protection</key>
    <string>NSFileProtectionComplete</string>
</dict>
```

For other capabilities such as HealthKit, the user has to be asked for permission, therefore it is not enough to add the entitlements, special keys and strings have to be added to the Info.plist file of the app.

The following sections go more into detail about the mentioned files and how to perform static and dynamic analysis using them.

Reference

- owasp-mastg Testing App Permissions (MSTG-PLATFORM-1) Overview Entitlements

Rulebook

- *iOS 10 or later, items that require permission description (Required)*

6.1.3 Static Analysis

Since iOS 10, these are the main areas which you need to inspect for permissions:

- Purpose Strings in the Info.plist File
- Code Signing Entitlements File
- Embedded Provisioning Profile File
- Entitlements Embedded in the Compiled App Binary
- Source Code Inspection

Reference

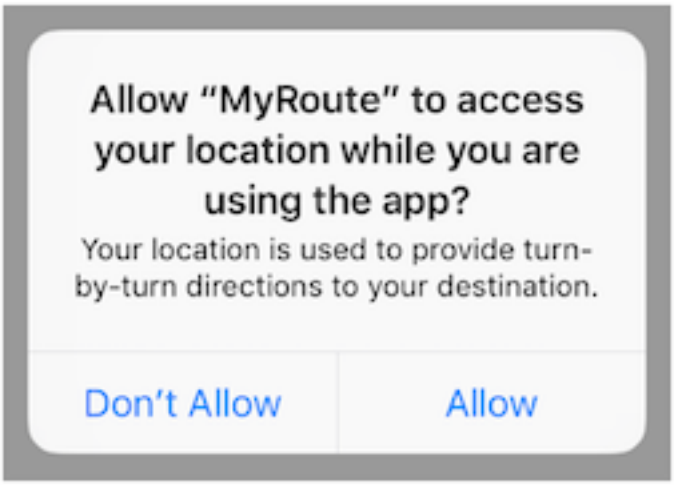
- owasp-mastg Testing App Permissions (MSTG-PLATFORM-1) Static Analysis

Rulebook

- *iOS 10 or later, items that require permission description (Required)*

6.1.3.1 Purpose Strings in the Info.plist File

Purpose strings or_usage description strings_ are custom texts that are offered to users in the system' s permission request alert when requesting permission to access protected data or resources.

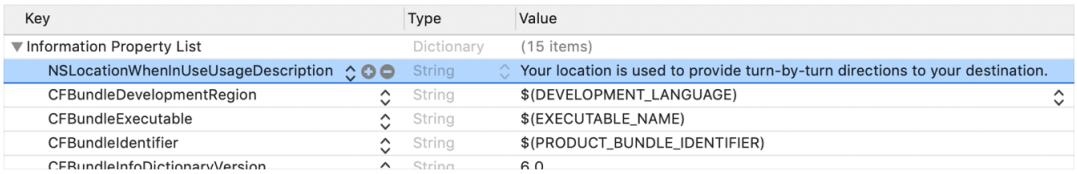


If linking on or after iOS 10, developers are required to include purpose strings in their app's Info.plist file. Otherwise, if the app attempts to access protected data or resources without having provided the corresponding purpose string, the access will fail and the app might even crash.

If having the original source code, you can verify the permissions included in the Info.plist file:

- Open the project with Xcode.
- Find and open the Info.plist file in the default editor and search for the keys starting with "Privacy -".

You may switch the view to display the raw values by right-clicking and selecting "Show Raw Keys/Values" (this way for example "Privacy - Location When In Use Usage Description" will turn into NSLocationWhenInUseUsageDescription).



If only having the IPA:

- Unzip the IPA.
- The Info.plist is located in Payload/<appname>.app/Info.plist.
- Convert it if needed (e.g. plutil -convert xml1 Info.plist) as explained in the chapter "iOS Basic Security Testing", section "The Info.plist File".
- Inspect all purpose strings Info.plist keys, usually ending with UsageDescription:

```
<plist version="1.0">
<dict>
    <key>NSLocationWhenInUseUsageDescription</key>
    <string>Your location is used to provide turn-by-turn directions to your␣
↪destination.</string>
```

For an overview of the different purpose strings Info.plist keys available see Table 1-2 at the Apple App Programming Guide for iOS. Click on the provided links to see the full description of each key in the CocoaKeys reference.

Following these guidelines should make it relatively simple to evaluate each and every entry in the Info.plist file to check if the permission makes sense.

For example, imagine the following lines were extracted from a Info.plist file used by a Solitaire game:

```
<key>NSHealthClinicalHealthRecordsShareUsageDescription</key>
<string>Share your health data with us!</string>
<key>NSCameraUsageDescription</key>
<string>We want to access your camera</string>
```

It should be suspicious that a regular solitaire game requests this kind of resource access as it probably does not have any need for accessing the camera nor a user's health-records.

Apart from simply checking if the permissions make sense, further analysis steps might be derived from analyzing purpose strings e.g. if they are related to storage sensitive data. For example, NSPhotoLibraryUsageDescription can be considered as a storage permission giving access to files that are outside of the app's sandbox and might also be accessible by other apps. In this case, it should be tested that no sensitive data is being stored there (photos in this case). For other purpose strings like NSLocationAlwaysUsageDescription, it must be also considered if the app is storing this data securely. Refer to the "Testing Data Storage" chapter for more information and best practices on securely storing sensitive data.

Reference

- owasp-mastg Testing App Permissions (MSTG-PLATFORM-1) Static Analysis Purpose Strings in the Info.plist File

6.1.3.2 Code Signing Entitlements File

Certain capabilities require a code signing entitlements file (<appname>.entitlements). It is automatically generated by Xcode but may be manually edited and/or extended by the developer as well.

Here is an example of entitlements file of the open source app Telegram including the App Groups entitlement (application-groups):

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/
↪PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
...
    <key>com.apple.security.application-groups</key>
    <array>
        <string>group.ph.telegra.Telegraph</string>
    </array>
</dict>
...
</plist>
```

The entitlement outlined above does not require any additional permissions from the user. However, it is always a good practice to check all entitlements, as the app might overask the user in terms of permissions and thereby leak information.

As documented at Apple Developer Documentation, the App Groups entitlement is required to share information between different apps through IPC or a shared file container, which means that data can be shared on the device directly between the apps. This entitlement is also required if an app extension requires to share information with its containing app.

Depending on the data to-be-shared it might be more appropriate to share it using another method such as through a backend where this data could be potentially verified, avoiding tampering by e.g. the user himself.

Reference

- owasp-mastg Testing App Permissions (MSTG-PLATFORM-1) Static Analysis Code Signing Entitlements File

6.1.3.3 Embedded Provisioning Profile File

When you do not have the original source code, you should analyze the IPA and search inside for the embedded provisioning profile that is usually located in the root app bundle folder (Payload/<appname>.app/) under the name embedded.mobileprovision.

This file is not a .plist, it is encoded using Cryptographic Message Syntax. On macOS you can inspect an embedded provisioning profile's entitlements using the following command:

```
security cms -D -i embedded.mobileprovision
```

and then search for the Entitlements key region (<key>Entitlements</key>).

Reference

- owasp-mastg Testing App Permissions (MSTG-PLATFORM-1) Static Analysis Embedded Provisioning Profile File

6.1.3.4 Entitlements Embedded in the Compiled App Binary

If you only have the app's IPA or simply the installed app on a jailbroken device, you normally won't be able to find .entitlements files. This could be also the case for the embedded.mobileprovision file. Still, you should be able to extract the entitlements property lists from the app binary yourself (which you've previously obtained as explained in the "iOS Basic Security Testing" chapter, section "Acquiring the App Binary").

The following steps should work even when targeting an encrypted binary. If for some reason they don't, you'll have to decrypt and extract the app with e.g. Clutch (if compatible with your iOS version), frida-ios-dump or similar.

- Extracting the Entitlements Plist from the App Binary

If you have the app binary in your computer, one approach is to use binwalk to extract (-e) all XML files (-y=xml):

```
$ binwalk -e -y=xml ./Telegram\ X

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
1430180       0x15D2A4        XML document, version: "1.0"
1458814       0x16427E        XML document, version: "1.0"
```

Or you can use radare2 (-qc to quietly run one command and exit) to search all strings on the app binary (izz) containing "PropertyList" (~PropertyList):

```
$ r2 -qc 'izz~PropertyList' ./Telegram\ X

0x0015d2a4 ascii <?xml version="1.0" encoding="UTF-8" standalone="yes"?>\n<!
↪DOCTYPE plist PUBLIC
"-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">\n
↪<plist version="1.0">
...<key>com.apple.security.application-groups</key>\n\t\t<array>
\n\t\t\t<string>group.ph.telegra.Telegraph</string>...

0x0016427d ascii H<?xml version="1.0" encoding="UTF-8"?>\n<!DOCTYPE plist PUBLIC
"-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">\n
↪<plist version="1.0">\n
<dict>\n\t<key>cdhashes</key>...
```

In both cases (binwalk or radare2) we were able to extract the same two plist files. If we inspect the first one (0x0015d2a4) we see that we were able to completely recover the original entitlements file from Telegram.

Note: the strings command will not help here as it will not be able to find this information. Better use grep with the -a flag directly on the binary or use radare2 (izz)/rabin2 (-zz).

If you access the app binary on the jailbroken device (e.g via SSH), you can use grep with the -a, – text flag (treats all files as ASCII text):

```
$ grep -a -A 5 'PropertyList' /var/containers/Bundle/Application/
    15E6A58F-1CA7-44A4-A9E0-6CA85B65FA35/Telegram X.app/Telegram\ X

<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/
↪PropertyList-1.0.dtd">
<plist version="1.0">
    <dict>
        <key>com.apple.security.application-groups</key>
        <array>
        ...
```

Play with the `A num, --after-context=num` flag to display more or less lines. You may use tools like the ones we presented above as well, if you have them also installed on your jailbroken iOS device.

This method should work even if the app binary is still encrypted (it was tested against several App Store apps).

Reference

- owasp-mastg Testing App Permissions (MSTG-PLATFORM-1) Static Analysis Entitlements Embedded in the Compiled App Binary

6.1.3.5 Source Code Inspection

After having checked the <appname>.entitlements file and the Info.plist file, it is time to verify how the requested permissions and assigned capabilities are put to use. For this, a source code review should be enough. However, if you don't have the original source code, verifying the use of permissions might be specially challenging as you might need to reverse engineer the app, refer to the "Dynamic Analysis" for more details on how to proceed.

When doing a source code review, pay attention to:

- whether the purpose strings in the Info.plist file match the programmatic implementations.
- whether the registered capabilities are used in such a way that no confidential information is leaking.

Users can grant or revoke authorization at any time via "Settings", therefore apps normally check the authorization status of a feature before accessing it. This can be done by using dedicated APIs available for many system frameworks that provide access to protected resources.

You can use the Apple Developer Documentation as a starting point. For example:

- Bluetooth: the state property of the CBCentralManager class is used to check system-authorization status for using Bluetooth peripherals.
- Location: search for methods of CLLocationManager, e.g. locationServicesEnabled.

```
func checkForLocationServices() {
    if CLLocationManager.locationServicesEnabled() {
        // Location services are available, so query the user's location.
    } else {
        // Update your app's UI to show that the location is unavailable.
    }
}
```

See Table1 in "Determining the Availability of Location Services" (Apple Developer Documentation) for a complete list.

Go through the application searching for usages of these APIs and check what happens to sensitive data that might be obtained from them. For example, it might be stored or transmitted over the network, if this is the case, proper data protection and transport security should be additionally verified.

Reference

- owasp-mastg Testing App Permissions (MSTG-PLATFORM-1) Static Analysis Source Code Inspection

6.1.4 Dynamic Analysis

With help of the static analysis you should already have a list of the included permissions and app capabilities in use. However, as mentioned in "Source Code Inspection", spotting the sensitive data and APIs related to those permissions and app capabilities might be a challenging task when you don't have the original source code. Dynamic analysis can help here getting inputs to iterate onto the static analysis.

Following an approach like the one presented below should help you spotting the mentioned sensitive data and APIs:

1. Consider the list of permissions / capabilities identified in the static analysis (e.g. NSLocationWhenInUseUsageDescription).
2. Map them to the dedicated APIs available for the corresponding system frameworks (e.g. Core Location). You may use the Apple Developer Documentation for this.
3. Trace classes or specific methods of those APIs (e.g. CLLocationManager), for example, using frida-trace.
4. Identify which methods are being really used by the app while accessing the related feature (e.g. "Share your location").

5. Get a backtrace for those methods and try to build a call graph.

Once all methods were identified, you might use this knowledge to reverse engineer the app and try to find out how the data is being handled. While doing that you might spot new methods involved in the process which you can again feed to step 3. above and keep iterating between static and dynamic analysis.

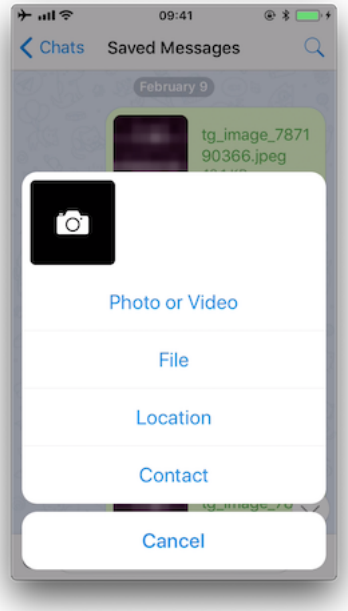
In the following example we use Telegram to open the share dialog from a chat and frida-trace to identify which methods are being called.

First we launch Telegram and start a trace for all methods matching the string "authorizationStatus" (this is a general approach because more classes apart from CLLocationManager implement this method):

```
frida-trace -U "Telegram" -m "*[* *authorizationStatus*]"
```

-U connects to the USB device. -m includes an Objective-C method to the traces. You can use a glob pattern (e.g. with the "*" wildcard, -m "*[* *authorizationStatus*]" means "include any Objective-C method of any class containing 'authorizationStatus' "). Type frida-trace -h for more information.

Now we open the share dialog:



The following methods are displayed:

```
  1942 ms  +[PHPhotoLibrary authorizationStatus]
  1959 ms  +[TGMediaAssetsLibrary authorizationStatusSignal]
  1959 ms     | +[TGMediaAssetsModernLibrary authorizationStatusSignal]
```

If we click on Location, another method will be traced:

```
 11186 ms  +[CLLocationManager authorizationStatus]
 11186 ms     | +[CLLocationManager _authorizationStatus]
 11186 ms     |    | +[CLLocationManager _
↪authorizationStatusForBundleIdentifier:0x0 bundle:0x0]
```

Use the auto-generated stubs of frida-trace to get more information like the return values and a backtrace. Do the following modifications to the JavaScript file below (the path is relative to the current directory):

```
// __handlers__/__CLLocationManager_authorizationStatus_.js
```

(continues on next page)

(continued from previous page)

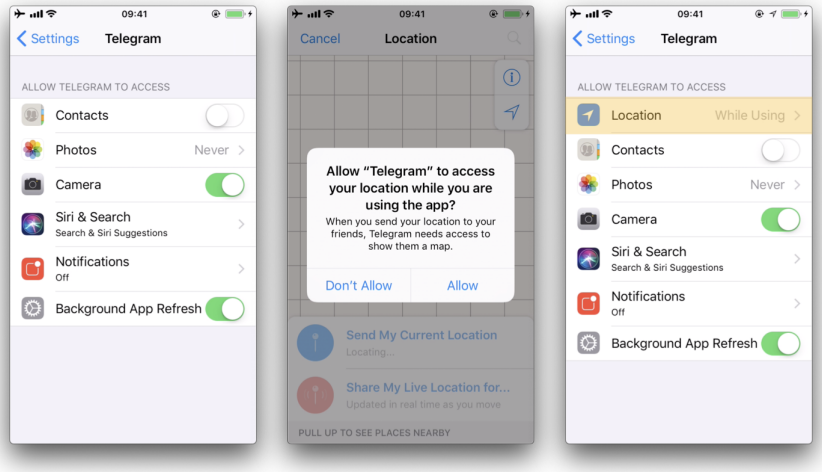
```
onEnter: function (log, args, state) {
    log("+[CLLocationManager authorizationStatus]");
    log("Called from:\n" +
        Thread.backtrace(this.context, Backtracer.ACCURATE)
        .map(DebugSymbol.fromAddress).join("\n\t") + "\n");
  },
  onLeave: function (log, retval, state) {
    console.log('RET :' + retval.toString());
  }
```

Clicking again on "Location" reveals more information:

```
  3630 ms  -[CLLocationManager init]
  3630 ms      | -[CLLocationManager initWithEffectiveBundleIdentifier:0x0␣
↪bundle:0x0]
  3634 ms  -[CLLocationManager setDelegate:0x14c9ab000]
  3641 ms  +[CLLocationManager authorizationStatus]
RET: 0x4
  3641 ms  Called from:
0x1031aa158 TelegramUI!+[TGLocationUtils␣
↪requestWhenInUserLocationAuthorizationWithLocationManager:]
    0x10337e2c0 TelegramUI!-[TGLocationPickerController initWithContext:intent:]
    0x101ee93ac TelegramUI!0x1013ac
```

We see that +[CLLocationManager authorizationStatus] returned 0x4 (CLAuthorizationStatus.authorizedWhenInUse) and was called by +[TGLocationUtils requestWhenInUserLocationAuthorizationWithLocationManager:]. As we anticipated before, you might use this kind of information as an entry point when reverse engineering the app and from there get inputs (e.g. names of classes or methods) to keep feeding the dynamic analysis.

Next, there is a visual way to inspect the status of some app permissions when using the iPhone/iPad by opening "Settings" and scrolling down until you find the app you' re interested in. When clicking on it, this will open the "ALLOW APP_NAME TO ACCESS" screen. However, not all permissions might be displayed yet. You will have to trigger them in order to be listed on that screen.



For example, in the previous example, the "Location" entry was not being listed until we triggered the permission dialogue for the first time. Once we did it, no matter if we allowed the access or not, the the "Location" entry will be displayed.

Reference

- owasp-mastg Testing App Permissions (MSTG-PLATFORM-1) Dynamic Analysis

6.1.5 Rulebook

1. *How to request permission (Required)*
2. *Settings for accessing photos (Required)*
3. *Implement appropriate data protection and transfer security (Required)*
4. *iOS 10 or later, items that require permission description (Required)*

6.1.5.1 How to request permission (Required)

User permission must be obtained in order to access the user's resources. As a means for that, it is necessary to add an explanation text to the OS standard alert at the time of permission request. The description should be in the plist and you should write the source that handles the permission request at the right time (like just before the screen where you use the camera). Do not unnecessarily request permissions immediately after starting the app.

plist description of permission:

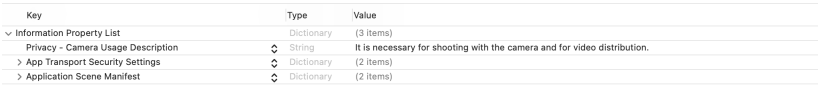


Fig 6.1.5.1.1 Add camera permission

Automatic plist generation xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/
↪PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
    <key>NSCameraUsageDescription</key>
    <string>It is necessary for shooting with the camera and for video␣
↪distribution.</string>
    <key>NSAppTransportSecurity</key>
    <dict>
        <key>NSAllowsArbitraryLoads</key>
        <true/>
        <key>NSAllowsArbitraryLoadsForMedia</key>
        <true/>
    </dict>
    <key>UIApplicationSceneManifest</key>
    <dict>
        <key>UIApplicationSupportsMultipleScenes</key>
        <false/>
        <key>UISceneConfigurations</key>
        <dict>
            <key>UIWindowSceneSessionRoleApplication</key>
            <array>
                <dict>
                    <key>UISceneConfigurationName</key>
                    <string>Default Configuration</string>
                    <key>UISceneDelegateClassName</key>
                    <string>$(PRODUCT_MODULE_NAME).SceneDelegate</string>
```

(continues on next page)

(continued from previous page)

```
                    <key>UISceneStoryboardFile</key>
                    <string>Main</string>
                </dict>
            </array>
        </dict>
    </dict>
</dict>
</plist>
```

Camera permission check and request handling:

```
import Foundation
import AVFoundation

class CameraHelper {
    func request(completion: @escaping (Bool) -> Void) {

        let status = AVCaptureDevice.authorizationStatus(for: .video)
        switch status {
        case .authorized:

            completion(true)
        case .denied:
            completion(false)
        case .notDetermined: ()
            AVCaptureDevice.requestAccess(for: .video) { granted in
                completion(granted)
            }

        case .restricted:
            completion(false)

        @unknown default:
            // not support case
            ()
        }
    }
}
```

If this is violated, the following may occur.

- Requesting unnecessary permissions can lead to unintended data disclosure.

6.1.5.2 Settings for accessing photos (Required)

If you use PHPickerViewController in the PhotosUI library when retrieving photos from the album, you can display a screen for the user to select an image without checking permission. This should be selected unless you need to retrieve photos other than those selected by the user.

How to select image in album via PHPickerViewController:

```
import PhotosUI

class PViewController: UIViewController,PHPickerViewControllerDelegate {

    var selectedImages: [UIImage] = []

    override func viewDidLoad() {
        super.viewDidLoad()
    }
    func makeViewContoroller() {
```

(continues on next page)

(continued from previous page)

```
        var configuration = PHPickerConfiguration()
        configuration.selectionLimit = 36 // Selection limit. Set to 0 for␣
↪unlimited.
        configuration.filter = .images // The type of media that can be retrieved.

        let picker = PHPickerViewController(configuration: configuration)
        picker.delegate = self
        present(picker, animated: true)
    }


    func picker(_ picker: PHPickerViewController, didFinishPicking results:␣
↪[PHPickerResult]) {

        for image in results {

            image.itemProvider.loadObject(ofClass: UIImage.self) { (selectedImage,␣
↪error) in

                guard let wrapImage = selectedImage as? UIImage else {
                    return
                }

                self.selectedImages.append(wrapImage)

            }
        }
    }
}
```

If this is violated, the following may occur.

- Requesting unnecessary permissions can lead to unintended data disclosure.

6.1.5.3 Implement appropriate data protection and transfer security (Required)

When sending data over the network. Enabling his ATS on iOS.

Rulebook

- *Use App Transport Security (ATS) (Required)*

If this is violated, the following may occur.

- Unintended data leakage may occur.

6.1.5.4 iOS 10 or later, items that require permission description (Required)

Operations that need to be described as permissions in plist and the associated keys are shown below.

- NSAppleMusicUsageDescription Access to media library
- NSCalendarsUsageDescription Access to calendars
- NSContactsUsageDescription Access to contacts
- NSPhotoLibraryUsageDescription Access Photo Library
- NSRemindersUsageDescription Access reminders
- NSCameraUsageDescription Access to camera
- NSMicrophoneUsageDescription Access to microphone
- NSMotionUsageDescription Access to accelerometer

- NSHealthShareUsageDescription Access to health data
- NSHealthUpdateUsageDescription Modify health data
- NSHomeKitUsageDescription Access HomeKit configuration data
- NSSiriUsageDescription Send user data to Siri
- NSSpeechRecognitionUsageDescription Send user data to Speech Recognition Server
- NSLocationWhenInUseUsageDescription Access location information (allowed only when in use)
- NFCReaderUsageDescription Access to device' s NFC hardware
- NSFaceIDUsageDescription Authenticate with Face ID
- NSPhotoLibraryAddUsageDescription Add-only access to photo library
- NSLocationAlwaysAndWhenInUseUsageDescription Access to location information (always allowed)
- NSHealthClinicalHealthRecordsShareUsageDescription Request permission to read clinical records
- NSHealthRequiredReadAuthorizationTypeIdentifiers Type of clinical records data for which read permission must be obtained
- NSBluetoothAlwaysUsageDescription Access to Bluetooth
- NSLocationTemporaryUsageDescriptionDictionary Access to location information (allowed only once)
- NSWidgetWantsLocation widget uses location information
- NSLocationDefaultAccuracyReduced Require location accuracy reduction by default
- NSLocalNetworkUsageDescription Access to local network
- NSUserTrackingUsageDescription Permission to use data to track users and devices
- NSSensorKitUsageDescription Briefly describe the purpose of the research study
- NSGKFriendListUsageDescription Access Game Center friend list
- NSNearbyInteractionUsageDescription Initiate an interaction session with a nearby device
- NSIdentityUsageDescription Request ID information
- NSSensorKitPrivacyPolicyURL Hyperlink to a web page that displays the privacy policy
- UIRequiresPersistentWiFi Requires Wi-Fi connection or
- NSSensorKitUsageDetail Dictionary containing keys to specific information collected by the app
  - SRSensorUsageKeyboardMetrics Observes keyboard activity
  - SRSensorUsageDeviceUsage Observe how often devices are activated
  - SRSensorUsageWristDetection Observe how the watch is worn
  - SRSensorUsagePhoneUsage Observe phone operations
  - SRSensorUsageMessageUsage Observe user activity in messages
  - SRSensorUsageVisits Observe frequently visited locations
  - SRSensorUsagePedometer Observe step information
  - SRSensorUsageMotion Observe motion data
  - SRSensorUsageSpeechMetrics Analyze user speech
  - SRSensorUsageAmbientLightSensor Observe light intensity in the environment

* NSLocationAlwaysAndWhenInUseUsageDescription should be used since NSLocationAlwaysUsageDescription has been deprecated since iOS 11.

* NSNearbyInteractionUsageDescription should be used since NSNearbyInteractionAllowOnceUsageDescription has been deprecated since iOS 15.

* NSBluetoothPeripheralUsageDescription is required if you are using an API to access Bluetooth peripherals and have a deployment target prior to iOS 13.

If this is violated, the following may occur.

- Cannot access the specified function.

6.2 MSTG-PLATFORM-2

All inputs from external sources and the user are validated and if necessary sanitized. This includes data received via the UI, IPC mechanisms such as intents, custom URLs, and network sources.

6.2.1 Cross-Site Scripting Flaws

Cross-site scripting (XSS) issues allow attackers to inject client-side scripts into web pages viewed by users. This type of vulnerability is prevalent in web applications. When a user views the injected script in a browser, the attacker gains the ability to bypass the same origin policy, enabling a wide variety of exploits (e.g. stealing session cookies, logging key presses, performing arbitrary actions, etc.).

In the context of native apps, XSS risks are far less prevalent for the simple reason these kinds of applications do not rely on a web browser. However, apps using WebView components, such as WKWebView or the deprecated UIWebView on iOS and WebView on Android, are potentially vulnerable to such attacks.

An older but well-known example is the local XSS issue in the Skype app for iOS, first identified by Phil Purviance. The Skype app failed to properly encode the name of the message sender, allowing an attacker to inject malicious JavaScript to be executed when a user views the message. In his proof-of-concept, Phil showed how to exploit the issue and steal a user' s address book.

Reference

- owasp-mastg Cross-Site Scripting Flaws (MSTG-PLATFORM-2)

6.2.1.1 Static Analysis

Take a close look at any WebViews present and investigate for untrusted input rendered by the app.

If a WebView is used to display a remote website, the burden of escaping HTML shifts to the server side. If an XSS flaw exists on the web server, this can be used to execute script in the context of the WebView. As such, it is important to perform static analysis of the web application source code.

Verify that the following best practices have been followed:

- No untrusted data is rendered in HTML, JavaScript or other interpreted contexts unless it is absolutely necessary.
- Appropriate encoding is applied to escape characters, such as HTML entity encoding. Note: escaping rules become complicated when HTML is nested within other code, for example, rendering a URL located inside a JavaScript block.

Consider how data will be rendered in a response. For example, if data is rendered in a HTML context, six control characters that must be escaped:

Table 6.2.1.1.1 List of control characters that must be escaped

Character	Escaped
&	&amp;
<	&lt;
>	&gt;
"	&quot;
'	&#x27;
/	&#x2F;

For a comprehensive list of escaping rules and other prevention measures, refer to the OWASP XSS Prevention Cheat Sheet.

Reference

- owasp-mastg Cross-Site Scripting Flaws (MSTG-PLATFORM-2) Static Analysis

Rulebook

- *Check your WebView for untrusted input rendered from your app (Required)*

6.2.1.2 Dynamic Analysis

XSS issues can be best detected using manual and/or automated input fuzzing, i.e. injecting HTML tags and special characters into all available input fields to verify the web application denies invalid inputs or escapes the HTML meta-characters in its output.

A reflected XSS attack refers to an exploit where malicious code is injected via a malicious link. To test for these attacks, automated input fuzzing is considered to be an effective method. For example, the BURP Scanner is highly effective in identifying reflected XSS vulnerabilities. As always with automated analysis, ensure all input vectors are covered with a manual review of testing parameters.

Reference

- owasp-mastg Cross-Site Scripting Flaws (MSTG-PLATFORM-2) Dynamic Analysis

6.2.2 Rulebook

1. *Check your WebView for untrusted input rendered from your app (Required)*

6.2.2.1 Check your WebView for untrusted input rendered from your app (Required)

Verify that the following best practices have been followed:

- No untrusted data is rendered in HTML, JavaScript or other interpreted contexts unless it is absolutely necessary.
- Appropriate encoding is applied to escape characters, such as HTML entity encoding. Note: escaping rules become complicated when HTML is nested within other code, for example, rendering a URL located inside a JavaScript block.

Consider how data will be rendered in a response. For example, if data is rendered in a HTML context, six control characters that must be escaped:

Table 6.2.2.1.1 List of control characters that must be escaped

Character	Escaped
&	&amp;
<	&lt;
>	&gt;
“	&quot;
‘	&#x27;
/	&#x2F;

For a comprehensive list of escaping rules and other prevention measures, refer to the OWASP XSS Prevention Cheat Sheet.

If this is violated, the following may occur.

- An XSS problem may exist.

6.3 MSTG-PLATFORM-3

The app does not export sensitive functionality via custom URL schemes, unless these mechanisms are properly protected.

6.3.1 Custom URL Schemes

Custom URL schemes allow apps to communicate via a custom protocol. An app must declare support for the schemes and handle incoming URLs that use those schemes.

Apple warns about the improper use of custom URL schemes in the Apple Developer Documentation:

URL schemes offer a potential attack vector into your app, so make sure to validate all URL parameters and discard any malformed URLs. In addition, limit the available actions to those that do not risk the user's data. For example, do not allow other apps to directly delete content or access sensitive information about the user. When testing your URL-handling code, make sure your test cases include improperly formatted URLs.

They also suggest using universal links instead, if the purpose is to implement deep linking:

While custom URL schemes are an acceptable form of deep linking, universal links are strongly recommended as a best practice.

Supporting a custom URL scheme is done by:

- defining the format for the app's URLs,
- registering the scheme so that the system directs appropriate URLs to the app,
- handling the URLs that the app receives.

Security issues arise when an app processes calls to its URL scheme without properly validating the URL and its parameters and when users aren't prompted for confirmation before triggering an important action.

One example is the following bug in the Skype Mobile app, discovered in 2010: The Skype app registered the skype:// protocol handler, which allowed other apps to trigger calls to other Skype users and phone numbers. Unfortunately, Skype didn't ask users for permission before placing the calls, so any app could call arbitrary numbers without the user's knowledge. Attackers exploited this vulnerability by putting an invisible `<iframe src="skype://xxx?call"></iframe>` (where xxx was replaced by a premium number), so any Skype user who inadvertently visited a malicious website called the premium number.

As a developer, you should carefully validate any URL before calling it. You can allow only certain applications which may be opened via the registered protocol handler. Prompting users to confirm the URL-invoked action is another helpful control.

All URLs are passed to the app delegate, either at launch time or while the app is running or in the background. To handle incoming URLs, the delegate should implement methods to:

- retrieve information about the URL and decide whether you want to open it,
- open the resource specified by the URL.

More information can be found in the archived App Programming Guide for iOS and in the Apple Secure Coding Guide.

In addition, an app may also want to send URL requests (aka. queries) to other apps. This is done by:

- registering the application query schemes that the app wants to query,
- optionally querying other apps to know if they can open a certain URL,
- sending the URL requests.

All of this presents a wide attack surface that we will address in the static and dynamic analysis sections.

Reference

- owasp-mastg Testing Custom URL Schemes (MSTG-PLATFORM-3)

Rulebook

- *If you want to implement deep linking, use universal links (Recommended)*
- *How to use custom URL schemes (Required)*
- *If your app handles URL scheme calls, it should properly validate URLs and their arguments and ask for user consent before performing sensitive actions (Required)*
- *How to use universal links (Recommended)*
- *Validate deep link URLs and their arguments (Required)*

6.3.2 Static Analysis

There are a couple of things that we can do in the static analysis. In the next sections we will see the following:

- Testing custom URL schemes registration
- Testing application query schemes registration
- Testing URL handling and validation
- Testing URL requests to other apps
- Testing for deprecated methods

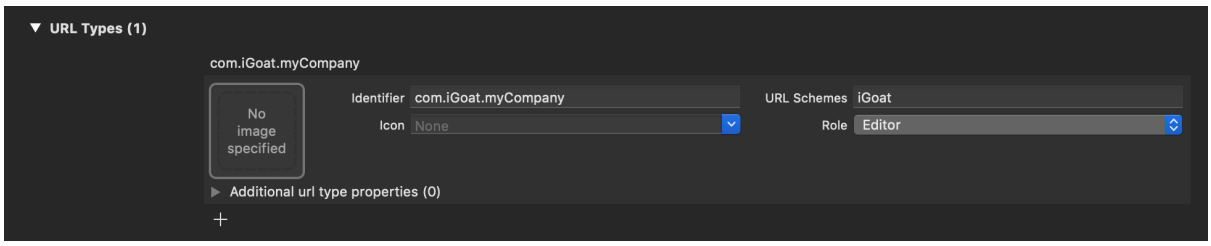
Reference

- owasp-mastg Testing Custom URL Schemes (MSTG-PLATFORM-3) Static Analysis

6.3.2.1 Custom URL Schemes Registration

The first step to test custom URL schemes is finding out whether an application registers any protocol handlers.

If you have the original source code and want to view registered protocol handlers, simply open the project in Xcode, go to the **Info** tab and open the **URL Types** section as presented in the screenshot below:



Also in Xcode you can find this by searching for the CFBundleURLTypes key in the app's Info.plist file (example from iGoat-Swift):

```
<key>CFBundleURLTypes</key>
<array>
    <dict>
        <key>CFBundleURLName</key>
        <string>com.iGoat.myCompany</string>
        <key>CFBundleURLSchemes</key>
        <array>
            <string>iGoat</string>
        </array>
    </dict>
</array>
```

In a compiled application (or IPA), registered protocol handlers are found in the file Info.plist in the app bundle's root folder. Open it and search for the CFBundleURLSchemes key, if present, it should contain an array of strings (example from iGoat-Swift):

```
grep -A 5 -nri urlsch Info.plist
Info.plist:45:    <key>CFBundleURLSchemes</key>
Info.plist-46-    <array>
Info.plist-47-        <string>iGoat</string>
Info.plist-48-    </array>
```

Once the URL scheme is registered, other apps can open the app that registered the scheme, and pass parameters by creating appropriately formatted URLs and opening them with the UIApplication openURL:options:completionHandler: method.

Note from the App Programming Guide for iOS:

If more than one third-party app registers to handle the same URL scheme, there is currently no process for determining which app will be given that scheme.

This could lead to a URL scheme hijacking attack (see page 136 in [#thiel2]).

Reference

- owasp-mastg Testing Custom URL Schemes (MSTG-PLATFORM-3) Static Analysis Testing Custom URL Schemes Registration

Rulebook

- *How to use custom URL schemes (Required)*
- *How to use universal links (Recommended)*
- *Validate deep link URLs and their arguments (Required)*

6.3.2.2 Application Query Schemes Registration

Before calling the openURL:options:completionHandler: method, apps can call canOpenURL: to verify that the target app is available. However, as this method was being used by malicious app as a way to enumerate installed apps, from iOS 9.0 the URL schemes passed to it must be also declared by adding the LSApplicationQueriesSchemes key to the app's Info.plist file and an array of up to 50 URL schemes.

```
<key>LSApplicationQueriesSchemes</key>
    <array>
        <string>url_scheme1</string>
        <string>url_scheme2</string>
    </array>
```

canOpenURL will always return NO for undeclared schemes, whether or not an appropriate app is installed. However, this restriction only applies to canOpenURL.

The openURL:options:completionHandler: method will still open any URL scheme, even if the LSApplicationQueriesSchemes array was declared, and return YES / NO depending on the result.

As an example, Telegram declares in its Info.plist these Queries Schemes, among others:

```
    <key>LSApplicationQueriesSchemes</key>
    <array>
        <string>dbapi-3</string>
        <string>instagram</string>
        <string>googledrive</string>
        <string>comgooglemaps-x-callback</string>
        <string>foursquare</string>
        <string>here-location</string>
        <string>yandexmaps</string>
        <string>yandexnavi</string>
        <string>comgooglemaps</string>
        <string>youtube</string>
        <string>twitter</string>
        ...
```

Reference

- owasp-mastg Testing Custom URL Schemes (MSTG-PLATFORM-3) Static Analysis Testing Application Query Schemes Registration

6.3.2.3 URL Handling and Validation

In order to determine how a URL path is built and validated, if you have the original source code, you can search for the following methods:

- application:didFinishLaunchingWithOptions: method or application:will-FinishLaunchingWithOptions:: verify how the decision is made and how the information about the URL is retrieved.
- application:openURL:options:: verify how the resource is being opened, i.e. how the data is being parsed, verify the options, especially if access by the calling app (sourceApplication) should be allowed or denied. The app might also need user permission when using the custom URL scheme.

In Telegram you will find four different methods being used:

```
func application(_ application: UIApplication, open url: URL, sourceApplication:␣
↪String?) -> Bool {
    self.openUrl(url: url)
    return true
}

func application(_ application: UIApplication, open url: URL, sourceApplication:␣
↪String?,
annotation: Any) -> Bool {
    self.openUrl(url: url)
    return true
}

func application(_ app: UIApplication, open url: URL,
options: [UIApplicationOpenURLOptionsKey : Any] = [:]) -> Bool {
    self.openUrl(url: url)
    return true
}

func application(_ application: UIApplication, handleOpen url: URL) -> Bool {
    self.openUrl(url: url)
    return true
}
```

We can observe some things here:

- The app implements also deprecated methods like application:handleOpenURL: and application:openURL:sourceApplication:annotation:.
- The source application is not being verified in any of those methods.
- All of them call a private openUrl method. You can inspect it to learn more about how the URL request is handled.

Reference

- owasp-mastg Testing Custom URL Schemes (MSTG-PLATFORM-3) Static Analysis Testing URL Handling and Validation

Rulebook

- *How to use custom URL schemes (Required)*
- *How to use universal links (Recommended)*
- *Validate deep link URLs and their arguments (Required)*

6.3.2.4 URL Requests to Other Apps

The method openURL:options:completionHandler: and the deprecated openURL: method of UIApplication are responsible for opening URLs (i.e. to send requests / make queries to other apps) that may be local to the current app or it may be one that must be provided by a different app. If you have the original source code you can search directly for usages of those methods.

Additionally, if you are interested into knowing if the app is querying specific services or apps, and if the app is well-known, you can also search for common URL schemes online and include them in your greps. For example, a quick Google search reveals:

```
Apple Music - music:// or musics:// or audio-player-event://
Calendar - calshow:// or x-apple-calevent://
Contacts - contacts://
Diagnostics - diagnostics:// or diags://
GarageBand - garageband://
iBooks - ibooks:// or itms-books:// or itms-bookss://
Mail - message:// or mailto://emailaddress
Messages - sms://phonenumber
Notes - mobilenotes://
...
```

We search for this method in the Telegram source code, this time without using Xcode, just with egrep:

```
$ egrep -nr "open.*options.*completionHandler" ./Telegram-iOS/

./AppDelegate.swift:552: return UIApplication.shared.open(parsedUrl,
    options: [UIApplicationOpenURLOptionUniversalLinksOnly: true as NSNumber],
    completionHandler: { value in
./AppDelegate.swift:556: return UIApplication.shared.open(parsedUrl,
    options: [UIApplicationOpenURLOptionUniversalLinksOnly: true as NSNumber],
    completionHandler: { value in
```

If we inspect the results we will see that openURL:options:completionHandler: is actually being used for universal links, so we have to keep searching. For example, we can search for openURL(:

```
$ egrep -nr "openURL\(" ./Telegram-iOS/

./ApplicationContext.swift:763:  UIApplication.shared.openURL(parsedUrl)
./ApplicationContext.swift:792:  UIApplication.shared.openURL(URL(
                                        string: "https://telegram.org/deactivate?
↪phone=\(phone)")!
                                 )
./AppDelegate.swift:423:         UIApplication.shared.openURL(url)
./AppDelegate.swift:538:         UIApplication.shared.openURL(parsedUrl)
...
```

If we inspect those lines we will see how this method is also being used to open "Settings" or to open the "App Store Page" .

When just searching for :// we see:

```
if documentUri.hasPrefix("file://"), let path = URL(string: documentUri)?.path {
if !url.hasPrefix("mt-encrypted-file://?") {
guard let dict = TGStringUtils.argumentDictionary(inUrlString: String(url[url.
↪index(url.startIndex,
    offsetBy: "mt-encrypted-file://?".count)...])) else {
parsedUrl = URL(string: "https://\(url)")
if let url = URL(string: "itms-apps://itunes.apple.com/app/id\(appStoreId)") {
} else if let url = url as? String, url.lowercased().hasPrefix("tg://") {
[[WKExtension sharedExtension] openSystemURL:[NSURL URLWithString:[NSString
    stringWithFormat:@"tel://%@", userHandle.data]]];
```

After combining the results of both searches and carefully inspecting the source code we find the following piece of code:

```
openUrl: { url in
            var parsedUrl = URL(string: url)
            if let parsed = parsedUrl {
                if parsed.scheme == nil || parsed.scheme!.isEmpty {
                    parsedUrl = URL(string: "https://\(url)")
                }
                if parsed.scheme == "tg" {
                    return
                }
            }

            if let parsedUrl = parsedUrl {
                UIApplication.shared.openURL(parsedUrl)
```

Before opening a URL, the scheme is validated, "https" will be added if necessary and it won't open any URL with the "tg" scheme. When ready it will use the deprecated openURL method.

If only having the compiled application (IPA) you can still try to identify which URL schemes are being used to query other apps:

- Check if LSApplicationQueriesSchemes was declared or search for common URL schemes.
- Also use the string :// or build a regular expression to match URLs as the app might not be declaring some schemes.

You can do that by first verifying that the app binary contains those strings by e.g. using unix strings command:

```
strings <yourapp> | grep "someURLscheme://"
```

or even better, use radare2's iz/izz command or rafind2, both will find strings where the unix strings command won't. Example from iGoat-Swift:

```
$ r2 -qc izz~iGoat:// iGoat-Swift
37436 0x001ee610 0x001ee610  23  24 (4.__TEXT.__cstring) ascii iGoat://?
↪contactNumber=
```

Reference

- owasp-mastg Testing Custom URL Schemes (MSTG-PLATFORM-3) Static Analysis Testing URL Requests to Other Apps

6.3.2.5 Deprecated Methods

Search for deprecated methods like:

- application:handleOpenURL:
- openURL:
- application:openURL:sourceApplication:annotation:

For example, here we find those three:

```
$ rabin2 -zzq Telegram\ X.app/Telegram\ X | grep -i "openurl"

0x1000d9e90 31 30 UIApplicationOpenURLOptionsKey
0x1000dee3f 50 49 application:openURL:sourceApplication:annotation:
0x1000dee71 29 28 application:openURL:options:
0x1000dee8e 27 26 application:handleOpenURL:
0x1000df2c9 9 8 openURL:
0x1000df766 12 11 canOpenURL:
```

(continues on next page)

(continued from previous page)

```
0x1000df772 35 34 openURL:options:completionHandler:
...
```

Reference

- owasp-mastg Testing Custom URL Schemes (MSTG-PLATFORM-3) Static Analysis Testing for Deprecated Methods

6.3.3 Dynamic Analysis

Once you've identified the custom URL schemes the app has registered, there are several methods that you can use to test them:

- Performing URL requests
- Identifying and hooking the URL handler method
- Testing URL schemes source validation
- Fuzzing URL schemes

Reference

- owasp-mastg Testing Custom URL Schemes (MSTG-PLATFORM-3) Dynamic Analysis

6.3.3.1 Performing URL Requests

Using Safari

To quickly test one URL scheme you can open the URLs on Safari and observe how the app behaves. For example, if you write tel://123456789 in the address bar of Safari, a pop up will appear with the telephone number and the options "Cancel" and "Call". If you press "Call" it will open the Phone app and directly make the call.

You may also know already about pages that trigger custom URL schemes, you can just navigate normally to those pages and Safari will automatically ask when it finds a custom URL scheme.

Reference

- owasp-mastg Testing Custom URL Schemes (MSTG-PLATFORM-3) Dynamic Analysis Performing URL Requests Using Safari

Using the Notes App

As already seen in "Triggering Universal Links", you may use the Notes app and long press the links you've written in order to test custom URL schemes. Remember to exit the editing mode in order to be able to open them. Note that you can click or long press links including custom URL schemes only if the app is installed, if not they won't be highlighted as clickable links.

Reference

- owasp-mastg Testing Custom URL Schemes (MSTG-PLATFORM-3) Dynamic Analysis Performing URL Requests Using the Notes App

Using Frida

If you simply want to open the URL scheme you can do it using Frida:

```
$ frida -U iGoat-Swift

[iPhone::iGoat-Swift]-> function openURL(url) {
                            var UIApplication = ObjC.classes.UIApplication.
↪sharedApplication();
                            var toOpen = ObjC.classes.NSURL.URLWithString_(url);
                            return UIApplication.openURL_(toOpen);
```

(continues on next page)

(continued from previous page)

```
                        }
[iPhone::iGoat-Swift]-> openURL("tel://234234234")
true
```

In this example from Frida CodeShare the author uses the non-public API LSApplication Workspace.openSensitiveURL:withOptions: to open the URLs (from the SpringBoard app):

```
function openURL(url) {
    var w = ObjC.classes.LSApplicationWorkspace.defaultWorkspace();
    var toOpen = ObjC.classes.NSURL.URLWithString_(url);
    return w.openSensitiveURL_withOptions_(toOpen, null);
}
```

Note that the use of non-public APIs is not permitted on the App Store, that's why we don't even test these but we are allowed to use them for our dynamic analysis.

Reference

- owasp-mastg Testing Custom URL Schemes (MSTG-PLATFORM-3) Dynamic Analysis Performing URL Requests Using Frida

6.3.3.2 Identifying and Hooking the URL Handler Method

If you can't look into the original source code you will have to find out yourself which method does the app use to handle the URL scheme requests that it receives. You cannot know if it is an Objective-C method or a Swift one, or even if the app is using a deprecated one.

Reference

- owasp-mastg Testing Custom URL Schemes (MSTG-PLATFORM-3) Dynamic Analysis Performing URL Requests Identifying and Hooking the URL Handler Method

Crafting the Link Yourself and Letting Safari Open It

For this we will use the ObjC method observer from Frida CodeShare, which is an extremely handy script that allows you to quickly observe any collection of methods or classes just by providing a simple pattern.

In this case we are interested into all methods containing "openURL", therefore our pattern will be *[* *openURL*]:

- The first asterisk will match all instance - and class + methods.
- The second matches all Objective-C classes.
- The third and forth allow to match any method containing the string openURL.

```
$ frida -U iGoat-Swift --codeshare mrmacete/objc-method-observer

[iPhone::iGoat-Swift]-> observeSomething("*[* *openURL*]");
Observing  -[_UIDICActivityItemProvider␣
↪activityViewController:openURLAnnotationForActivityType:]
Observing  -[CNQuickActionsManager _openURL:]
Observing  -[SUClientController openURL:]
Observing  -[SUClientController openURL:inClientWithIdentifier:]
Observing  -[FBSSystemService openURL:application:options:clientPort:withResult:]
Observing  -[iGoat_Swift.AppDelegate application:openURL:options:]
Observing  -[PrefsUILinkLabel openURL:]
Observing  -[UIApplication openURL:]
Observing  -[UIApplication _openURL:]
Observing  -[UIApplication openURL:options:completionHandler:]
Observing  -[UIApplication openURL:withCompletionHandler:]
Observing  -[UIApplication _openURL:originatingView:completionHandler:]
Observing  -[SUApplication application:openURL:sourceApplication:annotation:]
...
```

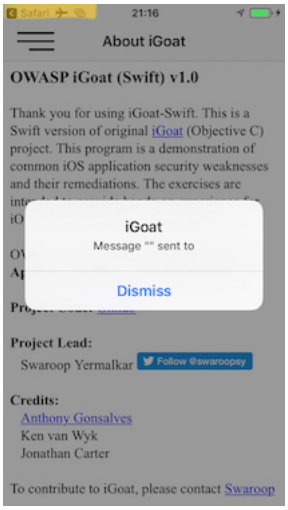
The list is very long and includes the methods we have already mentioned. If we trigger now one URL scheme, for example "igoat://" from Safari and accept to open it in the app we will see the following:

```
[iPhone::iGoat-Swift]-> (0x1c4038280)  -[iGoat_Swift.AppDelegate␣
↪application:openURL:options:]
application: <UIApplication: 0x101d0fad0>
openURL: igoat://
options: {
    UIApplicationOpenURLOptionsOpenInPlaceKey = 0;
    UIApplicationOpenURLOptionsSourceApplicationKey = "com.apple.mobilesafari";
}
0x18b5030d8 UIKit!__58-[UIApplication _applicationOpenURLAction:payload:origin:]_
↪block_invoke
0x18b502a94 UIKit!-[UIApplication _applicationOpenURLAction:payload:origin:]
...
0x1817e1048 libdispatch.dylib!_dispatch_client_callout
0x1817e86c8 libdispatch.dylib!_dispatch_block_invoke_direct$VARIANT$mp
0x18453d9f4 FrontBoardServices!__FBSSERIALQUEUE_IS_CALLING_OUT_TO_A_BLOCK__
0x18453d698 FrontBoardServices!-[FBSSerialQueue _performNext]
RET: 0x1
```

Now we know that:

- The method -[iGoat_Swift.AppDelegate application:openURL:options:] gets called. As we have seen before, it is the recommended way and it is not deprecated.
- It receives our URL as a parameter: igoat://.
- We also can verify the source application: com.apple.mobilesafari.
- We can also know from where it was called, as expected from -[UIApplication _applicationOpenURLAction:payload:origin:].
- The method returns 0x1 which means YES (the delegate successfully handled the request).

The call was successful and we see now that the iGoat app was open:



Notice that we can also see that the caller (source application) was Safari if we look in the upper-left corner of the screenshot.

Reference

- owasp-mastg Testing Custom URL Schemes (MSTG-PLATFORM-3) Dynamic Analysis Performing URL Requests Crafting the Link Yourself and Letting Safari Open It

Dynamically Opening the Link from the App Itself

It is also interesting to see which other methods get called on the way. To change the result a little bit we will call the same URL scheme from the iGoat app itself. We will use again ObjC method observer and the Frida REPL:

```
$ frida -U iGoat-Swift --codeshare mrmacete/objc-method-observer

[iPhone::iGoat-Swift]-> function openURL(url) {
                            var UIApplication = ObjC.classes.UIApplication.
↪sharedApplication();
                            var toOpen = ObjC.classes.NSURL.URLWithString_(url);
                            return UIApplication.openURL_(toOpen);
                        }

[iPhone::iGoat-Swift]-> observeSomething("*[* *openURL*]");
[iPhone::iGoat-Swift]-> openURL("iGoat://?contactNumber=123456789&message=hola")

(0x1c409e460)  -[__NSXPCInterfaceProxy__LSDOpenProtocol␣
↪openURL:options:completionHandler:]
openURL: iGoat://?contactNumber=123456789&message=hola
options: nil
completionHandler: <__NSStackBlock__: 0x16fc89c38>
0x183befbec MobileCoreServices!-[LSApplicationWorkspace openURL:withOptions:error:]
0x10ba6400c
...
RET: nil

...

(0x101d0fad0)  -[UIApplication openURL:]
openURL: iGoat://?contactNumber=123456789&message=hola
0x10a610044
...
RET: 0x1

true
(0x1c4038280)  -[iGoat_Swift.AppDelegate application:openURL:options:]
application: <UIApplication: 0x101d0fad0>
openURL: iGoat://?contactNumber=123456789&message=hola
options: {
    UIApplicationOpenURLOptionsOpenInPlaceKey = 0;
    UIApplicationOpenURLOptionsSourceApplicationKey = "OWASP.iGoat-Swift";
}
0x18b5030d8 UIKit!__58-[UIApplication _applicationOpenURLAction:payload:origin:]_
↪block_invoke
0x18b502a94 UIKit!-[UIApplication _applicationOpenURLAction:payload:origin:]
...
RET: 0x1
```

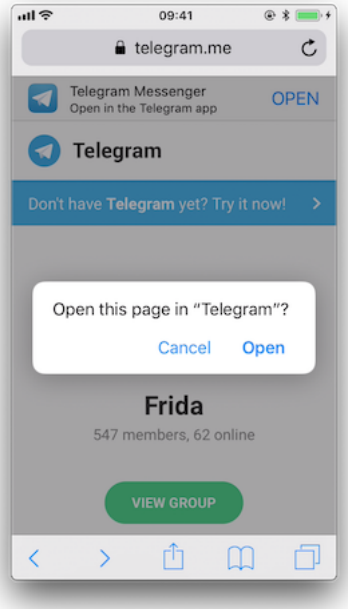
The output is truncated for better readability. This time you see that UIApplicationOpenURLOptionsSourceApplicationKey has changed to OWASP.iGoat-Swift, which makes sense. In addition, a long list of openURL-like methods were called. Considering this information can be very useful for some scenarios as it will help you to decide what you next steps will be, e.g. which method you will hook or tamper with next.

Reference

- owasp-mastg Testing Custom URL Schemes (MSTG-PLATFORM-3) Dynamic Analysis Performing URL Requests Dynamically Opening the Link from the App Itself

Opening a Link by Navigating to a Page and Letting Safari Open It

You can now test the same situation when clicking on a link contained on a page. Safari will identify and process the URL scheme and choose which action to execute. Opening this link "https://telegram.me/fridadotre" will trigger this behavior.



First of all we let frida-trace generate the stubs for us:

```
$ frida-trace -U Telegram -m "*[* *restorationHandler*]" -i "*open*Url*"
    -m "*[* *application*URL*]" -m "*[* openURL]"

...
7310 ms  -[UIApplication _applicationOpenURLAction: 0x1c44ff900 payload:␣
↪0x10c5ee4c0 origin: 0x0]
7311 ms     | -[AppDelegate application: 0x105a59980 openURL: 0x1c46ebb80 options:␣
↪0x1c0e222c0]
7312 ms     |
↪$S10TelegramUI15openExternalUrl7account7context3url05forceD016presentationData
          ␣
↪18applicationContext20navigationController12dismissInputy0A4Core7AccountC_
↪AA14Open
            URLContextOSSSbAA012PresentationK0CAA0a11ApplicationM0C7Display010Navi-
gationO0CSgyyctF()
```

Now we can simply modify by hand the stubs we are interested in:

- The Objective-C method application:openURL:options::

```
// __handlers__/__AppDelegate_application_openUR_3679fadc.js

onEnter: function (log, args, state) {
    log("-[AppDelegate application: " + args[2] +
                " openURL: " + args[3] + " options: " + args[4] + "]");
    log("\tapplication :" + ObjC.Object(args[2]).toString());
    log("\topenURL :" + ObjC.Object(args[3]).toString());
    log("\toptions :" + ObjC.Object(args[4]).toString());
},
```

- The Swift method $S10TelegramUI15openExternalUrl···:

```
// __handlers__/TelegramUI/_S10TelegramUI15openExternalUrl7_b1a3234e.js

onEnter: function (log, args, state) {
```

(continues on next page)

(continued from previous page)

```
    log("TelegramUI.openExternalUrl(account, url, presentationData," +
                "applicationContext, navigationController, dismissInput)");
    log("\taccount: " + ObjC.Object(args[1]).toString());
    log("\turl: " + ObjC.Object(args[2]).toString());
    log("\tpresentationData: " + args[3]);
    log("\tapplicationContext: " + ObjC.Object(args[4]).toString());
    log("\tnavigationController: " + ObjC.Object(args[5]).toString());
},
```

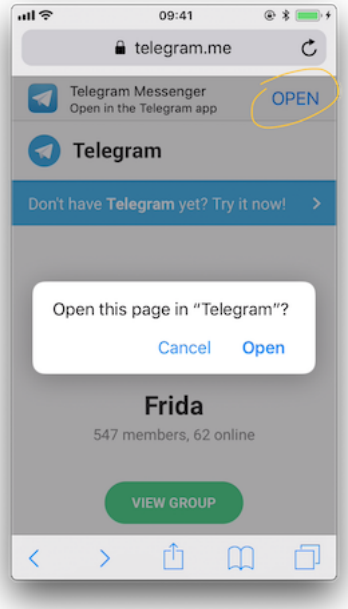
The next time we run it, we see the following output:

```
$ frida-trace -U Telegram -m "*[* *restorationHandler*]" -i "*open*Url*"
    -m "*[* *application*URL*]" -m "*[* openURL]"

  8144 ms  -[UIApplication _applicationOpenURLAction: 0x1c44ff900 payload:␣
↪0x10c5ee4c0 origin: 0x0]
  8145 ms     | -[AppDelegate application: 0x105a59980 openURL: 0x1c46ebb80␣
↪options: 0x1c0e222c0]
  8145 ms     |     application: <Application: 0x105a59980>
  8145 ms     |     openURL: tg://resolve?domain=fridadotre
  8145 ms     |     options :{
                        UIApplicationOpenURLOptionsOpenInPlaceKey = 0;
                        UIApplicationOpenURLOptionsSourceApplicationKey = "com.
↪apple.mobilesafari";
                    }
  8269 ms     |    | TelegramUI.openExternalUrl(account, url, presentationData,
                                    applicationContext, navigationController,␣
↪dismissInput)
  8269 ms     |    |    account: nil
  8269 ms     |    |    url: tg://resolve?domain=fridadotre
  8269 ms     |    |    presentationData: 0x1c4c51741
  8269 ms     |    |    applicationContext: nil
  8269 ms     |    |    navigationController: TelegramUI.PresentationData
  8274 ms     | -[UIApplication applicationOpenURL:0x1c46ebb80]
```

There you can observe the following:

- It calls application:openURL:options: from the app delegate as expected.
- The source application is Safari (“com.apple.mobilesafari”).
- application:openURL:options: handles the URL but does not open it, it calls TelegramUI.openExternalUrl for that.
- The URL being opened is tg://resolve?domain=fridadotre.
- It uses the tg:// custom URL scheme from Telegram.

It is interesting to see that if you navigate again to “https://telegram.me/fridadotre” , click on cancel and then click on the link offered by the page itself (“Open in the Telegram app”), instead of opening via custom URL scheme it will open via universal links.



You can try this while tracing both methods:

```
$ frida-trace -U Telegram -m "*[* *restorationHandler*]" -m "*[*␣
↪*application*openURL*options*]"

// After clicking "Open" on the pop-up

 16374 ms  -[AppDelegate application :0x10556b3c0 openURL :0x1c4ae0080 options␣
↪:0x1c7a28400]
 16374 ms    application :<Application: 0x10556b3c0>
 16374 ms    openURL :tg://resolve?domain=fridadotre
 16374 ms    options :{
    UIApplicationOpenURLOptionsOpenInPlaceKey = 0;
    UIApplicationOpenURLOptionsSourceApplicationKey = "com.apple.mobilesafari";
}

// After clicking "Cancel" on the pop-up and "OPEN" in the page

406575 ms  -[AppDelegate application:0x10556b3c0 continueUserActivity:0x1c063d0c0
                restorationHandler:0x16f27a898]
406575 ms  application:<Application: 0x10556b3c0>
406575 ms  continueUserActivity:<NSUserActivity: 0x1c063d0c0>
406575 ms       webpageURL:https://telegram.me/fridadotre
406575 ms       activityType:NSUserActivityTypeBrowsingWeb
406575 ms       userInfo:{
}
406575 ms  restorationHandler:<__NSStackBlock__: 0x16f27a898>
```

Reference

- owasp-mastg Testing Custom URL Schemes (MSTG-PLATFORM-3) Dynamic Analysis Performing URL Requests Opening a Link by Navigating to a Page and Letting Safari Open It

Testing for Deprecated Methods

Search for deprecated methods like:

- application:handleOpenURL:
- openURL:

- application:openURL:sourceApplication:annotation:

You may simply use frida-trace for this, to see if any of those methods are being used.

Reference

- owasp-mastg Testing Custom URL Schemes (MSTG-PLATFORM-3) Dynamic Analysis Performing URL Requests Testing for Deprecated Methods

6.3.3.3 Testing URL Schemes Source Validation

A way to discard or confirm validation could be by hooking typical methods that might be used for that. For example isEqualToString::

```
// - (BOOL)isEqualToString:(NSString *)aString;

var isEqualToString = ObjC.classes.NSString["- isEqualToString:"];

Interceptor.attach(isEqualToString.implementation, {
  onEnter: function(args) {
    var message = ObjC.Object(args[2]);
    console.log(message)
  }
});
```

If we apply this hook and call the URL scheme again:

```
$ frida -U iGoat-Swift

[iPhone::iGoat-Swift]-> var isEqualToString = ObjC.classes.NSString["-␣
↪isEqualToString:"];

                    Interceptor.attach(isEqualToString.implementation, {
                      onEnter: function(args) {
                        var message = ObjC.Object(args[2]);
                        console.log(message)
                      }
                    });
{}
[iPhone::iGoat-Swift]-> openURL("iGoat://?contactNumber=123456789&message=hola")
true
nil
```

Nothing happens. This tells us already that this method is not being used for that as we cannot find any app-package-looking string like OWASP.iGoat-Swift or com.apple.mobilesafari between the hook and the text of the tweet. However, consider that we are just probing one method, the app might be using other approach for the comparison.

Reference

- owasp-mastg Testing Custom URL Schemes (MSTG-PLATFORM-3) Dynamic Analysis Testing URL Schemes Source Validation

6.3.3.4 Fuzzing URL Schemes

If the app parses parts of the URL, you can also perform input fuzzing to detect memory corruption bugs.

What we have learned above can be now used to build your own fuzzer on the language of your choice, e.g. in Python and call the openURL using Frida' s RPC. That fuzzer should do the following:

- Generate payloads.
- For each of them call openURL.
- Check if the app generates a crash report (.ips) in /private/var/mobile/Library/Logs/CrashReporter.

The FuzzDB project offers fuzzing dictionaries that you can use as payloads.

Reference

- owasp-mastg Testing Custom URL Schemes (MSTG-PLATFORM-3) Dynamic Analysis Fuzzing URL Schemes

Using Frida

Doing this with Frida is pretty easy, you can refer to this blog post to see an example that fuzzes the iGoat-Swift app (working on iOS 11.1.2).

Before running the fuzzer we need the URL schemes as inputs. From the static analysis we know that the iGoat-Swift app supports the following URL scheme and parameters: iGoat://?contactNumber={0}&message={0}.

```
$ frida -U SpringBoard -l ios-url-scheme-fuzzing.js
[iPhone::SpringBoard]-> fuzz("iGoat", "iGoat://?contactNumber={0}&message={0}")
Watching for crashes from iGoat...
No logs were moved.
Opened URL: iGoat://?contactNumber=0&message=0
OK!
Opened URL: iGoat://?contactNumber=1&message=1
OK!
Opened URL: iGoat://?contactNumber=-1&message=-1
OK!
Opened URL: iGoat://?contactNumber=null&message=null
OK!
Opened URL: iGoat://?contactNumber=nil&message=nil
OK!
Opened URL: iGoat://?contactNumber=99999999999999999999999999999999999
&message=99999999999999999999999999999999999
OK!
Opened URL: iGoat://?
↪contactNumber=AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
...
&message=AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
...
OK!
Opened URL: iGoat://?
↪contactNumber=AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
...
&mes-
sage=AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

(continues on next page)

(continued from previous page)

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
...
OK!
Opened URL: iGoat://?contactNumber='&message='
OK!
Opened URL: iGoat://?contactNumber=%20d&message=%20d
OK!
Opened URL: iGoat://?contactNumber=%20n&message=%20n
OK!
Opened URL: iGoat://?contactNumber=%20x&message=%20x
OK!
Opened URL: iGoat://?contactNumber=%20s&message=%20s
OK!
```

The script will detect if a crash occurred. On this run it did not detect any crashed but for other apps this could be the case. We would be able to inspect the crash reports in /private/var/mobile/Library/Logs/CrashReporter or in /tmp if it was moved by the script.

Reference

- owasp-mastg Testing Custom URL Schemes (MSTG-PLATFORM-3) Fuzzing URL Schemes Using Frida

6.3.4 Rulebook

1. *If you want to implement deep linking, use universal links (Recommended)*
2. *How to use custom URL schemes (Required)*
3. *If your app handles URL scheme calls, it should properly validate URLs and their arguments and ask for user consent before performing sensitive actions (Required)*
4. *How to use universal links (Recommended)*
5. *Validate deep link URLs and their arguments (Required)*

6.3.4.1 If you want to implement deep linking, use universal links (Recommended)

The use of universal linking is recommended for the purpose of deep linking implementation.

apple-app-site-association Configuration:

```
{
    "applinks": {
        "apps": [],
        "details": [
            {
                "appID": "9JA89QQLNQ.com.apple.wwdc",
                "paths": [ "/wwdc/news/", "/videos/wwdc/2015/*"]
            },
            {
                "appID": "ABCD1234.com.apple.wwdc",
                "paths": [ "*" ]
            }
        ]
    }
}
```

Handling Universal Links setting in Swift AppDelegate:

```
import UIKit

func application(_ application: UIApplication,
                 continue userActivity: NSUserActivity,
                 restorationHandler: @escaping ([Any]?) -> Void) -> Bool
{
    guard userActivity.activityType == NSUserActivityTypeBrowsingWeb,
        let incomingURL = userActivity.webpageURL,
        let components = NSURLComponents(url: incomingURL,␣
↪resolvingAgainstBaseURL: true),
        let path = components.path,
        let params = components.queryItems else {
            return false
    }

    print("path = \(path)")

    if let albumName = params.first(where: { $0.name == "albumname" } )?.value,
        let photoIndex = params.first(where: { $0.name == "index" })?.value {

        print("album = \(albumName)")
        print("photoIndex = \(photoIndex)")
        return true

    } else {
        print("Either album name or photo index missing")
        return false
    }
}
```

If this is not noted, the following may occur.

- URL scheme hijacking attacks and unauthorized URLs may be accessed.

6.3.4.2 How to use custom URL schemes (Required)

When another app opens a URL containing a custom scheme, the system launches the app and moves it to the foreground if necessary. The system calls the app's delegate method to send data from the URL to the app. Code is added to the application method to parse the contents of the URL and perform the appropriate action.

Custom URL scheme's plist description:

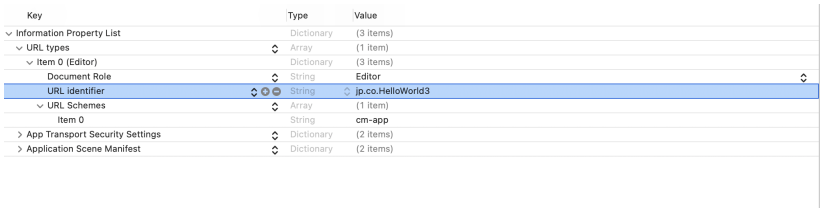


Fig 6.3.4.2.1 Adding a custom URL scheme

Automatic generation of plist xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/
↪PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
        <key>CFBundleURLTypes</key>
        <array>
```

(continues on next page)

(continued from previous page)

```
                <dict>
                        <key>CFBundleTypeRole</key>
                        <string>Editor</string>
                        <key>CFBundleURLName</key>
                        <string>jp.co.HelloWorld3</string>
                        <key>CFBundleURLSchemes</key>
                        <array>
                                <string>cm-app</string>
                        </array>
                </dict>
        </array>
        <key>NSAppTransportSecurity</key>
        <dict>
                <key>NSAllowsArbitraryLoads</key>
                <true/>
                <key>NSAllowsArbitraryLoadsForMedia</key>
                <true/>
        </dict>
        <key>UIApplicationSceneManifest</key>
        <dict>
                <key>UIApplicationSupportsMultipleScenes</key>
                <false/>
                <key>UISceneConfigurations</key>
                <dict>
                        <key>UIWindowSceneSessionRoleApplication</key>
                        <array>
                                <dict>
                                        <key>UISceneConfigurationName</key>
                                        <string>Default Configuration</string>
                                        <key>UISceneDelegateClassName</key>
                                        <string>$(PRODUCT_MODULE_NAME).
↪SceneDelegate</string>
                                        <key>UISceneStoryboardFile</key>
                                        <string>Main</string>
                                </dict>
                        </array>
                </dict>
        </dict>
</dict>
</plist>
```

Running the app via a custom URL scheme:

```
import UIKit

@main
class AppDelegate: UIResponder, UIApplicationDelegate {

    func application(_ application: UIApplication,
                     open url: URL,
                     options: [UIApplication.OpenURLOptionsKey : Any] = [:] ) ->␣
↪Bool {

        // Determine who sent the URL.
        let sendingAppID = options[.sourceApplication]
        print("source application = \(sendingAppID ?? "Unknown")")

        // Process the URL.
        guard let components = NSURLComponents(url: url, resolvingAgainstBaseURL:␣
↪true),
            let albumPath = components.path,
```

(continues on next page)

(continued from previous page)

```
            let params = components.queryItems else {
                print("Invalid URL or album path missing")
                return false
        }

        if let photoIndex = params.first(where: { $0.name == "index" })?.value {
            print("albumPath = \(albumPath)")
            print("photoIndex = \(photoIndex)")
            return true
        } else {
            print("Photo index missing")
            return false
        }
    }

    func application(_ application: UIApplication, didFinishLaunchingWithOptions␣
↪launchOptions: [UIApplication.LaunchOptionsKey: Any]?) -> Bool {
        // Override point for customization after application launch.
        return true
    }
}
```

If this is violated, the following may occur.

- Unintended data may be sent to the application.

6.3.4.3 If your app handles URL scheme calls, it should properly validate URLs and their arguments and ask for user consent before performing sensitive actions (Required)

Failure to ask the user for consent before performing a critical operation creates a security problem.

One example is the following flaw in the Skype Mobile app discovered in 2010. The Skype app registered a skype:// protocol handler and did not ask the user for permission before another app made a call to another Skype user's phone number, allowing any app to call any number without the user's knowledge. Attackers used this vulnerability to place an invisible `<iframe src="skype://xxx?call"></iframe>` (where xxx is replaced with a premium number) to allow Skype users who visit a malicious website to call premium numbers. Premium Numbers.

As a developer, you should carefully verify the URL before calling it. Only specific applications invoked via registered protocol handlers can be allowed. Another effective control method is to prompt the user to verify the operation invoked by the URL.

For argument validation, see *Validate deep link URLs and their arguments (Required)*.

All URLs are passed to the app's delegate at app startup, during app execution, or in the background. To process the received URLs, the delegate must implement the following methods.

- Obtain information about the URL and determine whether to open it.
- Open the resource specified by the URL.

For more information, please refer to the previous App Programming Guide for iOS and Apple Secure Coding Guide.

In addition, apps can send URL requests (a.k.a. queries) to other apps. This is done as follows

- Register the application query scheme that the app is requesting.
- Optionally query other apps to see if they can open a particular URL.
- Send a URL request.

All of these represent a broad attack surface that will be covered in the static and dynamic analysis sections.

Reference

- owasp-mastg Testing Custom URL Schemes (MSTG-PLATFORM-3)

If this is violated, the following may occur.

- There is a risk of transmitting users' personal information to outside parties without the users themselves being aware of it.

6.3.4.4 How to use universal links (Recommended)

When the user taps the universal link, the system can redirect the link directly to the installed app without going through Safari or the website. If the user does not have the app installed, the system will open the URL in Safari and allow the website to handle it. This can be done by defining the specification of which app to redirect to in the apple-app-site-accosiation file and setting the Domain of the file in Xcode's Associated Domains.

Specify the associated file Place on a web server (anywhere accessible, such as AWS S3)

- Place apple-app-site-accosiation file on the web server
  - appID is team name + bundleID

json description of related files:

```
{
    "webcredentials": {
        "apps": [ "${TeamID}.${BundleID}" ]
    },
    "applinks": {
        "apps": [],
        "details": [{
            "appID": "${TeamID}.${BundleID}",
            "paths": ["NOT /tests/*",
                      "NOT /settings/*",
                      "/*"]
            }]
    }
}
```

Notes on apple-app-site-association acquisition for iOS14 and later. The acquisition route has changed since iOS14, and apple-app-site-association is acquired via Apple's CDN. The problem with this change is that if the apple-app-site-association file is placed on a server with IP restrictions, Apple's CDN will not be able to acquire it, and UniversalLinks will not function.

Configuration in Xcode

- Configure Capabilities on Xcode. Add "Associated Domains" and set Domains
  - Domain is the domain of the site where the apple-app-site-association file is located.

Associated Domains description:

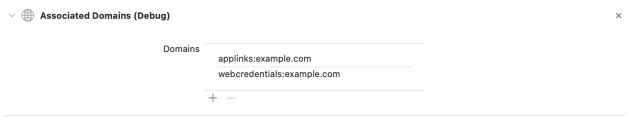


Fig 6.3.4.4.1 Adding Associated Domains

Source via Universal Link :

```
import UIKit

@main
class AppDelegate: UIResponder, UIApplicationDelegate {
```

(continues on next page)

(continued from previous page)

```
    func application(application: UIApplication, continueUserActivity␣
↪userActivity: NSUserActivity, restorationHandler: ([AnyObject]?) -> Void) ->␣
↪Bool {

        if userActivity.activityType == NSUserActivityTypeBrowsingWeb {
            // Processing Universal Links
            print(userActivity.webpageURL!)
        }

        return true
    }
}
```

If this is not noted, the following may occur.

- May redirect to an unintended application.

6.3.4.5 Validate deep link URLs and their arguments (Required)

Verify that the query parameter sent by the deep link is really the one expected by the app. It is important to do this for all parameters provided by the app. If not applicable, return false in the deep link function.

Deeplink Parameter Argument Validation:

```
import UIKit

@main
class AppDelegate: UIResponder, UIApplicationDelegate {

  func application(application: UIApplication, continueUserActivity userActivity:␣
↪NSUserActivity, restorationHandler: ([AnyObject]?) -> Void) -> Bool {

          if userActivity.activityType == NSUserActivityTypeBrowsingWeb {
              // Processing Universal Links
              guard let webpageURL = userActivity.webpageURL else {
                  return false
              }
              guard let components = NSURLComponents(url: webpageURL,␣
↪resolvingAgainstBaseURL: true),
                  let albumPath = components.path,
                  let params = components.queryItems else {
                      print("Invalid URL or album path missing")
                      return false
              }
              if (validationParam(params: params)) {
                  // validation failed
                  return false
              }
          }

          // validation success
          return true
      }

      func application(_ application: UIApplication,
                       open url: URL,
                       options: [UIApplication.OpenURLOptionsKey : Any] = [:] ) ->␣
↪Bool {

          // Determine who sent the URL.
```

(continues on next page)

(continued from previous page)

```
        let sendingAppID = options[.sourceApplication]
        print("source application = \(sendingAppID ?? "Unknown")")

        // Process the URL.
        guard let components = NSURLComponents(url: url,␣
↪resolvingAgainstBaseURL: true),
            let albumPath = components.path,
            let params = components.queryItems else {
                print("Invalid URL or album path missing")
                return false
        }

        if (validationParam(params: params)) {
            // validation failed
            return false
        }
        // validation success
        return true
    }

    func validationParam(params: [URLQueryItem]) -> Bool {

        if let index = params.first(where: { $0.name == "index" })?.value {
            return true
        } else {
            print("params index missing")
            return false
        }
    }
}
```

If this is violated, the following may occur.

- The app may send query parameters that are not expected.
- If the parameters are used in WebView as they are, there is a possibility that an unauthorized authentication input site etc. will be displayed.

6.4 MSTG-PLATFORM-4

The app does not export sensitive functionality through IPC facilities, unless these mechanisms are properly protected.

During implementation of a mobile application, developers may apply traditional techniques for IPC (such as using shared files or network sockets). The IPC system functionality offered by mobile application platforms should be used because it is much more mature than traditional techniques. Using IPC mechanisms with no security in mind may cause the application to leak or expose sensitive data.

In contrast to Android's rich Inter-Process Communication (IPC) capability, iOS offers some rather limited options for communication between apps. In fact, there's no way for apps to communicate directly. In this section we will present the different types of indirect communication offered by iOS and how to test them. Here's an overview:

- Custom URL Schemes
- Universal Links
- UIActivity Sharing
- App Extensions
- UIPasteboard

Reference

- owasp-mastg Testing for Sensitive Functionality Exposure Through IPC (MSTG-PLATFORM-4)

6.4.1 Custom URL Schemes

Please refer to the section "*Custom URL Schemes*" for more information on what custom URL schemes are and how to test them.

Reference

- owasp-mastg Testing for Sensitive Functionality Exposure Through IPC (MSTG-PLATFORM-4) Custom URL Schemes

Rulebook

- *How to use custom URL schemes (Required)*

6.4.2 Universal Links

6.4.2.1 Overview

Universal links are the iOS equivalent to Android App Links (aka. Digital Asset Links) and are used for deep linking. When tapping a universal link (to the app's website), the user will seamlessly be redirected to the corresponding installed app without going through Safari. If the app isn't installed, the link will open in Safari.

Universal links are standard web links (HTTP/HTTPS) and are not to be confused with custom URL schemes, which originally were also used for deep linking.

For example, the Telegram app supports both custom URL schemes and universal links:

- tg://resolve?domain=fridadotre is a custom URL scheme and uses the tg:// scheme.
- https://telegram.me/fridadotre is a universal link and uses the https:// scheme.

Both result in the same action, the user will be redirected to the specified chat in Telegram ("fridadotre" in this case). However, universal links give several key benefits that are not applicable when using custom URL schemes and are the recommended way to implement deep linking, according to the Apple Developer Documentation. Specifically, universal links are:

- **Unique**: Unlike custom URL schemes, universal links can't be claimed by other apps, because they use standard HTTP or HTTPS links to the app's website. They were introduced as a way to prevent URL scheme hijacking attacks (an app installed after the original app may declare the same scheme and the system might target all new requests to the last installed app).
- **Secure**: When users install the app, iOS downloads and checks a file (the Apple App Site Association or AASA) that was uploaded to the web server to make sure that the website allows the app to open URLs on its behalf. Only the legitimate owners of the URL can upload this file, so the association of their website with the app is secure.
- **Flexible**: Universal links work even when the app is not installed. Tapping a link to the website would open the content in Safari, as users expect.
- **Simple**: One URL works for both the website and the app.
- **Private**: Other apps can communicate with the app without needing to know whether it is installed.

Reference

- owasp-mastg Testing for Sensitive Functionality Exposure Through IPC (MSTG-PLATFORM-4) Universal Links

6.4.2.2 Static Analysis

Testing universal links on a static approach includes doing the following:

- Checking the Associated Domains entitlement
- Retrieving the Apple App Site Association file
- Checking the link receiver method
- Checking the data handler method
- Checking if the app is calling other app's universal links

Reference

- owasp-mastg Testing for Sensitive Functionality Exposure Through IPC (MSTG-PLATFORM-4) Universal Links Static Analysis

Checking the Associated Domains Entitlement

Universal links require the developer to add the Associated Domains entitlement and include in it a list of the domains that the app supports.

In Xcode, go to the **Capabilities** tab and search for **Associated Domains**. You can also inspect the .entitlements file looking for com.apple.developer.associated-domains. Each of the domains must be prefixed with applinks:, such as applinks:www.mywebsite.com.

Here's an example from Telegram's .entitlements file:

```
<key>com.apple.developer.associated-domains</key>
<array>
    <string>applinks:telegram.me</string>
    <string>applinks:t.me</string>
</array>
```

More detailed information can be found in the archived Apple Developer Documentation.

If you don't have the original source code you can still search for them, as explained in "Entitlements Embedded in the Compiled App Binary".

Reference

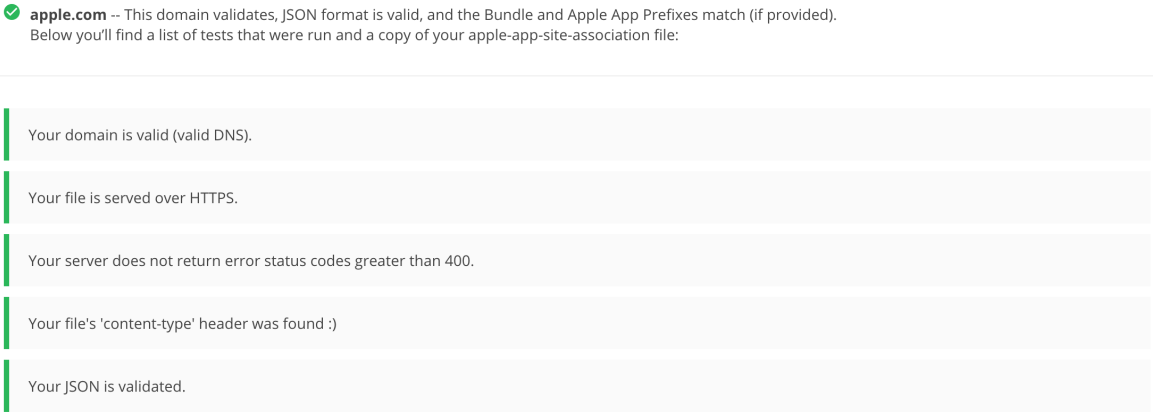
- owasp-mastg Testing for Sensitive Functionality Exposure Through IPC (MSTG-PLATFORM-4) Universal Links Static Analysis Checking the Associated Domains Entitlement

Retrieving the Apple App Site Association File

Try to retrieve the apple-app-site-association file from the server using the associated domains you got from the previous step. This file needs to be accessible via HTTPS, without any redirects, at https://<domain>/apple-app-site-association or https://<domain>/.well-known/apple-app-site-association.

You can retrieve it yourself using your browser and navigating to https://<domain>/apple-app-site-association, https://<domain>/.well-known/apple-app-site-association or using Apple's CDN at https://app-site-association.cdn-apple.com/a/v1/<domain>.

Alternatively, you can use the Apple App Site Association (AASA) Validator. After entering the domain, it will display the file, verify it for you and show the results (e.g. if it is not being properly served over HTTPS). See the following example from apple.com https://www.apple.com/.well-known/apple-app-site-association:




```
{
    "activitycontinuation": {
    "apps": [
        "W74U47NE8E.com.apple.store.Jolly"
    ]
    },
    "applinks": {
        "apps": [],
        "details": [
            {
            "appID": "W74U47NE8E.com.apple.store.Jolly",
            "paths": [
                "NOT /shop/buy-iphone/*",
                "NOT /us/shop/buy-iphone/*",
                "/xc/*",
                "/shop/buy-*",
                "/shop/product/*",
                "/shop/bag/shared_bag/*",
                "/shop/order/list",
                "/today",
                "/shop/watch/watch-accessories",
                "/shop/watch/watch-accessories/*",
                "/shop/watch/bands",
            ] } ] }
}
```

The "details" key inside "applinks" contains a JSON representation of an array that might contain one or more apps. The "appID" should match the "application-identifier" key from the app' s entitlements. Next, using the "paths" key, the developers can specify certain paths to be handled on a per app basis. Some apps, like Telegram use a standalone * ("paths" : ["*"]) in order to allow all possible paths. Only if specific areas of the website should not be handled by some app, the developer can restrict access by excluding them by prepending a "NOT " (note the whitespace after the T) to the corresponding path. Also remember that the system will look for matches by following the order of the dictionaries in the array (first match wins).

This path exclusion mechanism is not to be seen as a security feature but rather as a filter that developer might use to specify which apps open which links. By default, iOS does not open any unverified links.

Remember that universal links verification occurs at installation time. iOS retrieves the AASA file for the declared domains (applinks) in its com.apple.developer.associated-domains entitlement. iOS will refuse to open those links if the verification did not succeed. Some reasons to fail verification might include:

- The AASA file is not served over HTTPS.
- The AASA is not available.
- The appIDs do not match (this would be the case of a malicious app). iOS would successfully prevent any possible hijacking attacks.

Reference

- owasp-mastg Testing for Sensitive Functionality Exposure Through IPC (MSTG-PLATFORM-4) Universal Links Static Analysis Retrieving the Apple App Site Association File

Checking the Link Receiver Method

In order to receive links and handle them appropriately, the app delegate has to implement application:continueUserActivity:restorationHandler:. If you have the original project try searching for this method.

Please note that if the app uses openURL:options:completionHandler: to open a universal link to the app's website, the link won't open in the app. As the call originates from the app, it won't be handled as a universal link.

From Apple Docs: When iOS launches your app after a user taps a universal link, you receive an NSUserActivity object with an activityType value of NSUserActivityTypeBrowsingWeb. The activity object's webpageURL property contains the URL that the user is accessing. The webpage URL property always contains an HTTP or HTTPS URL, and you can use NSURLComponents APIs to manipulate the components of the URL. [···] To protect user's privacy and security, you should not use HTTP when you need to transport data; instead, use a secure transport protocol such as HTTPS.

From the note above we can highlight that:

- The mentioned NSUserActivity object comes from the continueUserActivity parameter, as seen in the method above.
- The scheme of the webpageURL must be HTTP or HTTPS (any other scheme should throw an exception). The scheme instance property of URLComponents / NSURLComponents can be used to verify this.

If you don't have the original source code you can use radare2 or rabin2 to search the binary strings for the link receiver method:

```
$ rabin2 -zq Telegram\ X.app/Telegram\ X | grep restorationHan

0x1000deea9 53 52 application:continueUserActivity:restorationHandler:
```

Reference

- owasp-mastg Testing for Sensitive Functionality Exposure Through IPC (MSTG-PLATFORM-4) Universal Links Static Analysis Checking the Link Receiver Method

Checking the Data Handler Method

You should check how the received data is validated. Apple explicitly warns about this:

Universal links offer a potential attack vector into your app, so make sure to validate all URL parameters and discard any malformed URLs. In addition, limit the available actions to those that do not risk the user's data. For example, do not allow universal links to directly delete content or access sensitive information about the user. When testing your URL-handling code, make sure your test cases include improperly formatted URLs.

As stated in the Apple Developer Documentation, when iOS opens an app as the result of a universal link, the app receives an NSUserActivity object with an activityType value of NSUserActivityTypeBrowsingWeb. The activity object's webpageURL property contains the HTTP or HTTPS URL that the user accesses. The following example in Swift verifies exactly this before opening the URL:

```
func application(_ application: UIApplication, continue userActivity:␣
↪NSUserActivity,
                 restorationHandler: @escaping ([UIUserActivityRestoring]?) ->␣
↪Void) -> Bool {
    // ...
    if userActivity.activityType == NSUserActivityTypeBrowsingWeb, let url =␣
↪userActivity.webpageURL {
        application.open(url, options: [:], completionHandler: nil)
    }

    return true
}
```

In addition, remember that if the URL includes parameters, they should not be trusted before being carefully sanitized and validated (even when coming from trusted domain). For example, they might have been spoofed by an attacker or might include malformed data. If that is the case, the whole URL and therefore the universal link request must be discarded.

The NSURLComponents API can be used to parse and manipulate the components of the URL. This can be also part of the method application:continueUserActivity:restorationHandler: itself or might occur on a separate method being called from it. The following example demonstrates this:

```
func application(_ application: UIApplication,
                 continue userActivity: NSUserActivity,
                 restorationHandler: @escaping ([Any]?) -> Void) -> Bool {
    guard userActivity.activityType == NSUserActivityTypeBrowsingWeb,
        let incomingURL = userActivity.webpageURL,
        let components = NSURLComponents(url: incomingURL,␣
↪resolvingAgainstBaseURL: true),
        let path = components.path,
        let params = components.queryItems else {
        return false
    }

    if let albumName = params.first(where: { $0.name == "albumname" })?.value,
        let photoIndex = params.first(where: { $0.name == "index" })?.value {
        // Interact with album name and photo index

        return true

    } else {
        // Handle when album and/or album name or photo index missing

        return false
    }
}
```

Finally, as stated above, be sure to verify that the actions triggered by the URL do not expose sensitive information or risk the user's data on any way.

Reference

- owasp-mastg Testing for Sensitive Functionality Exposure Through IPC (MSTG-PLATFORM-4) Universal Links Static Analysis Checking the Data Handler Method

Rulebook

- *How to validate data received as a result of Universal Link (Required)*
- *Use secure transfer protocols to protect user privacy and security (Required)*
- *If the URL contains parameters, do not trust the URL without careful sanitization and validation (Recommended)*

Checking if the App is Calling Other App's Universal Links

An app might be calling other apps via universal links in order to simply trigger some actions or to transfer information, in that case, it should be verified that it is not leaking sensitive information.

If you have the original source code, you can search it for the openURL:options: completionHandler: method and check the data being handled.

Note that the openURL:options:completionHandler: method is not only used to open universal links but also to call custom URL schemes.

This is an example from the Telegram app:

```
}, openUniversalUrl: { url, completion in
    if #available(iOS 10.0, *) {
        var parsedUrl = URL(string: url)
```

(continues on next page)

(continued from previous page)

```
        if let parsed = parsedUrl {
            if parsed.scheme == nil || parsed.scheme!.isEmpty {
                parsedUrl = URL(string: "https://\(url)")
            }
        }

        if let parsedUrl = parsedUrl {
            return UIApplication.shared.open(parsedUrl,
                        options: [UIApplicationOpenURLOptionUniversalLinksOnly:␣
↪true as NSNumber],
                        completionHandler: { value in completion.completion(value)}
            )
```

Note how the app adapts the scheme to "https" before opening it and how it uses the option UIApplicationOpenURLOptionUniversalLinksOnly: true that opens the URL only if the URL is a valid universal link and there is an installed app capable of opening that URL.

If you don't have the original source code, search in the symbols and in the strings of the app binary. For example, we will search for Objective-C methods that contain "openURL":

```
$ rabin2 -zq Telegram\ X.app/Telegram\ X | grep openURL

0x1000dee3f 50 49 application:openURL:sourceApplication:annotation:
0x1000dee71 29 28 application:openURL:options:
0x1000df2c9 9 8 openURL:
0x1000df772 35 34 openURL:options:completionHandler:
```

As expected, openURL:options:completionHandler: is among the ones found (remember that it might be also present because the app opens custom URL schemes). Next, to ensure that no sensitive information is being leaked you'll have to perform dynamic analysis and inspect the data being transmitted. Please refer to "*Identifying and Hooking the URL Handler Method*" in the "Dynamic Analysis" of "Custom URL Schemes" section for some examples on hooking and tracing this method.

Reference

- owasp-mastg Testing for Sensitive Functionality Exposure Through IPC (MSTG-PLATFORM-4) Universal Links Static Analysis Checking if the App is Calling Other App's Universal Links

Rulebook

- *Make sure you are not exposing sensitive information when calling other apps via universal links (Required)*

6.4.2.3 Dynamic Analysis

If an app is implementing universal links, you should have the following outputs from the static analysis:

- the associated domains
- the Apple App Site Association file
- the link receiver method
- the data handler method

You can use this now to dynamically test them:

- Triggering universal links
- Identifying valid universal links
- Tracing the link receiver method
- Checking how the links are opened

Reference

- owasp-mastg Testing for Sensitive Functionality Exposure Through IPC (MSTG-PLATFORM-4) Universal Links Dynamic Analysis

Triggering Universal Links

Unlike custom URL schemes, unfortunately you cannot test universal links from Safari just by typing them in the search bar directly as this is not allowed by Apple. But you can test them anytime using other apps like the Notes app:

- Open the Notes app and create a new note.
- Write the links including the domain.
- Leave the editing mode in the Notes app.
- Long press the links to open them (remember that a standard click triggers the default option).

To do it from Safari you will have to find an existing link on a website that once clicked, it will be recognized as a Universal Link. This can be a bit time consuming.

Alternatively you can also use Frida for this, see the section "*Performing URL Requests*" for more details.

Reference

- owasp-mastg Testing for Sensitive Functionality Exposure Through IPC (MSTG-PLATFORM-4) Universal Links Dynamic Analysis Triggering Universal Links

Identifying Valid Universal Links

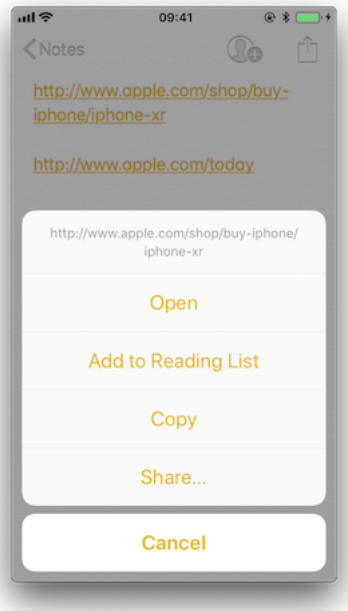
First of all we will see the difference between opening an allowed Universal Link and one that shouldn't be allowed.

From the apple-app-site-association of apple.com we have seen above we chose the following paths:

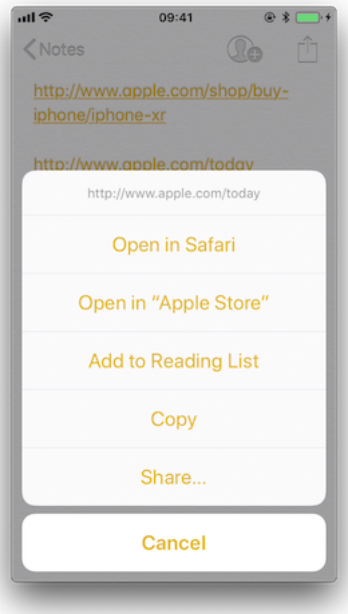
```
"paths": [
    "NOT /shop/buy-iphone/*",
    ...
    "/today",
```

One of them should offer the "Open in app" option and the other should not.

If we long press on the first one (http://www.apple.com/shop/buy-iphone/iphone-xr) it only offers the option to open it (in the browser).



If we long press on the second (http://www.apple.com/today) it shows options to open it in Safari and in "Apple Store" :



Note that there is a difference between a click and a long press. Once we long press a link and select an option, e.g. "Open in Safari" , this will become the default option for all future clicks until we long press again and select another option.

If we repeat the process on the method application:continueUserActivity: restorationHandler: by either hooking or tracing, we will see how it gets called as soon as we open the allowed universal link. For this you can use for example frida-trace:

```
frida-trace -U "Apple Store" -m "*[* *restorationHandler*]"
```

Reference

- owasp-mastg Testing for Sensitive Functionality Exposure Through IPC (MSTG-PLATFORM-4) Universal Links Dynamic Analysis Identifying Valid Universal Links

Tracing the Link Receiver Method

This section explains how to trace the link receiver method and how to extract additional information. For this example, we will use Telegram, as there are no restrictions in its apple-app-site-association file:

```
{
    "applinks": {
        "apps": [],
        "details": [
            {
                "appID": "X834Q8SBVP.org.telegram.TelegramEnterprise",
                "paths": [
                    "*"
                ]
            },
            {
                "appID": "C67CF9S4VU.ph.telegra.Telegraph",
                "paths": [
                    "*"
                ]
```

(continues on next page)

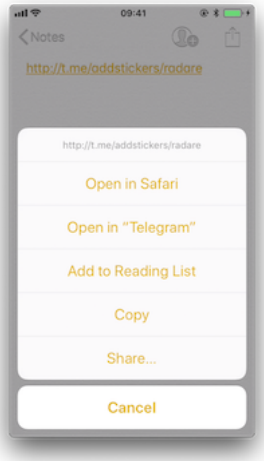
(continued from previous page)

```
            },
            {
                "appID": "X834Q8SBVP.org.telegram.Telegram-iOS",
                "paths": [
                    "*"
                ]
            }
        ]
    }
}
```

In order to open the links we will also use the Notes app and frida-trace with the following pattern:

```
frida-trace -U Telegram -m "*[* *restorationHandler*]"
```

Write https://t.me/addstickers/radare (found through a quick Internet research) and open it from the Notes app.



First we let frida-trace generate the stubs in __handlers__/:

```
$ frida-trace -U Telegram -m "*[* *restorationHandler*]"
Instrumenting functions...
-[AppDelegate application:continueUserActivity:restorationHandler:]
```

You can see that only one function was found and is being instrumented. Trigger now the universal link and observe the traces.

```
298382 ms  -[AppDelegate application:0x10556b3c0 continueUserActivity:0x1c4237780
               restorationHandler:0x16f27a898]
```

You can observe that the function is in fact being called. You can now add code to the stubs in __handlers__/ to obtain more details:

```
// __handlers__/__AppDelegate_application_contin_8e36bbb1.js

onEnter: function (log, args, state) {
    log("-[AppDelegate application: " + args[2] + " continueUserActivity: " +␣
↪args[3] +
                    " restorationHandler: " + args[4] + "]");
    log("\tapplication: " + ObjC.Object(args[2]).toString());
    log("\tcontinueUserActivity: " + ObjC.Object(args[3]).toString());
    log("\t\twebpageURL: " + ObjC.Object(args[3]).webpageURL().toString());
```

(continues on next page)

(continued from previous page)

```
    log("\t\tactivityType: " + ObjC.Object(args[3]).activityType().toString());
    log("\t\tuserInfo: " + ObjC.Object(args[3]).userInfo().toString());
    log("\trestorationHandler: " +ObjC.Object(args[4]).toString());
  },
```

The new output is:

```
298382 ms  -[AppDelegate application:0x10556b3c0 continueUserActivity:0x1c4237780
               restorationHandler:0x16f27a898]
298382 ms  application:<Application: 0x10556b3c0>
298382 ms  continueUserActivity:<NSUserActivity: 0x1c4237780>
298382 ms       webpageURL:http://t.me/addstickers/radare
298382 ms       activityType:NSUserActivityTypeBrowsingWeb
298382 ms       userInfo:{
}
298382 ms  restorationHandler:<__NSStackBlock__: 0x16f27a898>
```

Apart from the function parameters we have added more information by calling some methods from them to get more details, in this case about the NSUserActivity. If we look in the Apple Developer Documentation we can see what else we can call from this object.

Reference

- owasp-mastg Testing for Sensitive Functionality Exposure Through IPC (MSTG-PLATFORM-4) Universal Links Dynamic Analysis Tracing the Link Receiver Method

Checking How the Links Are Opened

If you want to know more about which function actually opens the URL and how the data is actually being handled you should keep investigating.

Extend the previous command in order to find out if there are any other functions involved into opening the URL.

```
frida-trace -U Telegram -m "*[* *restorationHandler*]" -i "*open*Url*"
```

-i includes any method. You can also use a glob pattern here (e.g. -i "*open*Url*" means "include any function containing 'open', then 'Url' and something else")

Again, we first let frida-trace generate the stubs in __handlers__/:

```
$ frida-trace -U Telegram -m "*[* *restorationHandler*]" -i "*open*Url*"
Instrumenting functions...
-[AppDelegate application:continueUserActivity:restorationHandler:]
$S10TelegramUI0A19ApplicationBindingsC16openUniversalUrlyySS_
 ↪AA0ac4OpenG10Completion...
$S10TelegramUI15openExternalUrl7account7context3url05forceD016presentationData18ap-
plication...
$S10TelegramUI31AuthorizationSequenceControllerC7ac-
count7strings7openUrl5apiId0J4HashAC0A4Core19...
...
```

Now you can see a long list of functions but we still don't know which ones will be called. Trigger the universal link again and observe the traces.

```
           /* TID 0x303 */
298382 ms  -[AppDelegate application:0x10556b3c0 continueUserActivity:0x1c4237780
               restorationHandler:0x16f27a898]
298619 ms     |
 ↪$S10TelegramUI15openExternalUrl7account7context3url05forceD016presentationData
                ⌙
 ↪18applicationContext20navigationController12dismissInputy0A4Core7AccountC_AA
                ⌙
```

(continues on next page)

(continued from previous page)

```
↪14OpenURLContextOSSSbAA012PresentationK0CAA0a11ApplicationM0C7Display0
                10NavigationO0CSgyyctF()
```

Apart from the Objective-C method, now there is one Swift function that is also of your interest.

There is probably no documentation for that Swift function but you can just demangle its symbol using swift-demangle via xcrun:

xcrun can be used invoke Xcode developer tools from the command-line, without having them in the path. In this case it will locate and run swift-demangle, an Xcode tool that demangles Swift symbols.

```
$ xcrun swift-demangle␣
↪S10TelegramUI15openExternalUrl7account7context3url05forceD016presentationData
18applicationContext20navigationController12dismissInputy0A4Core7AccountC_
↪AA14OpenURLContextOSSSbAA0
12PresentationK0CAA0a11ApplicationM0C7Display010NavigationO0CSgyyctF
```

Resulting in:

```
---> TelegramUI.openExternalUrl(
    account: TelegramCore.Account, context: TelegramUI.OpenURLContext, url: Swift.
↪String,
    forceExternal: Swift.Bool, presentationData: TelegramUI.PresentationData,
    applicationContext: TelegramUI.TelegramApplicationContext,
    navigationController: Display.NavigationController?, dismissInput: () -> ()) ->
↪ ()
```

This not only gives you the class (or module) of the method, its name and the parameters but also reveals the parameter types and return type, so in case you need to dive deeper now you know where to start.

For now we will use this information to properly print the parameters by editing the stub file:

```
// __handlers__/TelegramUI/_S10TelegramUI15openExternalUrl7_b1a3234e.js

onEnter: function (log, args, state) {

    log("TelegramUI.openExternalUrl(account: TelegramCore.Account,
        context: TelegramUI.OpenURLContext, url: Swift.String, forceExternal:␣
↪Swift.Bool,
        presentationData: TelegramUI.PresentationData,
        applicationContext: TelegramUI.TelegramApplicationContext,
        navigationController: Display.NavigationController?, dismissInput: () ->␣
↪()) -> ()");
    log("\taccount: " + ObjC.Object(args[0]).toString());
    log("\tcontext: " + ObjC.Object(args[1]).toString());
    log("\turl: " + ObjC.Object(args[2]).toString());
    log("\tpresentationData: " + args[3]);
    log("\tapplicationContext: " + ObjC.Object(args[4]).toString());
    log("\tnavigationController: " + ObjC.Object(args[5]).toString());
  },
```

This way, the next time we run it we get a much more detailed output:

```
298382 ms  -[AppDelegate application:0x10556b3c0 continueUserActivity:0x1c4237780
                restorationHandler:0x16f27a898]
298382 ms  application:<Application: 0x10556b3c0>
298382 ms  continueUserActivity:<NSUserActivity: 0x1c4237780>
298382 ms       webpageURL:http://t.me/addstickers/radare
298382 ms       activityType:NSUserActivityTypeBrowsingWeb
298382 ms       userInfo:{
}
298382 ms  restorationHandler:<__NSStackBlock__: 0x16f27a898>
```

(continues on next page)

(continued from previous page)

```
298619 ms     | TelegramUI.openExternalUrl(account: TelegramCore.Account,
context: TelegramUI.OpenURLContext, url: Swift.String, forceExternal: Swift.Bool,
presentationData: TelegramUI.PresentationData, applicationContext:
TelegramUI.TelegramApplicationContext, navigationController: Display.
↪NavigationController?,
dismissInput: () -> ()) -> ()
298619 ms     |     account: TelegramCore.Account
298619 ms     |     context: nil
298619 ms     |     url: http://t.me/addstickers/radare
298619 ms     |     presentationData: 0x1c4e40fd1
298619 ms     |     applicationContext: nil
298619 ms     |     navigationController: TelegramUI.PresentationData
```

There you can observe the following:

- It calls application:continueUserActivity:restorationHandler: from the app delegate as expected.
- application:continueUserActivity:restorationHandler: handles the URL but does not open it, it calls TelegramUI.openExternalUrl for that.
- The URL being opened is https://t.me/addstickers/radare.

You can now keep going and try to trace and verify how the data is being validated. For example, if you have two apps that communicate via universal links you can use this to see if the sending app is leaking sensitive data by hooking these methods in the receiving app. This is especially useful when you don't have the source code as you will be able to retrieve the full URL that you wouldn't see other way as it might be the result of clicking some button or triggering some functionality.

In some cases, you might find data in userInfo of the NSUserActivity object. In the previous case there was no data being transferred but it might be the case for other scenarios. To see this, be sure to hook the userInfo property or access it directly from the continueUserActivity object in your hook (e.g. by adding a line like this log("userInfo:" + ObjC.Object(args[3]).userInfo().toString());).

Reference

- owasp-mastg Testing for Sensitive Functionality Exposure Through IPC (MSTG-PLATFORM-4) Universal Links Dynamic Analysis Checking How the Links Are Opened

Final Notes about Universal Links and Handoff

Universal links and Apple's Handoff feature are related:

- Both rely on the same method when receiving data:

```
application:continueUserActivity:restorationHandler:
```

- Like universal links, the Handoff's Activity Continuation must be declared in the com.apple.developer.associated-domains entitlement and in the server's apple-app-site-association file (in both cases via the keyword "activitycontinuation":). See "Retrieving the Apple App Site Association File" above for an example.

Actually, the previous example in "Checking How the Links Are Opened" is very similar to the "Web Browser-to-Native App Handoff" scenario described in the "Handoff Programming Guide":

If the user is using a web browser on the originating device, and the receiving device is an iOS device with a native app that claims the domain portion of the webpageURL property, then iOS launches the native app and sends it an NSUserActivity object with an activityType value of NSUserActivityTypeBrowsingWeb. The webpageURL property contains the URL the user was visiting, while the userInfo dictionary is empty.

In the detailed output above you can see that NSUserActivity object we've received meets exactly the mentioned points:

```
298382 ms  -[AppDelegate application:0x10556b3c0 continueUserActivity:0x1c4237780
               restorationHandler:0x16f27a898]
298382 ms  application:<Application: 0x10556b3c0>
298382 ms  continueUserActivity:<NSUserActivity: 0x1c4237780>
298382 ms       webpageURL:http://t.me/addstickers/radare
298382 ms       activityType:NSUserActivityTypeBrowsingWeb
298382 ms       userInfo:{
}
298382 ms  restorationHandler:<__NSStackBlock__: 0x16f27a898>
```

This knowledge should help you when testing apps supporting Handoff.

Reference

- owasp-mastg Testing for Sensitive Functionality Exposure Through IPC (MSTG-PLATFORM-4) Universal Links Dynamic Analysis Final Notes about Universal Links and Handoff

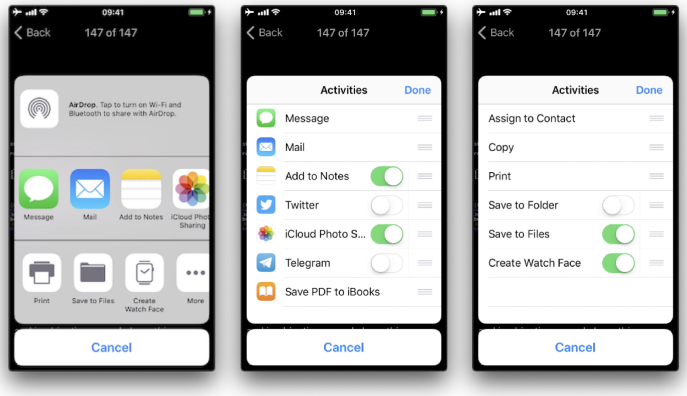
Rulebook

- *Universal Links and Handoff (Required)*

6.4.3 UIActivity Sharing

6.4.3.1 Overview

Starting on iOS 6 it is possible for third-party apps to share data (items) via specific mechanisms like AirDrop, for example. From a user perspective, this feature is the well-known system-wide "Share Activity Sheet" that appears after clicking on the "Share" button.



The available built-in sharing mechanisms (aka. Activity Types) include:

- airDrop
- assignToContact
- copyToPasteboard
- mail
- message
- postToFacebook
- postToTwitter

A full list can be found in UIActivity.ActivityType. If not considered appropriate for the app, the developers have the possibility to exclude some of these sharing mechanisms.

Reference

- owasp-mastg Testing for Sensitive Functionality Exposure Through IPC (MSTG-PLATFORM-4) UIActivity Sharing

6.4.3.2 Static Analysis

Sending Items

When testing UIActivity Sharing you should pay special attention to:

- the data (items) being shared,
- the custom activities,
- the excluded activity types.

Data sharing via UIActivity works by creating a UIActivityViewController and passing it the desired items (URLs, text, a picture) on init(activityItems: applicationActivities:).

As we mentioned before, it is possible to exclude some of the sharing mechanisms via the controller's excludedActivityTypes property. It is highly recommended to do the tests using the latest versions of iOS as the number of activity types that can be excluded can increase. The developers have to be aware of this and **explicitly exclude** the ones that are not appropriate for the app data. Some activity types might not be even documented like "Create Watch Face".

If having the source code, you should take a look at the UIActivityViewController:

- Inspect the activities passed to the init(activityItems:applicationActivities:) method.
- Check if it defines custom activities (also being passed to the previous method).
- Verify the excludedActivityTypes, if any.

If you only have the compiled/installed app, try searching for the previous method and property, for example:

```
$ rabin2 -zq Telegram\ X.app/Telegram\ X | grep -i activityItems
0x1000df034 45 44 initWithActivityItems:applicationActivities:
```

Reference

- owasp-mastg Testing for Sensitive Functionality Exposure Through IPC (MSTG-PLATFORM-4) UIActivity Sharing Static Analysis Sending Items

Rulebook

- *Explicit exclusion of activity types targeted by UIActivity Sharing (Required)*

Receiving Items

When receiving items, you should check:

- if the app declares custom document types by looking into Exported/Imported UTIs ("Info" tab of the Xcode project). The list of all system declared UTIs (Uniform Type Identifiers) can be found in the archived Apple Developer Documentation.
- if the app specifies any document types that it can open by looking into Document Types ("Info" tab of the Xcode project). If present, they consist of name and one or more UTIs that represent the data type (e.g. "public.png" for PNG files). iOS uses this to determine if the app is eligible to open a given document (specifying Exported/Imported UTIs is not enough).
- if the app properly verifies the received data by looking into the implementation of application:openURL:options: (or its deprecated version UIApplicationDelegate application:openURL:sourceApplication:annotation:) in the app delegate.

If not having the source code you can still take a look into the Info.plist file and search for:

- UTExportedTypeDeclarations/UTImportedTypeDeclarations if the app declares exported/imported custom document types.
- CFBundleDocumentTypes to see if the app specifies any document types that it can open.

A very complete explanation about the use of these keys can be found on Stackoverflow.

Let's see a real-world example. We will take a File Manager app and take a look at these keys. We used objection here to read the Info.plist file.

```
objection --gadget SomeFileManager run ios plist cat Info.plist
```

Note that this is the same as if we would retrieve the IPA from the phone or accessed via e.g. SSH and navigated to the corresponding folder in the IPA / app sandbox. However, with objection we are just one command away from our goal and this can be still considered static analysis.

The first thing we noticed is that app does not declare any imported custom document types but we could find a couple of exported ones:

```
UTExportedTypeDeclarations =     (
    {
UTTypeConformsTo =             (
    "public.data"
);
UTTypeDescription = "SomeFileManager Files";
UTTypeIdentifier = "com.some.filemanager.custom";
UTTypeTagSpecification =             {
    "public.filename-extension" =                 (
        ipa,
        deb,
        zip,
        rar,
        tar,
        gz,
        ...
        key,
        pem,
        p12,
        cer
    );
};
}
);
```

The app also declares the document types it opens as we can find the key CFBundleDocumentTypes:

```
CFBundleDocumentTypes =     (
        {
        ...
        CFBundleTypeName = "SomeFileManager Files";
        LSItemContentTypes =             (
            "public.content",
            "public.data",
            "public.archive",
            "public.item",
            "public.database",
            "public.calendar-event",
            ...
        );
    }
);
```

We can see that this File Manager will try to open anything that conforms to any of the UTIs listed in LSItemContentTypes and it's ready to open files with the extensions listed in UTTypeTagSpecification/" public.filename-extension" .

Please take a note of this because it will be useful if you want to search for vulnerabilities when dealing with the different types of files when performing dynamic analysis.

Reference

- owasp-mastg Testing for Sensitive Functionality Exposure Through IPC (MSTG-PLATFORM-4) UIActivity Sharing Static Analysis Receiving Items

Rulebook

- *Always confirm when receiving files from other apps (Required)*

6.4.3.3 Dynamic Analysis

Sending Items

There are three main things you can easily inspect by performing dynamic instrumentation:

- The activityItems: an array of the items being shared. They might be of different types, e.g. one string and one picture to be shared via a messaging app.
- The applicationActivities: an array of UIActivity objects representing the app's custom services.
- The excludedActivityTypes: an array of the Activity Types that are not supported, e.g. postToFacebook.

To achieve this you can do two things:

- Hook the method we have seen in the static analysis (init(activityItems: applicationActivities:)) to get the activityItems and applicationActivities.
- Find out the excluded activities by hooking excludedActivityTypes property.

Let's see an example using Telegram to share a picture and a text file. First prepare the hooks, we will use the Frida REPL and write a script for this:

```
Interceptor.attach(
    ObjC.classes.
        UIActivityViewController['- initWithActivityItems:applicationActivities:'].
↪implementation, {
      onEnter: function (args) {

        printHeader(args)

        this.initWithActivityItems = ObjC.Object(args[2]);
        this.applicationActivities = ObjC.Object(args[3]);

        console.log("initWithActivityItems: " + this.initWithActivityItems);
        console.log("applicationActivities: " + this.applicationActivities);

      },
      onLeave: function (retval) {
        printRet(retval);
      }
    });

    Interceptor.attach(
    ObjC.classes.UIActivityViewController['- excludedActivityTypes'].
↪implementation, {
      onEnter: function (args) {
        printHeader(args)
      },
      onLeave: function (retval) {
        printRet(retval);
      }
    });
```

(continues on next page)

(continued from previous page)

```
    function printHeader(args) {
      console.log(Memory.readUtf8String(args[1]) + " @ " + args[1])
    };

    function printRet(retval) {
      console.log('RET @ ' + retval + ': ' );
      try {
        console.log(new ObjC.Object(retval).toString());
      } catch (e) {
        console.log(retval.toString());
      }
    };
```

You can store this as a JavaScript file, e.g. inspect_send_activity_data.js and load it like this:

```
frida -U Telegram -l inspect_send_activity_data.js
```

Now observe the output when you first share a picture:

```
[*] initWithActivityItems:applicationActivities: @ 0x18c130c07
initWithActivityItems: (
    "<UIImage: 0x1c4aa0b40> size {571, 264} orientation 0 scale 1.000000"
)
applicationActivities: nil
RET @ 0x13cb2b800:
<UIActivityViewController: 0x13cb2b800>

[*] excludedActivityTypes @ 0x18c0f8429
RET @ 0x0:
nil
```

and then a text file:

```
[*] initWithActivityItems:applicationActivities: @ 0x18c130c07
initWithActivityItems: (
    "<QLActivityItemProvider: 0x1c4a30140>",
    "<UIPrintInfo: 0x1c0699a50>"
)
applicationActivities: (
)
RET @ 0x13c4bdc00:
<_UIDICActivityViewController: 0x13c4bdc00>

[*] excludedActivityTypes @ 0x18c0f8429
RET @ 0x1c001b1d0:
(
    "com.apple.UIKit.activity.MarkupAsPDF"
)
```

You can see that:

- For the picture, the activity item is a UIImage and there are no excluded activities.
- For the text file there are two different activity items and com.apple.UIKit.activity. MarkupAsPDF is excluded.

In the previous example, there were no custom applicationActivities and only one excluded activity. However, to better illustrate what you can expect from other apps we have shared a picture using another app, here you can see a bunch of application activities and excluded activities (output was edited to hide the name of the originating app):

```
[*] initWithActivityItems:applicationActivities: @ 0x18c130c07
initWithActivityItems: (
    "<SomeActivityItemProvider: 0x1c04bd580>"
```

(continues on next page)

(continued from previous page)

```
)
applicationActivities: (
    "<SomeActionItemActivityAdapter: 0x141de83b0>",
    "<SomeActionItemActivityAdapter: 0x147971cf0>",
    "<SomeOpenInSafariActivity: 0x1479f0030>",
    "<SomeOpenInChromeActivity: 0x1c0c8a500>"
)
RET @ 0x142138a00:
<SomeActivityViewController: 0x142138a00>

[*] excludedActivityTypes @ 0x18c0f8429
RET @ 0x14797c3e0:
(
    "com.apple.UIKit.activity.Print",
    "com.apple.UIKit.activity.AssignToContact",
    "com.apple.UIKit.activity.SaveToCameraRoll",
    "com.apple.UIKit.activity.CopyToPasteboard",
)
```

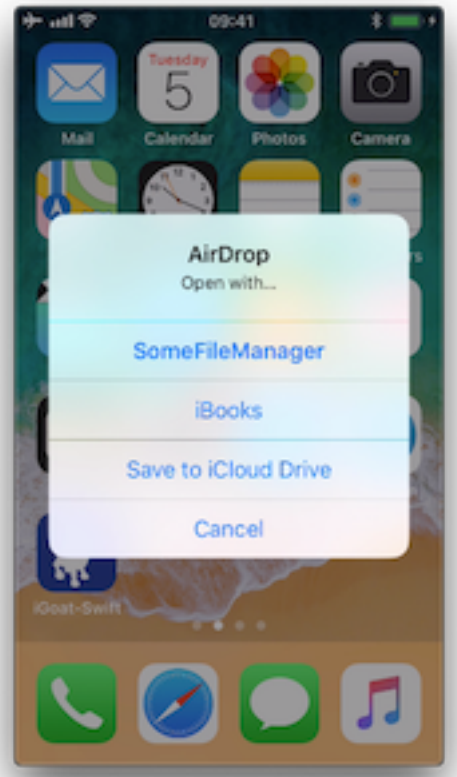
Receiving Items

After performing the static analysis you would know the document types that the app can open and if it declares any custom document types and (part of) the methods involved. You can use this now to test the receiving part:

- Share a file with the app from another app or send it via AirDrop or e-mail. Choose the file so that it will trigger the "Open with⋯" dialogue (that is, there is no default app that will open the file, a PDF for example).
- Hook application:openURL:options: and any other methods that were identified in a previous static analysis.
- Observe the app behavior.
- In addition, you could send specific malformed files and/or use a fuzzing technique.

To illustrate this with an example we have chosen the same real-world file manager app from the static analysis section and followed these steps:

1. Send a PDF file from another Apple device (e.g. a MacBook) via Airdrop.
2. Wait for the **AirDrop** popup to appear and click on **Accept**.
3. As there is no default app that will open the file, it switches to the **Open with⋯** popup. There, we can select the app that will open our file. The next screenshot shows this (we have modified the display name using Frida to conceal the app' s real name):



4. After selecting **SomeFileManager** we can see the following:

```
(0x1c4077000)  -[AppDelegate application:openURL:options:]
application: <UIApplication: 0x101c00950>
openURL: file:///var/mobile/Library/Application%20Support
                    /Containers/com.some.filemanager/Documents/Inbox/OWASP_MASVS.
↪pdf
options: {
    UIApplicationOpenURLOptionsAnnotationKey =     {
        LSMoveDocumentOnOpen = 1;
    };
    UIApplicationOpenURLOptionsOpenInPlaceKey = 0;
    UIApplicationOpenURLOptionsSourceApplicationKey = "com.apple.sharingd";
    "_UIApplicationOpenURLOptionsSourceProcessHandleKey" = "<FBSProcessHandle:␣
↪0x1c3a63140;
                                                              sharingd:605;␣
↪valid: YES>";
}
0x18c7930d8 UIKit!__58-[UIApplication _applicationOpenURLAction:payload:origin:]_
↪block_invoke
...
0x1857cdc34 FrontBoardServices!-[FBSSerialQueue _performNextFromRunLoopSource]
RET: 0x1
```

As you can see, the sending application is com.apple.sharingd and the URL's scheme is file://. Note that once we select the app that should open the file, the system already moved the file to the corresponding destination, that is to the app's Inbox. The apps are then responsible for deleting the files inside their Inboxes. This app, for example, moves the file to /var/mobile/Documents/ and removes it from the Inbox.

```
(0x1c002c760)  -[XXFileManager moveItemAtPath:toPath:error:]
moveItemAtPath: /var/mobile/Library/Application Support/Containers
```

(continues on next page)

(continued from previous page)

```
                              /com.some.filemanager/Documents/Inbox/OWASP_MASVS.pdf
toPath: /var/mobile/Documents/OWASP_MASVS (1).pdf
error: 0x16f095bf8
0x100f24e90 SomeFileManager!-[AppDelegate __handleOpenURL:]
0x100f25198 SomeFileManager!-[AppDelegate application:openURL:options:]
0x18c7930d8 UIKit!__58-[UIApplication _applicationOpenURLAction:payload:origin:]_
↪block_invoke
...
0x1857cd9f4 FrontBoardServices!__FBSSERIALQUEUE_IS_CALLING_OUT_TO_A_BLOCK__
RET: 0x1
```

If you look at the stack trace, you can see how application:openURL:options: called __handleOpenURL:, which called moveItemAtPath:toPath:error:. Notice that we have now this information without having the source code for the target app. The first thing that we had to do was clear: hook application:openURL:options:. Regarding the rest, we had to think a little bit and come up with methods that we could start tracing and are related to the file manager, for example, all methods containing the strings "copy" , "move" , "remove" , etc. until we have found that the one being called was moveItemAtPath:toPath:error:.

A final thing worth noticing here is that this way of handling incoming files is the same for custom URL schemes. Please refer to the "*Custom URL Schemes*" section for more information.

Reference

- owasp-mastg Testing for Sensitive Functionality Exposure Through IPC (MSTG-PLATFORM-4) UIActivity Sharing Dynamic Analysis

6.4.4 App Extensions

6.4.4.1 Overview

What are app extensions

Together with iOS 8, Apple introduced App Extensions. According to Apple App Extension Programming Guide, app extensions let apps offer custom functionality and content to users while they' re interacting with other apps or the system. In order to do this, they implement specific, well scoped tasks like, for example, define what happens after the user clicks on the "Share" button and selects some app or action, provide the content for a Today widget or enable a custom keyboard.

Depending on the task, the app extension will have a particular type (and only one), the so-called extension points. Some notable ones are:

- Custom Keyboard: replaces the iOS system keyboard with a custom keyboard for use in all apps.
- Share: post to a sharing website or share content with others.
- Today: also called widgets, they offer content or perform quick tasks in the Today view of Notification Center.

Reference

- owasp-mastg Testing for Sensitive Functionality Exposure Through IPC (MSTG-PLATFORM-4) App Extensions Overview What are app extensions

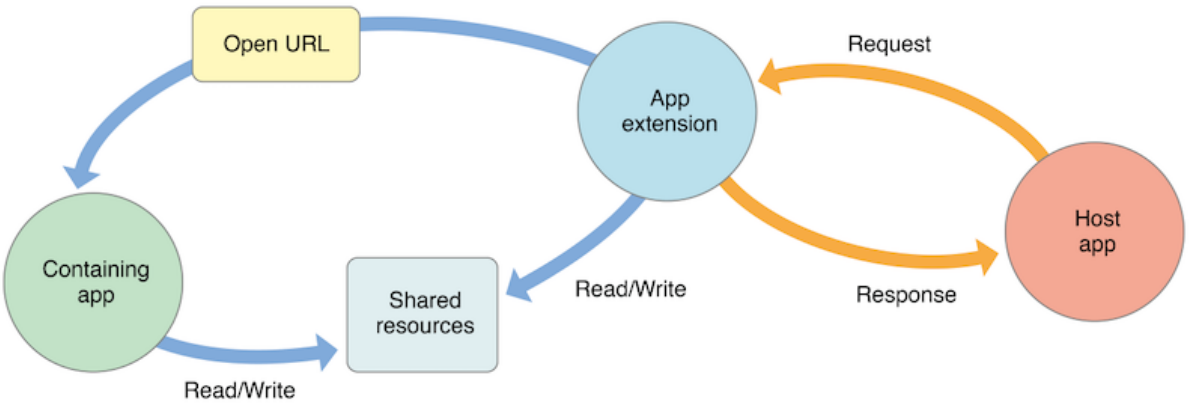
How do app extensions interact with other apps

There are three important elements here:

- App extension: is the one bundled inside a containing app. Host apps interact with it.
- Host app: is the (third-party) app that triggers the app extension of another app.
- Containing app: is the app that contains the app extension bundled into it.

For example, the user selects text in the host app, clicks on the "Share" button and selects one "app" or action from the list. This triggers the app extension of the containing app. The app extension displays its view within the context of the host app and uses the items provided by the host app, the selected text in this case, to perform a specific task

(post it on a social network, for example). See this picture from the Apple App Extension Programming Guide which pretty good summarizes this:



Reference

- owasp-mastg Testing for Sensitive Functionality Exposure Through IPC (MSTG-PLATFORM-4) App Extensions Overview How do app extensions interact with other apps

Security Considerations

From the security point of view it is important to note that:

- An app extension does never communicate directly with its containing app (typically, it isn't even running while the contained app extension is running).
- An app extension and the host app communicate via inter-process communication.
- An app extension's containing app and the host app don't communicate at all.
- A Today widget (and no other app extension type) can ask the system to open its containing app by calling the openURL:completionHandler: method of the NSExtensionContext class.
- Any app extension and its containing app can access shared data in a privately defined shared container.

In addition:

- App extensions cannot access some APIs, for example, HealthKit.
- They cannot receive data using AirDrop but do can send data.
- No long-running background tasks are allowed but uploads or downloads can be initiated.
- App extensions cannot access the camera or microphone on an iOS device (except for iMessage app extensions).

Reference

- owasp-mastg Testing for Sensitive Functionality Exposure Through IPC (MSTG-PLATFORM-4) App Extensions Overview Security Considerations

6.4.4.2 Static Analysis

The static analysis will take care of:

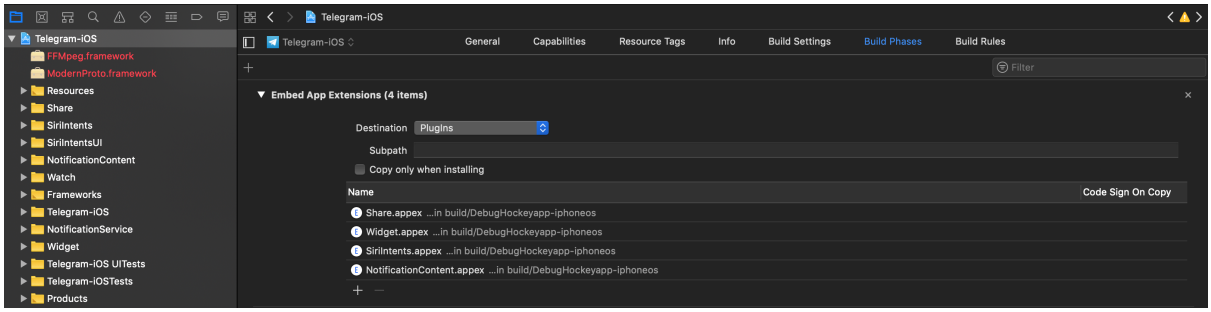
- Verifying if the app contains app extensions
- Determining the supported data types
- Checking data sharing with the containing app
- Verifying if the app restricts the use of app extensions

Reference

- owasp-mastg Testing for Sensitive Functionality Exposure Through IPC (MSTG-PLATFORM-4) App extensions Static Analysis

Verifying if the App Contains App Extensions

If you have the original source code you can search for all occurrences of NSExtensionPointIdentifier with Xcode (cmd+shift+f) or take a look into "Build Phases / Embed App extensions" :



There you can find the names of all embedded app extensions followed by .appex, now you can navigate to the individual app extensions in the project.

If not having the original source code:

Grep for NSExtensionPointIdentifier among all files inside the app bundle (IPA or installed app):

```
$ grep -nr NSExtensionPointIdentifier Payload/Telegram\ X.app/
Binary file Payload/Telegram X.app//PlugIns/SiriIntents.appex/Info.plist matches
Binary file Payload/Telegram X.app//PlugIns/Share.appex/Info.plist matches
Binary file Payload/Telegram X.app//PlugIns/NotificationContent.appex/Info.plist␣
↪matches
Binary file Payload/Telegram X.app//PlugIns/Widget.appex/Info.plist matches
Binary file Payload/Telegram X.app//Watch/Watch.app/PlugIns/Watch Extension.appex/
↪Info.plist matches
```

You can also access per SSH, find the app bundle and list all inside PlugIns (they are placed there by default) or do it with objection:

```
ph.telegra.Telegraph on (iPhone: 11.1.2) [usb] # cd PlugIns
    /var/containers/Bundle/Application/15E6A58F-1CA7-44A4-A9E0-6CA85B65FA35/
    Telegram X.app/PlugIns

ph.telegra.Telegraph on (iPhone: 11.1.2) [usb] # ls
NSFileType      Perms  NSFileProtection     Read    Write     Name
------------  -------  ------------------  ------  -------  ----------------------
↪---
Directory         493  None                True    False     NotificationContent.
↪appex
Directory         493  None                True    False     Widget.appex
Directory         493  None                True    False     Share.appex
Directory         493  None                True    False     SiriIntents.appex
```

We can see now the same four app extensions that we saw in Xcode before.

Reference

- owasp-mastg Testing for Sensitive Functionality Exposure Through IPC (MSTG-PLATFORM-4) App extensions Static Analysis Verifying if the App Contains App Extensions

Determining the Supported Data Types

This is important for data being shared with host apps (e.g. via Share or Action Extensions). When the user selects some data type in a host app and it matches the data types define here, the host app will offer the extension. It is worth noticing the difference between this and data sharing via UIActivity where we had to define the document types, also using UTIs. An app does not need to have an extension for that. It is possible to share data using only UIActivity.

Inspect the app extension's Info.plist file and search for NSExtensionActivationRule. That key specifies the data being supported as well as e.g. maximum of items supported. For example:

```
<key>NSExtensionAttributes</key>
    <dict>
        <key>NSExtensionActivationRule</key>
        <dict>
            <key>NSExtensionActivationSupportsImageWithMaxCount</key>
            <integer>10</integer>
            <key>NSExtensionActivationSupportsMovieWithMaxCount</key>
            <integer>1</integer>
            <key>NSExtensionActivationSupportsWebURLWithMaxCount</key>
            <integer>1</integer>
        </dict>
    </dict>
```

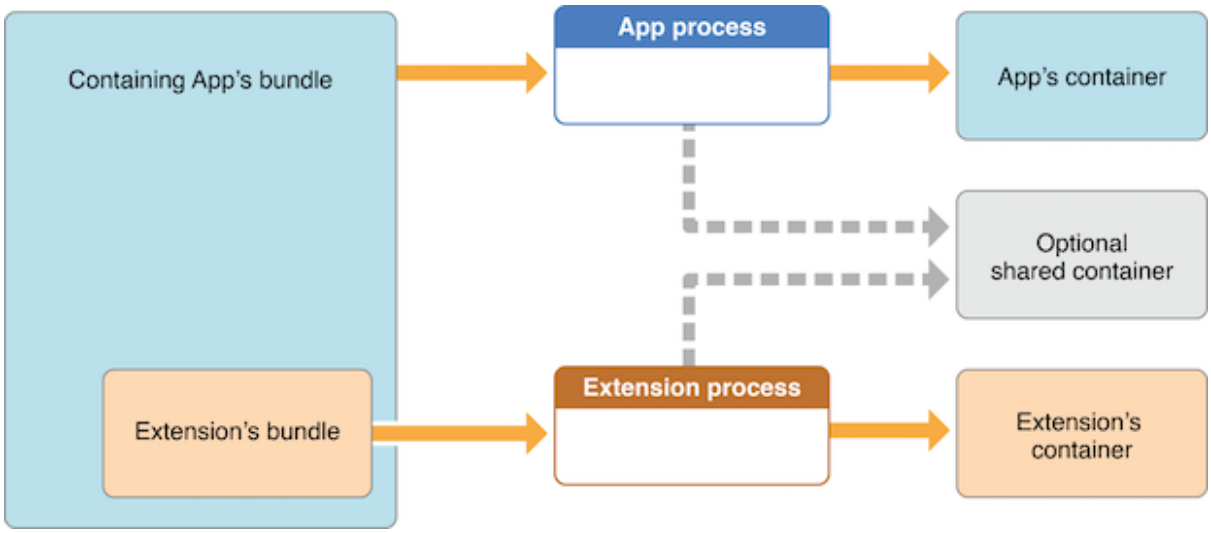
Only the data types present here and not having 0 as MaxCount will be supported. However, more complex filtering is possible by using a so-called predicate string that will evaluate the UTIs given. Please refer to the Apple App Extension Programming Guide for more detailed information about this.

Reference

- owasp-mastg Testing for Sensitive Functionality Exposure Through IPC (MSTG-PLATFORM-4) App extensions Static Analysis Determining the Supported Data Types

Checking Data Sharing with the Containing App

Remember that app extensions and their containing apps do not have direct access to each other's containers. However, data sharing can be enabled. This is done via "App Groups" and the NSUserDefaults API. See this figure from Apple App Extension Programming Guide:



As also mentioned in the guide, the app must set up a shared container if the app extension uses the NSURLSession

class to perform a background upload or download, so that both the extension and its containing app can access the transferred data.

Reference

- owasp-mastg Testing for Sensitive Functionality Exposure Through IPC (MSTG-PLATFORM-4) App extensions Static Analysis Checking Data Sharing with the Containing App

6.4.4.3 Verifying if the App Restricts the Use of App Extensions

It is possible to reject a specific type of app extension by using the following method:

- application:shouldAllowExtensionPointIdentifier:

However, it is currently only possible for "custom keyboard" app extensions (and should be verified when testing apps handling sensitive data via the keyboard like e.g. banking apps).

Reference

- owasp-mastg Testing for Sensitive Functionality Exposure Through IPC (MSTG-PLATFORM-4) App extensions Static Analysis Verifying if the App Restricts the Use of App Extensions

6.4.4.4 Dynamic Analysis

For the dynamic analysis we can do the following to gain knowledge without having the source code:

- Inspecting the items being shared
- Identifying the app extensions involved

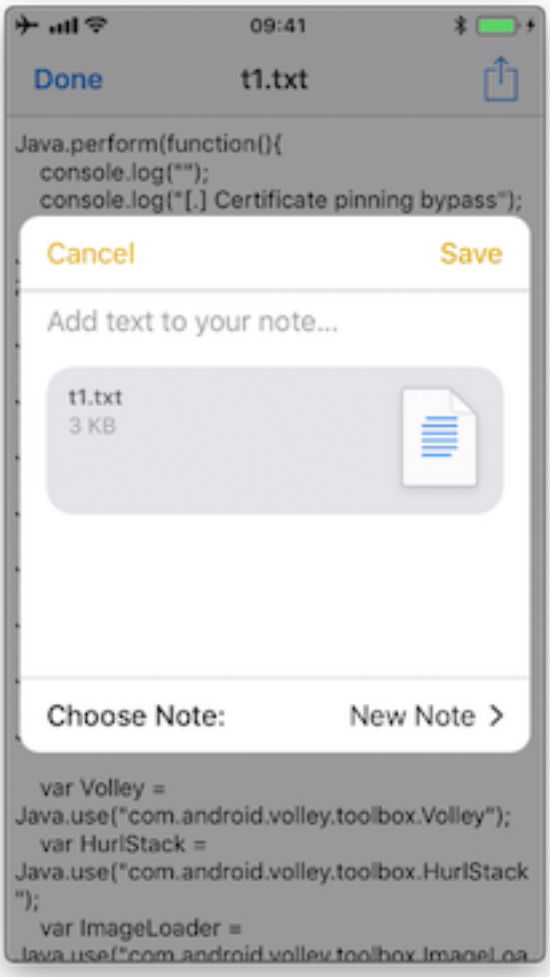
Reference

- owasp-mastg Testing for Sensitive Functionality Exposure Through IPC (MSTG-PLATFORM-4) App extensions Dynamic Analysis

Inspecting the Items Being Shared

For this we should hook NSExtensionContext - inputItems in the data originating app.

Following the previous example of Telegram we will now use the "Share" button on a text file (that was received from a chat) to create a note in the Notes app with it:



If we run a trace, we' d see the following output:

```
(0x1c06bb420) NSExtensionContext - inputItems
0x18284355c Foundation!-[NSExtension _itemProviderForPayload:extensionContext:]
0x1828447a4 Foundation!-[NSExtension _
↪loadItemForPayload:contextIdentifier:completionHandler:]
0x182973224 Foundation!__NSXPCCONNECTION_IS_CALLING_OUT_TO_EXPORTED_OBJECT_S3__
0x182971968 Foundation!-[NSXPCConnection _decodeAndInvokeMessageWithEvent:flags:]
0x182748830 Foundation!message_handler
0x181ac27d0 libxpc.dylib!_xpc_connection_call_event_handler
0x181ac0168 libxpc.dylib!_xpc_connection_mach_event
...
RET: (
"<NSExtensionItem: 0x1c420a540> - userInfo:
{
    NSExtensionItemAttachmentsKey =     (
    "<NSItemProvider: 0x1c46b30e0> {types = (\n \"public.plain-text\",\n \"public.
↪file-url\"\n)}"
    );
}"
)
```

Here we can observe that:

- This occurred under-the-hood via XPC, concretely it is implemented via a NSXPCConnection that uses the libxpc.dylib Framework.

- The UTIs included in the NSItemProvider are public.plain-text and public.file-url, the latter being included in NSExtensionActivationRule from the Info.plist of the "Share Extension" of Telegram.

Reference

- owasp-mastg Testing for Sensitive Functionality Exposure Through IPC (MSTG-PLATFORM-4) App extensions Dynamic Analysis Inspecting the Items Being Shared

Identifying the App Extensions Involved

You can also find out which app extension is taking care of your the requests and responses by hooking NSExtension - _plugIn:

We run the same example again:

```
(0x1c0370200) NSExtension - _plugIn
RET: <PKPlugin: 0x1163637f0 ph.telegra.Telegraph.Share(5.3) 5B6DE177-F09B-47DA-
↪90CD-34D73121C785
1(2) /private/var/containers/Bundle/Application/15E6A58F-1CA7-44A4-A9E0-
↪6CA85B65FA35
/Telegram X.app/PlugIns/Share.appex>

(0x1c0372300)  -[NSExtension _plugIn]
RET: <PKPlugin: 0x10bff7910 com.apple.mobilenotes.SharingExtension(1.5) 73E4F137-
↪5184-4459-A70A-83
F90A1414DC 1(2) /private/var/containers/Bundle/Application/5E267B56-F104-41D0-835B-
↪F1DAB9AE076D
/MobileNotes.app/PlugIns/com.apple.mobilenotes.SharingExtension.appex>
```

As you can see there are two app extensions involved:

- Share.appex is sending the text file (public.plain-text and public.file-url).
- com.apple.mobilenotes.SharingExtension.appex which is receiving and will process the text file.

If you want to learn more about what's happening under-the-hood in terms of XPC, we recommend to take a look at the internal calls from "libxpc.dylib". For example you can use frida-trace and then dig deeper into the methods that you find more interesting by extending the automatically generated stubs.

Reference

- owasp-mastg Testing for Sensitive Functionality Exposure Through IPC (MSTG-PLATFORM-4) App extensions Dynamic Analysis Identifying the App Extensions Involved

6.4.5 UIPasteboard

6.4.5.1 Overview

When typing data into input fields, the clipboard can be used to copy in data. The clipboard is accessible system-wide and is therefore shared by apps. This sharing can be misused by malicious apps to get sensitive data that has been stored in the clipboard.

When using an app you should be aware that other apps might be reading the clipboard continuously, as the Facebook app did. Before iOS 9, a malicious app might monitor the pasteboard in the background while periodically retrieving [UIPasteboard generalPasteboard].string. As of iOS 9, pasteboard content is accessible to apps in the foreground only, which reduces the attack surface of password sniffing from the clipboard dramatically. Still, copy-pasting passwords is a security risk you should be aware of, but also cannot be solved by an app.

- Preventing pasting into input fields of an app, does not prevent that a user will copy sensitive information anyway. Since the information has already been copied before the user notices that it's not possible to paste it in, a malicious app has already sniffed the clipboard.
- If pasting is disabled on password fields users might even choose weaker passwords that they can remember and they cannot use password managers anymore, which would contradict the original intention of making the app more secure.

The UIPasteboard enables sharing data within an app, and from an app to other apps. There are two kinds of pasteboards:

- **systemwide general pasteboard**: for sharing data with any app. Persistent by default across device restarts and app uninstalls (since iOS 10).
- **custom / named pasteboards**: for sharing data with another app (having the same team ID as the app to share from) or with the app itself (they are only available in the process that creates them). Non-persistent by default (since iOS 10), that is, they exist only until the owning (creating) app quits.

Some security considerations:

- Users cannot grant or deny permission for apps to read the pasteboard.
- Since iOS 9, apps cannot access the pasteboard while in background, this mitigates background pasteboard monitoring. However, if the malicious app is brought to foreground again and the data remains in the pasteboard, it will be able to retrieve it programmatically without the knowledge nor the consent of the user.
- Apple warns about persistent named pasteboards and discourages their use. Instead, shared containers should be used.
- Starting in iOS 10 there is a new Handoff feature called Universal Clipboard that is enabled by default. It allows the general pasteboard contents to automatically transfer between devices. This feature can be disabled if the developer chooses to do so and it is also possible to set an expiration time and date for copied data.

Reference

- owasp-mastg Testing for Sensitive Functionality Exposure Through IPC (MSTG-PLATFORM-4) UIPasteboard

Rulebook

- *Be aware that other apps may be continuously reading your clipboard (Required)*
- *Notes on pasteboard security (Required)*

6.4.5.2 Static Analysis

The **systemwide general pasteboard** can be obtained by using generalPasteboard, search the source code or the compiled binary for this method. Using the systemwide general pasteboard should be avoided when dealing with sensitive data.

Custom pasteboards can be created with pasteboardWithName:create: or pasteboardWithUniqueName. Verify if custom pasteboards are set to be persistent as this is deprecated since iOS 10. A shared container should be used instead.

In addition, the following can be inspected:

- Check if pasteboards are being removed with removePasteboardWithName:, which invalidates an app pasteboard, freeing up all resources used by it (no effect for the general pasteboard).
- Check if there are excluded pasteboards, there should be a call to setItems:options: with the UIPasteboardOptionLocalOnly option.
- Check if there are expiring pasteboards, there should be a call to setItems:options: with the UIPasteboardOptionExpirationDate option.
- Check if the app swipes the pasteboard items when going to background or when terminating. This is done by some password manager apps trying to restrict sensitive data exposure.

Reference

- owasp-mastg Testing for Sensitive Functionality Exposure Through IPC (MSTG-PLATFORM-4) UIPasteboard Static Analysis

Rulebook

- *Notes on pasteboard security (Required)*

6.4.5.3 Dynamic Analysis

Detect Pasteboard Usage

Hook or trace the following:

- generalPasteboard for the system-wide general pasteboard.
- pasteboardWithName:create: and pasteboardWithUniqueName for custom pasteboards.

Detect Persistent Pasteboard Usage

Hook or trace the deprecated setPersistent: method and verify if it's being called.

Monitoring and Inspecting Pasteboard Items

When monitoring the pasteboards, there is several details that may be dynamically retrieved:

- Obtain pasteboard name by hooking pasteboardWithName:create: and inspecting its input parameters or pasteboardWithUniqueName and inspecting its return value.
- Get the first available pasteboard item: e.g. for strings use string method. Or use any of the other methods for the standard data types.
- Get the number of items with numberOfItems.
- Check for existence of standard data types with the convenience methods, e.g. hasImages, hasStrings, hasURLs (starting in iOS 10).
- Check for other data types (typically UTIs) with containsPasteboardTypes: inItemSet:. You may inspect for more concrete data types like, for example an picture as public.png and public.tiff (UTIs) or for custom data such as com.mycompany.myapp.mytype. Remember that, in this case, only those apps that declare knowledge of the type are able to understand the data written to the pasteboard. This is the same as we have seen in the "*UIActivity Sharing*" section. Retrieve them using itemSetWithPasteboardTypes: and setting the corresponding UTIs.
- Check for excluded or expiring items by hooking setItems:options: and inspecting its options for UIPasteboardOptionLocalOnly or UIPasteboardOptionExpirationDate.

If only looking for strings you may want to use objection's command ios pasteboard monitor:

Hooks into the iOS UIPasteboard class and polls the generalPasteboard every 5 seconds for data. If new data is found, different from the previous poll, that data will be dumped to screen.

You may also build your own pasteboard monitor that monitors specific information as seen above.

For example, this script (inspired from the script behind objection's pasteboard monitor) reads the pasteboard items every 5 seconds, if there's something new it will print it:

```
const UIPasteboard = ObjC.classes.UIPasteboard;
    const Pasteboard = UIPasteboard.generalPasteboard();
    var items = "";
    var count = Pasteboard.changeCount().toString();

setInterval(function () {
      const currentCount = Pasteboard.changeCount().toString();
      const currentItems = Pasteboard.items().toString();

      if (currentCount === count) { return; }

      items = currentItems;
      count = currentCount;

      console.log('[* Pasteboard changed] count: ' + count +
      ' hasStrings: ' + Pasteboard.hasStrings().toString() +
      ' hasURLs: ' + Pasteboard.hasURLs().toString() +
      ' hasImages: ' + Pasteboard.hasImages().toString());
      console.log(items);
```

(continues on next page)

(continued from previous page)

```
    }, 1000 * 5);
```

In the output we can see the following:

```
[* Pasteboard changed] count: 64 hasStrings: true hasURLs: false hasImages: false
(
    {
        "public.utf8-plain-text" = hola;
    }
)
[* Pasteboard changed] count: 65 hasStrings: true hasURLs: true hasImages: false
(
    {
        "public.url" = "https://codeshare.frida.re/";
        "public.utf8-plain-text" = "https://codeshare.frida.re/";
    }
)
[* Pasteboard changed] count: 66 hasStrings: false hasURLs: false hasImages: true
(
    {
        "com.apple.uikit.image" = "<UIImage: 0x1c42b23c0> size {571, 264}␣
↪orientation 0 scale 1.000000";
        "public.jpeg" = "<UIImage: 0x1c44a1260> size {571, 264} orientation 0␣
↪scale 1.000000";
        "public.png" = "<UIImage: 0x1c04aaaa0> size {571, 264} orientation 0 scale␣
↪1.000000";
    }
)
```

You see that first a text was copied including the string “hola” , after that a URL was copied and finally a picture was copied. Some of them are available via different UTIs. Other apps will consider these UTIs to allow pasting of this data or not.

Reference

- owasp-mastg Testing for Sensitive Functionality Exposure Through IPC (MSTG-PLATFORM-4) UIPasteboard Dynamic Analysis

6.4.6 Rulebook

1. *How to validate data received as a result of Universal Link (Required)*
2. *Use secure transfer protocols to protect user privacy and security (Required)*
3. *If the URL contains parameters, do not trust the URL without careful sanitization and validation (Recommended)*
4. *Make sure you are not exposing sensitive information when calling other apps via universal links (Required)*
5. *Explicit exclusion of activity types targeted by UIActivity Sharing (Required)*
6. *Always confirm when receiving files from other apps (Required)*
7. *Universal Links and Handoff (Required)*
8. *Be aware that other apps may be continuously reading your clipboard (Required)*
9. *Notes on pasteboard security (Required)*

6.4.6.1 How to validate data received as a result of Universal Link (Required)

Universal links offer a potential attack vector into your app, so make sure to validate all URL parameters and discard any malformed URLs.

when iOS opens an app as the result of a universal link, the app receives an NSUserActivity object with an activityType value of NSUserActivityTypeBrowsingWeb. The activity object's webpageURL property contains the HTTP or HTTPS URL that the user accesses. The following example in Swift verifies exactly this before opening the URL:

```
func application(_ application: UIApplication, continue userActivity:␣
↪NSUserActivity,
                restorationHandler: @escaping ([UIUserActivityRestoring]?) ->␣
↪Void) -> Bool {
    // ...
    if userActivity.activityType == NSUserActivityTypeBrowsingWeb, let url =␣
↪userActivity.webpageURL {
        application.open(url, options: [:], completionHandler: nil)
    }

    return true
}
```

If this is violated, the following may occur.

- Vulnerable to URL scheme hijacking attacks.
- Unsafe associations between websites and apps.

6.4.6.2 Use secure transfer protocols to protect user privacy and security (Required)

In HTTP communication, the website does not use an SSL server certificate, so there is a risk that the content of the communication may be illegally acquired or altered. For example, if you enter personal information such as your name, address, or credit card number on an online site, there is a risk that your information will be leaked to a malicious third party. So don't use HTTP.

On the other hand, in HTTPS communication, SSL is used for encryption to enhance the security of the website.

See the rulebook sample code below for ATS enablement.

Rulebook

- *Use App Transport Security (ATS) (Required)*

If this is violated, the following may occur.

- A man-in-the-middle attack may eavesdrop on or tamper with communication contents.

6.4.6.3 If the URL contains parameters, do not trust the URL without careful sanitization and validation (Recommended)

If the URL contains parameters, it is recommended not to trust the URL without careful sanitization and validation as the parameters may have been tampered with even if the URL is correct. Parameters are variables added to the end of the URL to send information to the server.

Basic Structure of Parameters

The key and value are given with "?" after the URL. If there are multiple keys, combine them with "&".

Example) https//example.com?key=value&key=value&key=value

Therefore, it is necessary to perform the following verification to determine whether the URL is safe.

- Is there a key other than the expected parameter?
- Are the values in the parameters as expected (within upper and lower limits of numbers, allowed characters, etc.)?

  - Contains one value per parameter
  - Only usable characters (a-z A-Z 0-9 - _ . ! ‘ ( ) *) must be used

If the above validation is violated, discard even if the URL is as expected.

```
func application(_ application: UIApplication,
                 continue userActivity: NSUserActivity,
                 restorationHandler: @escaping ([Any]?) -> Void) -> Bool {
    guard userActivity.activityType == NSUserActivityTypeBrowsingWeb,
        let incomingURL = userActivity.webpageURL,
        let components = NSURLComponents(url: incomingURL,␣
↪resolvingAgainstBaseURL: true),
        let path = components.path,
        let params = components.queryItems else {
        return false
    }

    if let albumName = params.first(where: { $0.name == "albumname" })?.value,
        let photoIndex = params.first(where: { $0.name == "index" })?.value {
        // Interact with album name and photo index

        return true

    } else {
        // Handle when album and/or album name or photo index missing

        return false
    }
}
```

If this is not noted, the following may occur.

- The intended web page is not displayed. Furthermore, there is a possibility that an attacker may impersonate and gain unauthorized access unintentionally.

6.4.6.4 Make sure you are not exposing sensitive information when calling other apps via universal links (Required)

Your app may call other apps via Universal Links simply to cause some action or transfer information, but you should make sure that you are not exposing sensitive information. .

Given the original source code, you can search for the openURL:options: completionHandler: method to see what data is processed.

Note that the openURL:options:completionHandler: method is used not only for opening universal links, but also for calling custom URL schemes.

Telegram app example:

```
}, openUniversalUrl: { url, completion in
    if #available(iOS 10.0, *) {
        var parsedUrl = URL(string: url)
        if let parsed = parsedUrl {
            if parsed.scheme == nil || parsed.scheme!.isEmpty {
                parsedUrl = URL(string: "https://\(url)")
            }
        }

        if let parsedUrl = parsedUrl {
            return UIApplication.shared.open(parsedUrl,
                                             options: [UIApplicationOpenURLOptionUniversalLinksOnly:␣
↪true as NSNumber],
```

(continues on next page)

(continued from previous page)

```
                        completionHandler: { value in completion.completion(value)}
            )
```

Note how the app adapts the scheme to "https" before opening it and how it uses the option UIApplicationOpenURLOptionUniversalLinksOnly: true that opens the URL only if the URL is a valid universal link and there is an installed app capable of opening that URL.

If this is violated, the following may occur.

- Failure to do so may result in the disclosure of confidential information.

6.4.6.5 Explicit exclusion of activity types targeted by UIActivity Sharing (Required)

Data can be shared via UIActivity, but it is possible to explicitly exclude target activity types. By specifying this, it is possible to prevent unnecessary data sharing to other applications.

However, since the number of activity types that can be excluded may increase with OS version upgrades, it is recommended to reconfirm the items to be excluded in the new OS.

Code to exclude:

```
import UIKit
import Accounts

class ShareViewController: UIViewController {

    @IBAction func share(sender: AnyObject) {
        // Items to share
        let shareText = "Hello world"
        let shareWebsite = NSURL(string: "https://www.apple.com/jp/watch/")!

        let activityItems = [shareText, shareWebsite] as [Any]

        // Initialization process
        let activityVC = UIActivityViewController(activityItems: activityItems,␣
↪applicationActivities: nil)

        // Activity type exclusions
        let excludedActivityTypes = [
            UIActivity.ActivityType.postToFacebook,
            UIActivity.ActivityType.postToTwitter,
            UIActivity.ActivityType.saveToCameraRoll,
            UIActivity.ActivityType.print
        ]

        activityVC.excludedActivityTypes = excludedActivityTypes

        // Show UIActivityViewController
        self.present(activityVC, animated: true, completion: nil)
    }
    override func viewDidLoad() {
        super.viewDidLoad()
    }

}
```

Activity types that can be specified in the list

- UIActivityTypeAddToReadingList: Add URL to Safari Reading List
- UIActivityTypeAirDrop: Use content with AirDrop
- UIActivityTypeAssignToContact: Assign image to contact

- UIActivityTypeCollaborationCopyLink: * No description in Developer
- UIActivityTypeCollaborationInviteWithLink: * No description in Developer
- UIActivityTypeCopyToPasteboard: Post content to the pasteboard
- UIActivityTypeMail: Post content to new email message
- UIActivityTypeMarkupAsPDF: Mark up your content as a PDF file
- UIActivityTypeMessage: Post content to messaging apps
- UIActivityTypeOpenInIBooks: Open content in iBooks
- UIActivityTypePostToFacebook: Post the provided content to the user' s wall on Facebook
- UIActivityTypePostToFlickr: Post the provided image to the user' s girlfriend' s Flickr account
- UIActivityTypePostToTencentWeibo: Post the provided content to the user' s girlfriend' s Tencent Weibo feed
- UIActivityTypePostToTwitter: Post the provided content to the user' s girlfriend' s Twitter feed
- UIActivityTypePostToVimeo: Posting the provided video to the user' s girlfriend' s Vimeo account
- UIActivityTypePostToWeibo: Post the provided content to the user' s girlfriend' s Weibo feed
- UIActivityTypePrint: Output the provided content
- UIActivityTypeSaveToCameraRoll: Assign an image or video to a user' s camera roll
- UIActivityTypeSharePlay: Making Contributed Content Available Through SharePlay

If this is violated, the following may occur.

- There is a possibility of sending confidential information etc. to an unexpected sharing application side.

6.4.6.6 Always confirm when receiving files from other apps (Required)

Confirm on the source that the file name of the received file and the character string data in the contents are not directly used as DB queries.

- if the app declares custom document types by looking into Exported/Imported UTIs ("Info" tab of the Xcode project). The list of all system declared UTIs (Uniform Type Identifiers) can be found in the archived Apple Developer Documentation.
- if the app specifies any document types that it can open by looking into Document Types ("Info" tab of the Xcode project). If present, they consist of name and one or more UTIs that represent the data type (e.g. "public.png" for PNG files). iOS uses this to determine if the app is eligible to open a given document (specifying Exported/Imported UTIs is not enough).
- if the app properly verifies the received data by looking into the implementation of application:openURL:options: in the app delegate.

Below is a sample code that determines the BundleId of the application that sent the URL request in sourceApplication.

```
import UIKit

@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate
{
    // omit

    func application(_ app: UIApplication, open url: URL, options: [UIApplication.
↪OpenURLOptionsKey : Any] = [:]) -> Bool {
        guard let sourceApplication = options[.sourceApplication] as? String else {
            return false
        }
```

(continues on next page)

(continued from previous page)

```
        if sourceApplication.hasPrefix("--BundleId of other apps--") {
            // Receive query parameters and perform various processing
            // ...
        }

        return true
    }
}
```

If not having the source code you can still take a look into the Info.plist file and search for:

- UTExportedTypeDeclarations/UTImportedTypeDeclarations if the app declares exported/imported custom document types.
- CFBundleDocumentTypes to see if the app specifies any document types that it can open.

A very complete explanation about the use of these keys can be found on Stackoverflow.

Let's see a real-world example. We will take a File Manager app and take a look at these keys. We used objection here to read the Info.plist file.

```
objection --gadget SomeFileManager run ios plist cat Info.plist
```

Note that this is the same as if we would retrieve the IPA from the phone or accessed via e.g. SSH and navigated to the corresponding folder in the IPA / app sandbox. However, with objection we are just one command away from our goal and this can be still considered static analysis.

The first thing we noticed is that app does not declare any imported custom document types but we could find a couple of exported ones:

```
UTExportedTypeDeclarations =     (
    {
UTTypeConformsTo =             (
    "public.data"
);
UTTypeDescription = "SomeFileManager Files";
UTTypeIdentifier = "com.some.filemanager.custom";
UTTypeTagSpecification =             {
    "public.filename-extension" =                 (
        ipa,
        deb,
        zip,
        rar,
        tar,
        gz,
        ...
        key,
        pem,
        p12,
        cer
    );
};
}
);
```

The app also declares the document types it opens as we can find the key CFBundleDocumentTypes:

```
CFBundleDocumentTypes =     (
        {
        ...
        CFBundleTypeName = "SomeFileManager Files";
```

(continues on next page)

(continued from previous page)

```
        LSItemContentTypes =              (
            "public.content",
            "public.data",
            "public.archive",
            "public.item",
            "public.database",
            "public.calendar-event",
            ...
        );
    }
);
```

We can see that this File Manager will try to open anything that conforms to any of the UTIs listed in LSItemContentTypes and it' s ready to open files with the extensions listed in UTTypeTagSpecification/" public.filename-extension" . Please take a note of this because it will be useful if you want to search for vulnerabilities when dealing with the different types of files when performing dynamic analysis.

If this is violated, the following may occur.

- If you use the string as it is, SQL injection may occur.
- If the character string is used in WebView as it is, there is a possibility that an unauthorized authentication input site etc. will be displayed.

6.4.6.7 Universal Links and Handoff (Required)

Implement application:continueUserActivity:restorationHandler: method of AppDelegate when the application is launched with universal link. Note that this method is called not only by Universal Links, but also by SearchAPI or Handoff, so it is necessary to check the activityType.

The sample code below is the universal link check process.

```
import UIKit

extension AppDelegate {
  func application(application: UIApplication, continueUserActivity userActivity:␣
↪NSUserActivity, restorationHandler: ([AnyObject]?) -> Void) -> Bool {
      if userActivity.activityType == NSUserActivityTypeBrowsingWeb {
          // Check activity type
          // Universal Links
          if UniversalLinkHandler.handleUniversalLink(userActivity.webpageURL) ==␣
↪false {
              UIApplication.sharedApplication().openURL(userActivity.webpageURL!)
                print(userActivity.activityType)
          }
          else {
              print(userActivity.activityType)
          }
      }
      return true
    }
}
```

The sample code below is Handoff check process.

```
import UIKit

extension AppDelegate {
  func handoffApplication(application: UIApplication, continueUserActivity␣
↪userActivity: NSUserActivity, restorationHandler: ([AnyObject]?) -> Void) ->␣
↪Bool {
```

(continues on next page)

(continued from previous page)

```
        if userActivity.activityType == NSUserActivityTypeBrowsingWeb {
            // Check activity type
            print(userActivity.activityType)
            return true
        } else if userActivity.activityType == NSUserActivity.myHandoffActivityType {
            // Restore state for userActivity and userInfo

            return true
        }

        return false
    }



    override func updateUserActivityState(_ activity: NSUserActivity) {
        if activity.activityType == NSUserActivity.myHandoffActivityType {
            let updateDict:  [AnyHashable : Any] = [
                "shape-type" : "com.example.myapp.create-shape",
                "activity-version" : 1
            ]
            activity.addUserInfoEntries(from: updateDict)
        }
    }

}


extension NSUserActivity {

    public static let myHandoffActivityType = "com.myapp.name.my-activity-type"

    public static var myActivity: NSUserActivity {
        let activity = NSUserActivity(activityType: myHandoffActivityType)
        activity.isEligibleForHandoff = true
        activity.requiredUserInfoKeys = ["shape-type"]
        activity.title = NSLocalizedString("Creating shape", comment: "Creating␣
↪shape activity")

        return activity
    }
}
```

If this is violated, the following may occur.

- Vulnerable to URL scheme hijacking attacks.
- Unsafe associations between websites and apps.

6.4.6.8 Be aware that other apps may be continuously reading your clipboard (Required)

UIPasteboard allows you to copy/paste to clipboard. It is possible to get the character string copied in the application in another application. It is important not to copy sensitive information such as passwords.

How to copy with UIPasteboard:

```
class UIPasteboardSample {

    func uiPasteboardSample() {
        //  System-wide general pasteboard
        UIPasteboard.general.string = "copy"
```

(continues on next page)

(continued from previous page)

```
        // Get the string in UIPasteboard.general.string (shared by other apps)
        let parst = UIPasteboard.general.string

        // custom / named pasteboard
        guard let customPasteboard = UIPasteboard(name: UIPasteboard.
↪Name(rawValue: "myData"), create: false) else {
            return
        }
        customPasteboard.string = "aaaa"
    }
}
```

If this is violated, the following may occur.

- A malicious app can exploit shared state to obtain sensitive data stored on the clipboard.

6.4.6.9 Notes on pasteboard security (Required)

Apple warns about persistent named pasteboards and discourages their use.

When you get text copied from another app from iOS14 from the clipboard, an alert is displayed at the top of the screen due to security concerns. His UIPasteboard, where the universal clipboard was introduced, adds two of his features to reduce security risks:

- Restrict scope to local only
- set expiration date

This can be used when the terminals are close to each other and each is set as follows.

- Signed in to iCloud with the same Apple ID on each device.
- Bluetooth is turned on on each device.
- Wi-Fi is turned on on each device.
- Handoff is turned on on each device.

Restrict scope to local only An example of restricting scope to local only would be:

```
import UIKit

class Pasteboard: UIViewController {

    override func viewDidLoad() {
        super.viewDidLoad()

        let pasteboard = UIPasteboard.general
        pasteboard.setItems([["key" : "value"]], options: [UIPasteboard.OptionsKey.
↪localOnly : true])
    }
}
```

Set an expiration date In the example below, after 24 hours have passed since copying, the copied data is automatically deleted.

```
import UIKit

class Pasteboard: UIViewController {

    override func viewDidLoad() {
        super.viewDidLoad()
```

(continues on next page)

(continued from previous page)

```
        let options: [UIPasteboard.OptionsKey : Any] = [.expirationDate: Date().
↪addingTimeInterval(60 * 60 * 24)]
        UIPasteboard.general.setItems([["key" : "value"]], options: options)
    }
}
```

If this is violated, the following may occur.

- A malicious app can exploit shared state to obtain sensitive data stored on the clipboard.

6.5 MSTG-PLATFORM-5

JavaScript is disabled in WebViews unless explicitly required.

WebViews are in-app browser components for displaying interactive web content. They can be used to embed web content directly into an app's user interface. iOS WebViews support JavaScript execution by default, so script injection and Cross-Site Scripting attacks can affect them.

Reference

- owasp-mastg Testing iOS WebViews (MSTG-PLATFORM-5)

6.5.1 UIWebView

UIWebView is deprecated starting on iOS 12 and should not be used. Make sure that either WKWebView or SFSafariViewController are used to embed web content. In addition to that, JavaScript cannot be disabled for UIWebView which is another reason to refrain from using it.

Reference

- owasp-mastg Testing iOS WebViews (MSTG-PLATFORM-5) Overview UIWebView

Rulebook

- *Do not use UIWebView as it has been deprecated since iOS 12 (Required)*

6.5.2 WKWebView

WKWebView was introduced with iOS 8 and is the appropriate choice for extending app functionality, controlling displayed content (i.e., prevent the user from navigating to arbitrary URLs) and customizing. WKWebView also increases the performance of apps that are using WebViews significantly, through the Nitro JavaScript engine [#thiel2].

WKWebView comes with several security advantages over UIWebView:

- JavaScript is enabled by default but thanks to the javaScriptEnabled property of WKWebView, it can be completely disabled, preventing all script injection flaws.
- The JavaScriptCanOpenWindowsAutomatically can be used to prevent JavaScript from opening new windows, such as pop-ups.
- The hasOnlySecureContent property can be used to verify resources loaded by the WebView are retrieved through encrypted connections.
- WKWebView implements out-of-process rendering, so memory corruption bugs won't affect the main app process.

A JavaScript Bridge can be enabled when using WKWebViews (and UIWebViews). See Section "*Bridge between JavaScript and native in WebView*" below for more information.

Reference

- owasp-mastg Testing iOS WebViews (MSTG-PLATFORM-5) Overview WKWebView

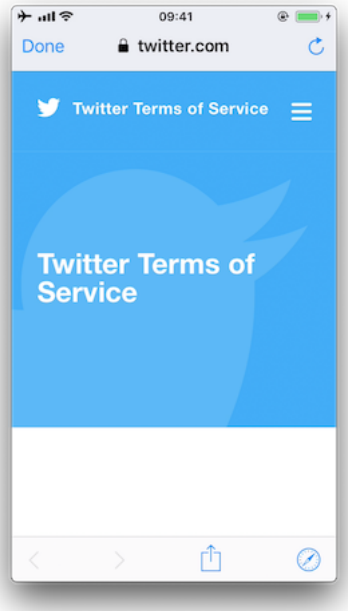
Rulebook

- *How to use WKWebView (Required)*

6.5.3 SFSafariViewController

SFSafariViewController is available starting on iOS 9 and should be used to provide a generalized web viewing experience. These WebViews can be easily spotted as they have a characteristic layout which includes the following elements:

- A read-only address field with a security indicator.
- An Action ("Share") button.
- A Done button, back and forward navigation buttons, and a "Safari" button to open the page directly in Safari.



There are a couple of things to consider:

- JavaScript cannot be disabled in SFSafariViewController and this is one of the reasons why the usage of WKWebView is recommended when the goal is extending the app' s user interface.
- SFSafariViewController also shares cookies and other website data with Safari.
- The user' s activity and interaction with a SFSafariViewController are not visible to the app, which cannot access AutoFill data, browsing history, or website data.
- According to the App Store Review Guidelines, SFSafariViewControllers may not be hidden or obscured by other views or layers.

This should be sufficient for an app analysis and therefore, SFSafariViewControllers are out of scope for the Static and Dynamic Analysis sections.

Reference

- owasp-mastg Testing iOS WebViews (MSTG-PLATFORM-5) Overview SFSafariViewController

Rulebook

- *How to use SFSafariViewController (Required)*

6.5.4 Safari Web Inspector

Enabling Safari web inspection on iOS allows you to inspect the contents of a WebView remotely from a macOS device and it does not require a jailbroken iOS device. Enabling the Safari Web Inspector is especially interesting in applications that expose native APIs using a JavaScript bridge, for example in hybrid applications.

To activate the web inspection you have to follow these steps:

1. On the iOS device open the Settings app: Go to **Safari** -> **Advanced** and toggle on Web Inspector.
2. On the macOS device, open Safari: in the menu bar, go to **Safari** -> **Preferences** -> **Advanced** and enable Show Develop menu in menu bar.
3. Connect your iOS device to the macOS device and unlock it: the iOS device name should appear in the Develop menu.
4. (If not yet trusted) On macOS' s Safari, go to the Develop menu, click on the iOS device name, then on "Use for Development" and enable trust.

To open the web inspector and debug a WebView:

1. In iOS, open the app and navigate to the screen that should contain a WebView.
2. In macOS Safari, go to **Developer** -> **'iOS Device Name'** and you should see the name of the WebView based context. Click on it to open the Web Inspector.

Now you' re able to debug the WebView as you would with a regular web page on your desktop browser.

Reference

- owasp-mastg Testing iOS WebViews (MSTG-PLATFORM-5) Overview Safari Web Inspector

Rulebook

- *How to use WKWebView (Required)*
- *How to use SFSafariViewController (Required)*

6.5.5 Static Analysis

For the static analysis we will focus mostly on the following points having UIWebView and WKWebView under scope.

- Identifying WebView usage
- Testing JavaScript configuration
- Testing for mixed content
- Testing for WebView URI manipulation

Reference

- owasp-mastg Testing iOS WebViews (MSTG-PLATFORM-5) Static Analysis

6.5.5.1 Identifying WebView Usage

Look out for usages of the above mentioned WebView classes by searching in Xcode.

In the compiled binary you can search in its symbols or strings like this:

UIWebView

```
$ rabin2 -zz ./WheresMyBrowser | egrep "UIWebView$"
489 0x0002fee9 0x10002fee9   9  10 (5.__TEXT.__cstring) ascii UIWebView
896 0x0003c813 0x0003c813  24  25 () ascii @_OBJC_CLASS_$_UIWebView
1754 0x00059599 0x00059599  23  24 () ascii _OBJC_CLASS_$_UIWebView
```

WKWebView

```
$ rabin2 -zz ./WheresMyBrowser | egrep "WKWebView$"
490 0x0002fef3 0x10002fef3   9  10 (5.__TEXT.__cstring) ascii WKWebView
625 0x00031670 0x100031670  17  18 (5.__TEXT.__cstring) ascii unwindToWKWebView
904 0x0003c960 0x0003c960  24  25 () ascii @_OBJC_CLASS_$_WKWebView
1757 0x000595e4 0x000595e4  23  24 () ascii _OBJC_CLASS_$_WKWebView
```

Alternatively you can also search for known methods of these WebView classes. For example, search for the method used to initialize a WKWebView (init(frame:configuration:)):

```
$ rabin2 -zzq ./WheresMyBrowser | egrep "WKWebView.*frame"
0x5c3ac 77 76 __T0So9WKWebViewCABSC6CGRectV5frame_
↪So0aB13ConfigurationC13configurationtcfC
0x5d97a 79 78 __T0So9WKWebViewCABSC6CGRectV5frame_
↪So0aB13ConfigurationC13configurationtcfcTO
0x6b5d5 77 76 __T0So9WKWebViewCABSC6CGRectV5frame_
↪So0aB13ConfigurationC13configurationtcfC
0x6c3fa 79 78 __T0So9WKWebViewCABSC6CGRectV5frame_
↪So0aB13ConfigurationC13configurationtcfcTO
```

You can also demangle it:

```
$ xcrun swift-demangle __T0So9WKWebViewCABSC6CGRectV5frame_
↪So0aB13ConfigurationC13configurationtcfcTO

---> @nonobjc __C.WKWebView.init(frame: __C_Synthesized.CGRect,
                                 configuration: __C.WKWebViewConfiguration) -> __C.
↪WKWebView
```

Reference

- owasp-mastg Testing iOS WebViews (MSTG-PLATFORM-5) Static Analysis Identifying WebView Usage

Rulebook

- *How to use WKWebView (Required)*

6.5.5.2 Testing JavaScript Configuration

First of all, remember that JavaScript cannot be disabled for UIWebVIews.

For WKWebViews, as a best practice, JavaScript should be disabled unless it is explicitly required. To verify that JavaScript was properly disabled search the project for usages of WKPreferences and ensure that the javaScriptEnabled property is set to false:

```
let webPreferences = WKPreferences()
webPreferences.javaScriptEnabled = false // Note that javaScriptEnabled is now␣
↪deprecated.
```

If only having the compiled binary you can search for this in it:

```
$ rabin2 -zz ./WheresMyBrowser | grep -i "javascriptenabled" // Note that␣
↪javaScriptEnabled is now deprecated.
391 0x0002f2c7 0x10002f2c7  17  18 (4.__TEXT.__objc_methname) ascii␣
↪javaScriptEnabled
392 0x0002f2d9 0x10002f2d9  21  22 (4.__TEXT.__objc_methname) ascii␣
↪setJavaScriptEnabled:
```

If user scripts were defined, they will continue running as the javaScriptEnabled property won't affect them. See WKUserContentController and WKUserScript for more information on injecting user scripts to WKWebViews.

Reference

- owasp-mastg Testing iOS WebViews (MSTG-PLATFORM-5) Static Analysis Testing JavaScript Configuration

Rulebook

- *How to use WKWebView (Required)*
- *Disable JavaScript unless explicitly required (Recommended)*

6.5.5.3 Testing for Mixed Content

In contrast to UIWebViews, when using WKWebViews it is possible to detect mixed content (HTTP content loaded from a HTTPS page). By using the method hasOnlySecureContent it can be verified whether all resources on the page have been loaded through securely encrypted connections. This example from [#thiel2] (see page 159 and 160) uses this to ensure that only content loaded via HTTPS is shown to the user, otherwise an alert is displayed telling the user that mixed content was detected.

In the compiled binary:

```
$ rabin2 -zz ./WheresMyBrowser | grep -i "hasonlysecurecontent"

# nothing found
```

In this case, the app does not make use of this.

In addition, if you have the original source code or the IPA, you can inspect the embedded HTML files and verify that they do not include mixed content. Search for http:// in the source and inside tag attributes, but remember that this might give false positives as, for example, finding an anchor tag <a> that includes a http:// inside its href attribute does not always present a mixed content issue. Learn more about mixed content in the MDN Web Docs.

Reference

- owasp-mastg Testing iOS WebViews (MSTG-PLATFORM-5) Static Analysis Testing for Mixed Content

Rulebook

- *How to use WKWebView (Required)*

6.5.6 Dynamic Analysis

For the dynamic analysis we will address the same points from the static analysis.

- Enumerating WebView instances
- Checking if JavaScript is enabled
- Verifying that only secure content is allowed

It is possible to identify WebViews and obtain all their properties on runtime by performing dynamic instrumentation. This is very useful when you don' t have the original source code.

For the following examples, we will keep using the "Where' s My Browser?" app and Frida REPL.

Reference

- owasp-mastg Testing iOS WebViews (MSTG-PLATFORM-5) Dynamic Analysis

6.5.6.1 Enumerating WebView Instances

Once you've identified a WebView in the app, you may inspect the heap in order to find instances of one or several of the WebViews that we have seen above.

For example, if you use Frida you can do so by inspecting the heap via "ObjC.choose()"

```
ObjC.choose(ObjC.classes['UIWebView'], {
  onMatch: function (ui) {
    console.log('onMatch: ', ui);
    console.log('URL: ', ui.request().toString());
  },
  onComplete: function () {
    console.log('done for UIWebView!');
  }
});

ObjC.choose(ObjC.classes['WKWebView'], {
  onMatch: function (wk) {
    console.log('onMatch: ', wk);
    console.log('URL: ', wk.URL().toString());
  },
  onComplete: function () {
    console.log('done for WKWebView!');
  }
});

ObjC.choose(ObjC.classes['SFSafariViewController'], {
  onMatch: function (sf) {
    console.log('onMatch: ', sf);
  },
  onComplete: function () {
    console.log('done for SFSafariViewController!');
  }
});
```

For the UIWebView and WKWebView WebViews we also print the associated URL for the sake of completion.

In order to ensure that you will be able to find the instances of the WebViews in the heap, be sure to first navigate to the WebView you've found. Once there, run the code above, e.g. by copying into the Frida REPL:

```
$ frida -U com.authenticationfailure.WheresMyBrowser

# copy the code and wait ...

onMatch:  <UIWebView: 0x14fd25e50; frame = (0 126; 320 393);
                autoresize = RM+BM; layer = <CALayer: 0x1c422d100>>
URL:  <NSMutableURLRequest: 0x1c000ef00> {
  URL: file:///var/mobile/Containers/Data/Application/A654D169-1DB7-429C-9DB9-
↪A871389A8BAA/
          Library/UIWebView/scenario1.html, Method GET, Headers {
    Accept =     (
        "text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8"
    );
    "Upgrade-Insecure-Requests" =     (
        1
    );
    "User-Agent" =     (
        "Mozilla/5.0 (iPhone; CPU iPhone ... AppleWebKit/604.3.5 (KHTML, like␣
↪Gecko) Mobile/..."
    );
} }
```

Now we quit with q and open another WebView (WKWebView in this case). It also gets detected if we repeat the

previous steps:

```
$ frida -U com.authenticationfailure.WheresMyBrowser

# copy the code and wait ...

onMatch:  <WKWebView: 0x1508b1200; frame = (0 0; 320 393); layer = <CALayer:␣
↪0x1c4238f20>>
URL:  file:///var/mobile/Containers/Data/Application/A654D169-1DB7-429C-9DB9-
↪A871389A8BAA/
            Library/WKWebView/scenario1.html
```

We will extend this example in the following sections in order to get more information from the WebViews. We recommend to store this code to a file, e.g. webviews_inspector.js and run it like this:

```
frida -U com.authenticationfailure.WheresMyBrowser -l webviews_inspector.js
```

Reference

- owasp-mastg Testing iOS WebViews (MSTG-PLATFORM-5) Dynamic Analysis Enumerating WebView Instances

6.5.6.2 Checking if JavaScript is Enabled

Remember that if a UIWebView is being used, JavaScript is enabled by default and there's no possibility to disable it.

For WKWebView, you should verify if JavaScript is enabled. Use javaScriptEnabled from WKPreferences for this.

Extend the previous script with the following line:

```
ObjC.choose(ObjC.classes['WKWebView'], {
  onMatch: function (wk) {
    console.log('onMatch: ', wk);
    console.log('javaScriptEnabled:', wk.configuration().preferences().
↪javaScriptEnabled()); // Note that javaScriptEnabled is now deprecated.
//...
  }
});
```

The output shows now that, in fact, JavaScript is enabled:

```
$ frida -U com.authenticationfailure.WheresMyBrowser -l webviews_inspector.js

onMatch:  <WKWebView: 0x1508b1200; frame = (0 0; 320 393); layer = <CALayer:␣
↪0x1c4238f20>>

javaScriptEnabled:  true
```

Reference

- owasp-mastg Testing iOS WebViews (MSTG-PLATFORM-5) Dynamic Analysis Checking if JavaScript is Enabled

Rulebook

- *How to use WKWebView (Required)*

6.5.6.3 Verifying that Only Secure Content is Allowed

UIWebView's do not provide a method for this. However, you may inspect if the system enables the "Upgrade-Insecure-Requests" CSP (Content Security Policy) directive by calling the request method of each UIWebView instance ("Upgrade-Insecure-Requests" should be available starting on iOS 10 which included a new version of WebKit, the browser engine powering the iOS WebViews). See an example in the previous section "Enumerating WebView Instances".

For WKWebView's, you may call the method hasOnlySecureContent for each of the WKWebViews found in the heap. Remember to do so once the WebView has loaded.

Extend the previous script with the following line:

```
ObjC.choose(ObjC.classes['WKWebView'], {
  onMatch: function (wk) {
    console.log('onMatch: ', wk);
    console.log('hasOnlySecureContent: ', wk.hasOnlySecureContent().toString());
    //...
      }
    });
```

The output shows that some of the resources on the page have been loaded through insecure connections:

```
$ frida -U com.authenticationfailure.WheresMyBrowser -l webviews_inspector.js

onMatch:  <WKWebView: 0x1508b1200; frame = (0 0; 320 393); layer = <CALayer:␣
↪0x1c4238f20>>

hasOnlySecureContent:  false
```

Reference

- owasp-mastg Testing iOS WebViews (MSTG-PLATFORM-5) Dynamic Analysis Verifying that Only Secure Content is Allowed

Rulebook

- *How to use WKWebView (Required)*
- *After loading the WKWebView, verify that the loaded resource was obtained over an encrypted connection with the hasOnlySecureContent method (Required)*

6.5.6.4 Testing for WebView URI Manipulation

Make sure that the WebView's URI cannot be manipulated by the user in order to load other types of resources than necessary for the functioning of the WebView. This can be specifically dangerous when the WebView's content is loaded from the local file system, allowing the user to navigate to other resources within the application.

Reference

- owasp-mastg Testing iOS WebViews (MSTG-PLATFORM-5) Dynamic Analysis Testing for WebView URI Manipulation

6.5.7 Rulebook

1. *Do not use UIWebView as it has been deprecated since iOS 12 (Required)*
2. *How to use WKWebView (Required)*
3. *Disable JavaScript unless explicitly required (Recommended)*
4. *After loading the WKWebView, verify that the loaded resource was obtained over an encrypted connection with the hasOnlySecureContent method (Required)*
5. *How to use SFSafariViewController (Required)*

6.5.7.1 Do not use UIWebView as it has been deprecated since iOS 12 (Required)

Since UIWebView has been abolished since iOS 12, it is necessary to use WKWebView or SFSafariViewController.

Refer to the following for whether to use WKWebView or SFSafariViewController.

- *How to use WKWebView (Required)*
- *How to use SFSafariViewController (Required)*

* There is no sample code because it is a deprecated rule.

If this is violated, the following may occur.

- App submissions, uploads to App Store Connect, and TestFlight are no longer possible.

6.5.7.2 How to use WKWebView (Required)

WKWebView is part of the WebKit framework. WKWebView lets you seamlessly integrate web content into your app' s UI. You can display all or part of web content directly within your app by loading views that work with existing HTML , CSS , and JavaScript content.

Since JavaScript cannot be disabled in SFSafariViewController , it is recommended to use WKWebView if the purpose is to extend the user interface of the application.

WKWebView implementation:

```
import UIKit
import WebKit

class ViewController: UIViewController, WKUIDelegate {

    var webView: WKWebView!

    override func loadView() {
        let webConfiguration = WKWebViewConfiguration()
        webView = WKWebView(frame: .zero, configuration: webConfiguration)
        webView.uiDelegate = self
        view = webView
    }

    override func viewDidLoad() {
        super.viewDidLoad()

        let myURL = URL(string:"https://www.apple.com")
        let myRequest = URLRequest(url: myURL!)
        webView.load(myRequest)
    }
}
```

If this is violated, the following may occur.

- Vulnerable to script injection if JavaScript usage is enabled.

6.5.7.3 Disable JavaScript unless explicitly required (Recommended)

WKWebView has stronger cooperation with JavaScript than UIWebView. By using WKWebView, it is now possible to easily load and execute application-specific JavaScript on an existing web page.

The default value of javaScriptEnable is true. Setting this property to false disables any JavaScript loaded or executed by the web page. This setting has no effect on user scripts. This item is currently deprecated.

```
let webPreferences = WKPreferences()
webPreferences.javaScriptEnabled = false // Note that javaScriptEnabled is now␣
↪deprecated.
```

Since javaScriptEnabled has been abolished from iOS 14, it is necessary to use allowsContentJavaScript . Setting this property to false disables any JavaScript loaded or executed by the web page.

```
let webView = WKWebView()
webView.configuration.defaultWebpagePreferences.allowsContentJavaScript = false
```

If this is not noted, the following may occur.

- JavaScript loaded or executed by the web page is enabled.

6.5.7.4 After loading the WKWebView, verify that the loaded resource was obtained over an encrypted connection with the hasOnlySecureContent method (Required)

hasOnlySecureContent is a boolean value that indicates whether the WebView is loading all resources on the page over a secure encrypted connection.

Once the WebView is loaded, do the following and make sure it' s fetched over an encrypted connection:

```
ObjC.choose(ObjC.classes['WKWebView'], {
  onMatch: function (wk) {
    console.log('onMatch: ', wk);
    console.log('hasOnlySecureContent: ', wk.hasOnlySecureContent().toString());
    //...
      }
    });
```

If this is violated, the following may occur.

- Unable to see if a resource loaded by a WebView was obtained over an insecure connection.

6.5.7.5 How to use SFSafariViewController (Required)

SFSafariViewController is part of the SafariServices framework. Using this API, users can browse web pages or websites within her app. It also provides similar behavior to Safari, including features such as password autofill, readers, and secure browsing.

SFSafariViewController implementation:

```
import UIKit
import SafariServices

class SimpleSafariViewController: UIViewController {

    override func viewDidLoad() {
        super.viewDidLoad()
    }

    @IBAction func tappedButton(_ sender: Any) {
        let url = URL(string: "https://www.google.co.jp/")
        let safariView = SFSafariViewController(url: url!)
        present(safariView, animated: true)
```

(continues on next page)

(continued from previous page)

```
    }
}
```

If this is not noted, the following may occur.

- The use of JavaScript cannot be disabled, making it vulnerable to script injection.

* Note that the following cases can only be implemented with WKWebView.

- I want to do something when the page loads.
- I want to know the URL of the loaded page or the page I tried to load.
- I want to run JavaScript on the page being displayed.

6.6 MSTG-PLATFORM-6

WebViews are configured to allow only the minimum set of protocol handlers required (ideally, only https is supported). Potentially dangerous handlers, such as file, tel and app-id, are disabled.

6.6.1 Testing WebView Protocol Handlers

Several default schemes are available that are being interpreted in a WebView on iOS, for example:

- http(s)://
- file://
- tel://

WebViews can load remote content from an endpoint, but they can also load local content from the app data directory. If the local content is loaded, the user shouldn't be able to influence the filename or the path used to load the file, and users shouldn't be able to edit the loaded file.

Use the following best practices as defensive-in-depth measures:

- Create a list that defines local and remote web pages and URL schemes that are allowed to be loaded.
- Create checksums of the local HTML/JavaScript files and check them while the app is starting up. Minify JavaScript files "Minification (programming)") to make them harder to read.

Reference

- owasp-mastg Testing WebView Protocol Handlers (MSTG-PLATFORM-6) Overview

Rulebook

- *How to load local content in WebView (Required)*

6.6.2 Static Analysis

- Testing how WebViews are loaded
- Testing WebView file access
- Checking telephone number detection

6.6.2.1 Testing How WebViews are Loaded

If a WebView is loading content from the app data directory, users should not be able to change the filename or path from which the file is loaded, and they shouldn't be able to edit the loaded file.

This presents an issue especially in UIWebViews loading untrusted content via the deprecated methods loadHTMLString:baseURL: or loadData:MIMEType:textEncodingName: baseURL: and setting the baseURL parameter to nil or to a file: or applewebdata: URL schemes. In this case, in order to prevent unauthorized access to local files, the best option is to set it instead to about:blank. However, the recommendation is to avoid the use of UIWebViews and switch to WKWebViews instead.

Here's an example of a vulnerable UIWebView from "Where's My Browser?":

```
let scenario2HtmlPath = Bundle.main.url(forResource: "web/UIWebView/scenario2.html
↪", withExtension: nil)
do {
    let scenario2Html = try String(contentsOf: scenario2HtmlPath!, encoding: .utf8)
    uiWebView.loadHTMLString(scenario2Html, baseURL: nil)
} catch {}
```

The page loads resources from the internet using HTTP, enabling a potential MITM to exfiltrate secrets contained in local files, e.g. in shared preferences.

When working with WKWebViews, Apple recommends using loadHTMLString:baseURL: or loadData:MIMEType:textEncodingName:baseURL: to load local HTML files and loadRequest: for web content. Typically, the local files are loaded in combination with methods including, among others: pathForResource:ofType:, URLForResource:withExtension: or init(contentsOf:encoding:).

Search the source code for the mentioned methods and inspect their parameters.

Example in Objective-C:

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    WKWebViewConfiguration *configuration = [[WKWebViewConfiguration alloc] init];

    self.webView = [[WKWebView alloc] initWithFrame:CGRectMake(10, 20,
        CGRectGetWidth([UIScreen mainScreen].bounds) - 20,
        CGRectGetHeight([UIScreen mainScreen].bounds) - 84)␣
↪configuration:configuration];
    self.webView.navigationDelegate = self;
    [self.view addSubview:self.webView];

    NSString *filePath = [[NSBundle mainBundle] pathForResource:@"example_file"␣
↪ofType:@"html"];
    NSString *html = [NSString stringWithContentsOfFile:filePath
                                encoding:NSUTF8StringEncoding error:nil];
    [self.webView loadHTMLString:html baseURL:[NSBundle mainBundle].resourceURL];
}
```

Example in Swift from "Where's My Browser?":

```
let scenario2HtmlPath = Bundle.main.url(forResource: "web/WKWebView/scenario2.html
↪", withExtension: nil)
do {
    let scenario2Html = try String(contentsOf: scenario2HtmlPath!, encoding: .utf8)
    wkWebView.loadHTMLString(scenario2Html, baseURL: nil)
} catch {}
```

If only having the compiled binary, you can also search for these methods, e.g.:

```
$ rabin2 -zz ./WheresMyBrowser | grep -i "loadHTMLString"
231 0x0002df6c 24 (4.__TEXT.__objc_methname) ascii loadHTMLString:baseURL:
```

In a case like this, it is recommended to perform dynamic analysis to ensure that this is in fact being used and from which kind of WebView. The baseURL parameter here doesn't present an issue as it will be set to "null" but could be an issue if not set properly when using a UIWebView. See "Checking How WebViews are Loaded" for an example about this.

* See "*Testing How WebViews are Loaded*" for proper settings.

In addition, you should also verify if the app is using the method loadFileURL: allowingReadAccessToURL:. Its first parameter is URL and contains the URL to be loaded in the WebView, its second parameter allowingReadAccessToURL may contain a single file or a directory. If containing a single file, that file will be available to the WebView. However, if it contains a directory, all files on that directory will be made available to the WebView. Therefore, it is worth inspecting this and in case it is a directory, verifying that no sensitive data can be found inside it.

Example in Swift from "Where's My Browser?" :

```
var scenario1Url = FileManager.default.urls(for: .libraryDirectory, in: .
↪userDomainMask)[0]
scenario1Url = scenario1Url.appendingPathComponent("WKWebView/scenario1.html")
wkWebView.loadFileURL(scenario1Url, allowingReadAccessTo: scenario1Url)
```

In this case, the parameter allowingReadAccessToURL contains a single file "WKWebView/scenario1.html" , meaning that the WebView has exclusively access to that file.

In the compiled binary:

```
$ rabin2 -zz ./WheresMyBrowser | grep -i "loadFileURL"
237 0x0002dff1 37 (4.__TEXT.__objc_methname) ascii␣
↪loadFileURL:allowingReadAccessToURL:
```

Reference

- owasp-mastg Testing WebView Protocol Handlers (MSTG-PLATFORM-6) Static Analysis Testing How WebViews are Loaded

Rulebook

- *How to load local content in WebView (Required)*
- *How to load local content in WKWebView (Recommended)*
- *Do not specify a directory containing sensitive data in the parameter "allowingReadAccessTo" of WKWebView.loadFileURL (Required)*

6.6.2.2 Testing WebView File Access

If you have found a UIWebView being used, then the following applies:

- The file:// scheme is always enabled.
- File access from file:// URLs is always enabled.
- Universal access from file:// URLs is always enabled.

Regarding WKWebViews:

- The file:// scheme is also always enabled and it **cannot be disabled**.
- It disables file access from file:// URLs by default but it can be enabled.

The following WebView properties can be used to configure file access:

- allowFileAccessFromFileURLs (WKPreferences, false by default): it enables JavaScript running in the context of a file:// scheme URL to access content from other file:// scheme URLs.
- allowUniversalAccessFromFileURLs (WKWebViewConfiguration, false by default): it enables JavaScript running in the context of a file:// scheme URL to access content from any origin.

For example, it is possible to set the **undocumented property** allowFileAccessFromFileURLs by doing this:

Objective-C:

```
[webView.configuration.preferences setValue:@YES forKey:@
↪"allowFileAccessFromFileURLs"];
```

Swift:

```
webView.configuration.preferences.setValue(true, forKey:
↪"allowFileAccessFromFileURLs")
```

If one or more of the above properties are activated, you should determine whether they are really necessary for the app to work properly.

Reference

- owasp-mastg Testing WebView Protocol Handlers (MSTG-PLATFORM-6) Static Analysis Testing WebView File Access

Rulebook

- *Configure file access using WebView properties (Required)*

6.6.2.3 Checking Telephone Number Detection

In Safari on iOS, telephone number detection is on by default. However, you might want to turn it off if your HTML page contains numbers that can be interpreted as phone numbers, but are not phone numbers, or to prevent the DOM document from being modified when parsed by the browser. To turn off telephone number detection in Safari on iOS, use the format-detection meta tag (<meta name = "format-detection" content = "telephone=no" >). An example of this can be found in the Apple developer documentation. Phone links should be then used (e.g. 1-408-555-5555) to explicitly create a link.

Reference

- owasp-mastg Testing WebView Protocol Handlers (MSTG-PLATFORM-6) Static Analysis Checking Telephone Number Detection

Rulebook

- *Phone link should be used to explicitly create the link (Required)*

6.6.3 Dynamic Analysis

If it' s possible to load local files via a WebView, the app might be vulnerable to directory traversal attacks. This would allow access to all files within the sandbox or even to escape the sandbox with full access to the file system (if the device is jailbroken). It should therefore be verified if a user can change the filename or path from which the file is loaded, and they shouldn' t be able to edit the loaded file.

To simulate an attack, you may inject your own JavaScript into the WebView with an interception proxy or simply by using dynamic instrumentation. Attempt to access local storage and any native methods and properties that might be exposed to the JavaScript context.

In a real-world scenario, JavaScript can only be injected through a permanent backend Cross-Site Scripting vulnerability or a MITM attack. See the OWASP XSS Prevention Cheat Sheet and the chapter "iOS Network Communication" for more information.

For what concerns this section we will learn about:

- Checking how WebViews are loaded
- Determining WebView file access

Reference

- owasp-mastg Testing WebView Protocol Handlers (MSTG-PLATFORM-6) Dynamic Analysis

(Checking_How_WebViews_are_Loaded)

6.6.3.1 Checking How WebViews are Loaded

As we have seen above in "Testing How WebViews are Loaded", if "scenario 2" of the WKWebViews is loaded, the app will do so by calling URLForResource:withExtension: and loadHTMLString:baseURL.

To quicky inspect this, you can use frida-trace and trace all "loadHTMLString" and "URLForResource:withExtension:" methods.

```
$ frida-trace -U "Where's My Browser?"
    -m "*[WKWebView *loadHTMLString*]" -m "*[* URLForResource:withExtension:]"

 14131 ms  -[NSBundle URLForResource:0x1c0255390 withExtension:0x0]
 14131 ms  URLForResource: web/WKWebView/scenario2.html
 14131 ms  withExtension: 0x0
 14190 ms  -[WKWebView loadHTMLString:0x1c0255390 baseURL:0x0]
 14190 ms   HTMLString: <!DOCTYPE html>
    <html>
        ...
        </html>

 14190 ms  baseURL: nil
```

In this case, baseURL is set to nil, meaning that the effective origin is "null". You can obtain the effective origin by running window.origin from the JavaScript of the page (this app has an exploitation helper that allows to write and run JavaScript, but you could also implement a MITM or simply use Frida to inject JavaScript, e.g. via evaluateJavaScript:completionHandler of WKWebView).

As an additional note regarding UIWebViews, if you retrieve the effective origin from a UIWebView where baseURL is also set to nil you will see that it is not set to "null", instead you'll obtain something similar to the following:

```
applewebdata://5361016c-f4a0-4305-816b-65411fc1d780
```

This origin "applewebdata://" is similar to the "file://" origin as it does not implement Same-Origin Policy and allow access to local files and any web resources. In this case, it would be better to set baseURL to "about:blank", this way, the Same-Origin Policy would prevent cross-origin access. However, the recommendation here is to completely avoid using UIWebViews and go for WKWebViews instead.

Reference

- owasp-mastg Testing WebView Protocol Handlers (MSTG-PLATFORM-6) Dynamic Analysis Checking How WebViews are Loaded

6.6.3.2 Determining WebView File Access

Even if not having the original source code, you can quickly determine if the app's WebViews do allow file access and which kind. For this, simply navigate to the target WebView in the app and inspect all its instances, for each of them get the values mentioned in the static analysis, that is, allowFileAccessFromFileURLs and allowUniversalAccessFromFileURLs. This only applies to WKWebViews (UIWebVIews always allow file access).

We continue with our example using the "Where's My Browser?" app and Frida REPL, extend the script with the following content:

```
ObjC.choose(ObjC.classes['WKWebView'], {
  onMatch: function (wk) {
    console.log('onMatch: ', wk);
    console.log('URL: ', wk.URL().toString());
    console.log('javaScriptEnabled: ', wk.configuration().preferences().
↪javaScriptEnabled()); // Note that javaScriptEnabled is now deprecated.
```

(continues on next page)

(continued from previous page)

```
    console.log('allowFileAccessFromFileURLs: ',
            wk.configuration().preferences().valueForKey_(
↪'allowFileAccessFromFileURLs').toString());
    console.log('hasOnlySecureContent: ', wk.hasOnlySecureContent().toString());
    console.log('allowUniversalAccessFromFileURLs: ',
            wk.configuration().valueForKey_('allowUniversalAccessFromFileURLs').
↪toString());
  },
  onComplete: function () {
    console.log('done for WKWebView!');
  }
});
```

If you run it now, you'll have all the information you need:

```
$ frida -U -f com.authenticationfailure.WheresMyBrowser -l webviews_inspector.js

onMatch:  <WKWebView: 0x1508b1200; frame = (0 0; 320 393); layer = <CALayer:␣
↪0x1c4238f20>>
URL:  file:///var/mobile/Containers/Data/Application/A654D169-1DB7-429C-9DB9-
↪A871389A8BAA/
        Library/WKWebView/scenario1.html
javaScriptEnabled:  true // Note that javaScriptEnabled is now deprecated.
allowFileAccessFromFileURLs:  0
hasOnlySecureContent:  false
allowUniversalAccessFromFileURLs:  0
```

Both allowFileAccessFromFileURLs and allowUniversalAccessFromFileURLs are set to "0", meaning that they are disabled. In this app we can go to the WebView configuration and enable allowFileAccessFromFileURLs. If we do so and re-run the script we will see how it is set to "1" this time:

```
$ frida -U -f com.authenticationfailure.WheresMyBrowser -l webviews_inspector.js
...

allowFileAccessFromFileURLs:  1
```

Reference

- owasp-mastg Testing WebView Protocol Handlers (MSTG-PLATFORM-6) Dynamic Analysis Determining WebView File Access

6.6.4 Rulebook

1. *How to load local content in WebView (Required)*
2. *How to load local content in WKWebView (Recommended)*
3. *Do not specify a directory containing sensitive data in the parameter "allowingReadAccessTo" of WKWebView.loadFileURL (Required)*
4. *Configure file access using WebView properties (Required)*
5. *Phone link should be used to explicitly create the link (Required)*

6.6.4.1 How to load local content in WebView (Required)

WebViews can load remote content from an endpoint, but they can also load local content from the app data directory. If the local content is loaded, the user shouldn't be able to influence the filename or the path used to load the file, and users shouldn't be able to edit the loaded file.

Use the following best practices as defensive-in-depth measures:

- Create a list that defines local and remote web pages and URL schemes that are allowed to be loaded.
- Create checksums of the local HTML/JavaScript files and check them while the app is starting up. Minify JavaScript files "Minification (programming)") to make them harder to read.

The sample code below shows how to load local content in WebView.

```
import UIKit
import WebKit

class webViewSample {

    var webView: WKWebView!
    // create webView
    // ...
    func loadLocalHTML() {
        guard let path: String = Bundle.main.path(forResource: "index", ofType:
↪"html") else { return }
        let localHTMLUrl = URL(fileURLWithPath: path, isDirectory: false)
        webView.loadFileURL(localHTMLUrl, allowingReadAccessTo: localHTMLUrl)
    }
}
```

If this is violated, the following may occur.

- Malicious local content may be loaded.

6.6.4.2 How to load local content in WKWebView (Recommended)

When working with WKWebViews, Apple recommends using loadHTMLString:baseURL: or loadData:MIMEType:textEncodingName:baseURL: to load local HTML files and loadRequest: for web content. Typically, the local files are loaded in combination with methods including, among others: pathForResource:ofType:, URLForResource:withExtension: or init(contentsOf:encoding:).

Search the source code for the mentioned methods and inspect their parameters.

Example in Objective-C

```
#import <Foundation/Foundation.h>
#import <WebKit/WebKit.h>
#import <UIKit/UIKit.h>

@interface MyWKWebView : UIView <WKNavigationDelegate, WKUIDelegate> {}

@property (strong, nonatomic) WKWebView *webView;

- (void)settingWebView;
@end


@implementation MyWKWebView {}

-(void)settingWebView {

    WKWebViewConfiguration *configuration = [[WKWebViewConfiguration alloc] init];
```

(continues on next page)

(continued from previous page)

```
    // create WKWebView
    self.webView = [[WKWebView alloc] initWithFrame:CGRectMake(10, 20,
        CGRectGetWidth([UIScreen mainScreen].bounds) - 20,
        CGRectGetHeight([UIScreen mainScreen].bounds) - 84)␣
↪configuration:configuration];

    self.webView.navigationDelegate = self;

    // Add subview to view of UIView
    [self addSubview:self.webView];

    NSString *filePath = [[NSBundle mainBundle] pathForResource:@"example_file"␣
↪ofType:@"html"];
    NSString *html = [NSString stringWithContentsOfFile:filePath
                                    encoding:NSUTF8StringEncoding error:nil];
    [self.webView loadHTMLString:html baseURL:[NSBundle mainBundle].resourceURL];

}

@end
```

Example in Swift from "Where's My Browser?"

```
let scenario2HtmlPath = Bundle.main.url(forResource: "web/WKWebView/scenario2.html
↪", withExtension: nil)
do {
    let scenario2Html = try String(contentsOf: scenario2HtmlPath!, encoding: .utf8)
    wkWebView.loadHTMLString(scenario2Html, baseURL: nil)
} catch {}
```

If it is possible to read local files through a WebView, your app may be vulnerable to directory traversal attacks. In this case, it is possible to exit the sandbox with access to all files in the sandbox or (if the device is jailbroken) full access to the filesystem. Therefore, when loading a local file, you should verify that the user can change the filename or the path from which the file was loaded, and should not be allowed to edit the loaded file.

If this is not noted, the following may occur.

- Malicious local content may be loaded.

6.6.4.3 Do not specify a directory containing sensitive data in the parameter "allowingReadAccessTo" of WKWebView.loadFileURL (Required)

allowReadAccessToURL can contain a single file or directory. If it contains a single file, that file is available in her WebView. But if it contains a directory, all files on that directory will be available in his WebView. So if you include a directory, you should make sure there is no sensitive data in it.

The sample code below shows how to load local HTML in WKWebView.

```
import UIKit
import WebKit

class ViewController: UIViewController {

    func load(_ bundleFileName: String) {
        let wkWebView = WKWebView()
        var url = FileManager.default.urls(for: .libraryDirectory, in: .
↪userDomainMask)[0]
        url = url.appendingPathComponent("WKWebView/scenario1.html")
        wkWebView.loadFileURL(url, allowingReadAccessTo: url)
    }
}
```

If it contains a directory, If there is no scenario1 file in the personal directory, a list of files in personal may be displayed.

```
import UIKit
import WebKit

class ViewController: UIViewController {

    func load(_ bundleFileName: String) {
        let wkWebView = WKWebView()
        var url = FileManager.default.urls(for: .libraryDirectory, in: .
↪userDomainMask)[0]
        url = url.appendingPathComponent("WKWebView/personal/scenario1.html")
        wkWebView.loadFileURL(url, allowingReadAccessTo: url)
    }
}
```

If this is violated, the following may occur.

- All files on the directory will be available in WebView, potentially exposing sensitive data.

6.6.4.4 Configure file access using WebView properties (Required)

The following WebView properties can be used to configure file access:

- allowFileAccessFromFileURLs (WKPreferences, false by default): it enables JavaScript running in the context of a file:// scheme URL to access content from other file:// scheme URLs.
- allowUniversalAccessFromFileURLs (WKWebViewConfiguration, false by default): it enables JavaScript running in the context of a file:// scheme URL to access content from any origin.

For example, it is possible to set the **undocumented property** allowFileAccessFromFileURLs by doing this:

Objective-C:

```
#import <Foundation/Foundation.h>
#import <WebKit/WebKit.h>
#import <UIKit/UIKit.h>

@interface MyWKWebViewAllowFileAccessFromFileURLs : UIView <WKNavigationDelegate,␣
↪WKUIDelegate> {}

@property (strong, nonatomic) WKWebView *webView;

- (void)settingWebView;
- (void)setConfiguration;
@end


@implementation MyWKWebViewAllowFileAccessFromFileURLs {}

-(void)settingWebView {

    WKWebViewConfiguration *configuration = [[WKWebViewConfiguration alloc] init];

    // Set to webview generation property
    self.webView = [[WKWebView alloc] initWithFrame:CGRectMake(10, 20,
        CGRectGetWidth([UIScreen mainScreen].bounds) - 20,
        CGRectGetHeight([UIScreen mainScreen].bounds) - 84)␣
↪configuration:configuration];

    self.webView.navigationDelegate = self;
    [self setConfiguration];
```

(continues on next page)

(continued from previous page)

```
    [self addSubview:self.webView];


    NSString *filePath = [[NSBundle mainBundle] pathForResource:@"example_file"␣
↪ofType:@"html"];
    NSString *html = [NSString stringWithContentsOfFile:filePath
                                encoding:NSUTF8StringEncoding error:nil];
    [self.webView loadHTMLString:html baseURL:[NSBundle mainBundle].resourceURL];




}

-(void)setConfiguration {

    // set allowFileAccessFromFileURLs
    [self.webView.configuration.preferences setValue:@YES forKey:@
↪"allowFileAccessFromFileURLs"];
}

@end
```

Swift:

```
import Foundation
import UIKit
import WebKit

class MyWKWebViewAllowFileAccessFromFileURLs: UIView {

    var webView: WKWebView!

    func settingWebView() {

        let webConfiguration = WKWebViewConfiguration()
        webView = WKWebView(frame: .zero, configuration: webConfiguration)
        webView.uiDelegate = self
        webView.navigationDelegate = self

        self.setConfiguration()

        self.addSubview(webView)


        guard let filepath = Bundle.main.url(forResource: "example_file",␣
↪withExtension: "html") else {
            return
        }

        webView.loadFileURL(filepath, allowingReadAccessTo: filepath)

    }

    func setConfiguration() {

        webView.configuration.preferences.setValue(true, forKey:
↪"allowFileAccessFromFileURLs")
    }
```

(continues on next page)

(continued from previous page)

```
}

// MARK: - WKWebView ui delegate
extension MyWKWebViewAllowFileAccessFromFileURLs: WKUIDelegate {

}

// MARK: - WKWebView WKNavigation delegate
extension MyWKWebViewAllowFileAccessFromFileURLs: WKNavigationDelegate {

}
```

If one or more of the above properties are activated, you should determine whether they are really necessary for the app to work properly.

If this is violated, the following may occur.

- If you use the file scheme, you can access all files that the app can access.
- Accepting requests with unintended file schemes.

6.6.4.5 Phone link should be used to explicitly create the link (Required)

In Safari on iOS, telephone number detection is on by default. However, you might want to turn it off if your HTML page contains numbers that can be interpreted as phone numbers, but are not phone numbers, or to prevent the DOM document from being modified when parsed by the browser. To turn off telephone number detection in Safari on iOS, use the format-detection meta tag (<meta name = "format-detection" content = "telephone=no" >). An example of this can be found in the Apple developer documentation. Phone links should be then used (e.g. 1-408-555-5555) to explicitly create a link.

Reference

- owasp-mastg Testing WebView Protocol Handlers (MSTG-PLATFORM-6) Static Analysis Checking Telephone Number Detection

If this is violated, the following may occur.

- Numbers that are not phone numbers are interpreted as phone numbers on HTML.
- The DOM document is modified when parsed in a browser.

6.7 MSTG-PLATFORM-7

If native methods of the app are exposed to a WebView, verify that the WebView only renders JavaScript contained within the app package.

6.7.1 Bridge between JavaScript and native in WebView

Since iOS 7, Apple introduced APIs that allow communication between the JavaScript runtime in the WebView and the native Swift or Objective-C objects. If these APIs are used carelessly, important functionality might be exposed to attackers who manage to inject malicious scripts into the WebView (e.g., through a successful Cross-Site Scripting attack).

6.7.2 Static Analysis

Both UIWebView and WKWebView provide a means of communication between the WebView and the native app. Any important data or native functionality exposed to the WebView JavaScript engine would also be accessible to rogue JavaScript running in the WebView.

Reference

- owasp-mastg Determining Whether Native Methods Are Exposed Through WebViews (MSTG-PLATFORM-7) Static Analysis

6.7.2.1 Testing UIWebView JavaScript to Native Bridges

There are two fundamental ways of how native code and JavaScript can communicate:

- JSContext: When an Objective-C or Swift block is assigned to an identifier in a JSContext, JavaScriptCore automatically wraps the block in a JavaScript function.
- JSExport protocol: Properties, instance methods and class methods declared in a JSExport-inherited protocol are mapped to JavaScript objects that are available to all JavaScript code. Modifications of objects that are in the JavaScript environment are reflected in the native environment.

Note that only class members defined in the JSExport protocol are made accessible to JavaScript code.

Look out for code that maps native objects to the JSContext associated with a WebView and analyze what functionality it exposes, for example no sensitive data should be accessible and exposed to WebViews.

In Objective-C, the JSContext associated with a UIWebView is obtained as follows:

```
[webView valueForKeyPath:@"documentView.webView.mainFrame.javaScriptContext"]
```

Reference

- owasp-mastg Determining Whether Native Methods Are Exposed Through WebViews (MSTG-PLATFORM-7) Static Analysis Testing UIWebView JavaScript to Native Bridges

Rulebook

- *How native code and JavaScript communicate (Required)*

6.7.2.2 Testing WKWebView JavaScript to Native Bridges

JavaScript code in a WKWebView can still send messages back to the native app but in contrast to UIWebView, it is not possible to directly reference the JSContext of a WKWebView. Instead, communication is implemented using a messaging system and using the postMessage function, which automatically serializes JavaScript objects into native Objective-C or Swift objects. Message handlers are configured using the method add(_ scriptMessageHandler:name:).

Verify if a JavaScript to native bridge exists by searching for WKScriptMessageHandler and check all exposed methods. Then verify how the methods are called.

The following example from "Where's My Browser?" demonstrates this.

First we see how the JavaScript bridge is enabled:

```
func enableJavaScriptBridge(_ enabled: Bool) {
    options_dict["javaScriptBridge"]?.value = enabled
    let userContentController = wkWebViewConfiguration.userContentController
    userContentController.removeScriptMessageHandler(forName: "javaScriptBridge")

    if enabled {
            let javaScriptBridgeMessageHandler = JavaScriptBridgeMessageHandler()
            userContentController.add(javaScriptBridgeMessageHandler, name:
↪"javaScriptBridge")
    }
}
```

Adding a script message handler with name "name" (or "javaScriptBridge" in the example above) causes the JavaScript function window.webkit.messageHandlers.myJavaScriptMessageHandler.postMessage to be defined in all frames in all web views that use the user content controller. It can be then used from the HTML file like this:

```
function invokeNativeOperation() {
    value1 = document.getElementById("value1").value
    value2 = document.getElementById("value2").value
    window.webkit.messageHandlers.javaScriptBridge.postMessage(["multiplyNumbers",␣
↪value1, value2]);
}
```

The called function resides in JavaScriptBridgeMessageHandler.swift:

```
class JavaScriptBridgeMessageHandler: NSObject, WKScriptMessageHandler {

//...

case "multiplyNumbers":

        let arg1 = Double(messageArray[1])!
        let arg2 = Double(messageArray[2])!
        result = String(arg1 * arg2)
//...

let javaScriptCallBack = "javascriptBridgeCallBack('\(functionFromJS)','\(result)')
↪"
message.webView?.evaluateJavaScript(javaScriptCallBack, completionHandler: nil)
```

The problem here is that the JavaScriptBridgeMessageHandler not only contains that function, it also exposes a sensitive function:

```
case "getSecret":
        result = "XSRSOGKC342"
```

Reference

- owasp-mastg Determining Whether Native Methods Are Exposed Through WebViews (MSTG-PLATFORM-7) Static Analysis Testing UIWebView JavaScript to Native Bridges

Rulebook

- *How WKWebView's JavaScript code sends messages back to the native app (Required)*

6.7.3 Dynamic Analysis

At this point you've surely identified all potentially interesting WebViews in the iOS app and got an overview of the potential attack surface (via static analysis, the dynamic analysis techniques that we have seen in previous sections or a combination of them). This would include HTML and JavaScript files, usage of the JSContext / JSExport for UIWebView and WKScriptMessageHandler for WKWebView, as well as which functions are exposed and present in a WebView.

Further dynamic analysis can help you exploit those functions and get sensitive data that they might be exposing. As we have seen in the static analysis, in the previous example it was trivial to get the secret value by performing reverse engineering (the secret value was found in plain text inside the source code) but imagine that the exposed function retrieves the secret from secure storage. In this case, only dynamic analysis and exploitation would help.

The procedure for exploiting the functions starts with producing a JavaScript payload and injecting it into the file that the app is requesting. The injection can be accomplished via various techniques, for example:

- If some of the content is loaded insecurely from the Internet over HTTP (mixed content), you can try to implement a MITM attack.

- You can always perform dynamic instrumentation and inject the JavaScript payload by using frameworks like Frida and the corresponding JavaScript evaluation functions available for the iOS WebViews (stringByEvaluatingJavaScriptFromString: for UIWebView and evaluateJavaScript:completionHandler: for WKWebView).

In order to get the secret from the previous example of the "Where's My Browser?" app, you can use one of these techniques to inject the following payload that will reveal the secret by writing it to the "result" field of the WebView:

```
function javascriptBridgeCallBack(name, value) {
    document.getElementById("result").innerHTML=value;
};
window.webkit.messageHandlers.javaScriptBridge.postMessage(["getSecret"]);
```

Of course, you may also use the Exploitation Helper it provides:

See another example for a vulnerable iOS app and function that is exposed to a WebView in [#thiel2] page 156.

Reference

- owasp-mastg Determining Whether Native Methods Are Exposed Through WebViews (MSTG-PLATFORM-7) Dynamic Analysis

6.7.4 Rulebook

1. *How native code and JavaScript communicate (Required)*
2. *How WKWebView' s JavaScript code sends messages back to the native app (Required)*

6.7.4.1 How native code and JavaScript communicate (Required)

There are basically two ways native code and JavaScript communicate.

- JSContext : When an Objective-C or Swift block is assigned an identifier in JSContext, JavaScriptCore automatically wraps that block in a JavaScript function.
- JSExport protocol: Properties, instance methods, and class methods declared with the JSExport-inherited protocol are mapped to JavaScript objects and are available to all JavaScript code. Modifications to objects in the JavaScript environment are reflected in the native environment.

Note, however, that only class members defined with the JSExport protocol will be accessible from JavaScript code.

Focus on the code that maps native objects to the JSContext associated with the WebView and analyze what functionality it exposes. For example, sensitive data should not be accessed and exposed in WebView.

In Objective-C, the JSContext associated with UIWebView is obtained like this:

```
#import <Foundation/Foundation.h>
#import <WebKit/WebKit.h>
#import <UIKit/UIKit.h>

@interface MyWKWebView : UIView <WKNavigationDelegate, WKUIDelegate> {}

@property (strong, nonatomic) WKWebView *webView;

- (void)settingWebView;
@end


@implementation MyWKWebView {}

-(void)settingWebView {

    WKWebViewConfiguration *configuration = [[WKWebViewConfiguration alloc] init];

    self.webView = [[WKWebView alloc] initWithFrame:CGRectMake(10, 20,
        CGRectGetWidth([UIScreen mainScreen].bounds) - 20,
        CGRectGetHeight([UIScreen mainScreen].bounds) - 84)␣
↪configuration:configuration];

    self.webView.navigationDelegate = self;

    // javaScriptContext
    [self.webView valueForKeyPath:@"documentView.webView.mainFrame.
↪javaScriptContext"];


    [self addSubview:self.webView];

    NSString *filePath = [[NSBundle mainBundle] pathForResource:@"example_file"␣
↪ofType:@"html"];
    NSString *html = [NSString stringWithContentsOfFile:filePath
                                encoding:NSUTF8StringEncoding error:nil];
    [self.webView loadHTMLString:html baseURL:[NSBundle mainBundle].resourceURL];
```

(continues on next page)

(continued from previous page)

```
}

@end
```

JSExport protocol

```
#import <Foundation/Foundation.h>
#import <JavaScriptCore/JavaScriptCore.h>

// JSExport obj
@protocol MyJSExport <JSExport>
- (void)method1:(NSString *)param1;
@end

@interface MyJSCode : NSObject <MyJSExport>
@end

@implementation MyJSCode
- (void)method1:(NSString *)param1 {
    NSLog(@"method1");
}

@end


// JSContext
@interface MyJSExample : NSObject
- (void)execute;
@end

@implementation MyJSExample
- (void)method1:(NSString *)param1 {
    NSLog(@"method1");
}

- (void)execute {
    JSContext *sContext = [[JSContext alloc] init];
    if (sContext)
    {
        sContext[@"mycode"] = [[MyJSCode alloc] init];
        [sContext evaluateScript:@"mycode.method1(\"foo\");"];

    }
}

@end
```

Reference

- owasp-mastg Determining Whether Native Methods Are Exposed Through WebViews (MSTG-PLATFORM-7) Static Analysis Testing UIWebView JavaScript to Native Bridges
- JSExport

If this is violated, the following may occur.

- This could expose critical functionality to an attacker who injects malicious script into the WebView.

6.7.4.2 How WKWebView' s JavaScript code sends messages back to the native app (Required)

WKWebView' s JavaScript code can send messages back to the native app, but in contrast to UIWebView, it is not possible to directly reference WKWebView' s JSContext. Instead, communication is implemented using a messaging system and the postMessage function, which automatically serializes JavaScript objects to native Objective-C or Swift objects.The message handler is set with a method called add(_ scriptMessageHandler:name:).

Check for the existence of a JavaScript-to-native bridge by searching for WKScriptMessageHandler and checking all exposed methods. And see how the method is called.

The following example for "Where' s My Browser?" illustrates this.

First, check how the JavaScript bridge is enabled.

```
func enableJavaScriptBridge(_ enabled: Bool) {
    options_dict["javaScriptBridge"]?.value = enabled
    let userContentController = wkWebViewConfiguration.userContentController
    userContentController.removeScriptMessageHandler(forName: "javaScriptBridge")

    if enabled {
            let javaScriptBridgeMessageHandler = JavaScriptBridgeMessageHandler()
            userContentController.add(javaScriptBridgeMessageHandler, name:
↪"javaScriptBridge")
    }
}
```

Check whether a method that should not be published is specified in the message handler.

If this is violated, the following may occur.

- This could expose critical functionality to an attacker who injects malicious script into the WebView.

6.8 MSTG-PLATFORM-8

Object deserialization, if any, is implemented using safe serialization APIs.

6.8.1 Object Serialization in iOS

There are several ways to persist an object on iOS:

6.8.1.1 Object Encoding

iOS comes with two protocols for object encoding and decoding for Objective-C or NSObjects: NSCoding and NSSecureCoding. When a class conforms to either of the protocols, the data is serialized to NSData: a wrapper for byte buffers. Note that Data in Swift is the same as NSData or its mutable counterpart: NSMutableData. The NSCoding protocol declares the two methods that must be implemented in order to encode/decode its instance-variables. A class using NSCoding needs to implement NSObject or be annotated as an @objc class. The NSCoding protocol requires to implement encode and init as shown below.

```
class CustomPoint: NSObject, NSCoding {

    //required by NSCoding:
    func encode(with aCoder: NSCoder) {
        aCoder.encode(x, forKey: "x")
        aCoder.encode(name, forKey: "name")
    }

    var x: Double = 0.0
    var name: String = ""
```

(continues on next page)

(continued from previous page)

```
    init(x: Double, name: String) {
            self.x = x
            self.name = name
    }

    // required by NSCoding: initialize members using a decoder.
    required convenience init?(coder aDecoder: NSCoder) {
            guard let name = aDecoder.decodeObject(forKey: "name") as? String
                    else {return nil}
            self.init(x:aDecoder.decodeDouble(forKey:"x"),
                              name:name)
    }

    //getters/setters/etc.
}
```

The issue with NSCoding is that the object is often already constructed and inserted before you can evaluate the class-type. This allows an attacker to easily inject all sorts of data. Therefore, the NSSecureCoding protocol has been introduced. When conforming to NSSecureCoding you need to include:

```
static var supportsSecureCoding: Bool {
        return true
}
```

when init(coder:) is part of the class. Next, when decoding the object, a check should be made, e.g.:

```
let obj = decoder.decodeObject(of:MyClass.self, forKey: "myKey")
```

The conformance to NSSecureCoding ensures that objects being instantiated are indeed the ones that were expected. However, there are no additional integrity checks done over the data and the data is not encrypted. Therefore, any secret data needs additional encryption and data of which the integrity must be protected, should get an additional HMAC.

Note, when NSData (Objective-C) or the keyword let (Swift) is used: then the data is immutable in memory and cannot be easily removed.

Reference

- owasp-mastg Testing Object Persistence (MSTG-PLATFORM-8) Overview Object Encoding

Rulebook

- *Use NSSecureCoding for object encoding and decoding (Required)*

6.8.1.2 Object Archiving with NSKeyedArchiver

NSKeyedArchiver is a concrete subclass of NSCoder and provides a way to encode objects and store them in a file. The NSKeyedUnarchiver decodes the data and recreates the original data. Let's take the example of the NSCoding section and now archive and unarchive them:

```
// archiving:
NSKeyedArchiver.archiveRootObject(customPoint, toFile: "/path/to/archive")

// unarchiving:
guard let customPoint = NSKeyedUnarchiver.unarchiveObjectWithFile("/path/to/archive
↪") as?
    CustomPoint else { return nil }
```

When decoding a keyed archive, because values are requested by name, values can be decoded out of sequence or not at all. Keyed archives, therefore, provide better support for forward and backward compatibility. This means that an archive on disk could actually contain additional data which is not detected by the program, unless the key for that given data is provided at a later stage.

Note that additional protection needs to be in place to secure the file in case of confidential data, as the data is not encrypted within the file. See the chapter "Data Storage on iOS" for more details.

Reference

- owasp-mastg Testing Object Persistence (MSTG-PLATFORM-8) Overview Object Archiving with NSKeyedArchiver

Rulebook

- *Encrypt data when storing sensitive information on the device using object persistence (Required)*

6.8.1.3 Codable

With Swift 4, the Codable type alias arrived: it is a combination of the Decodable and Encodable protocols. A String, Int, Double, Date, Data and URL are Codable by nature: meaning they can easily be encoded and decoded without any additional work. Let' s take the following example:

```
struct CustomPointStruct:Codable {
    var x: Double
    var name: String
}
```

By adding Codable to the inheritance list for the CustomPointStruct in the example, the methods init(from:) and encode(to:) are automatically supported. Fore more details about the workings of Codable check the Apple Developer Documentation. The Codables can easily be encoded / decoded into various representations: NSData using NSCoding/NSSecureCoding, JSON, Property Lists, XML, etc. See the subsections below for more details.

Reference

- owasp-mastg Testing Object Persistence (MSTG-PLATFORM-8) Overview Codable

Rulebook

- *In Swift 4, use a combination of Decodable and Encodable protocols (Required)*

6.8.1.4 JSON and Codable

There are various ways to encode and decode JSON within iOS by using different 3rd party libraries:

- Mantle
- JSONModel library
- SwiftyJSON library
- ObjectMapper library
- JSONKit
- JSONModel
- YYModel
- SBJson 5
- Unbox
- Gloss
- Mapper
- JASON
- Arrow

The libraries differ in their support for certain versions of Swift and Objective-C, whether they return (im)mutable results, speed, memory consumption and actual library size. Again, note in case of immutability: confidential information cannot be removed from memory easily.

Next, Apple provides support for JSON encoding/decoding directly by combining Codable together with a JSONEncoder and a JSONDecoder:

```
struct CustomPointStruct: Codable {
    var point: Double
    var name: String
}

let encoder = JSONEncoder()
encoder.outputFormatting = .prettyPrinted

let test = CustomPointStruct(point: 10, name: "test")
let data = try encoder.encode(test)
let stringData = String(data: data, encoding: .utf8)

// stringData = Optional ({
// "point" : 10,
// "name" : "test"
// })
```

JSON itself can be stored anywhere, e.g., a (NoSQL) database or a file. You just need to make sure that any JSON that contains secrets has been appropriately protected (e.g., encrypted/HMACed). See the chapter "Data Storage on iOS" for more details.

Reference

- owasp-mastg Testing Object Persistence (MSTG-PLATFORM-8) Overview JSON and Codable

Rulebook

- *Codable/JSONEncoder/JSONDecoder is used for JSON encoding/decoding (Required)*
- *Encrypt data when storing sensitive information on the device using object persistence (Required)*
- *Always verify the HMAC/signature before processing the actual information stored in the object (Required)*

6.8.1.5 Property Lists and Codable

You can persist objects to property lists (also called plists in previous sections). You can find two examples below of how to use it:

```
// archiving:
let data = NSKeyedArchiver.archivedDataWithRootObject(customPoint) // Note that␣
↪archivedDataWithRootObject is now deprecated.
NSUserDefaults.standardUserDefaults().setObject(data, forKey: "customPoint")

// unarchiving:
if let data = NSUserDefaults.standardUserDefaults().objectForKey("customPoint") as?
↪ NSData {
    let customPoint = NSKeyedUnarchiver.unarchiveObjectWithData(data) // Note that␣
↪unarchiveObjectWithData is now deprecated.
}
```

In this first example, the NSUserDefaults are used, which is the primary property list. We can do the same with the Codable version:

```
struct CustomPointStruct: Codable {
        var point: Double
        var name: String
    }

    var points: [CustomPointStruct] = [
        CustomPointStruct(point: 1, name: "test"),
        CustomPointStruct(point: 2, name: "test"),
```

(continues on next page)

(continued from previous page)

```
        CustomPointStruct(point: 3, name: "test"),
    ]

    UserDefaults.standard.set(try? PropertyListEncoder().encode(points), forKey:
↪"points")
    if let data = UserDefaults.standard.value(forKey: "points") as? Data {
        let points2 = try? PropertyListDecoder().decode([CustomPointStruct].self,␣
↪from: data)
    }
```

Note that plist files are not meant to store secret information. They are designed to hold user preferences for an app.

Reference

- owasp-mastg Testing Object Persistence (MSTG-PLATFORM-8) Overview Property Lists and Codable

Rulebook

- *How to persist an object to a property list (Required)*

6.8.1.6 XML

There are multiple ways to do XML encoding. Similar to JSON parsing, there are various third party libraries, such as:

- Fuzi
- Ono
- AEXML
- RaptureXML
- SwiftyXMLParser
- SWXMLHash

They vary in terms of speed, memory usage, object persistence and more important: differ in how they handle XML external entities. See XXE in the Apple iOS Office viewer as an example. Therefore, it is key to disable external entity parsing if possible. See the OWASP XXE prevention cheatsheet for more details. Next to the libraries, you can make use of Apple's XMLParser class

When not using third party libraries, but Apple's XMLParser, be sure to let shouldResolveExternalEntities return false.

Reference

- owasp-mastg Testing Object Persistence (MSTG-PLATFORM-8) Overview XML

Rulebook

- *Disable external entity parsing (Recommended)*

6.8.1.7 Object-Relational Mapping (CoreData and Realm)

There are various ORM-like solutions for iOS. The first one is Realm, which comes with its own storage engine. Realm has settings to encrypt the data as explained in Realm's documentation. This allows for handling secure data. Note that the encryption is turned off by default.

Apple itself supplies CoreData, which is well explained in the Apple Developer Documentation. It supports various storage backends as described in Apple's Persistent Store Types and Behaviors documentation. The issue with the storage backends recommended by Apple, is that none of the type of data stores is encrypted, nor checked for integrity. Therefore, additional actions are necessary in case of confidential data. An alternative can be found in project iMas, which does supply out of the box encryption.

Reference

- owasp-mastg Testing Object Persistence (MSTG-PLATFORM-8) Overview Object-Relational Mapping (CoreData and Realm)

Rulebook

- *Realm enforces and leverages data encryption (Recommended)*

6.8.1.8 Protocol Buffers

Protocol Buffers by Google, are a platform- and language-neutral mechanism for serializing structured data by means of the Binary Data Format. They are available for iOS by means of the Protobuf library. There have been a few vulnerabilities with Protocol Buffers, such as CVE-2015-5237. Note that Protocol Buffers do not provide any protection for confidentiality as no built-in encryption is available.

Reference

- owasp-mastg Testing Object Persistence (MSTG-PLATFORM-8) Protocol Buffers

6.8.2 Static Analysis

All different flavors of object persistence share the following concerns:

- If you use object persistence to store sensitive information on the device, then make sure that the data is encrypted: either at the database level, or specifically at the value level.
- Need to guarantee the integrity of the information? Use an HMAC mechanism or sign the information stored. Always verify the HMAC/signature before processing the actual information stored in the objects.
- Make sure that keys used in the two notions above are safely stored in the KeyChain and well protected. See the chapter "Data Storage on iOS" for more details.
- Ensure that the data within the deserialized object is carefully validated before it is actively used (e.g., no exploit of business/application logic is possible).
- Do not use persistence mechanisms that use Runtime Reference to serialize/deserialize objects in high-risk applications, as the attacker might be able to manipulate the steps to execute business logic via this mechanism (see the chapter "iOS Anti-Reversing Defenses" for more details).
- Note that in Swift 2 and beyond, a Mirror can be used to read parts of an object, but cannot be used to write against the object.

Reference

- owasp-mastg Testing Object Persistence (MSTG-PLATFORM-8) Static Analysis

Rulebook

- *Encrypt data when storing sensitive information on the device using object persistence (Required)*
- *Always verify the HMAC/signature before processing the actual information stored in the object (Required)*
- *Risky applications do not use persistence mechanisms that use Runtime References to serialize/deserialize objects (Required)*
- *Carefully validate data in deserialized objects (Required)*

6.8.3 Dynamic Analysis

There are several ways to perform dynamic analysis:

- For the actual persistence: Use the techniques described in the "Data Storage on iOS" chapter.
- For the serialization itself: Use a debug build or use Frida / objection to see how the serialization methods are handled (e.g., whether the application crashes or extra information can be extracted by enriching the objects).

Reference

- owasp-mastg Testing Object Persistence (MSTG-PLATFORM-8) Dynamic Analysis

6.8.4 Rulebook

1. *Use NSSecureCoding for object encoding and decoding (Required)*
2. *In Swift 4, use a combination of Decodable and Encodable protocols (Required)*
3. *Codable/JSONEncoder/JSONDecoder is used for JSON encoding/decoding (Required)*
4. *How to persist an object to a property list (Required)*
5. *Disable external entity parsing (Recommended)*
6. *Realm enforces and leverages data encryption (Recommended)*
7. *Encrypt data when storing sensitive information on the device using object persistence (Required)*
8. *Always verify the HMAC/signature before processing the actual information stored in the object (Required)*
9. *Risky applications do not use persistence mechanisms that use Runtime References to serialize/deserialize objects (Required)*
10. *Carefully validate data in deserialized objects (Required)*

6.8.4.1 Use NSSecureCoding for object encoding and decoding (Required)

The iOS provides two protocols for object encoding and decoding in Objective-C or NSObjects. NSCoding and NSSecureCoding. If the class conforms to one of the protocols, the data is serialized into a byte buffer wrapper NSData . Note that data in Swift is the same as NSData or mutable NSMutableData. The NSCoding protocol declares two methods that must be implemented to encode/decode its instance variables. Classes using NSCoding must either implement NSObject or be annotated as @objc classes.

However, with NSCoding, when decoding an encoded object, you can' t be sure that you saved it in the expected type. Therefore, there is a risk of decoding with an unexpected type. With NSSecureCoding, you can specify the expected type, so you can get the result of decoding only for objects of the expected type.

```
import Foundation

class MyClass : NSObject, NSSecureCoding {

    static var supportsSecureCoding: Bool { return true }

    var x: Double = 0.0
    var name: String = ""

    func encode(with coder: NSCoder) {
        coder.encode(x, forKey: "x")
        coder.encode(name, forKey: "name")
    }

    init(x: Double, name: String) {
            self.x = x
            self.name = name
```

(continues on next page)

(continued from previous page)

```
    }

    required convenience init?(coder: NSCoder) {
        guard let name = coder.decodeObject(forKey: "name") as? String
                else {return nil}
        self.init(x:coder.decodeDouble(forKey:"x"),
                                name:name)
    }

}
```

when init(coder:) is part of a class. Next, when decoding the object, for example, the following checks are performed.

```
let obj = decoder.decodeObject(of:MyClass.self, forKey: "myKey")
```

Conformance to NSSecureCoding ensures that the instantiated object is indeed what was expected. However, no additional integrity checks are performed on the data and the data is not encrypted. Therefore, secret data needs additional encryption, and data whose integrity must be protected needs to get an additional HMAC.

If this is violated, the following may occur.

- There is no guarantee that the instantiated object is really what was expected.

6.8.4.2 In Swift 4, use a combination of Decodable and Encodable protocols (Required)

Swift 4 introduces Codable type aliases: This is a combination of Decodable and Encodable protocols.Strings, Ints, Doubles, Dates, Data and URLs are inherently Codable:This means it can be easily encoded and decoded without additional work. Consider the following example.

```
struct CustomPointStruct:Codable {
    var x: Double
    var name: String
}
```

By adding Codable to the example CustomPointStruct' s inheritance list, the init(from:) and encode(to:) methods are now automatically supported. See Apple Developer Documentation for details on how Codable works. Codable can easily encode/decode into various representations. NSData using NSCoding/NSSecureCoding , JSON , Property Lists, XML, etc. See the section below for details.

* There is nothing that can be described as an event of violation or carelessness.

6.8.4.3 Codable/JSONEncoder/JSONDecoder is used for JSON encoding/decoding (Required)

Apple has added direct support for JSON encoding/decoding by combining Codable with JSONEncoder and JSON-Decoder.

```
struct CustomPointStruct: Codable {
    var point: Double
    var name: String
}

let encoder = JSONEncoder()
encoder.outputFormatting = .prettyPrinted

let test = CustomPointStruct(point: 10, name: "test")
let data = try encoder.encode(test)
let stringData = String(data: data, encoding: .utf8)

// stringData = Optional ({
// "point" : 10,
```

(continues on next page)

(continued from previous page)

```
// "name" : "test"
// })
```

* There is nothing that can be described as an event of violation or carelessness.

6.8.4.4 How to persist an object to a property list (Required)

Objects can be persisted in property lists (also called plists in the previous section). Below are two examples of how to use it.

```
// archiving:
let data = NSKeyedArchiver.archivedDataWithRootObject(customPoint) // Note that␣
↪archivedDataWithRootObject is now deprecated.
NSUserDefaults.standardUserDefaults().setObject(data, forKey: "customPoint")

// unarchiving:
if let data = NSUserDefaults.standardUserDefaults().objectForKey("customPoint") as?
↪ NSData {
    let customPoint = NSKeyedUnarchiver.unarchiveObjectWithData(data) // Note that␣
↪unarchiveObjectWithData is now deprecated.
}
```

This first example uses the main property list, NSUserDefaults. You can do the same with the Codable version.

```
struct CustomPointStruct: Codable {
        var point: Double
        var name: String
    }

    var points: [CustomPointStruct] = [
        CustomPointStruct(point: 1, name: "test"),
        CustomPointStruct(point: 2, name: "test"),
        CustomPointStruct(point: 3, name: "test"),
    ]

    UserDefaults.standard.set(try? PropertyListEncoder().encode(points), forKey:
↪"points")
    if let data = UserDefaults.standard.value(forKey: "points") as? Data {
        let points2 = try? PropertyListDecoder().decode([CustomPointStruct].self,␣
↪from: data)
    }
```

Since archivedDataWithRootObject and unarchiveObjectWithData have been deprecated since iOS 12, archivedDataWithRootObject:requiringSecureCoding: , unarchivedObject:ofClass:from should be used.

```
class MyClass: NSObject, NSSecureCoding {
    static var supportsSecureCoding: Bool = true
    var dataValue:String
    init(value: String){
        self.dataValue = value
    }

    func encode(with aCoder: NSCoder) {
        aCoder.encode(dataValue, forKey: "dataKey")
    }

    required init?(coder aDecoder: NSCoder) {
        self.dataValue = (aDecoder.decodeObject(forKey: "dataKey") as! String)
    }
}
```

(continues on next page)

(continued from previous page)

```
// archiving:
func saveData(_ value : MyClass) {
    guard let archiveData = try? NSKeyedArchiver.archivedData(withRootObject:␣
↪value, requiringSecureCoding: true) else {
        fatalError("Archive failed")
    }
    UserDefaults.standard.set(archiveData, forKey: "key")
}

// unarchiving:
func loadData() -> MyClass? {
    if let loadedData = UserDefaults().data(forKey: "key") {
        return try? NSKeyedUnarchiver.unarchivedObject(ofClass: MyClass.self,␣
↪from: loadedData)
    }
    return nil
}
```

Note that plist files are not meant to store sensitive information. Designed to hold app user settings.

If this is violated, the following may occur.

- Sensitive information can be read from plist files.

6.8.4.5 Disable external entity parsing (Recommended)

It is recommended to disable external entity parsing.See OWASP XXE prevention cheatsheet for details.Alternatively, you can take advantage of Apple's XMLParser class.

```
import UIKit

class ParseSample : NSObject {

    var parser = XMLParser()
    var arrDetail : [String] = []
    var arrFinal : [[String]] = []
    var content : String = ""

    func parseSample() {
        let str = "https://economictimes.indiatimes.com/rssfeedstopstories.cms"
         let url = URL(string: str)
         parser = XMLParser(contentsOf: url!) ?? XMLParser()
         parser.delegate = self
         parser.shouldResolveExternalEntities = false
         parser.parse()
    }
}


extension ParseSample: XMLParserDelegate {
    func parserDidStartDocument(_ parser: XMLParser) {
             arrFinal.removeAll()
    }

    func parser(_ parser: XMLParser, didStartElement elementName: String,␣
↪namespaceURI: String?, qualifiedName qName: String?, attributes attributeDict:␣
↪[String : String] = [:]) {

            if elementName == "item" {
                arrDetail.removeAll()
```

(continues on next page)

(continued from previous page)

```
            }
    }

    func parser(_ parser: XMLParser, didEndElement elementName: String,␣
↪namespaceURI: String?, qualifiedName qName: String?) {

            if elementName == "title" || elementName == "link" {
                arrDetail.append(content)
            }
            else if elementName == "item" {
                arrFinal.append(arrDetail)
            }
    }

    func parser(_ parser: XMLParser, foundCharacters string: String) {
           content = string
    }

    func parserDidEndDocument(_ parser: XMLParser) {
           print(arrFinal)
    }
}
```

If this is not noted, the following may occur.

- Vulnerable to XXE attacks.

6.8.4.6 Realm enforces and leverages data encryption (Recommended)

When leveraging Realm for iOS ORM-like solutions, encryption should be implemented. Realms have settings for encrypting data as described in Realm's documentation. This makes it possible to handle safe data. Encryption is turned off by default.

Example: Realm DB Encryption Method

```
#import <Foundation/Foundation.h>
#import <Realm/Realm.h>

// model
@interface Model1 : RLMObject
@property (strong, nonatomic) NSString *tid;
@property (strong, nonatomic) NSString *name;
@end

@interface MyRealm : RLMObject
- (void)save;
@end


@implementation MyRealm {}

-(void)save {

    // Use an autorelease pool to close the Realm at the end of the block, so
    // that we can try to reopen it with different keys
    @autoreleasepool {
        RLMRealmConfiguration *configuration = [RLMRealmConfiguration␣
↪defaultConfiguration];
        configuration.encryptionKey = [self getKey];
        RLMRealm *realm = [RLMRealm realmWithConfiguration:configuration
                                                    error:nil];
```

(continues on next page)

(continued from previous page)

```
        // Add an object
        [realm beginWriteTransaction];
        Model1 *obj = [[Model1 alloc] init];
        obj.tid = @"1";
        obj.name = @"abcd";
        [realm addObject:obj];
        [realm commitWriteTransaction];
    }
}


- (NSData *)getKey {
    // Identifier for our keychain entry - should be unique for your application
    static const uint8_t kKeychainIdentifier[] = "io.Realm.EncryptionKey";
    NSData *tag = [[NSData alloc] initWithBytesNoCopy:(void *)kKeychainIdentifier
                                               length:sizeof(kKeychainIdentifier)
                                         freeWhenDone:NO];

    // First check in the keychain for an existing key
    NSDictionary *query = @{(__bridge id)kSecClass: (__bridge id)kSecClassKey,
                            (__bridge id)kSecAttrApplicationTag: tag,
                            (__bridge id)kSecAttrKeySizeInBits: @512,
                            (__bridge id)kSecReturnData: @YES};

    CFTypeRef dataRef = NULL;
    OSStatus status = SecItemCopyMatching((__bridge CFDictionaryRef)query, &
↪dataRef);
    if (status == errSecSuccess) {
        return (__bridge NSData *)dataRef;
    }

    // No pre-existing key from this application, so generate a new one
    uint8_t buffer[64];
    status = SecRandomCopyBytes(kSecRandomDefault, 64, buffer);
    NSAssert(status == 0, @"Failed to generate random bytes for key");
    NSData *keyData = [[NSData alloc] initWithBytes:buffer length:sizeof(buffer)];

    // Store the key in the keychain
    query = @{(__bridge id)kSecClass: (__bridge id)kSecClassKey,
              (__bridge id)kSecAttrApplicationTag: tag,
              (__bridge id)kSecAttrKeySizeInBits: @512,
              (__bridge id)kSecValueData: keyData};

    status = SecItemAdd((__bridge CFDictionaryRef)query, NULL);
    NSAssert(status == errSecSuccess, @"Failed to insert new key in the keychain");

    return keyData;
}

@end
```

Podfile:

```
# Uncomment the next line to define a global platform for your project
# platform :ios, '9.0'

target 'OBCHelloworld' do
  # Comment the next line if you don't want to use dynamic frameworks
  use_frameworks!
```

(continues on next page)

(continued from previous page)

```
  # Pods for OBCHelloworld
  pod "Realm"

end
```

Reference

- RealmDB encryption

If this is not noted, the following may occur.

- Data can be read by an attackers or malicious applications.

6.8.4.7 Encrypt data when storing sensitive information on the device using object persistence (Required)

If object persistence is used to store sensitive information on the device, ensure that the data is encrypted. Encryption at the database level, or specifically at the value level.

An example of encrypting a string using a symmetric key:

```
class EncryptionSample {

    func EncryptionSample( rawString:String ) {

        // Symmetric key generation and release:
        let encryptionKey = SymmetricKey(size: .bits256)
        // ...

        // SHA-2 512-bit digest calculation:
        let rawString = "OWASP MTSG"
        let rawData = Data(rawString.utf8)
        let hash = SHA512.hash(data: rawData) // Compute the digest
        let textHash = String(describing: hash)
        print(textHash) // Print hash text
    }
}
```

If this is violated, the following may occur.

- Sensitive information contained in persisted data can be read.

6.8.4.8 Always verify the HMAC/signature before processing the actual information stored in the object (Required)

Do you need to ensure information integrity? Use the HMAC mechanism or sign the stored information. Always verify the HMAC/signature before processing the actual information stored in the object.

Hash conversion sample code using HMAC mechanism.

```
import Foundation
import CryptoSwift

class HMACExample {

    func hexString(secretKey: String, text:String) -> String {
        // String to Hash
        guard let hmac = try? HMAC(key: secretKey, variant: .sha2(.sha256)).
↪authenticate(text.bytes) else {
            return ""
        }
```

(continues on next page)

(continued from previous page)

```
        return hmac.toHexString()

    }
}
```

For how to sign the stored information (signature), see the sample code in the rulebook below.

Rulebook

- *Securely store values using the Keychain Services API (Required)*

If this is violated, the following may occur.

- Changes to sensitive information contained in persisted data may go unnoticed.

6.8.4.9 Risky applications do not use persistence mechanisms that use Runtime References to serialize/deserialize objects (Required)

Serialization/deserialization itself is a convenient function, but you need to be careful when using it. High-risk applications using persistence mechanisms that use Runtime Reference for object serialization/deserialization There is a vulnerability that an unintended object is manipulated when deserializing data given from the outside, causing an illegal operation.

See the "iOS Anti-Reversing Defenses" chapter for details.

* No sample code as it is a conceptual rule.

If this is violated, the following may occur.

- Attackers can exploit this vulnerability to cause significant damage, including remote code execution, privilege escalation, arbitrary file access, and DoS.

6.8.4.10 Carefully validate data in deserialized objects (Required)

Deserializing creates an object based on the bytes received. Values that cannot be changed on the application can be freely manipulated from the outside. For example, if the User class has a field called status and judges whether it is an administrator or a user, it will be possible for the user to deserialize as an administrator. Also, if you have a name field etc. as a character string, it may be possible to insert a character string that causes SQL injection if it is used as it is in a database query.

The best countermeasure is to design so that externally supplied data is not deserialized. Alternatively, deserialize only data that does not affect others.

If you need to deserialize the data from the outside, implement a consistency check on the serialized data using a digital signature or the like.

Correspond by comparing the serialized value and the hash value using the following cryptographically secure hash.

- *Implementation of the iOS encryption algorithm Apple CryptoKit (Recommended)*

If this is violated, the following may occur.

- cause SQL injection.
- Unauthorized escalation of access privileges.

7 Code Quality and Build Setting Requirements

7.1 MSTG-CODE-1

The app is signed and provisioned with a valid certificate, of which the private key is properly protected.

7.1.1 Code Signing

Code signing your app assures users that the app has a known source and hasn't been modified since it was last signed. Before your app can integrate app services, be installed on a non-jailbroken device, or be submitted to the App Store, it must be signed with a certificate issued by Apple. For more information on how to request certificates and code sign your apps, review the App Distribution Guide.

Reference

- owasp-mastg Making Sure that the App Is Properly Signed (MSTG-CODE-1) Overview

Rulebook

- *App code signature method (Required)*

7.1.2 Static Analysis

You have to ensure that the app is using the latest code signature format. You can retrieve the signing certificate information from the application's .app file with codesign. Codesign is used to create, check, and display code signatures, as well as inquire into the dynamic status of signed code in the system.

After you get the application's IPA file, re-save it as a ZIP file and decompress the ZIP file. Navigate to the Payload directory, where the application's .app file will be.

Execute the following codesign command to display the signing information:

```
$ codesign -dvvv YOURAPP.app
Executable=/Users/Documents/YOURAPP/Payload/YOURAPP.app/YOURNAME
Identifier=com.example.example
Format=app bundle with Mach-O universal (armv7 arm64)
CodeDirectory v=20200 size=154808 flags=0x0(none) hashes=4830+5 location=embedded
Hash type=sha256 size=32
CandidateCDHash sha1=455758418a5f6a878bb8fdb709ccfca52c0b5b9e
CandidateCDHash sha256=fd44efd7d03fb03563b90037f92b6ffff3270c46
Hash choices=sha1,sha256
CDHash=fd44efd7d03fb03563b90037f92b6ffff3270c46
Signature size=4678
```

(continues on next page)

(continued from previous page)

```
Authority=iPhone Distribution: Example Ltd
Authority=Apple Worldwide Developer Relations Certification Authority
Authority=Apple Root CA
Signed Time=4 Aug 2017, 12:42:52
Info.plist entries=66
TeamIdentifier=8LAMR92KJ8
Sealed Resources version=2 rules=12 files=1410
Internal requirements count=1 size=176
```

There are various ways to distribute your app as described at the Apple documentation, which include using the App Store or via Apple Business Manager for custom or in-house distribution. In case of an in-house distribution scheme, make sure that no ad hoc certificates are used when the app is signed for distribution.

Reference

- owasp-mastg Making Sure that the App Is Properly Signed (MSTG-CODE-1) Static Analysis

Rulebook

- *It is necessary to confirm that the app uses the latest code signature format (Required)*

7.1.3 Rulebook

1. *App code signature method (Required)*
2. *It is necessary to confirm that the app uses the latest code signature format (Required)*

7.1.3.1 App code signature method (Required)

Follow the steps below to sign the code to the application.

1. Add an Apple ID to your account settings

   Add the user's Apple ID account to the account preferences to identify the user and download information about the user's team. Xcode will use the Apple ID credentials to download information about all teams to which the user belongs.

2. Assign teams to targets within the project

   Assign each target in the user's project to a team. Xcode stores the signing authorizations (certificates, identifiers, and provisioning profiles) in the associated team account. If the user registers as an organization, the program role determines what tasks can be performed in Xcode. If the user registers as an individual, the user is the account holder for a single team. If you are not a member of the Apple Developer Program, Xcode creates a personal team for you.

3. Add functionality to the app

   Under "Signatures and Features" activate the app service you wish to use. Xcode will configure the user's project and update the signature permissions accordingly. If necessary, Xcode will enable the app service for the associated App ID and regenerate the provisioning profile it manages. To fully enable some app services, you may need to sign in to your developer account or App Store Connect.

4. Running the application on the terminal

   The first time you run your app on a connected or wireless device (iOS, tvOS), Xcode creates the necessary development signature permissions for you. Xcode registers the device of your choice and adds it to the provisioning profile it manages. For macOS apps, Xcode registers the Mac running Xcode.

5. Export of signing certificates and provisioning profiles

   The signing certificate and provisioning profile used to launch the application on the device are stored on the user's Mac. Since the private key for the signing certificate is only stored in the keychain, this is a good time to export the signing certificate and provisioning profile for the developer account. You can also export the developer account and move the signing authority to another Mac.

If this is violated, the following may occur.

- Cannot guarantee to the user that the source has not been changed after the app is signed.

7.1.3.2 It is necessary to confirm that the app uses the latest code signature format (Required)

You have to ensure that the app is using the latest code signature format. You can retrieve the signing certificate information from the application's .app file with codesign. Codesign is used to create, check, and display code signatures, as well as inquire into the dynamic status of signed code in the system.

After you get the application's IPA file, re-save it as a ZIP file and decompress the ZIP file. Navigate to the Payload directory, where the application's .app file will be.

Execute the following codesign command to display the signing information:

```
$ codesign -dvvv YOURAPP.app
Executable=/Users/Documents/YOURAPP/Payload/YOURAPP.app/YOURNAME
Identifier=com.example.example
Format=app bundle with Mach-O universal (armv7 arm64)
CodeDirectory v=20200 size=154808 flags=0x0(none) hashes=4830+5 location=embedded
Hash type=sha256 size=32
CandidateCDHash sha1=455758418a5f6a878bb8fdb709ccfca52c0b5b9e
CandidateCDHash sha256=fd44efd7d03fb03563b90037f92b6ffff3270c46
Hash choices=sha1,sha256
CDHash=fd44efd7d03fb03563b90037f92b6ffff3270c46
Signature size=4678
Authority=iPhone Distribution: Example Ltd
Authority=Apple Worldwide Developer Relations Certification Authority
Authority=Apple Root CA
Signed Time=4 Aug 2017, 12:42:52
Info.plist entries=66
TeamIdentifier=8LAMR92KJ8
Sealed Resources version=2 rules=12 files=1410
Internal requirements count=1 size=176
```

If this is violated, the following may occur.

- May not be able to register to the store.

7.2 MSTG-CODE-2

The app has been built in release mode, with settings appropriate for a release build (e.g. non-debuggable).

7.2.1 Build Mode Setting

Debugging iOS applications can be done using Xcode, which embeds a powerful debugger called lldb. Lldb is the default debugger since Xcode5 where it replaced GNU tools like gdb and is fully integrated in the development environment. While debugging is a useful feature when developing an app, it has to be turned off before releasing apps to the App Store or within an enterprise program.

Generating an app in Build or Release mode depends on build settings in Xcode; when an app is generated in Debug mode, a DEBUG flag is inserted in the generated files.

Reference

- owasp-mastg Determining Whether the App is Debuggable (MSTG-CODE-2)

7.2.2 Static Analysis

At first you need to determine the mode in which your app is to be generated to check the flags in the environment:

- Select the build settings of the project
- Under 'Apple LVM - Preprocessing' and 'Preprocessor Macros' , make sure 'DEBUG' or 'DEBUG_MODE' is not selected (Objective-C)
- Or in the 'Swift Compiler - Custom Flags' section / 'Other Swift Flags' , make sure the '-D DEBUG' entry does not exist.
- Open Manage Schemes and make sure the "Debug Executable" option is not selected for the scheme you want to use for release.

Reference

- owasp-mastg Determining Whether the App is Debuggable (MSTG-CODE-2) Static Analysis

Rulebook

- *Build mode setting confirmation (Required)*

7.2.3 Dynamic Analysis

Check whether you can attach a debugger directly, using Xcode. Next, check if you can debug the app on a jailbroken device after Clutching it. This is done using the debug-server which comes from the BigBoss repository at Cydia.

Note: if the application is equipped with anti-reverse engineering controls, then the debugger can be detected and stopped.

Reference

- owasp-mastg Determining Whether the App is Debuggable (MSTG-CODE-2) Dynamic Analysis

7.2.4 Rulebook

1. *Build mode setting confirmation (Required)*

7.2.4.1 Build mode setting confirmation (Required)

Generating an application in build or release mode depends on the build settings in Xcode. Therefore, it is necessary to make sure that the release mode is set correctly in the application for release. If the app is built in debug mode, the DEBUG flag will be inserted so that it can be attached directly to the debugger.

How to check DEBUG flag settings in your environment.

- Select the build settings for the project.
- Make sure "DEBUG" or "DEBUG_MODE" is not selected in "Apple LVM - Preprocessing" and "Preprocessor Macros" . (Objective-C)
- Or, in the "Swift Custom Flags" section / "Other Swift Flags" , make sure there is no "-D DEBUG" entry.
- Open Manage Schemes and make sure the "Debug Executable" option is not selected for the scheme you want to use for release.

* No sample code, as the rules relate to Xcode settings.

If this is violated, the following may occur.

- Attachment to the debugger leaks application information.

7.3 MSTG-CODE-3

Debugging symbols have been removed from native binaries.

7.3.1 Presence or absence of debug symbols

As a good practice, as little explanatory information as possible should be provided with a compiled binary. The presence of additional metadata such as debug symbols might provide valuable information about the code, e.g. function names leaking information about what a function does. This metadata is not required to execute the binary and thus it is safe to discard it for the release build, which can be done by using proper compiler configurations. As a tester you should inspect all binaries delivered with the app and ensure that no debugging symbols are present (at least those revealing any valuable information about the code).

When an iOS application is compiled, the compiler generates a list of debug symbols for each binary file in an app (the main app executable, frameworks, and app extensions). These symbols include class names, global variables, and method and function names which are mapped to specific files and line numbers where they're defined. Debug builds of an app place the debug symbols in a compiled binary by default, while release builds of an app place them in a companion Debug Symbol file (dSYM) to reduce the size of the distributed app.

Reference

- owasp-mastg Finding Debugging Symbols (MSTG-CODE-3) Overview

Rulebook

- *Not output code information in the release build (Required)*

7.3.2 Static Analysis

To verify the existence of debug symbols you can use objdump from binutils or llvm-objdump to inspect all of the app binaries.

In the following snippet we run objdump over TargetApp (the iOS main app executable) to show the typical output of a binary containing debug symbols which are marked with the d (debug) flag. Check the objdump man page for information about various other symbol flag characters.

```
$ objdump --syms TargetApp

0000000100007dc8 l    d  *UND* -[ViewController handleSubmitButton:]
000000010000809c l    d  *UND* -[ViewController touchesBegan:withEvent:]
0000000100008158 l    d  *UND* -[ViewController viewDidLoad]
...
000000010000916c l    d  *UND* _disable_gdb
00000001000091d8 l    d  *UND* _detect_injected_dylds
00000001000092a4 l    d  *UND* _isDebugged
...
```

To prevent the inclusion of debug symbols, set Strip Debug Symbols During Copy to YES via the XCode project's build settings. Stripping debugging symbols will not only reduce the size of the binary but also increase the difficulty of reverse engineering.

Reference

- owasp-mastg Finding Debugging Symbols (MSTG-CODE-3) Static Analysis

7.3.3 Dynamic Analysis

Dynamic analysis is not applicable for finding debugging symbols.

Reference

- owasp-mastg Finding Debugging Symbols (MSTG-CODE-3) Dynamic Analysis

7.3.4 Rulebook

1. *Not output code information in the release build (Required)*

7.3.4.1 Not output code information in the release build (Required)

Compiled binaries should give as little detail as possible. The presence of additional metadata like debug symbols can provide function names that leak important information about the code, such as information about what the function does. This metadata is not needed to run the binary, so it's safe to discard it in release builds. This is possible by using the appropriate compiler settings.

To avoid including debug symbols, set Strip Debug Symbols During Copy to YES from your XCode project's Build Settings. Removing debug symbols not only reduces the size of the binary, but also increases the difficulty of reverse engineering.

* There is no sample code because it is a conceptual rule.

If this is violated, the following may occur.

- It can leak some metadata such as debugging information, line numbers, and descriptive function and method names in your code.

7.4 MSTG-CODE-4

Debugging code and developer assistance code (e.g. test code, backdoors, hidden settings) have been removed. The app does not log verbose errors or debugging messages.

7.4.1 Logging

To speed up verification and get a better understanding of errors, developers often include debugging code, such as verbose logging statements (using NSLog, println, print, dump, and debugPrint) about responses from their APIs and about their application's progress and/or state. Furthermore, there may be debugging code for "management-functionality", which is used by developers to set the application's state or mock responses from an API. Reverse engineers can easily use this information to track what's happening with the application. Therefore, debugging code should be removed from the application's release version.

Reference

- owasp-mastg Finding Debugging Code and Verbose Error Logging (MSTG-CODE-4)

Rulebook

- *Debug code processing (Required)*
- *The debug code must be deleted from the application release version (Required)*

7.4.2 Static Analysis

You can take the following static analysis approach for the logging statements:

1. Import the application's code into Xcode.
2. Search the code for the following printing functions: NSLog, println, print, dump, debugPrint.
3. When you find one of them, determine whether the developers used a wrapping function around the logging function for better mark up of the statements to be logged; if so, add that function to your search.
4. For every result of steps 2 and 3, determine whether macros or debug-state related guards have been set to turn the logging off in the release build. Please note the change in how Objective-C can use preprocessor macros:

```
#ifdef DEBUG
    // Debug-only code
#endif
```

The procedure for enabling this behavior in Swift has changed: you need to either set environment variables in your scheme or set them as custom flags in the target's build settings. Please note that the following functions (which allow you to determine whether the app was built in the Swift 2.1. release-configuration) aren't recommended, as Xcode 8 and Swift 3 don't support these functions:

- _isDebugAssertConfiguration
- _isReleaseAssertConfiguration
- _isFastAssertConfiguration

Depending on the application's setup, there may be more logging functions. For example, when CocoaLumberjack is used, static analysis is a bit different.

For the "debug-management" code (which is built-in): inspect the storyboards to see whether there are any flows and/or view-controllers that provide functionality different from the functionality the application should support. This functionality can be anything from debug views to printed error messages, from custom stub-response configurations to logs written to files on the application's file system or a remote server.

As a developer, incorporating debug statements into your application's debug version should not be a problem as long as you make sure that the debug statements are never present in the application's release version.

7.4.2.1 Objective-C Logging Statement

In Objective-C, developers can use preprocessor macros to filter out debug code:

```
#ifdef DEBUG
    // Debug-only code
#endif
```

7.4.2.2 Swift Logging Statement

In Swift 2 (with Xcode 7), you have to set custom compiler flags for every target, and compiler flags have to start with "-D". So you can use the following annotations when the debug flag DMSTG-DEBUG is set:

```
#if MSTG-DEBUG
    // Debug-only code
#endif
```

In Swift 3 (with Xcode 8), you can set Active Compilation Conditions in Build settings/Swift compiler - Custom flags. Instead of a preprocessor, Swift 3 uses conditional compilation blocks based on the defined conditions:

Reference

- owasp-mastg Finding Debugging Code and Verbose Error Logging (MSTG-CODE-4) Static Analysis

7.4.3 Dynamic Analysis

Dynamic analysis should be executed on both a simulator and a device because developers sometimes use target-based functions (instead of functions based on a release/debug-mode) to execute the debugging code.

1. Run the application on a simulator and check for output in the console during the app's execution.
2. Attach a device to your Mac, run the application on the device via Xcode, and check for output in the console during the app's execution.

For the other "manager-based" debug code: click through the application on both a simulator and a device to see if you can find any functionality that allows an app's profiles to be pre-set, allows the actual server to be selected or allows responses from the API to be selected.

Reference

- owasp-mastg Finding Debugging Code and Verbose Error Logging (MSTG-CODE-4) Dynamic Analysis

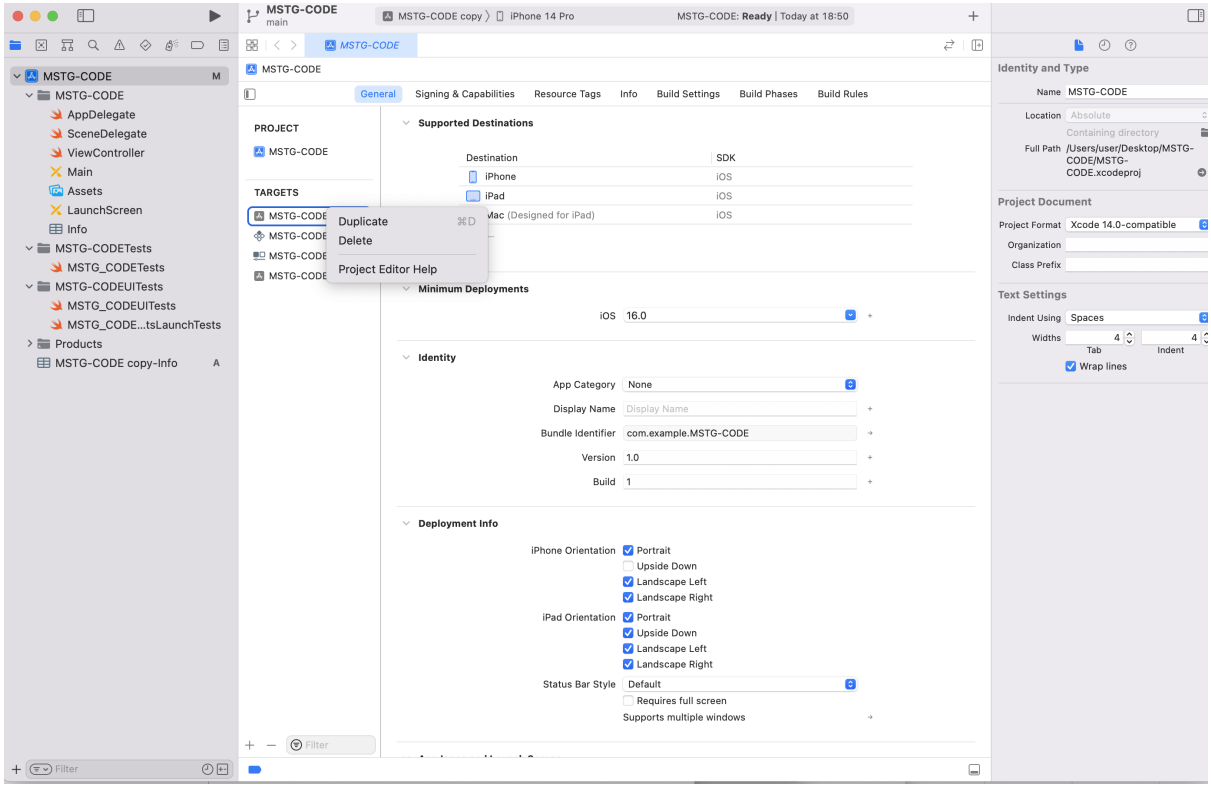
7.4.4 Rulebook

1. *Debug code processing (Required)*
2. *The debug code must be deleted from the application release version (Required)*

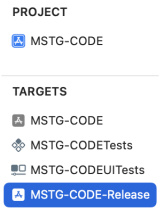
7.4.4.1 Debug code processing (Required)

How to set environment variables in scheme

1. Add build target Right-click on an existing target > select duplicate to create a new copy.

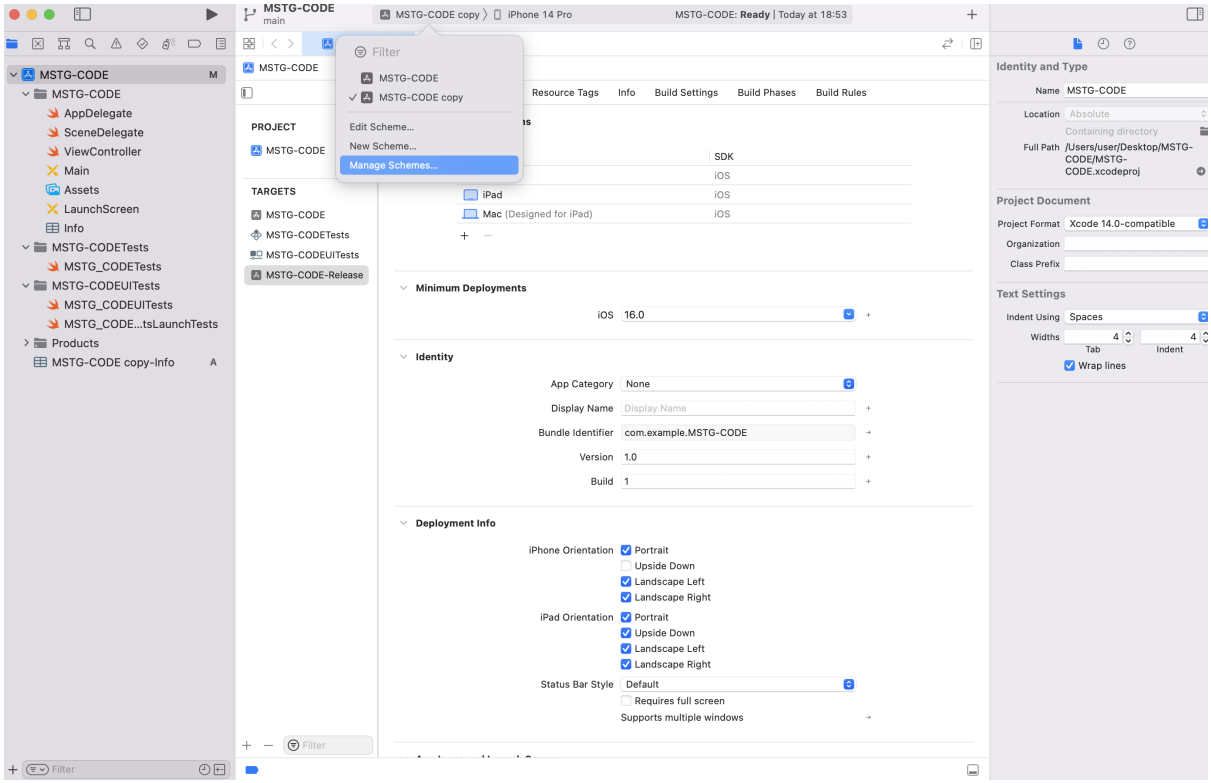


Once created, name it XXXXX-Release.

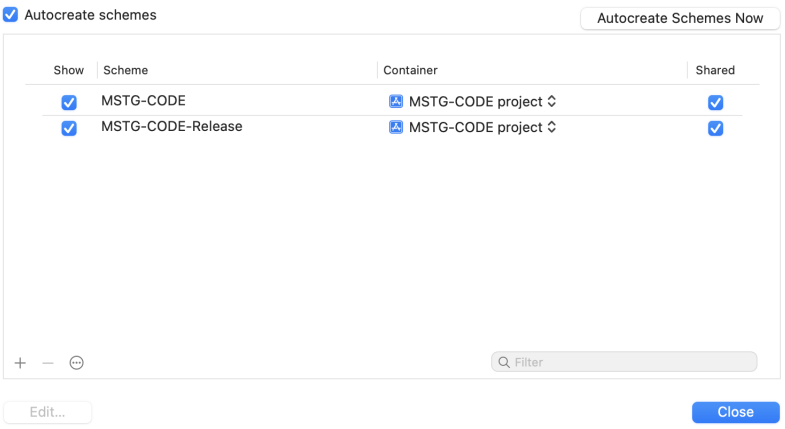


2. Add build scheme Adding a target automatically adds a scheme with the same name.

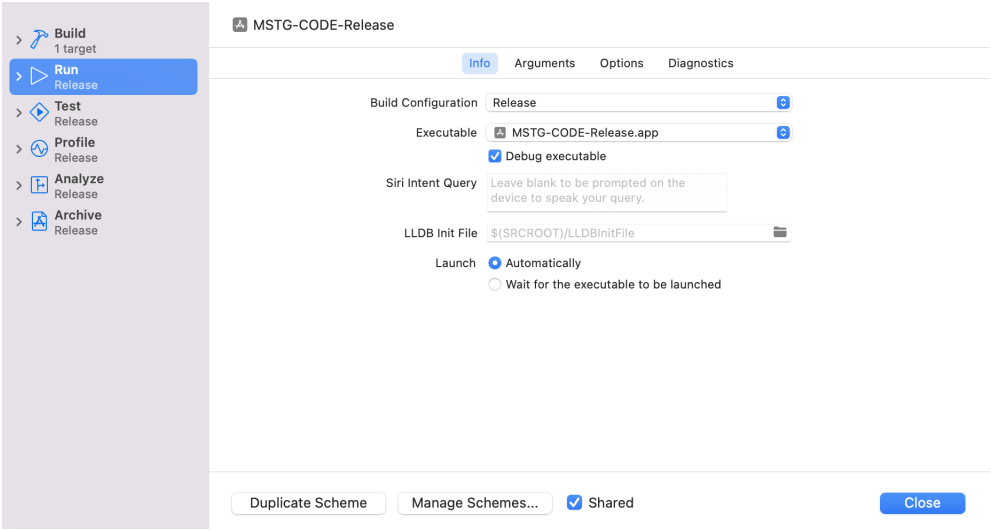
There is a target selection button to the right of the Run stop execution button in Xcode' s top bar. Clicking on it will bring up a list of targets. The name was changed in step 1, but the scheme was unchanged.



To change it, select Manage Schemes··· to display the scheme management screen. Rename the copy to XXXXX-Release, the same as the target.



3. Click Edit Scheme···Click Edit Scheme··· and change Test, Profile, Analyze, and Archive on the left tab to Release.

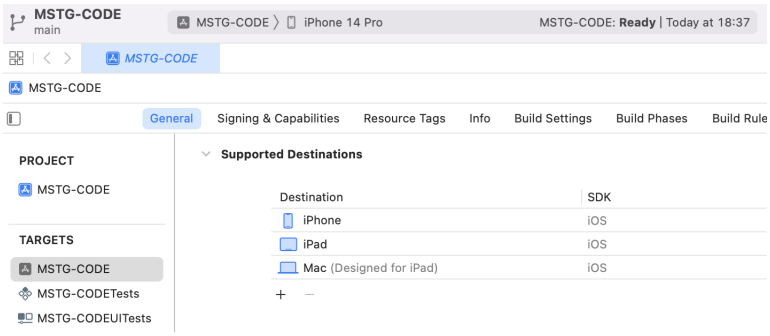


4. Macro description When the scheme is changed, only descriptions where the macro condition is true are automatically activated.

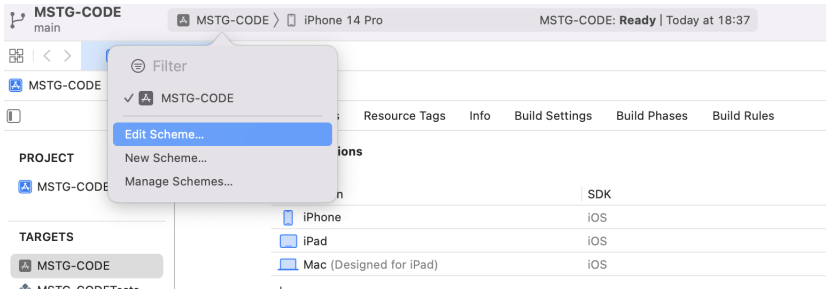
```
public class SampleClass {

    func execute() {
        // Do any additional setup after loading the view, typically from a nib.
#if PRODUCTION
        print("PRODUCTION Runing")
#endif

#if DEBUG
        print("DEBUG Runing")
#endif

#if STAGING
        print("STAGING Runing")
#endif
    }
}
```

How to set as custom compiler flags in target build settings

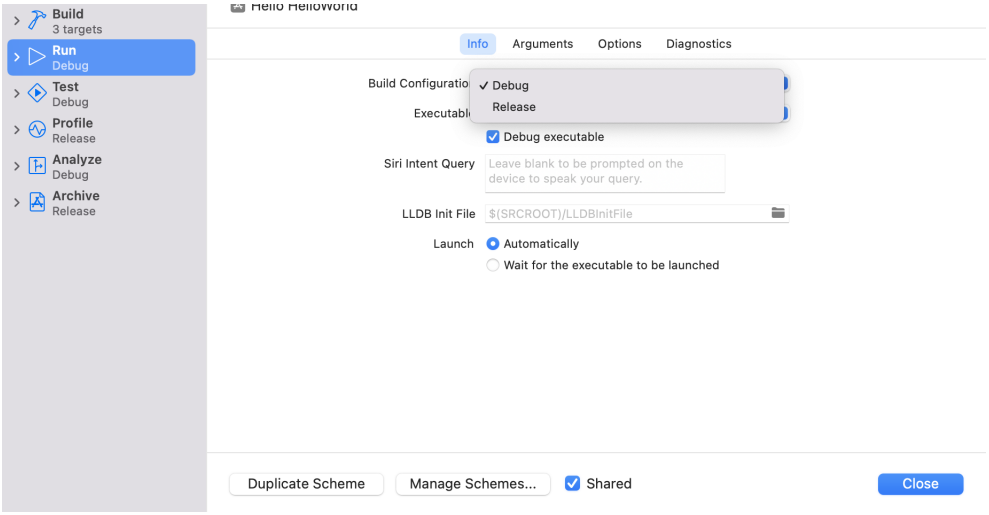
1. Click the project name



2. Click Edit Scheme···



3. Switch between Debug and Release in Build Configuration



How to use the preprocessor DEBUG is given to debug builds by default as a flag when compiling.

```
public class SampleClass {

    func sample() {
#if DEBUG
        // Development Connections
        let server = "api.development.com"
#else
        // Production connections
        let server = "api.com"
#endif

    }

}
```

* There is nothing that can be described as an event of violation or carelessness.

7.4.4.2 The debug code must be deleted from the application release version (Required)

Debug code added for logging should be removed when creating a release version.

If this is violated, the following may occur.

- May cause information leaks through debug code.

7.5 MSTG-CODE-5

All third party components used by the mobile app, such as libraries and frameworks, are identified, and checked for known vulnerabilities.

7.5.1 Third Party Library

iOS applications often make use of third party libraries which accelerate development as the developer has to write less code in order to solve a problem. However, third party libraries may contain vulnerabilities, incompatible licensing, or malicious content. Additionally, it is difficult for organizations and developers to manage application dependencies, including monitoring library releases and applying available security patches.

There are three widely used package management tools Swift Package Manager, Carthage, and CocoaPods:

- The Swift Package Manager is open source, included with the Swift language, integrated into Xcode (since Xcode 11) and supports Swift, Objective-C, Objective-C++, C, and C++ packages. It is written in Swift, decentralized and uses the Package.swift file to document and manage project dependencies.
- Carthage is open source and can be used for Swift and Objective-C packages. It is written in Swift, decentralized and uses the Cartfile file to document and manage project dependencies.
- CocoaPods is open source and can be used for Swift and Objective-C packages. It is written in Ruby, utilizes a centralized package registry for public and private packages and uses the Podfile file to document and manage project dependencies.

There are two categories of libraries:

- Libraries that are not (or should not) be packed within the actual production application, such as OHHTTPStubs used for testing.
- Libraries that are packed within the actual production application, such as Alamofire.

These libraries can lead to unwanted side-effects:

- A library can contain a vulnerability, which will make the application vulnerable. A good example is AFNetworking version 2.5.1, which contained a bug that disabled certificate validation. This vulnerability would allow attackers to execute man-in-the-middle attacks against apps that are using the library to connect to their APIs.
- A library can no longer be maintained or hardly be used, which is why no vulnerabilities are reported and/or fixed. This can lead to having bad and/or vulnerable code in your application through the library.
- A library can use a license, such as LGPL2.1, which requires the application author to provide access to the source code for those who use the application and request insight in its sources. In fact the application should then be allowed to be redistributed with modifications to its source code. This can endanger the intellectual property (IP) of the application.

Please note that this issue can hold on multiple levels: When you use webviews with JavaScript running in the webview, the JavaScript libraries can have these issues as well. The same holds for plugins/libraries for Cordova, React-native and Xamarin apps.

Reference

- owasp-mastg Checking for Weaknesses in Third Party Libraries (MSTG-CODE-5) Overview

Rulebook

- *Pay attention to the use of third party libraries (Recommended)*

7.5.2 Detecting vulnerabilities of third party libraries

In order to ensure that the libraries used by the apps are not carrying vulnerabilities, one can best check the dependencies installed by CocoaPods or Carthage.

Reference

- owasp-mastg Checking for Weaknesses in Third Party Libraries (MSTG-CODE-5) Detecting vulnerabilities of third party libraries

Rulebook

- *How to check the vulnerability of the library (Recommended)*

7.5.2.1 Swift Package Manager

In case Swift Package Manager is used for managing third party dependencies, the following steps can be taken to analyze the third party libraries for vulnerabilities:

First, at the root of the project, where the Package.swift file is located, type

```
swift build
```

Next, check the file Package.resolved for the actual versions used and inspect the given libraries for known vulnerabilities.

You can utilize the OWASP Dependency-Check's experimental Swift Package Manager Analyzer to identify the Common Platform Enumeration (CPE) naming scheme of all dependencies and any corresponding Common Vulnerability and Exposure (CVE) entries. Scan the application's Package.swift file and generate a report of known vulnerable libraries with the following command:

```
dependency-check  --enableExperimental --out . --scan Package.swift
```

Reference

- owasp-mastg Checking for Weaknesses in Third Party Libraries (MSTG-CODE-5) Swift Package Manager

7.5.2.2 CocoaPods

In case CocoaPods is used for managing third party dependencies, the following steps can be taken to analyze the third party libraries for vulnerabilities.

First, at the root of the project, where the Podfile is located, execute the following commands:

```
sudo gem install cocoapods
pod install
```

Next, now that the dependency tree has been built, you can create an overview of the dependencies and their versions by running the following commands:

```
sudo gem install cocoapods-dependencies
pod dependencies
```

The result of the steps above can now be used as input for searching different vulnerability feeds for known vulnerabilities.

Note:

1. If the developer packs all dependencies in terms of its own support library using a .podspec file, then this .podspec file can be checked with the experimental CocoaPods podspec checker.
2. If the project uses CocoaPods in combination with Objective-C, SourceClear can be used.

3. Using CocoaPods with HTTP-based links instead of HTTPS might allow for man-in-the-middle attacks during the download of the dependency, allowing an attacker to replace (parts of) the library with other content. Therefore, always use HTTPS.

You can utilize the OWASP Dependency-Check's experimental CocoaPods Analyzer to identify the Common Platform Enumeration (CPE) naming scheme of all dependencies and any corresponding Common Vulnerability and Exposure (CVE) entries. Scan the application's *.podspec and/or Podfile.lock files and generate a report of known vulnerable libraries with the following command:

```
dependency-check  --enableExperimental --out . --scan Podfile.lock
```

Reference

- owasp-mastg Checking for Weaknesses in Third Party Libraries (MSTG-CODE-5) CocoaPods

7.5.2.3 Carthage

In case Carthage is used for third party dependencies, then the following steps can be taken to analyze the third party libraries for vulnerabilities.

First, at the root of the project, where the Cartfile is located, type

```
brew install carthage
carthage update --platform iOS
```

Next, check the Cartfile.resolved for actual versions used and inspect the given libraries for known vulnerabilities.

Note, at the time of writing this chapter, there is no automated support for Carthage based dependency analysis known to the authors. At least, this feature was already requested for the OWASP DependencyCheck tool but not yet implemented (see the GitHub issue).

Reference

- owasp-mastg Checking for Weaknesses in Third Party Libraries (MSTG-CODE-5) Carthage

7.5.2.4 Discovered library vulnerabilities

When a library is found to contain vulnerabilities, then the following reasoning applies:

- Is the library packaged with the application? Then check whether the library has a version in which the vulnerability is patched. If not, check whether the vulnerability actually affects the application. If that is the case or might be the case in the future, then look for an alternative which provides similar functionality, but without the vulnerabilities.
- Is the library not packaged with the application? See if there is a patched version in which the vulnerability is fixed. If this is not the case, check if the implications of the vulnerability for the build process. Could the vulnerability impede a build or weaken the security of the build-pipeline? Then try looking for an alternative in which the vulnerability is fixed.

In case frameworks are added manually as linked libraries:

1. Open the xcodeproj file and check the project properties.
2. Go to the tab **Build Phases** and check the entries in **Link Binary With Libraries** for any of the libraries. See earlier sections on how to obtain similar information using MobSF.

In the case of copy-pasted sources: search the header files (in case of using Objective-C) and otherwise the Swift files for known method names for known libraries.

Next, note that for hybrid applications, you will have to check the JavaScript dependencies with RetireJS. Similarly for Xamarin, you will have to check the C# dependencies.

Last, if the application is a high-risk application, you will end up vetting the library manually. In that case there are specific requirements for native code, which are similar to the requirements established by the MASVS for the application as a whole. Next to that, it is good to vet whether all best practices for software engineering are applied.

Reference

- owasp-mastg Checking for Weaknesses in Third Party Libraries (MSTG-CODE-5) Carthage

Rulebook

- *Points to confirm the vulnerability of hybrid applications (Required)*

7.5.3 Detecting the Licenses Used by the Libraries of the Application

In order to ensure that the copyright laws are not infringed, one can best check the dependencies installed by Swift Packager Manager, CocoaPods, or Carthage.

Reference

- owasp-mastg Checking for Weaknesses in Third Party Libraries (MSTG-CODE-5) Detecting the Licenses Used by the Libraries of the Application

Rulebook

- *Analysis method of dependencies for applications such as libraries (Required)*

7.5.3.1 Swift Package Manager

When the application sources are available and Swift Package Manager is used, execute the following code in the root directory of the project, where the Package.swift file is located:

```
swift build
```

The sources of each of the dependencies have now been downloaded to /.build/checkouts/ folder in the project. Here you can find the license for each of the libraries in their respective folder.

Reference

- owasp-mastg Checking for Weaknesses in Third Party Libraries (MSTG-CODE-5) Swift Package Manager

7.5.3.2 CocoaPods

When the application sources are available and CocoaPods is used, then execute the following steps to get the different licenses: First, at the root of the project, where the Podfile is located, type

```
sudo gem install cocoapods
pod install
```

This will create a Pods folder where all libraries are installed, each in their own folder. You can now check the licenses for each of the libraries by inspecting the license files in each of the folders.

Reference

- owasp-mastg Checking for Weaknesses in Third Party Libraries (MSTG-CODE-5) CocoaPods

7.5.3.3 Carthage

When the application sources are available and Carthage is used, execute the following code in the root directory of the project, where the Cartfile is located:

```
brew install carthage
carthage update --platform iOS
```

The sources of each of the dependencies have now been downloaded to Carthage/Checkouts folder in the project. Here you can find the license for each of the libraries in their respective folder.

Reference

- owasp-mastg Checking for Weaknesses in Third Party Libraries (MSTG-CODE-5) Carthage

7.5.3.4 Issues with library licenses

When a library contains a license in which the app's IP needs to be open-sourced, check if there is an alternative for the library which can be used to provide similar functionalities.

Note: In case of a hybrid app, please check the build-tools used: most of them do have a license enumeration plugin to find the licenses being used.

Reference

- owasp-mastg Checking for Weaknesses in Third Party Libraries (MSTG-CODE-5) Issues with library licenses

Rulebook

- *Analysis method of dependencies for applications such as libraries (Required)*
- *Verification whether the license is complied (Required)*

7.5.4 Dynamic Analysis

The dynamic analysis of this section comprises of two parts: the actual license verification and checking which libraries are involved in case of missing sources.

It need to be validated whether the copyrights of the licenses have been adhered to. This often means that the application should have an about or EULA section in which the copy-right statements are noted as required by the license of the third party library.

Reference

- owasp-mastg Checking for Weaknesses in Third Party Libraries (MSTG-CODE-5) Dynamic Analysis

Rulebook

- *Verification whether the license is complied (Required)*

7.5.4.1 Listing Application Libraries

When performing app analysis, it is important to also analyze the app dependencies (usually in form of libraries or so-called iOS Frameworks) and ensure that they don't contain any vulnerabilities. Even when you don't have the source code, you can still identify some of the app dependencies using tools like Objection, MobSF or otool. Objection is the recommended tool, since it provides the most accurate results and it is easy to use. It contains a module to work with iOS Bundles, which offers two commands: list_bundles and list_frameworks.

The list_bundles command lists all of the application's bundles that are not related to Frameworks. The output contains executable name, bundle id, version of the library and path to the library.

```
...itudehacks.DVIAswiftv2.develop on (iPhone: 13.2.3) [usb] # ios bundles list_
↪bundles
Executable    Bundle                                        Version  Path
------------  ----------------------------------------  ---------  ---------------
↪-----------------------------
DVIA-v2       com.highaltitudehacks.DVIAswiftv2.develop          2  ...-1F0C-4DB1-
↪8C39-04ACBFFEE7C8/DVIA-v2.app
CoreGlyphs    com.apple.CoreGlyphs                               1  ...m/Library/
↪CoreServices/CoreGlyphs.bundle
```

The list_frameworks command lists all of the application's bundles that represent Frameworks.

```
...itudehacks.DVIAswiftv2.develop on (iPhone: 13.2.3) [usb] # ios bundles list_
↪frameworks
Executable      Bundle                                      Version    Path
--------------  ------------------------------------------  ---------  -------------
↪-------------------------------
Bolts           org.cocoapods.Bolts                         1.9.0      ...8/DVIA-v2.
↪app/Frameworks/Bolts.framework
RealmSwift      org.cocoapods.RealmSwift                    4.1.1      ...A-v2.app/
↪Frameworks/RealmSwift.framework
                                                                       ...ystem/
↪Library/Frameworks/IOKit.framework
...
```

Reference

- owasp-mastg Checking for Weaknesses in Third Party Libraries (MSTG-CODE-5) Dynamic Analysis Listing Application Libraries

Rulebook

- *Analysis method of dependencies for applications such as libraries (Required)*

7.5.5 Rulebook

1. *How to check the vulnerability of the library (Recommended)*
2. *Analysis method of dependencies for applications such as libraries (Required)*
3. *Verification whether the license is complied (Required)*
4. *Points to confirm the vulnerability of hybrid applications (Required)*
5. *Pay attention to the use of third party libraries (Recommended)*

7.5.5.1 How to check the vulnerability of the library (Recommended)

Checking the dependencies installed by CocoaPods or Carthage is the best way to make sure the libraries your app uses are not vulnerable.

If this is not noted, the following may occur.

- Apps can be vulnerable.

7.5.5.2 Analysis method of dependencies for applications such as libraries (Required)

Need to make sure you are not infringing copyright. If source code is available, it is best to check the dependencies installed by Swift Package Manager , CocoaPods and Carthage.

Swift Package Manager

When the application sources are available and Swift Package Manager is used, execute the following code in the root directory of the project, where the Package.swift file is located:

```
swift build
```

The sources of each of the dependencies have now been downloaded to /.build/checkouts/ folder in the project. Here you can find the license for each of the libraries in their respective folder.

CocoaPods

When the application sources are available and CocoaPods is used, then execute the following steps to get the different licenses: First, at the root of the project, where the Podfile is located, type

```
sudo gem install cocoapods
pod install
```

This will create a Pods folder where all libraries are installed, each in their own folder. You can now check the licenses for each of the libraries by inspecting the license files in each of the folders.

Carthage

When the application sources are available and Carthage is used, execute the following code in the root directory of the project, where the Cartfile is located:

```
brew install carthage
carthage update --platform iOS
```

The sources of each of the dependencies have now been downloaded to Carthage/Checkouts folder in the project. Here you can find the license for each of the libraries in their respective folder.

Even if you don't have the source code, you can use tools to identify some of your app's dependencies.

Objection

Objection is a Frida-powered run-time mobile exploration toolkit, built to help assess the security posture of mobile applications without the need for jailbreaking, with the following features:

- Support both iOS and Android.
- Inspect and manipulate container filesystems.
- Bypass SSL pinning.
- Dump keychain.
- Perform memory-related tasks such as dumps and patching.
- Explore and manipulate objects on the heap.

MobSF (Mobile-Security-Framework-MobSF)

MobSF is an all-in-one framework that automates Android/iOS pentesting, malware analysis, and security assessment, and can perform static and dynamic analysis. MobSF supports mobile app binaries (APK, XAPK, IPA, APPX) and source code in zip format.

otool

otool is a tool for viewing specified parts of object files and libraries. It understands both Mach-O files and the Universal File Format.

If this is violated, the following may occur.

- The library may contain licenses that require the app's intellectual property (IP) to be open sourced.

7.5.5.3 Verification whether the license is complied (Required)

A common feature of major licenses is that when distributing derived software, the "copyright", "license statement", and "disclaimer" of the source OSS must be displayed. Since it is **May vary for each license** to specifically display "what", "where" and "how", it is necessary to carefully check the individual license for details.

For example, the MIT License requires that a "copyright notice" and "license statement" be "in all copies or substantial portions of the software" by the following statement:

```
The above copyright notice and this permission notice shall be included in all␣
↪copies or substantial portions of the Software.
```

If this is violated, the following may occur.

- The library may contain licenses that require the app's intellectual property (IP) to be open sourced.

7.5.5.4 Points to confirm the vulnerability of hybrid applications (Required)

Note that for hybrid applications, RetireJS should check the JavaScript dependencies.

RetireJS simply visits a website and checks for known vulnerabilities in the JavaScript libraries used by that website. What Retire.js can do is "detection of known vulnerabilities in libraries", and it cannot find vulnerabilities that have not yet been announced.

If this is violated, the following may occur.

- May not detect known vulnerabilities in JavaScript libraries.

7.5.5.5 Pay attention to the use of third party libraries (Recommended)

Third-party libraries have the following shortcomings and should be examined when used.

- Vulnerabilities in libraries. Be aware that using a library that contains vulnerabilities may lead to the inclusion of malicious or vulnerable code in your application through the library. Even if vulnerabilities have not been discovered at this time, there is a possibility that they will be discovered in the future. In that case, update to a version that addresses the vulnerability, or refrain from using it if there is no updated version.
- License included in the library. Please note that some libraries have a license that requires the deployment of the source code of the application used when using the library.

Note that this problem can occur on multiple levels. If you use javascript in your webview, your javascript library may also have this problem. Plugins/libraries for Cordova, React-native and Xamarin apps are similar.

* There is no sample code due to the rules regarding the use of third-party libraries.

If this is not noted, the following may occur.

- The application contains malicious or vulnerable code that can be exploited.
- Licenses included in third-party libraries may require deployment of the app's source code.

7.6 MSTG-CODE-6

The app catches and handles possible exceptions.

7.6.1 Exception Handling

Exceptions often occur after an application enters an abnormal or erroneous state. Testing exception handling is about making sure that the application will handle the exception and get into a safe state without exposing any sensitive information via its logging mechanisms or the UI.

Bear in mind that exception handling in Objective-C is quite different from exception handling in Swift. Bridging the two approaches in an application that is written in both legacy Objective-C code and Swift code can be problematic.

Reference

- owasp-mastg Testing Exception Handling (MSTG-CODE-6) Overview

Rulebook

- *Exception treatment test purpose (Recommended)*

7.6.2 Exception handling in Objective-C

Objective-C has two types of errors:

Reference

- owasp-mastg Testing Exception Handling (MSTG-CODE-6) Exception handling in Objective-C

7.6.2.1 NSException

NSException is used to handle programming and low-level errors (e.g., division by 0 and out-of-bounds array access). An NSException can either be raised by raise or thrown with @throw. Unless caught, this exception will invoke the unhandled exception handler, with which you can log the statement (logging will halt the program). @catch allows you to recover from the exception if you're using a @try-@catch-block:

```
 @try {
    //do work here
 }

@catch (NSException *e) {
    //recover from exception
}

@finally {
    //cleanup
```

Bear in mind that using NSException comes with memory management pitfalls: you need to clean up allocations from the try block that are in the finally block. Note that you can promote NSException objects to NSError by instantiating an NSError in the @catch block.

7.6.2.2 NSError

NSError is used for all other types of errors. Some Cocoa framework APIs provide errors as objects in their failure callback in case something goes wrong; those that don't provide them pass a pointer to an NSError object by reference. It is a good practice to provide a BOOL return type to the method that takes a pointer to an NSError object to indicate success or failure. If there's a return type, make sure to return nil for errors. If NO or nil is returned, it allows you to inspect the error/reason for failure.

7.6.3 Exception Handling in Swift

Exception handing in Swift (2 - 5) is quite different. The try-catch block is not there to handle NSException. The block is used to handle errors that conform to the Error (Swift 3) or ErrorType (Swift 2) protocol. This can be challenging when Objective-C and Swift code are combined in an application. Therefore, NSError is preferable to NSException for programs written in both languages. Furthermore, error-handling is opt-in in Objective-C, but throws must be explicitly handled in Swift. To convert error-throwing, look at the Apple documentation. Methods that can throw errors use the throws keyword. The Result type represents a success or failure, see Result, How to use Result in Swift 5 and The power of Result types in Swift. There are four ways to handle errors in Swift:

Reference

- owasp-mastg Testing Exception Handling (MSTG-CODE-6) Exception Handling in Swift

Rulebook

- *Precautions when using NSException (Required)*
- *Exception processing (NSERROR) Precautions (Required)*
- *Precautions for Swift exception processing (Recommended)*

7.6.3.1 Exception handling in do-catch

- Propagate the error from a function to the code that calls that function. In this situation, there's no do-catch; there's only a throw throwing the actual error or a try to execute the method that throws. The method containing the try also requires the throws keyword:

```
func dosomething(argumentx:TypeX) throws {
    try functionThatThrows(argumentx: argumentx)
}
```

- Handle the error with a do-catch statement. You can use the following pattern:

```
func doTryExample() {
    do {
        try functionThatThrows(number: 203)
    } catch NumberError.lessThanZero {
        // Handle number is less than zero
    } catch let NumberError.tooLarge(delta) {
        // Handle number is too large (with delta value)
    } catch {
        // Handle any other errors
    }
}

enum NumberError: Error {
    case lessThanZero
    case tooLarge(Int)
    case tooSmall(Int)
}

func functionThatThrows(number: Int) throws -> Bool {
    if number < 0 {
        throw NumberError.lessThanZero
    } else if number < 10 {
        throw NumberError.tooSmall(10 - number)
    } else if number > 100 {
        throw NumberError.tooLarge(100 - number)
    } else {
        return true
    }
}
```

- Handle the error as an optional value:

```
    let x = try? functionThatThrows()
    // In this case the value of x is nil in case of an error.
```

- Use the try! expression to assert that the error won't occur.

7.6.3.2 Exception handling on Result type

- Handle the generic error as a Result return:

```
enum ErrorType: Error {
    case typeOne
    case typeTwo
}

func functionWithResult(param: String?) -> Result<String, ErrorType> {
    guard let value = param else {
        return .failure(.typeOne)
    }
```

(continues on next page)

(continued from previous page)

```
    return .success(value)
}

func callResultFunction() {
    let result = functionWithResult(param: "OWASP")

    switch result {
    case let .success(value):
        // Handle success
    case let .failure(error):
        // Handle failure (with error)
    }
}
```

- Handle network and JSON decoding errors with a Result type:

```
struct MSTG: Codable {
    var root: String
    var plugins: [String]
    var structure: MSTGStructure
    var title: String
    var language: String
    var description: String
}

struct MSTGStructure: Codable {
    var readme: String
}

enum RequestError: Error {
    case requestError(Error)
    case noData
    case jsonError
}

func getMSTGInfo() {
    guard let url = URL(string: "https://raw.githubusercontent.com/OWASP/owasp-
↪mastg/master/book.json") else {
        return
    }

    request(url: url) { result in
        switch result {
        case let .success(data):
            // Handle success with MSTG data
            let mstgTitle = data.title
            let mstgDescription = data.description
        case let .failure(error):
            // Handle failure
            switch error {
            case let .requestError(error):
                // Handle request error (with error)
            case .noData:
                // Handle no data received in response
            case .jsonError:
                // Handle error parsing JSON
            }
        }
    }
}
```

(continues on next page)

(continued from previous page)

```
func request(url: URL, completion: @escaping (Result<MSTG, RequestError>) -> Void)
↪{
    let task = URLSession.shared.dataTask(with: url) { data, _, error in
        if let error = error {
            return completion(.failure(.requestError(error)))
        } else {
            if let data = data {
                let decoder = JSONDecoder()
                guard let response = try? decoder.decode(MSTG.self, from: data)␣
↪else {
                    return completion(.failure(.jsonError))
                }
                return completion(.success(response))
            }
        }
    }
    task.resume()
}
```

7.6.4 Static Analysis

Review the source code to understand how the application handles various types of errors (IPC communications, remote services invocation, etc.). The following sections list examples of what you should check for each language at this stage.

Reference

- owasp-mastg Testing Exception Handling (MSTG-CODE-6) Static Analysis

7.6.4.1 Static Analysis in Objective-C

Make sure that

- the application uses a well-designed and unified scheme to handle exceptions and errors,
- the Cocoa framework exceptions are handled correctly,
- the allocated memory in the @try blocks is released in the @finally blocks,
- for every @throw, the calling method has a proper @catch at the level of either the calling method or the NSApplication/UIApplication objects to clean up sensitive information and possibly recover,
- the application doesn't expose sensitive information while handling errors in its UI or in its log statements, and the statements are verbose enough to explain the issue to the user,
- high-risk applications' confidential information, such as keying material and authentication information, is always wiped during the execution of @finally blocks,
- raise is rarely used (it's used when the program must be terminated without further warning),
- NSError objects don't contain data that might leak sensitive information.

Reference

- owasp-mastg Testing Exception Handling (MSTG-CODE-6) Static Analysis in Objective-C

Rulebook

- *Proper implementation and confirmation of exception/error processing (Required)*

7.6.4.2 Static Analysis in Swift

Make sure that

- the application uses a well-designed and unified scheme to handle errors,
- the application doesn't expose sensitive information while handling errors in its UI or in its log statements, and the statements are verbose enough to explain the issue to the user,
- high-risk applications' confidential information, such as keying material and authentication information, is always wiped during the execution of defer blocks,
- try! is used only with proper guarding up front (to programmatically verify that the method that's called with try! can't throw an error).

Reference

- owasp-mastg Testing Exception Handling (MSTG-CODE-6) Static Analysis in Swift

Rulebook

- *Proper implementation and confirmation of exception/error processing (Required)*

7.6.4.3 Proper Error Handling

Developers can implement proper error handling in several ways:

- Make sure that the application uses a well-designed and unified scheme to handle errors.
- Make sure that all logging is removed or guarded as described in the test case "Testing for Debugging Code and Verbose Error Logging".
- For a high-risk application written in Objective-C: create an exception handler that removes secrets that shouldn't be easily retrievable. The handler can be set via NSSetUncaughtExceptionHandler.
- Refrain from using try! in Swift unless you're certain that there's no error in the throwing method that's being called.
- Make sure that the Swift error doesn't propagate into too many intermediate methods.

Reference

- owasp-mastg Testing Exception Handling (MSTG-CODE-6) Proper Error Handling

Rulebook

- *Proper implementation and confirmation of exception/error processing (Required)*

7.6.5 Dynamic Analysis

There are several dynamic analysis methods:

- Enter unexpected values in the iOS application's UI fields.
- Test the custom URL schemes, pasteboard, and other inter-app communication controls by providing unexpected or exception-raising values.
- Tamper with the network communication and/or the files stored by the application.
- For Objective-C, you can use Cycript to hook into methods and provide them arguments that may cause the callee to throw an exception.

In most cases, the application should not crash. Instead, it should

- recover from the error or enter a state from which it can inform the user that it can't continue,
- provide a message (which shouldn't leak sensitive information) to get the user to take appropriate action,
- withhold information from the application's logging mechanisms.

Reference

- owasp-mastg Testing Exception Handling (MSTG-CODE-6) Dynamic Testing

7.6.6 Rulebook

1. *Exception treatment test purpose (Recommended)*
2. *Precautions when using NSException (Required)*
3. *Exception processing (NSERROR) Precautions (Required)*
4. *Precautions for Swift exception processing (Recommended)*
5. *Proper implementation and confirmation of exception/error processing (Required)*

7.6.6.1 Exception treatment test purpose (Recommended)

The purpose of testing exception handling is to transition to a safe state without exposing sensitive information through logging mechanisms or UI.

If this is not noted, the following may occur.

- Application crashes.
- Confidential information is leaked.

7.6.6.2 Precautions when using NSException (Required)

NSException is used to handle programming and low-level errors (eg division by 0, out-of-bounds array accesses, etc.).

If an exception occurs during the process described in try , the process immediately before the exception occurs is executed, and then the process described in catch is executed. Finally, the process described in finally is executed. Note that finally is always executed last regardless of whether try or catch is executed.

On the other hand, if no exception occurs, all the processing described in try is executed, and then finally processing is executed. The processing inside catch is not executed.

The way to intentionally generate an exception in try-catch is to execute the following code at the timing when you want to generate an exception.

```
#import <Foundation/Foundation.h>

@interface MyException : NSObject {}
- (void)execute;
@end


@implementation MyException {}

-(void)execute {
    // Process that deliberately causes an exception
    [[NSException exceptionWithName:@"Exception" reason:@"reason" userInfo:nil]␣
↪raise];
}

@end
```

* There is nothing that can be described as an event of violation or carelessness.

7.6.6.3 Exception processing (NSERROR) Precautions (Required)

NSError is used for all other types of Error. Some Cocoa framework APIs provide errors as objects in failure callbacks in case something goes wrong. Those that don't provide them pass a pointer to an NSError object by reference. It is good practice to provide methods that take a pointer to an NSError object with a BOOL return type that indicates success or failure. If you have a return type, you should always return nil on error. If NO or nil is returned, you can investigate the reason for the error/failure.

If you use it without entering an initial value, EXC_BAD_ACCESS will occur when you refer to NSError because the reference destination is an appropriate value.

This can be avoided by initializing with nil specified when generating NSError.

```
#import <Foundation/Foundation.h>

@interface MyErrorUse : NSObject {}
- (void)execute;
@end


@implementation MyErrorUse {}

-(void)execute {
    NSString *path = @"file/pass/";

    // Initializing NSError
    NSError *error = nil;
    [[NSFileManager defaultManager] removeItemAtPath: path error: &error];

    if (error != nil) {
       // error handling
    }

}

@end
```

The argument error passes a reference pointer of type NSError (see below). By writing &error when passing, if there is an error, this reference will be updated inside the method and the entity of NSError will be assigned.

In other words, if an error occurs while deleting a file, it can be handled by checking the content of this error. Conversely, when error handling is not to be performed, nil can be passed to error:.

* There is nothing that can be described as an event of violation or carelessness.

7.6.6.4 Precautions for Swift exception processing (Recommended)

Exception handling in Swift (2 - 5) uses try-catch blocks.

This block is used to handle errors conforming to the Error (Swift 3) or ErrorType (Swift 2) protocols and does not exist to handle NSExceptions.

NSError is recommended over NSException in application programs that combine Objective-C and Swift code. Additionally, error handling is opt-in in Objective-C, but throw must be handled explicitly in Swift.

If this is not noted, the following may occur.

- Exceptions may not be handled properly.

7.6.6.5 Proper implementation and confirmation of exception/error processing (Required)

When implementing exception/error handling in iOS, it is necessary to check the following.

Points to check in Objective-C

- The application uses a well-designed and uniform scheme for handling exceptions and errors.
- Cocoa framework exceptions are handled correctly.
- Memory allocated with try blocks is freed with @finally blocks.
- For every @throw there is a proper @catch either at the level of the calling method or the NSApplication/UIApplication object to clean up and possibly recover sensitive information.
- Applications do not expose sensitive information when handling errors in UI or log statements, and statements are verbose to explain the problem to the user.
- Sensitive information for high-risk applications, such as keying material and authentication information, is always removed during @finally blocks.
- raise is rarely used. (used when the program must exit without further warning)
- NSError objects do not contain sensitive data.

Points to check in Swift

- The application is well designed and handles errors with a uniform scheme.
- Applications do not expose sensitive information during error handling in UI or log statements, and statements are verbose to explain the problem to the user.
- High-risk application sensitive information such as keying material and credentials are always wiped during defer blocks.
- try! should be used after proper guarding (programmatically confirming that the method called by try! does not throw an error).

How to implement proper error handling

- Make sure your application uses a well-designed and uniform scheme for handling errors.
- Make sure all logs are removed or protected as described in the test case "Verification with debug code and redundant error logs".
- For high-risk applications written in Objective-C: You should write exception handlers that remove secrets that shouldn't be easily retrievable. Handlers can be set with NSSetUncaughtExceptionHandler .
- Avoid using try in Swift unless you are sure that the throwing method being called is error-free.
- Make sure Swift errors don't propagate to too many intermediate methods.

How to handle errors with try-catch

```
#import <Foundation/Foundation.h>

@interface MyErrorUse : NSObject {}
- (void)tryCatchExample;
@end


@implementation MyErrorUse {}

- (void)tryCatchExample {
    // what you want to do
    @try {
        // raise an exception
        [[NSException exceptionWithName:@"Exception" reason:@"TestCode"␣
↪userInfo:nil] raise];
    } @catch (NSException *exception) {
```

(continues on next page)

(continued from previous page)

```
        // action to be taken when an exception occurs
        NSLog(@"name:%@",exception.name);
        NSLog(@"reason:%@",exception.reason);

    } @finally {
        // Process to be executed when the try-catch process is finished
        NSLog(@"*** finally ***");
    }
}

@end
```

How to handle errors with do-catch

```
func doTryExample() {
    do {
        try functionThatThrows(number: 203)
    } catch NumberError.lessThanZero {
        // Handle number is less than zero
    } catch let NumberError.tooLarge(delta) {
        // Handle number is too large (with delta value)
    } catch {
        // Handle any other errors
    }
}

enum NumberError: Error {
    case lessThanZero
    case tooLarge(Int)
    case tooSmall(Int)
}

func functionThatThrows(number: Int) throws -> Bool {
    if number < 0 {
        throw NumberError.lessThanZero
    } else if number < 10 {
        throw NumberError.tooSmall(10 - number)
    } else if number > 100 {
        throw NumberError.tooLarge(100 - number)
    } else {
        return true
    }
}
```

Violation of this may result in:

- Application crashes.
- Confidential information is leaked.

7.7 MSTG-CODE-7

Error handling logic in security controls denies access by default.

* Since OWASP does not describe the problematic event (MSTG-CODE-7) on the iOS side, the description in this chapter is omitted.

7.8 MSTG-CODE-8

In unmanaged code, memory is allocated, freed and used securely.

7.8.1 Memory Corruption Bugs

Memory corruption bugs are a popular mainstay with hackers. This class of bug results from a programming error that causes the program to access an unintended memory location. Under the right conditions, attackers can capitalize on this behavior to hijack the execution flow of the vulnerable program and execute arbitrary code. This kind of vulnerability occurs in a number of ways:

The primary goal in exploiting memory corruption is usually to redirect program flow into a location where the attacker has placed assembled machine instructions referred to as shellcode. On iOS, the data execution prevention feature (as the name implies) prevents execution from memory defined as data segments. To bypass this protection, attackers leverage return-oriented programming (ROP). This process involves chaining together small, pre-existing code chunks (“gadgets”) in the text segment where these gadgets may execute a function useful to the attacker or, call mprotect to change memory protection settings for the location where the attacker stored the shellcode.

Android apps are, for the most part, implemented in Java which is inherently safe from memory corruption issues by design. However, native apps utilizing JNI libraries are susceptible to this kind of bug. Similarly, iOS apps can wrap C/C++ calls in Obj-C or Swift, making them susceptible to these kind of attacks.

Reference

- owasp-mastg Memory Corruption Bugs (MSTG-CODE-8)

7.8.1.1 Buffer overflows

This describes a programming error where an app writes beyond an allocated memory range for a particular operation. An attacker can use this flaw to overwrite important control data located in adjacent memory, such as function pointers. Buffer overflows were formerly the most common type of memory corruption flaw, but have become less prevalent over the years due to a number of factors. Notably, awareness among developers of the risks in using unsafe C library functions is now a common best practice plus, catching buffer overflow bugs is relatively simple. However, it is still worth testing for such defects.

Reference

- owasp-mastg Memory Corruption Bugs (MSTG-CODE-8)

7.8.1.2 Out-of-bounds-access

Buggy pointer arithmetic may cause a pointer or index to reference a position beyond the bounds of the intended memory structure (e.g. buffer or list). When an app attempts to write to an out-of-bounds address, a crash or unintended behavior occurs. If the attacker can control the target offset and manipulate the content written to some extent, code execution exploit is likely possible.

Reference

- owasp-mastg Memory Corruption Bugs (MSTG-CODE-8)

7.8.1.3 Dangling pointers

These occur when an object with an incoming reference to a memory location is deleted or deallocated, but the object pointer is not reset. If the program later uses the dangling pointer to call a virtual function of the already deallocated object, it is possible to hijack execution by overwriting the original vtable pointer. Alternatively, it is possible to read or write object variables or other memory structures referenced by a dangling pointer.

Reference

- owasp-mastg Memory Corruption Bugs (MSTG-CODE-8)

7.8.1.4 Use-after-free

This refers to a special case of dangling pointers referencing released (deallocated) memory. After a memory address is cleared, all pointers referencing the location become invalid, causing the memory manager to return the address to a pool of available memory. When this memory location is eventually re-allocated, accessing the original pointer will read or write the data contained in the newly allocated memory. This usually leads to data corruption and undefined behavior, but crafty attackers can set up the appropriate memory locations to leverage control of the instruction pointer.

Reference

- owasp-mastg Memory Corruption Bugs (MSTG-CODE-8)

7.8.1.5 Integer overflows

When the result of an arithmetic operation exceeds the maximum value for the integer type defined by the programmer, this results in the value "wrapping around" the maximum integer value, inevitably resulting in a small value being stored. Conversely, when the result of an arithmetic operation is smaller than the minimum value of the integer type, an integer underflow occurs where the result is larger than expected. Whether a particular integer overflow/underflow bug is exploitable depends on how the integer is used. For example, if the integer type were to represent the length of a buffer, this could create a buffer overflow vulnerability.

Reference

- owasp-mastg Memory Corruption Bugs (MSTG-CODE-8)

7.8.1.6 Format string vulnerabilities

When unchecked user input is passed to the format string parameter of the printf family of C functions, attackers may inject format tokens such as '%c' and '%n' to access memory. Format string bugs are convenient to exploit due to their flexibility. Should a program output the result of the string formatting operation, the attacker can read and write to memory arbitrarily, thus bypassing protection features such as ASLR.

Reference

- owasp-mastg Memory Corruption Bugs (MSTG-CODE-8)

7.8.2 Buffer and Integer Overflows

The following code snippet shows a simple example for a condition resulting in a buffer overflow vulnerability.

```
void copyData(char *userId) {
   char  smallBuffer[10]; // size of 10
   strcpy(smallBuffer, userId);
}
```

To identify potential buffer overflows, look for uses of unsafe string functions (strcpy, strcat, other functions beginning with the "str" prefix, etc.) and potentially vulnerable programming constructs, such as copying user input into a limited-size buffer. The following should be considered red flags for unsafe string functions:

- strcat

- strcpy
- strncat
- strlcat
- strncpy
- strlcpy
- sprintf
- snprintf
- gets

Also, look for instances of copy operations implemented as "for" or "while" loops and verify length checks are performed correctly.

Verify that the following best practices have been followed:

- When using integer variables for array indexing, buffer length calculations, or any other security-critical operation, verify that unsigned integer types are used and perform precondition tests are performed to prevent the possibility of integer wrapping.
- The app does not use unsafe string functions such as strcpy, most other functions beginning with the "str" prefix, sprint, vsprintf, gets, etc.;
- If the app contains C++ code, ANSI C++ string classes are used;
- In case of memcpy, make sure you check that the target buffer is at least of equal size as the source and that both buffers are not overlapping.
- iOS apps written in Objective-C use NSString class. C apps on iOS should use CFString, the Core Foundation representation of a string.
- No untrusted data is concatenated into format strings.

Reference

- owasp-mastg Memory Corruption Bugs (MSTG-CODE-8) Buffer and Integer Overflows

Rulebook

- *Do not use non safe character string functions that cause buffer overflow (Required)*
- *Buffa overflow best practices (Required)*

7.8.2.1 Static Analysis

Static code analysis of low-level code is a complex topic that could easily fill its own book. Automated tools such as RATS combined with limited manual inspection efforts are usually sufficient to identify low-hanging fruits. However, memory corruption conditions often stem from complex causes. For example, a use-after-free bug may actually be the result of an intricate, counter-intuitive race condition not immediately apparent. Bugs manifesting from deep instances of overlooked code deficiencies are generally discovered through dynamic analysis or by testers who invest time to gain a deep understanding of the program.

Reference

- owasp-mastg Memory Corruption Bugs (MSTG-CODE-8) Static Analysis

7.8.2.2 Dynamic Analysis

Memory corruption bugs are best discovered via input fuzzing: an automated black-box software testing technique in which malformed data is continually sent to an app to survey for potential vulnerability conditions. During this process, the application is monitored for malfunctions and crashes. Should a crash occur, the hope (at least for security testers) is that the conditions creating the crash reveal an exploitable security flaw.

Fuzz testing techniques or scripts (often called "fuzzers") will typically generate multiple instances of structured input in a semi-correct fashion. Essentially, the values or arguments generated are at least partially accepted by the target application, yet also contain invalid elements, potentially triggering input processing flaws and unexpected program behaviors. A good fuzzer exposes a substantial amount of possible program execution paths (i.e. high coverage output). Inputs are either generated from scratch ("generation-based") or derived from mutating known, valid input data ("mutation-based").

For more information on fuzzing, refer to the OWASP Fuzzing Guide.

Reference

- owasp-mastg Memory Corruption Bugs (MSTG-CODE-8) Dynamic Analysis

7.8.3 Memory corruption bug in Objective-C/Swift code

iOS applications have various ways to run into memory corruption bugs: first there are the native code issues which have been mentioned in the general Memory Corruption Bugs section. Next, there are various unsafe operations with both Objective-C and Swift to actually wrap around native code which can create issues. Last, both Swift and Objective-C implementations can result in memory leaks due to retaining objects which are no longer in use.

Reference

- owasp-mastg Memory Corruption Bugs (MSTG-CODE-8)

7.8.3.1 Static Analysis

Are there native code parts? If so: check for the given issues in the general memory corruption section. Native code is a little harder to spot when compiled. If you have the sources then you can see that C files use .c source files and .h header files and C++ uses .cpp files and .h files. This is a little different from the .swift and the .m source files for Swift and Objective-C. These files can be part of the sources, or part of third party libraries, registered as frameworks and imported through various tools, such as Carthage, the Swift Package Manager or Cocoapods.

For any managed code (Objective-C / Swift) in the project, check the following items:

- The doubleFree issue: when free is called twice for a given region instead of once.
- Retaining cycles: look for cyclic dependencies by means of strong references of components to one another which keep materials in memory.
- Using instances of UnsafePointer can be managed wrongly, which will allow for various memory corruption issues.
- Trying to manage the reference count to an object by Unmanaged manually, leading to wrong counter numbers and a too late/too soon release.

A great talk is given on this subject at Realm academy and a nice tutorial to see what is actually happening is provided by Ray Wenderlich on this subject.

Please note that with Swift 5 you can only deallocate full blocks, which means the playground has changed a bit.

Reference

- owasp-mastg Memory Corruption Bugs (MSTG-CODE-8) Static Analysis

Rulebook

- *Confirmation points in Objective-C / Swift (Recommended)*

7.8.3.2 Dynamic Analysis

There are various tools provided which help to identify memory bugs within Xcode, such as the Debug Memory graph introduced in Xcode 8 and the Allocations and Leaks instrument in Xcode.

Next, you can check whether memory is freed too fast or too slow by enabling NSAutoreleaseFreedObjectCheckEnabled, NSZombieEnabled, NSDebugEnabled in Xcode while testing the application.

There are various well written explanations which can help with taking care of memory management. These can be found in the reference list of this chapter.

Reference

- owasp-mastg Memory Corruption Bugs (MSTG-CODE-8) Dynamic Analysis
- https://developer.ibm.com/tutorials/mo-ios-memory/
- https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/MemoryMgmt/Articles/MemoryMgmt.html
- https://medium.com/zendesk-engineering/ios-identifying-memory-leaks-using-the-xcode-memory-graph-debugger-e84f097

7.8.4 Rulebook

1. *Do not use non safe character string functions that cause buffer overflow (Required)*
2. *Buffa overflow best practices (Required)*
3. *Confirmation points in Objective-C / Swift (Recommended)*

7.8.4.1 Do not use non safe character string functions that cause buffer overflow (Required)

The following functions exist as unsafe string functions that cause buffer overflows, and their use should be avoided.

- strcat
- strcpy
- strncat
- strlcat
- strncpy
- strlcpy
- sprintf
- snprintf
- gets

* There is no sample code because deprecated rules.

If this is violated, the following may occur.

- May cause buffer overflow.

7.8.4.2 Buffa overflow best practices (Required)

Make sure that you do not cause buffer overflows.

- When using integer variables for security-sensitive operations, such as array indexing or buffer length calculations, make sure that unsigned integer types are used, and make assumptions to prevent possible integer wrapping. Execute conditional tests.
- The app does not use unsafe string functions such as strcpy , most other functions prefixed with " str ", sprint , vsprintf , gets , etc.
- If your app contains C++ code, use the ANSI C++ string classes.
- For memcpy, make sure the target buffer is at least the same size as the source, and that they don' t overlap.
- No untrusted data is concatenated into the format string.

If this is violated, the following may occur.

- May cause buffer overflow.

7.8.4.3 Confirmation points in Objective-C / Swift (Recommended)

For any managed code (Objective-C / Swift) in the project, check the following items:

- The doubleFree issue: when free is called twice for a given region instead of once.
- Retaining cycles: look for cyclic dependencies by means of strong references of components to one another which keep materials in memory.
- Using instances of UnsafePointer can be managed wrongly, which will allow for various memory corruption issues.
- Trying to manage the reference count to an object by Unmanaged manually, leading to wrong counter numbers and a too late/too soon release.

A great talk is given on this subject at Realm academy and a nice tutorial to see what is actually happening is provided by Ray Wenderlich on this subject.

Please note that with Swift 5 you can only deallocate full blocks, which means the playground has changed a bit.

If this is not noted, the following may occur.

- May cause memory corruption bugs.

7.9 MSTG-CODE-9

Free security features offered by the toolchain, such as byte-code minification, stack protection, PIE support and automatic reference counting, are activated.

7.9.1 Use of Binary Protection Mechanisms

The tests used to detect the presence of binary protection mechanisms heavily depend on the language used for developing the application.

Although Xcode enables all binary security features by default, it may be relevant to verify this for old applications or to check for compiler flag misconfigurations. The following features are applicable:

- **PIE (Position Independent Executable)**:
  - PIE applies to executable binaries (Mach-O type MH_EXECUTE).
  - However it' s not applicable for libraries (Mach-O type MH_DYLIB).
- ***Memory management***:

  - Both pure Objective-C, Swift and hybrid binaries should have ARC (Automatic Reference Counting) enabled.
  - For C/C++ libraries, the developer is responsible for doing proper *Manual Memory Management*. See "*Memory Corruption Bugs*".

- **Stack Smashing Protection**: For pure Objective-C binaries, this should always be enabled. Since Swift is designed to be memory safe, if a library is purely written in Swift, and stack canaries weren't enabled, the risk will be minimal.

Learn more:

- OS X ABI Mach-O File Format Reference
- On iOS Binary Protections
- Security of runtime process in iOS and iPadOS
- Mach-O Programming Topics - Position-Independent Code

Tests to detect the presence of these protection mechanisms heavily depend on the language used for developing the application. For example, existing techniques for detecting the presence of stack canaries do not work for pure Swift apps.

Reference

- owasp-mastg Make Sure That Free Security Features Are Activated (MSTG-CODE-9) Overview

Rulebook

- *Confirmation of use of binary protection mechanism (Recommended)*

7.9.1.1 Xcode Project Settings

Stack Canary protection

Steps for enabling stack canary protection in an iOS application:

1. In Xcode, select your target in the "Targets" section, then click the "Build Settings" tab to view the target's settings.
2. Make sure that the "-fstack-protector-all" option is selected in the "Other C Flags" section.
3. Make sure that Position Independent Executables (PIE) support is enabled.

PIE protection

Steps for building an iOS application as PIE:

1. In Xcode, select your target in the "Targets" section, then click the "Build Settings" tab to view the target's settings.
2. Set the iOS Deployment Target to iOS 4.3 or later.
3. Make sure that "Generate Position-Dependent Code" (section "Apple Clang - Code Generation") is set to its default value ("NO").
4. Make sure that "Generate Position-Dependent Executable" (section "Linking") is set to its default value ("NO").

ARC protection

ARC is automatically enabled for Swift apps by the swiftc compiler. However, for Objective-C apps you'll have ensure that it's enabled by following these steps:

1. In Xcode, select your target in the "Targets" section, then click the "Build Settings" tab to view the target's settings.
2. Make sure that "Objective-C Automatic Reference Counting" is set to its default value ("YES").

See the Technical Q&A QA1788 Building a Position Independent Executable.

Reference

- owasp-mastg Make Sure That Free Security Features Are Activated (MSTG-CODE-9) Xcode Project Settings

7.9.2 Static Analysis

You can use otool to check the binary security features described above. All the features are enabled in these examples.

- PIE:

```
$ unzip DamnVulnerableiOSApp.ipa
$ cd Payload/DamnVulnerableIOSApp.app
$ otool -hv DamnVulnerableIOSApp
DamnVulnerableIOSApp (architecture armv7):
Mach header
magic cputype cpusubtype caps filetype ncmds sizeofcmds flags
MH_MAGIC ARM V7 0x00 EXECUTE 38 4292 NOUNDEFS DYLDLINK TWOLEVEL
WEAK_DEFINES BINDS_TO_WEAK PIE
DamnVulnerableIOSApp (architecture arm64):
Mach header
magic cputype cpusubtype caps filetype ncmds sizeofcmds flags
MH_MAGIC_64 ARM64 ALL 0x00 EXECUTE 38 4856 NOUNDEFS DYLDLINK TWOLEVEL
WEAK_DEFINES BINDS_TO_WEAK PIE
```

The output shows that the Mach-O flag for PIE is set. This check is applicable to all - Objective-C, Swift and hybrid apps but only to the main executable.

- Stack canary:

```
$ otool -Iv DamnVulnerableIOSApp | grep stack
0x0046040c 83177 ___stack_chk_fail
0x0046100c 83521 _sigaltstack
0x004fc010 83178 ___stack_chk_guard
0x004fe5c8 83177 ___stack_chk_fail
0x004fe8c8 83521 _sigaltstack
0x00000001004b3fd8 83077 ___stack_chk_fail
0x00000001004b4890 83414 _sigaltstack
0x0000000100590cf0 83078 ___stack_chk_guard
0x00000001005937f8 83077 ___stack_chk_fail
0x0000000100593dc8 83414 _sigaltstack
```

In the above output, the presence of __stack_chk_fail indicates that stack canaries are being used. This check is applicable to pure Objective-C and hybrid apps, but not necessarily to pure Swift apps (i.e. it is OK if it' s shown as disabled because Swift is memory safe by design).

- ARC:

```
$ otool -Iv DamnVulnerableIOSApp | grep release
0x0045b7dc 83156 ___cxa_guard_release
0x0045fd5c 83414 _objc_autorelease
0x0045fd6c 83415 _objc_autoreleasePoolPop
0x0045fd7c 83416 _objc_autoreleasePoolPush
0x0045fd8c 83417 _objc_autoreleaseReturnValue
0x0045ff0c 83441 _objc_release
[SNIP]
```

This check is applicable to all cases, including pure Swift apps where it' s automatically enabled.

Reference

- owasp-mastg Make Sure That Free Security Features Are Activated (MSTG-CODE-9) Static Analysis

7.9.3 Dynamic Analysis

These checks can be performed dynamically using Objection. Here's one example:

```
com.yourcompany.PPClient on (iPhone: 13.2.3) [usb] # ios info binary
Name                  Type     Encrypted    PIE    ARC    Canary    Stack Exec    ␣
↪RootSafe
--------------------  -------  -----------  -----  -----  --------  ------------  -
↪---------
PayPal                execute  True         True   True   True      False         ␣
↪False
CardinalMobile        dylib    False        False  True   True      False         ␣
↪False
FraudForce            dylib    False        False  True   True      False         ␣
↪False
...
```

Reference

- owasp-mastg Make Sure That Free Security Features Are Activated (MSTG-CODE-9) Dynamic Analysis

7.9.3.1 PIC (Position Independent Code)

PIC (Position Independent Code) is code that, being placed somewhere in the primary memory, executes properly regardless of its absolute address. PIC is commonly used for shared libraries, so that the same library code can be loaded in a location in each program address space where it does not overlap with other memory in use (for example, other shared libraries).

PIE (Position Independent Executable) are executable binaries made entirely from PIC. PIE binaries are used to enable ASLR (Address Space Layout Randomization) which randomly arranges the address space positions of key data areas of a process, including the base of the executable and the positions of the stack, heap and libraries.

Reference

- owasp-mastg Make Sure That Free Security Features Are Activated (MSTG-CODE-9) Position Independent Code

7.9.3.2 Memory management

Automatic Reference Counting

ARC (Automatic Reference Counting) is a memory management feature of the Clang compiler exclusive to Objective-C and Swift. ARC automatically frees up the memory used by class instances when those instances are no longer needed. ARC differs from tracing garbage collection in that there is no background process that deallocates the objects asynchronously at runtime.

Unlike tracing garbage collection, ARC does not handle reference cycles automatically. This means that as long as there are "strong" references to an object, it will not be deallocated. Strong cross-references can accordingly create deadlocks and memory leaks. It is up to the developer to break cycles by using weak references. You can learn more about how it differs from Garbage Collection here.

Garbage Collection

Garbage Collection (GC) is an automatic memory management feature of some languages such as Java/Kotlin/Dart. The garbage collector attempts to reclaim memory which was allocated by the program, but is no longer referenced —also called garbage. The Android runtime (ART) makes use of an improved version of GC. You can learn more about how it differs from ARC here.

Manual Memory Management Manual memory management is typically required in native libraries written in C/C++ where ARC and GC do not apply. The developer is responsible for doing proper memory management. Manual memory management is known to enable several major classes of bugs into a program when used incorrectly, notably violations of memory safety or memory leaks.

More information can be found in "*Memory Corruption Bugs*" .

Reference

- owasp-mastg Make Sure That Free Security Features Are Activated (MSTG-CODE-9) Memory management

7.9.3.3 Stack Smashing Protection

Stack canaries help prevent stack buffer overflow attacks by storing a hidden integer value on the stack right before the return pointer. This value is then validated before the return statement of the function is executed. A buffer overflow attack often overwrites a region of memory in order to overwrite the return pointer and take over the program flow. If stack canaries are enabled, they will be overwritten as well and the CPU will know that the memory has been tampered with.

Stack buffer overflow is a type of the more general programming vulnerability known as buffer overflow (or buffer overrun). Overfilling a buffer on the stack is more likely to **derail program execution** than overfilling a buffer on the heap because the stack contains the return addresses for all active function calls.

Reference

- owasp-mastg Make Sure That Free Security Features Are Activated (MSTG-CODE-9) Stack Smashing Protection

7.9.4 Rulebook

1. *Confirmation of use of binary protection mechanism (Recommended)*

7.9.4.1 Confirmation of use of binary protection mechanism (Recommended)

Xcode enables all binary security features by default, but you should check this for older applications or check for misconfigured compiler flags.

The following functions are applied for verification.

- **PIE (Position Independent Executable)**:
  - PIE applies to executable binaries (Mach-O type MH_EXECUTE).
  - However it's not applicable for libraries (Mach-O type MH_DYLIB).
- ***Memory management***:
  - Both pure Objective-C, Swift and hybrid binaries should have ARC (Automatic Reference Counting) enabled.
  - For C/C++ libraries, the developer is responsible for doing proper *Manual Memory Management*.
- **Stack Smashing Protection**: For pure Objective-C binaries, this should always be enabled. Since Swift is designed to be memory safe, if a library is purely written in Swift, and stack canaries weren't enabled, the risk will be minimal.

Learn more:

- OS X ABI Mach-O File Format Reference
- On iOS Binary Protections
- Security of runtime process in iOS and iPadOS
- Mach-O Programming Topics - Position-Independent Code

Tests to detect the presence of these protection mechanisms heavily depend on the language used for developing the application. For example, existing techniques for detecting the presence of stack canaries do not work for pure Swift apps.

If this is not noted, the following may occur.

- Binary protection function may not be available.

8 Revision history

2023-04-01

Initial English Edition