

MASDG

モバイルアプリケーション
セキュリティ設計ガイド
iOS 版



安田 良明



目次

第 1 章 アーキテクチャ・設計・脅威モデリング要件	3
1.1 MSTG-ARCH-1	3
1.2 MSTG-ARCH-2	6
1.3 MSTG-ARCH-3	13
1.4 MSTG-ARCH-4	13
1.5 MSTG-ARCH-12	15
第 2 章 データストレージとプライバシー要件	23
2.1 MSTG-STORAGE-1	23
2.2 MSTG-STORAGE-2	59
2.3 MSTG-STORAGE-3	60
2.4 MSTG-STORAGE-4	63
2.5 MSTG-STORAGE-5	65
2.6 MSTG-STORAGE-6	67
2.7 MSTG-STORAGE-7	69
2.8 MSTG-STORAGE-12	72
第 3 章 暗号化要件	75
3.1 MSTG-CRYPTO-1	75
3.2 MSTG-CRYPTO-2	94
3.3 MSTG-CRYPTO-3	107
3.4 MSTG-CRYPTO-4	108
3.5 MSTG-CRYPTO-5	110
3.6 MSTG-CRYPTO-6	113
第 4 章 認証とセッション管理要件	117
4.1 MSTG-AUTH-1	117
4.2 MSTG-AUTH-2	117
4.3 MSTG-AUTH-3	119
4.4 MSTG-AUTH-4	140
4.5 MSTG-AUTH-5	142

4.6	MSTG-AUTH-6	147
4.7	MSTG-AUTH-7	147
4.8	MSTG-AUTH-12	147
第 5 章 ネットワーク通信要件		149
5.1	MSTG-NETWORK-1	149
5.2	MSTG-NETWORK-2	156
5.3	MSTG-NETWORK-3	161
第 6 章 プラットフォーム連携要件		163
6.1	MSTG-PLATFORM-1	163
6.2	MSTG-PLATFORM-2	181
6.3	MSTG-PLATFORM-3	184
6.4	MSTG-PLATFORM-4	211
6.5	MSTG-PLATFORM-5	258
6.6	MSTG-PLATFORM-6	271
6.7	MSTG-PLATFORM-7	285
6.8	MSTG-PLATFORM-8	292
第 7 章 コード品質とビルド設定要件		311
7.1	MSTG-CODE-1	311
7.2	MSTG-CODE-2	314
7.3	MSTG-CODE-3	316
7.4	MSTG-CODE-4	318
7.5	MSTG-CODE-5	325
7.6	MSTG-CODE-6	335
7.7	MSTG-CODE-7	347
7.8	MSTG-CODE-8	347
7.9	MSTG-CODE-9	354
第 8 章 更新履歴		361

2023 年 04 月 01 日版

モバイルアプリケーションのセキュリティ設計ガイド (MASDG) iOS 版へようこそ。

モバイルアプリケーションのセキュリティ設計ガイド iOS 版は OWASP が公開している MASVS および MASTG に本プロジェクト独自の判定基準（ルールブック）やサンプルコードを組み込んだ iOS 上のセキュアなモバイルアプリケーションを設計、開発、テストするときに必要となるセキュリティ設計のフレームワークを確立するためのドキュメントです。

MASDG は、セキュリティ要件に対する具体的な設計内容に特化したベストプラクティスやサンプル等を取り扱うことで、MASVS を元に検討したセキュリティ要件からセキュリティ設計を作成することをサポートすることや、MASTG で示されているテスト方法を実施する前にセキュリティ設計に問題が無いかを評価することをサポートします。

本プロジェクトの独自のルールブックは MASVS L1 検証標準を対象としておりモバイルアプリを開発する際に網羅的なセキュリティベースラインを提供することを目的としています。これから創出される新しいテクノロジーは必ずリスクをもたらしプライバシー やセーフティー の課題を生じますがモバイルアプリに関する脅威に立ち向かうべく本ドキュメントの作成を行いました。

MASVS および MASTG に対しては様々なコミュニティや業界からフィードバックを受けていますが、本プロジェクトとしても社会生活に不可欠となったモバイルアプリケーションに対するセキュリティリスクの課題を取り組んで行きたいと考えておりますので MASDG を開発し公開しました。皆様からのフィードバックを歓迎します。

著作権とライセンス



Copyright © The OWASP Foundation. 本著作物は Creative Commons Attribution-ShareAlike 4.0 International License に基づいてライセンスされています。再使用または配布する場合は、他者に対し本著作物のライセンス条項を明らかにする必要があります。

- 本ガイドの内容は執筆時点のものです。サンプルコードを使用する場合はこの点にあらかじめご注意ください。
- 執筆関係者は、このガイド文書に関するいかなる責任も負うものではありません。全ては自己責任にてご活用ください。
- iOS は、Apple Inc. の商標または登録商標です。また、本文書に登場する会社名、製品名、サービス名は、一般に各社の登録商標または商標です。本文中では ®、TM、© マークは明記していません。
- この文書の内容の一部は、OWASP MASVS, OWASP MASTG が作成、提供しているコンテンツをベースに複製、改版したものです。

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 Willemse and Carlos Holguera	Alexander Antukh, Mesheryakov Aleksey, Elderov Ali, Bachevsky Artem, Jeroen Beckers, Jon-Anthonney 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 Willemse
Carlos Holguera
Romuald Szkulnarek
Jeroen Beckers
Vikas Gupta

OWASP MASVS ja Author

Koki Takeyama

OWASP MASTG ja Author

Koki Takeyama

Project Supporter

Riotaro Okada

1

アーキテクチャ・設計・脅威モデリング要件

1.1 MSTG-ARCH-1

アプリのすべてのコンポーネントを把握し、それらが必要とされている。

1.1.1 コンポーネント

アプリのすべてのコンポーネントを把握して、不要なコンポーネントを削除すること。

主なコンポーネントの種類は以下の通り。

- コンテンツ
 - Charts
 - Image views
 - Text views
 - Web views
- レイアウトと組織
 - Boxes
 - Collections
 - Column views
 - Disclosure Controls
 - Labels
 - Lists and tables
 - Lockups
 - Outline views

- Split views
- Tab views
- メニューとアクション
 - Activity views
 - Buttons
 - Context menus
 - Dock menus
 - Edit menus
 - Menus
 - Pop-up buttons
 - Pull-down buttons
 - Toolbars
- ナビゲーションと検索
 - Navigation bars
 - Path controls
 - Search fields
 - Sidebars
 - Tab bars
 - Token fields
- プrezentation
 - Action sheets
 - Alerts
 - Page controls
 - Panels
 - Popovers
 - Scroll views
 - Sheets
 - Windows
- 選択と入力

- Color wells
 - Combo boxes
 - Digit entry views
 - Image wells
 - Onscreen keyboards
 - Pickers
 - Segmented controls
 - Sliders
 - Steppers
 - Text fields
 - Toggles
- 状態
 - Activity rings
 - Gauges
 - Progress indicators
 - Rating indicators
 - システム
 - Complications
 - Home Screen quick actions
 - Live Activities
 - The menu bar
 - Notifications
 - Status bars
 - Top Shelf
 - Watch faces
 - Widgets

1.2 MSTG-ARCH-2

セキュリティコントロールはクライアント側だけではなくそれぞれのリモートエンドポイントで実施されている。

1.2.1 認証・認可情報の改竄

1.2.1.1 適切な認証対応

認証・認可のテストを行う場合は、以下の手順で行う。

- アプリが使用する追加の認証要素を特定する。
- 重要な機能を提供するすべてのエンドポイントを特定する。
- すべてのサーバ側エンドポイントにおいて、追加要素が厳密に実施されていることを確認する。

認証バイパスの脆弱性は、サーバ上で認証状態が一貫して実施されておらず、クライアントが認証状態を改ざんできる場合に存在する。バックエンドサービスは、モバイルクライアントからのリクエストを処理している間、リソースがリクエストされるたびに、ユーザがログインしているか、認可されているかを確認し、一貫した認可チェックを実施する必要がある。

[OWASP Web Testing Guide](#) の次の例を検討してください。この例では、Web リソースは URL を介してアクセスされ、認証状態は GET パラメータを介して渡される。

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

クライアントは、リクエストと共に送られる GET パラメータを任意に変更することができる。クライアントが単に `authenticated` パラメータの値を "yes" に変更することを妨げるものはなにもなく、効果的に認証をバイパスすることができてしまう。

これはおそらく実際には見られない単純な例だが、プログラマは認証状態を維持するために Cookie のような「隠れた」クライアント側パラメータに依存することがある。このようなパラメータは改ざんできないと想定している。例えば、[Nortel Contact Center Manager](#) における次のような典型的な脆弱性が考えられる。Nortel のアプライアンスの管理 Web アプリケーションは、cookie の「isAdmin」に依存して、ログインしたユーザに管理権限を付与する必要があるかどうかを判断していた。その結果、以下のようにクッキーの値を設定するだけで、管理者権限を取得することが可能であった。

```
isAdmin=True
```

セキュリティの専門家は、以前はセッションベースの認証を使用し、サーバ上でのみセッションデータを維持することを推奨していた。これにより、クライアント側でセッションの状態が改ざんされることを防ぐことができる。しかし、セッションベースの認証の代わりにステートレス認証を使用する要点は、サーバ上にセッションの状態を持たないことである。代わり、状態はクライアント側のトークンに保存され、リクエストごとに送信される。この場合、`isAdmin` のようなクライアント側のパラメータを見ることは、全く問題ない。

改ざんを防ぐために、クライアント側のトークンには暗号署名が付加されている。もちろん、うまくいかないこともあります。一般的なステートレス認証の実装は攻撃に対して脆弱である。例えば、いくつかの JSON Web

Token(JWT) 実装の署名検証は、署名タイプを「None」に設定することで無効化することが可能である。この攻撃については、「Testing JSON Web Token」の章で詳しく説明する。

参考資料

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

1.2.2 インジェクション欠陥

インジェクションの欠陥は、ユーザ入力がバックエンドのクエリやコマンドに挿入されたときに発生するセキュリティ上の脆弱性のことである。メタ文字を挿入することで、攻撃者は、コマンドやクエリの一部として誤って解釈される悪意のあるコードを実行できる。例えば、SQL クエリを操作することで、攻撃者は任意のデータベースレコードを取得したり、バックエンドのデータベースの内容を操作したりすることができる。

このクラスの脆弱性は、サーバ側の Web サービスに最も多く存在する。モバイルアプリにも悪用可能な例があるが、発生頻度は低く、攻撃対象領域も小さくなっている。

例えば、アプリがローカルの SQLite データベースにクエリを実行する場合、そのようなデータベースは通常、機密データを保存しない（開発者が基本的なセキュリティプラクティスに従っている場合）。このため、SQL インジェクションは攻撃ベクトルとしてふさわしくない。それにもかかわらず、インジェクションの脆弱性が悪用されることもあるため、適切な入力検証はプログラマにとって必要なベストプラクティスであると言える。

参考資料

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

ルールブック

- 適切な入力検証を実施する（必須）

1.2.2.1 SQL インジェクション

SQL インジェクション攻撃は、あらかじめ定義された SQL コマンドの構文を模倣して、入力データに SQL コマンドを統合する。SQL インジェクション攻撃が成功すると、攻撃者は、サーバから付与された権限に応じて、データベースへの読み取りまたは書き込みが可能になり、管理コマンドを実行できる可能性がある。

Android と iOS のアプリは、ローカルデータストレージを制御および整理する手段として SQLite データベースを使用する。例えば、Android アプリが、ローカルデータベースにユーザ情報を保存して、ローカルユーザ認証を処理しているとする（例のために見過ごしている不適切なプログラミング手法である）。ログイン時に、アプリはデータベースにクエリを実行し、ユーザが入力したユーザ名とパスワードを使用してレコードを検索する。

```
SQLiteDatabase db;

String sql = "SELECT * FROM users WHERE username = '" + username + "' AND_
password = '" + password + "'";

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

(次のページに続く)

(前のページからの続き)

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

さらに、攻撃者が「ユーザ名」と「パスワード」の欄に以下の値を入力したとする。

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

これにより、以下のようなクエリが発生する。

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

条件 '1'='1' は常に true と評価されるため、このクエリはデータベース内のすべてのレコードを返し、有効なユーザ アカウントが入力されていなくても、ログイン関数は true を返す。Ostorlab は、この SQL インジェクションのペイロードを使用して、adb を使用して Yahoo's weather mobile application のソートパラメータを悪用した。

Mark Woods 氏は、QNAP NAS ストレージ・アプライアンス上で動作する「Qnotes」および「Qget」Android アプリ内で、クライアント側 SQL インジェクションのもう 1 つの実例を発見した。これらのアプリは、SQL インジェクションに対して脆弱なコンテンツプロバイダをエクスポートし、攻撃者が NAS デバイスの認証情報を取得することを可能にする。この問題の詳細な説明は、Nettitude Blog にある。

参考資料

- [owasp-mastg Injection Flaws \(MSTG-ARCH-2 and MSTG-PLATFORM-2\) SQL Injection](#)

1.2.2.2 XML インジェクション

XML インジェクション攻撃では、攻撃者は XML メタ文字を挿入して XML コンテンツを構造的に変更する。これは、XML ベースのアプリケーションまたはサービスのロジックを危険にさらすだけでなく、コンテンツを処理する XML パーサーの操作を攻撃者が悪用できる可能性もある。

この攻撃の一般的な変種として、[XML eXternal Entity \(XXE\)](#) がある。攻撃者は、URI を含む外部エンティティ定義を入力 XML に挿入する。XML パーサーは、解析中に URI で指定されたリソースにアクセスし、攻撃者が定義したエンティティを展開する。

解析アプリケーションの完全性は、最終的に攻撃者の与えられる機能を決定し、悪意のあるユーザは次のいずれか（あるいはすべて）を行うことができる。ローカルファイルへのアクセス、任意のホストとポートへの HTTP リクエストの誘因、[クロスサイトリクエストフォージェリ（CSRF）](#) 攻撃の起動、サービス拒否状態の誘因。OWASP ウェブテストガイドには、XXE に関する以下の例がある。

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

この例では、ローカルファイル /dev/random が開かれ、そこで無限のバイトストリームが返されるため、サービス拒否が引き起こされる可能性がある。

現在のアプリ開発のトレンドは、XML が一般的でなくなってきたため、ほとんどが REST/JSON ベースのサービスに焦点を当てている。しかし、まれにユーザが提供したコンテンツや信頼できないコンテンツを使用して XML クエリを作成した場合、iOS の NSXMLParser などのローカル XML パーサーによって解釈される可能性がある。そのため、入力は常に検証する必要があり、メタ文字はエスケープする必要がある。

参考資料

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

ルールブック

- 入力は常に検証する必要があり、メタ文字はエスケープする必要がある（必須）

1.2.2.3 インジェクション攻撃ベクトル

モバイルアプリの攻撃対象は、一般的な Web アプリケーションやネットワークアプリケーションとは全く異なる。モバイルアプリは、ネットワーク上にサービスを公開するあまりなく、アプリのユーザインターフェース上で実行可能な攻撃ベクトルは稀である。アプリに対するインジェクション攻撃は、プロセス間通信（IPC）インターフェースを介して発生する可能性が最も高く、悪意のあるアプリがデバイス上で実行されている別のアプリを攻撃することがある。

潜在的な脆弱性を見つけるには、まず次のどちらかを行う。

- 信頼できない入力の可能性のあるエントリポイントを特定し、その場所からトレースして、宛先に潜在的な脆弱性のある関数が含まれているかどうかを確認する。
- 既知の危険なライブラリ/API 呼び出し（SQL クエリなど）を特定し、未チェックの入力がそれぞれのクエリと正常に連携しているかどうかを確認する。

手動のセキュリティ レビューでは、両方の手法を組み合わせて使用する必要がある。一般に、信頼できない入力は、次のチャネルを通じてモバイル アプリに侵入する。

- IPC コール
- カスタム URL スキーム
- QR コード
- Bluetooth、NFC などで受信した入力データ
- ペーストボード
- ユーザインターフェース

以下のベスト プラクティスに従っていることを確認する。

- 信頼できない入力は、許容値のリストを使用して型チェックや検証を行う。
- データベースクエリを実行するときは、変数バインディング（つまり、パラメータ化されたクエリ）を使用する prepared ステートメントが使用される。prepared ステートメントが定義されている場合、ユーザ提供のデータと SQL コードは自動的に分離される。

- XML データを解析するときは、XXE 攻撃を防ぐために、パーサーアプリケーションが外部エンティティの解決を拒否するように構成されていることを確認する。
- X.509 形式の証明書データを扱う場合は、安全なパーサーが使用されていることを確認する。例えば、バージョン 1.6 未満の Bouncy Castle では、安全でないリフレクションによるリモートコード実行が可能である。

各モバイル OS の入力ソースや脆弱性の可能性のある API に関する詳細は、OS 固有のテストガイドで参照。

参考資料

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

ルールブック

- 宛先に潜在的な脆弱性のある関数を含めない（必須）
- 未チェックの入力がそれぞれのクエリと正常に連携している（必須）
- 信頼できない入力を確認する（必須）
- パーサーアプリケーションは外部エンティティの解決を拒否するように構成する（必須）
- X.509 形式の証明書データを利用する場合は安全なパーサーを使用する（必須）

1.2.3 ルールブック

1. 適切な入力検証を実施する（必須）
2. 入力は常に検証する必要があり、メタ文字はエスケープする必要がある（必須）
3. 宛先に潜在的な脆弱性のある関数を含めない（必須）
4. 未チェックの入力がそれぞれのクエリと正常に連携している（必須）
5. 信頼できない入力を確認する（必須）
6. パーサーアプリケーションは外部エンティティの解決を拒否するように構成する（必須）
7. X.509 形式の証明書データを利用する場合は安全なパーサーを使用する（必須）

1.2.3.1 適切な入力検証を実施する（必須）

インジェクションの脆弱性が悪用されることもあるため、適切な入力検証はプログラマにとって必要なベストプラクティスであると言える。

下記は入力検証の一例。

- 正規表現チェック
- 長さ/サイズチェック

これに違反する場合、以下の可能性がある。

- インジェクションの脆弱性が悪用される可能性がある。

1.2.3.2 入力は常に検証する必要があり、メタ文字はエスケープする必要がある（必須）

ユーザが提供したコンテンツや信頼できないコンテンツを使用して XML クエリを作成した場合、ローカル XML パーサーによって XML メタ文字が XML コンテンツとして解釈される可能性がある。そのため、入力は常に検証する必要があり、メタ文字はエスケープする必要がある。

下記は入力検証の一例。

- 正規表現チェック
- 長さ/サイズチェック

下記はメタ文字の一例。

表 1.2.3.2.1 メタ文字一覧

文字	項目名	エンティティ参照による表記
<	右大なり	<
>	左大なり	>
&	アンパーサント	&
"	ダブルクオーテーション	"
'	シングルクオーテーション	'

これに違反する場合、以下の可能性がある。

- ローカル XML パーサーによって XML メタ文字が XML コンテンツとして解釈される可能性がある。

1.2.3.3 宛先に潜在的な脆弱性のある関数を含めない（必須）

信頼できない入力の可能性のあるエントリポイントを特定し、その場所からトレースして、宛先に潜在的な脆弱性のある関数（サードパーティ製の関数）が含まれているかどうかを確認する。

これに違反する場合、以下の可能性がある。

- プロセス間通信（IPC）インターフェースを介して、悪意のあるアプリがデバイス上で実行されている別のアプリを攻撃する可能性がある。

1.2.3.4 未チェックの入力がそれぞれのクエリと正常に連携している（必須）

既知の危険なライブラリ/API 呼び出し（SQL クエリなど）を特定し、未チェックの入力がそれぞれのクエリと正常に連携しているかどうかを確認する。また、利用するライブラリ/API のリファレンスを確認し、非推奨でないことを確認する。

iOS API リファレンス：<https://developer.apple.com/documentation/>

これに違反する場合、以下の可能性がある。

- プロセス間通信（IPC）インターフェースを介して、悪意のあるアプリがデバイス上で実行されている別のアプリを攻撃する可能性がある。

1.2.3.5 信頼できない入力を確認する（必須）

一般に、信頼できない入力は、次のチャネルを通じてモバイルアプリに侵入するため、チャネルの利用箇所をソースコードから特定し確認する。

下記はチャネルの利用を特定するキーワードの一例。

- IPC コール : NSXPConnection, XPC Services
- カスタム URL スキーム : deeplink, URL Schemes, identifier
- QR コード : qr, camera
- Bluetooth、NFC などで受信した入力データ : MCNearbyServiceBrowser, MCNearbyServiceAdvertiser, Core Bluetooth
- ペーストボード : UIPasteboard
- ユーザインターフェース : UITextField

これに違反する場合、以下の可能性がある。

- プロセス間通信（IPC）インターフェースを介して、悪意のあるアプリがデバイス上で実行されている別のアプリを攻撃する可能性がある。

1.2.3.6 パーサーアプリケーションは外部エンティティの解決を拒否するように構成する（必須）

XXE 攻撃を防ぐために、パーサーアプリケーションは外部エンティティの解決を拒否するように構成する必要がある。

XXE を防止する最も安全な方法は、常に DTD (外部エンティティ) を完全に無効にすることである。パーサーに応じて、メソッドは次のようになる。

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

DTD を無効にすると、パーサーは **Billion Laughs** などのサービス拒否 (DOS) 攻撃に対しても安全になる。DTD を完全に無効にすることができない場合は、各パーサーに固有の方法で、外部エンティティと外部ドキュメントタイプの宣言を無効にする必要がある。

これに違反する場合、以下の可能性がある。

- XXE 攻撃に対して脆弱となる。

参考資料

- OWASP Cheat Sheet Series XML External Entity Prevention https://cheatsheetseries.owasp.org/cheatsheets/XML_External_Entity_Prevention_Cheat_Sheet.html#general-guidance

1.2.3.7 X.509 形式の証明書データを利用する場合は安全なパーサーを使用する（必須）

X.509 形式の証明書データを扱う場合は、安全なパーサーを使用する。

下記は安全なパーサーの一例。

- SecCertificateCreateWithData

これに違反する場合、以下の可能性がある。

- 安全でないリフレクションによるリモートコード実行等、インジェクション攻撃に対して脆弱となる。

1.3 MSTG-ARCH-3

モバイルアプリと接続されるすべてのリモートサービスの高次のアーキテクチャが定義され、そのアーキテクチャにセキュリティが対応されている。

※リモートサービス側での対応に関する章であるため、本資料ではガイド記載を省略

1.4 MSTG-ARCH-4

モバイルアプリのコンテキストで機密とみなされるデータが明確に特定されている。

機密情報の分類は、業界や国によって異なる。さらに、組織は機密データに対して制限的な見方をしている場合があり、機密情報を明確に定義するデータ分類ポリシーを持っている場合がある。データ分類ポリシーが利用できない場合は、一般的に機密と見なされる情報の次のリストを使用する。

参考資料

- owasp-mastg Mobile Application Security Testing Identifying Sensitive Data

ルールブック

- データ分類ポリシーに従い、機密データを識別する（必須）

1.4.1 ユーザ認証情報

ユーザ認証情報（資格情報、PIN など）。

1.4.2 個人識別情報

個人情報の盗難に悪用される可能性のある個人識別情報 (PII): 社会保障番号、クレジットカード番号、銀行口座番号、健康情報。

1.4.3 デバイス識別子

個人を特定できるデバイス識別子。

1.4.4 機密性高いデータ

侵害されると風評被害や金銭的コストにつながる機密性の高いデータ。

1.4.5 保護が法的義務であるデータ

保護が法的義務であるデータ。

1.4.6 技術データ

アプリ (またはその関連システム) によって生成され、他のデータまたはシステム自体を保護するために使用される技術データ (暗号化キーなど)。

1.4.7 ルールブック

1. データ分類ポリシーに従い、機密データを識別する (必須)

1.4.7.1 データ分類ポリシーに従い、機密データを識別する (必須)

業界・国・組織で明確に定義されているデータ分類ポリシーに従い、機密データを識別する。データ分類ポリシーが利用できない場合は、一般的に機密と見なされる情報として以下のリストを使用する。

- ユーザ認証情報 (資格情報、PIN など)
- 個人情報の盗難に悪用される可能性のある個人識別情報 (PII): 社会保障番号、クレジットカード番号、銀行口座番号、健康情報。
- 個人を特定できるデバイス識別子
- 機密性高いデータ: 侵害されると風評被害や金銭的コストにつながる機密性の高いデータ。
- 保護が法的義務であるデータ
- アプリ (またはその関連システム) によって生成され、他のデータまたはシステム自体を保護するために使用される技術データ (暗号化キーなど)。

「機密データ」の定義なしで機密データの漏洩を検出することは不可能な場合があるため、定義はテストを開発する前に決定する必要がある。

これに違反する場合、以下の可能性がある。

- ペネトレーションテストの結果から侵害される可能性のある機密データを機密データとして認識できず、リスクとして把握・対策出来ない可能性がある。

1.5 MSTG-ARCH-12

アプリはプライバシーに関する法律および規制に準拠している。

1.5.1 プライバシーに関する一般的な法律および規則

参考資料

- V6.1 Data Classification
- V6.1 データ分類

1.5.1.1 個人情報・プライバシー

個人情報を扱っている場合、GDPR、個人情報保護法に準拠していること。

1.5.1.2 医療データ

医療データを扱っている場合、HIPAA,HITECH に準拠していること。

1.5.1.3 金融情報

クレジットカード情報

クレジットカード情報を扱っている場合、PCI DSS に準拠していること。

1.5.2 プライバシーに関する App Store での規則

Apple のエコシステムにおいて、ユーザのプライバシーの保護は最優先に扱われる。個人のデータは慎重に扱い、ユーザの期待に応えることはもちろん、プライバシーのベストプラクティス、すべての適用法および「Apple Developer Program 使用許諾契約」の規約に準拠することが求められる。以下に具体例を記載する。

参考資料

- App Store 法的事項 5.1 プライバシー

1.5.2.1 データの収集および保存

1. **プライバシーポリシー**すべての App には、App Store Connect のメタデータフィールドと各 App 内にアクセスしやすい形で、プライバシーポリシーへのリンクを必ず含める必要がある。プライバシーポリシーはわかりやすく明確なものである必要がある。
 - App/サービスが収集するデータの種類（該当する場合）、データの収集方法、データの用途はすべて明確に提示すること。
 - 本ガイドラインに準拠して、App のユーザデータをサードパーティ（分析ツール、広告ネットワーク、サードパーティ製の SDK、その他ユーザデータにアクセスできる親会社、子会社、その他の関連組織）と共有する場合は、そのサードパーティが App のプライバシーポリシーで定める内容や本ガイドラインの要求事項と同一、あるいは同等のレベルでユーザのデータを保護していることを確認する必要がある。
 - データ保存/削除のポリシーと、ユーザが同意を無効にする方法やユーザデータの削除をリクエストする方法を記載する必要がある。
2. **許可**ユーザデータや使用状況に関するデータを収集する App では、収集するデータが収集の時点またはその直後の時点で匿名であると考えられる場合でも、そのデータ収集に関してユーザから同意を得る必要がある。有料の機能は、ユーザデータへのアクセスをユーザが許可することに応じるもの、またはそれを条件とすることはできない。また、App では、簡単にアクセスできるわかりやすい方法でユーザが同意を撤回できるようにする必要がある。必ず、目的文字列でデータの用途を明確かつ十分に説明すること。正当な利益のため、同意を得ることなくデータを収集する App は、EU 一般 Data protection 規則（GDPR）の規約、またはそれに類する制定法のすべての条項を遵守する必要がある。詳しくは、「[許可のリクエスト](#)」を確認すること。
3. **必要最低限のデータ** App は、中心的な機能に関連するデータへのアクセスのみがリクエストされ、関連するタスクの実行に必要なデータのみが収集および使用されるものである必要がある。可能な場合、「写真」や「連絡先」といった保護されているリソースへの完全なアクセス権をリクエストする代わりに、プロセス外のピッカーやシェアシートを使用すること。
4. **アクセス** App では、ユーザのアクセス許可設定を尊重しなければならない。不要なデータアクセスに同意するようユーザを巧みに誘導したり、だましたり、強制したりすることはできない。たとえば、ソーシャルネットワークに写真を投稿できる App で、マイクへのアクセスに同意しなければ写真をアップロードできない仕様とすることは許可されない。可能であれば、アクセスに同意しないユーザ向けに別の方法を用意すること。たとえば、位置情報の共有に同意しないユーザには、住所を手動で入力できる機能を用意することができる。
5. **アカウントへのログイン** アカウント情報をベースにした重要な機能を実装しているものでない限り、App にログインせずに使用できるようにする。アカウントの作成に対応した App の場合は、App 内でアカウントの削除もできるようにする必要がある。App の中心的な機能に直接関連する場合、または法律で要求される場合を除き、App を動作させるためにユーザの個人情報の入力を要求することは許可されない。App の中心的な機能が特定のソーシャルネットワーク（Facebook、WeChat、Weibo、Twitter など）に関連するものでない場合は、ログインや別のメカニズムを介さずに使用できるようにする必要がある。基本的なプロフィール情報の取得、ソーシャルネットワークへの公開、App を利用するよう友達を招待することは、App の中心的な機能とはみなされない。また、ソーシャルネットワークの認証情

報や、App とソーシャルネットワーク間のデータアクセスを App 内で無効にできるメカニズムを用意する必要があります。App では、ソーシャルネットワークの認証情報やトークンをデバイスの外部に保存することはできない。そうした認証情報やトークンは、App の使用中に、App から直接ソーシャルネットワークに接続するときにのみ使用することができる。

6. App を利用して密かにユーザのパスワードやその他のプライベートデータを取得するデベロッパは、Apple Developer Program から除名される。
7. ユーザに情報を視覚的に提示する際は、必ず SafariViewController を使用する必要がある。SafariViewController を非表示にしたり、別のビューやレイヤーで隠したりすることは許可されない。また、SafariViewController を使用して、ユーザの認知や同意なしに App でユーザのトラッキングを行うことは許可されない。
8. ユーザ以外のソースから取得した個人情報、またはユーザの明示的な同意なしに取得された個人情報を収集する App は、その情報が公開データベースからのものであったとしても、App Store では許可されない。
9. 規制の多い分野（キャッシングや金融サービス、ヘルスケア、ギャンブル、合法大麻の使用、航空旅行など）でのサービスを提供する App、機密性の高いユーザ情報を必要とする App は、個人のデベロッパではなく、そうしたサービスを提供する法人によって提出される必要がある。大麻の合法的な販売を促進するための App は、それが合法とみなされる法的管轄地域でのみ利用できるよう、地域制限を設定する必要がある。
10. App は、ユーザの基本的な連絡先情報（たとえば名前やメールアドレスなど）の共有がユーザの任意の選択であり、いかなる機能やサービスの提供もこれらの情報の共有を条件にしておらず、本ガイドラインのその他の規定（子どもからの情報収集に関する制限を含む）にすべて遵守するものである限り、これらの情報をユーザにリクエストすることができる。

参考資料

- App Store 法的事項 5.1.1 データの収集および保存

1.5.2.2 データの使用と共有

1. 法律で許可されているものでない限り、事前にユーザの許可を取らずに、ユーザの個人データを使用、送信、共有することはできない。使用する場合は、どこでどのようにデータを使用するかに関する情報をユーザが確認できる手段を提供する必要がある。App で収集したデータは、App の改善や、「[Apple Developer Program 使用許諾契約](#)」に準拠した）広告の提示といった目的でのみサードパーティと共有することができる。ユーザアクティビティをトラッキングするには、App Tracking Transparency API を介して、ユーザの明示的な許可を得る必要がある。トラッキングについて、詳しくは[こちら](#)を確認すること。ユーザの同意なしに、またはプライバシー関連の法令を遵守せずにユーザデータを共有する App は App Store から削除される。さらに、デベロッパは Apple Developer Program から除名される場合がある。
2. 法律で明示的に許可されている場合を除き、特定の目的のために収集されたデータを、ユーザの同意をあらためて取ることなく、別の目的に使用することはできない。
3. App で収集したデータに基づき、密かにユーザプロファイルを構築することは許可されない。また、

Apple から提供された API で収集したデータ、「匿名化」されたデータ、「集積」されたデータ、または個人を識別できないその他の方法で収集されると説明されたデータに基づいて、ユーザの識別やユーザプロファイルの再構築を行おうとしたり、助長したり、それを他者に促したりすることは許可されない。

4. 「連絡先」、「写真」、ユーザデータにアクセスするその他の API から収集した情報を使用して、自身での使用またはサードパーティに販売したり配信したりすることを目的とした連絡先データベースを構築することは許可されない。また、分析や広告/マーケティングを目的として、ユーザのデバイスに他にどのような App がインストールされているかに関する情報を収集することも許可されない。
5. ユーザの「連絡先」や「写真」で収集した情報を使用して、第三者に連絡を取ることは許可されない。ただし、ユーザ本人が明示的かつ個別にリクエストする場合はこの限りではない。「すべて選択」のオプションを用意したり、デフォルトですべての連絡先が選択される仕様にしたりすることは許可されない。メッセージを送信する前に、どのようなメッセージが受信者に表示されるかをユーザに明確に伝える必要がある（メッセージの内容や、表示される送信者情報など）。
6. HomeKit API、HealthKit、Clinical Health Records API、MovementDisorder API、ClassKit、または深度測定ツールや（ARKit、Camera API、Photo API などの）フェイスマッピングツールで収集したデータを、サードパーティが行うものを含めて、マーケティング、広告、またはユーザベースのデータマイニングに使用することは許可されない。CallKit、HealthKit、ClassKit、ARKit の実装におけるベストプラクティスについて詳しくは、リンク先を確認すること。
7. Apple Pay を使用する App では、商品またはサービスの配信を円滑化または向上させる目的でのみ、Apple Pay を通じて取得したユーザデータをサードパーティと共有することができる。

参考資料

- App Store 法的事項 5.1.2 データの使用と共有

1.5.2.3 健康および健康に関する調査

健康、フィットネス、医療データは特に慎重に扱う必要があり、ユーザの個人情報の保護を徹底するために、この分野の App にはいくつかの追加ルールが適用される。

1. 健康、フィットネス、医療に関する調査のために収集されたデータを、許可を得た健康管理の向上または健康調査のため以外の目的（広告、マーケティング、またはその他のユーザベースのデータマイニングなど）で、App で使用またはサードパーティに公開することはできない。このデータは、Clinical Health Records API、HealthKit API、モーションとフィットネス、MovementDisorder API、健康関連の臨床調査から得られるデータなどを指す。ただし、ユーザに直接メリット（より低額な保険料など）を提供する場合には、メリットを提供する組織によって App が提出されており、データがサードパーティと共有されない限り、ユーザの健康やフィットネスに関するデータを App で使用することができる。その際、そのデバイスから収集される健康に関するデータについて具体的に明示する必要がある。
2. App の使用によって、HealthKit またはその他の健康調査 App や健康管理 App に虚偽のデータまたは誤ったデータが書き込まれないようにすること。また、個人の健康情報を iCloud に保存することはできない。
3. 健康に関する臨床調査を実施する App では、参加者本人、未成年の場合は親または保護者から同意を得る必要がある。このような同意には、(a) 調査の性質、目的、期間、(b) 手順、患者へのリスクおよ

び利点、(c) データの機密性と扱いに関する情報（サードパーティとの共有を含む）、(d) 患者が質問がある場合の連絡先、(e) 辞退プロセスを含める必要がある。

4. 健康関連の臨床調査を実施する App は、独立した倫理審査委員会の適切な承認を得る必要がある。承認を受けた証拠を要望に応じて提示していただく必要がある。

参考資料

- App Store 法的事項 5.1.3 健康および健康に関する調査

1.5.2.4 子どもに関する配慮

多くの理由から、子どもの個人データを扱う場合は厳重な注意が求められる。児童オンラインプライバシー保護法 (COPPA) や EU 一般 Data protection 規則 (GDPR) のような法律、およびその他の適用される規制または法律をすべて慎重に確認すること。

App では、これらの法律に準拠する目的でのみ生年月日や保護者の連絡先を要求することができる。ただし、ユーザの年齢に関係なく、その App がユーザにとって有益な機能またはエンターテイメントの価値を提供するものである必要がある。

主に子どもを対象とした App には、サードパーティ製の分析機能またはサードパーティ製の広告を組み込むことはできない。これにより、子ども達にとって、より安全な環境を提供することができます。限られたケースでは、サードパーティ製の分析機能およびサードパーティ製の広告が許可される場合もある。この場合は、そのサービスがガイドラインの 1.3 で定められている規約に準拠することが条件となる。

さらに、「子ども向け」カテゴリの App または未成年の個人情報（名前、住所、メールアドレス、位置情報、写真、ビデオ、絵、チャット対応の可否、その他の個人データ、上記の任意の組み合わせによる永続的識別子など）を収集、送信、共有する機能を持つ App にはプライバシーポリシーを設定し、子どものプライバシー保護法に関するすべての適用法に準拠する必要がある。「子ども向け」カテゴリの App のペアレンタルゲートと、プライバシー保護法に基づく個人データ収集への保護者の同意は、通常同じではないことに注意すること。

ガイドラインの 2.3.8 で定められている点として、App のメタデータに「子ども用」といった用語を使用できるのは、「子ども向け」カテゴリの App に限られる。「子ども向け」カテゴリに属さない App では、App 名、サブタイトル、アイコン、スクリーンショット、説明に、App の主な対象ユーザが子どもであることを暗示的に表す用語を含めることはできない。

参考資料

- App Store 法的事項 5.1.4 子どもに関する配慮

1.5.2.5 位置情報サービス

App で位置情報サービスを使用できるのは、位置情報サービスが App の機能またはサービスと直接関連する場合のみである。位置情報ベースの API は、救急サービスまたは自動車、飛行機、その他デバイスの自律制御のために使用することはできない。ただし、軽量のドローンや玩具などの小型デバイス、またはリモートコントロールの自動車警報システムなどは除く。位置情報を収集、送信、使用する際は、事前にユーザに通知し、同意を得る必要がある。App で位置情報サービスを使用する場合は、App におけるこのサービスの目的を説明する必要がある。説明に関するベストプラクティスについては、「[Human Interface Guidelines](#)」を参照すること。

参考資料

- [App Store 法的事項 5.1.5 位置情報サービス](#)

1.5.3 知的財産に関する App Store での規則

App には自分で作成したコンテンツ、または使用許可を取得したコンテンツのみを使用すること。許可なくコンテンツを使用した場合は App を削除する。もちろん、他のデベロッパが他者の作品を許可なく使用した場合はその App が削除される。ご自分の知的財産権が App Store の他のデベロッパに侵害されているおそれがある場合は、[Web フォーム](#)から申し立てを行うこと。国や地域によって法律は異なりますが、少なくとも以下の各項に留意する必要がある。

1. 全般商標、著作権取得済みの作品、特許取得済みのアイデアなどの保護されたサードパーティ製の素材を App で許可なく使用することはできない。また、誤解を招く、虚偽の、または模倣の描写、名前、メタデータを App バンドルやデベロッパ名に含めることは許可されない。App は、知的財産権およびその他の関連する権利を所有する、またはそのライセンスを受けている個人または法人によって提出される必要がある。
2. サードパーティのサイトおよびサービス App がサードパーティのサービスのコンテンツを使用、アクセス、または表示する場合、あるいはそのコンテンツへのアクセスを収益化する場合、当該サービスの利用規約に従って特別の許可を得る必要がある。また、要望に応じて承認書類を提示していただく必要がある。
3. オーディオおよびビデオのダウンロード App を使って違法なファイル共有を助長したり、コンテンツの供給元（Apple Music、YouTube、SoundCloud、Vimeo など）の明示的な承認を得ることなく、そのコンテンツを保存、変換、またはダウンロードする機能を App に搭載したりすることはできない。オーディオおよびビデオコンテンツのストリーミングも利用規約に違反する可能性があるため、App でこれらのサービスにアクセスする前に必ず確認すること。要望に応じて書類を提出していただく必要がある。
4. **Apple の承認** Apple が App の開発元またはサプライヤーであると示唆または暗示することは許可されない。また、Apple が App の品質または機能に関して推奨していると示唆または暗示しないこと。App が「スタッフのおすすめ」に選ばれた場合、バッジが自動的に適用される。
5. **Apple 製品**既存の Apple 製品、既存のインターフェース（Finder など）、既存の App（App Store、iTunes Store、メッセージなど）、Apple の既存広告などとの混同を招くような、類似した App を開発することは許可されない。サードパーティ製のキーボードやステッカーパックを含め、App や Extension に Apple の絵文字を含めることはできない。iTunes と Apple Music の音楽プレビューを娛樂的な目的で

(フォトコレージュの BGM、ゲームのサウンドトラックなど)、またはその他の承認されていない目的で使用することは許可されない。iTunes や Apple Music の音楽プレビューを使用する場合は、iTunes や Apple Music の該当曲のリンクを表示する必要がある。App でアクティビティリングを表示する場合は、アクティビティコントロールに類似する方法で、ムーブ、エクササイズ、スタンドのデータを表示することはできない。アクティビティリングの使用に関して詳しくは、「Human Interface Guidelines」を参照すること。App で Apple Weather のデータを表示する場合は、WeatherKit ドキュメントのアトリビューション要件に従う必要がある。

参考資料

- App Store 法的事項 5.2 知的財産

2

データストレージとプライバシー要件

2.1 MSTG-STORAGE-1

個人識別情報、ユーザ資格情報、暗号化キーなどの機密データを格納するために、システムの資格情報保存機能を使用している。

ローカルストレージに保存される機密データは、可能な限り少なくする必要がある。しかし、ほとんどの実用的なシナリオでは、少なくとも一部のユーザデータを保存する必要がある。幸いなことに、iOS はセキュアストレージ API を提供しており、開発者はすべての iOS デバイスに搭載されている暗号ハードウェアを使用することができる。これらの API を正しく使用すれば、ハードウェアに支えられた 256 ビット AES 暗号化によって、機密データやファイルを安全に保護することができる。

2.1.1 データ保護 API

アプリケーション開発者は、iOS Data Protection API を活用して、フラッシュメモリに保存されたユーザデータに対する詳細なアクセス制御を実装することができる。この API は、iPhone 5S で導入された Secure Enclave Processor (SEP) 上に構築されている。SEP は、データ保護とキー管理のための暗号化操作を提供するコプロセッサである。デバイス固有の Hardware Key であるデバイス UID (Unique ID) を Secure Enclave に埋め込むことで、OS カーネルが侵害された場合でもデータ保護の完全性を確保することができる。

データ保護アーキテクチャは、キーの階層構造に基づいている。UID とユーザの Passcode Key (PBKDF2 アルゴリズムによってユーザのパスフレーズから導き出される) は、この階層の最上位に位置する。これらは、異なるデバイスの状態（例：デバイスのロック/ロック解除）に関連付けられた、いわゆる Class Key の「ロック解除」に使用できる。

iOS のファイルシステムに保存されるすべてのファイルは、File Metadata に含まれるファイルごとのキーで暗号化される。メタデータは File System Key で暗号化され、ファイル作成時にアプリが選択した保護クラスに対応する Class Key でラップされる。

次の図は、iOS Data Protection Key Hierarchy を示している。

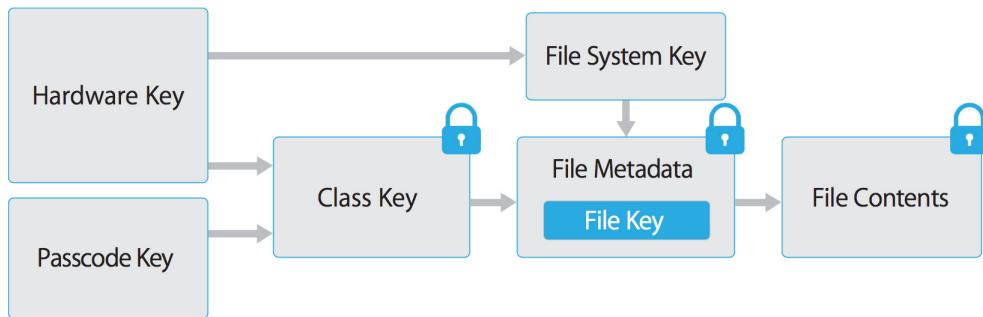


図 2.1.1.1 キーの階層構造に基づくデータ保護のフロー

ファイルは、4つの異なる保護クラスのいずれかに割り当てることができる。これらの保護クラスについては、[iOS Security Guide](#) を参照すること。

- Complete Protection (NSFileProtectionComplete) : ユーザパスコードとデバイス UID から派生したキーが、この Class Key を保護する。派生したキーはデバイスがロックされた直後にメモリから消去されるため、ユーザがデバイスのロックを解除するまでデータにアクセスできなくなる。
 - Protected Unless Open (NSFileProtectionCompleteUnlessOpen) : この保護クラスは Complete Protection と似ているが、ロック解除時にファイルが開かれると、ユーザがデバイスをロックしても、アプリはファイルへのアクセスを継続することができる。この保護クラスは、メールの添付ファイルをバックグラウンドでダウンロードする場合などに使用される。
 - Protected Until First User Authentication (NSFileProtectionCompleteUntilFirstUserAuthentication) : ユーザがデバイス起動後初めてロックを解除すると同時に、ファイルにアクセスできるようになる。その後、ユーザがデバイスをロックし、Class Key がメモリから削除されない場合でも、ファイルにアクセスできる。
 - No Protection (NSFileProtectionNone) : この保護 Class Key は、UID のみで保護される。Class Key は "Effaceable Storage" に保存される。これは、iOS デバイスのフラッシュメモリの領域で、少量のデータの保存が可能である。この保護クラスは、高速なリモートワイプ（Class Key を即座に削除し、データにアクセスできなくなる）のために存在する。

`NSFileProtectionNone` を除くすべての Class Key は、デバイスの UID とユーザのパスコードから派生したキーで暗号化されている。その結果、復号はデバイス自身でのみ可能であり、正しいパスコードが必要となる。

iOS 7 以降、デフォルトのデータ保護クラスは "Protected Until First User Authentication"。

參考資料

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

ルールブック

- *iOS Data Protection API* を活用して、フラッシュメモリに保存されたユーザデータにアクセス制御を実装する（必須）

2.1.2 Keychain

iOS の Keychain は暗号鍵やセッショントークンなどの短く機密性の高いデータを安全に保管するために使用される。これは Keychain API を介してのみアクセスできる SQLite データベースとして実装されている。

macOS では、すべてのユーザアプリケーションが好きなだけ Keychain を作成することができ、すべてのログインアカウントが独自の Keychain を持つことができる。

iOS の Keychain の構造は異なっており、すべてのアプリで使用できる Keychain は 1 つだけである。[項目](#)へのアクセスは、属性 `kSecAttrAccessGroup` のアクセスグループ機能を介して、同じ開発者によって署名されたアプリ間でアクセスを共有することができる。Keychain へのアクセスは `securityd` デーモンにより管理され、アプリの `Keychain-access-groups`, `application-identifier`, `application-group` エンタイトルメントに従ってアクセスを許可する。

Keychain API には、主に以下の操作が存在する。

- `SecItemAdd`
- `SecItemUpdate`
- `SecItemCopyMatching`
- `SecItemDelete`

Keychain に格納されるデータは、ファイルの暗号化に使用されるクラス構造と同様のクラス構造によって保護されている。Keychain に追加されたアイテムは、バイナリ plist としてエンコードされ、Galois/Counter Mode (GCM) でアイテムごとに 128 ビットの AES キーを使用して暗号化される。より大きなサイズのデータは Keychain に直接保存されないことに注意する。それが Data Protection API の目的である。`SecItemAdd` または `SecItemUpdate` のコールで、`kSecAttrAccessible` キーを設定することにより、Keychain アイテムのデータ保護を設定することができる。[kSecAttrAccessible](#) の設定可能なアクセシビリティ値は、以下の Keychain Data Protection クラスである。

- `kSecAttrAccessibleAlways`: Keychain アイテムのデータはデバイスがロックされているかどうかにかかわらず常にアクセスできる。なお、この項目は現在非推奨。
- `kSecAttrAccessibleAlwaysThisDeviceOnly`: Keychain アイテムのデータはデバイスがロックされているかどうかにかかわらず常にアクセスできる。データは iCloud やローカルのバックアップには含まれない。なお、この項目は現在非推奨。
- `kSecAttrAccessibleAfterFirstUnlock`: ユーザがデバイスのロックを一度解除するまで、再起動後に Keychain アイテムのデータにアクセスすることはできない。
- `kSecAttrAccessibleAfterFirstUnlockThisDeviceOnly`: ユーザがデバイスのロックを一度解除するまで、再起動後に Keychain アイテムのデータにアクセスすることはできない。この属性を持つアイテムは、新しいデバイスに移行されない。したがって、別のデバイスのバックアップから復元した後、これらのアイテムは存在しない。
- `kSecAttrAccessibleWhenUnlocked`: Keychain アイテムのデータはユーザによりデバイスがロック解除されている間のみアクセスできる。

- kSecAttrAccessibleWhenUnlockedThisDeviceOnly: Keychain アイテムのデータはユーザによりデバイスがロック解除されている間のみアクセスできる。データは iCloud やローカルのバックアップには含まれない。
- kSecAttrAccessibleWhenPasscodeSetThisDeviceOnly: Keychain のデータはデバイスがロック解除されている場合のみアクセスできる。この保護クラスは、デバイスにパスコードが設定されている場合のみ使用できる。データは iCloud やローカルのバックアップには含まれない。

AccessControlFlags は、ユーザがキーを認証するためのメカニズムを定義する (SecAccessControlCreateFlags):

- kSecAccessControlDevicePasscode: パスコードを介してアイテムにアクセスする。
- kSecAccessControlBiometryAny: Touch ID に登録された指紋のいずれかを介してアイテムにアクセスする。指紋を追加または削除してもアイテムは無効にならない。
- kSecAccessControlBiometryCurrentSet: Touch ID に登録された指紋のいずれかを介してアイテムにアクセスする。指紋を追加または削除すると、アイテムが無効になる。
- kSecAccessControlUserPresence: 登録済みの指紋 (Touch ID を使用) のいずれか、またはデフォルトのパスコードを使用してアイテムにアクセスする。

Touch ID によって (kSecAccessControlBiometryAny または kSecAccessControlBiometryCurrentSet を介して) 保護されたキーは、Secure Enclave によって保護されることに注意する。Keychain はトークンのみを保持し、実際のキーは保持しない。キーは Secure Enclave に存在する。

iOS 9 以降では、Secure Enclave で ECC ベースの署名操作を実行できる。そのシナリオでは、秘密キーと暗号化操作は Secure Enclave 内に存在する。ECC キーの作成については、「静的解析」を参照すること。iOS 9 は 256 ビット ECC のみをサポートする。さらに、公開キーは Secure Enclave に保存できないため、Keychain に保存する必要がある。キーを作成したら、kSecAttrKeyType を使用して、キーを使用するアルゴリズムのタイプを指定できる。

これらのメカニズムを使用する場合は、パスコードが設定されているかどうかをテストすることが推奨される。iOS8 では、kSecAttrAccessibleWhenPasscodeSetThisDeviceOnly 属性によって保護された Keychain 内のアイテムから読み書きできるかどうかを確認する必要がある。iOS9 以降では LAContext を使用して、ロック画面が設定されているかどうかを確認できる。

Swift での例 :

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

Objective-C での例:

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

(次のページに続く)

(前のページからの続き)

```

    }
}

```

Keychain データの永続性

iOS では、アプリケーションがアンインストールされると、ワイプされるアプリケーションサンドボックスによって保存されたデータとは異なり、アプリケーションによって使用される Keychain データはデバイスによって保持される。ユーザが工場出荷時設定へのリセットを実行せずにデバイスを販売した場合、デバイスの購入者は、以前のユーザが使用していたのと同じアプリケーションを再インストールすることで、以前のユーザのアプリケーションアカウントとデータにアクセスできる可能性がある。これを実行するのに技術的な能力は必要ない。

iOS アプリケーションを評価するときは、Keychain データの永続性を探す必要がある。これは通常、アプリケーションを使用して Keychain に保存できるサンプルデータを生成し、アプリケーションをアンインストールした後再インストールし、アプリケーションのインストール間でデータが保持されているかどうかを確認することによって行われる。objection (runtime mobile exploration toolkit) を使用して、Keychain データをダンプする。次の objection コマンドは、この手順を示す。

```

...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            None     ↳
↳ Data
-----
↳ -----
↳ -----
2020-02-11 13:26:52 +0000 WhenUnlocked      None     Password ↳
↳ keychainValue   com.highaltitudehacks.DVIAswiftv2.develop ↳
↳ mysecretpass123

```

アプリケーションがアンインストールされたときにデータを強制的に消去するために開発者が使用できる iOS API は存在しない。代わりに、開発者は次の手順を実行して、アプリケーションのインストール間で Keychain データが保持されないようにする必要がある。

- インストール後にアプリケーションを初めて起動するときに、アプリケーションに関連付けられているすべての Keychain データを消去する。これにより、デバイスの 2 番目のユーザが前のユーザのアカウントに誤ってアクセスするのを防ぐことができる。次の Swift の例は、この消去手順の基本的なデモである。

```

let userDefaults = UserDefaults.standard

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

    // Update the flag indicator
    userDefaults.set(true, forKey: "hasRunBefore")

```

(次のページに続く)

(前のページからの続き)

```
userDefaults.synchronize() // Forces the app to update UserDefaults
}
```

- iOS アプリケーションのログアウト機能を開発する時は、アカウント ログアウトの一部として Keychain データが消去されることを確認する。これにより、ユーザはアプリケーションをアンインストールする前にアカウントをクリアできる。

静的解析

iOS アプリのソースコードにアクセスできる場合、アプリ全体で保存および処理される機密データを特定する。これには、パスワード、シークレットキー、個人を特定できる情報（PII）などが含まれるが、業界の規制、法律、会社のポリシーによって機密であると特定された他のデータも含まれる可能性がある。以下に挙げるローカルストレージ API のいずれかを介して保存されているデータを探す。

機密性の高いデータは、適切な保護がないまま保存されることがないように注意する。たとえば、認証トークンを暗号化せずに NSUserDefaults に保存しない。また、暗号化 key を .plist ファイルに保存したり、コード内の文字列としてハードコードしたり、予測可能な難読化関数や安定した属性に基づく key 導出関数を使って生成したりすることは避ける。

機密性の高いデータは、Keychain API (セキュア・エンクレーブ内に保存される) を使用するか、エンベロープ暗号化を使って暗号化して保存する必要がある。エンベロープ暗号化、またはキーラッピングは、対称型暗号化を使用してキーの材料をカプセル化する暗号構造である。データ暗号化 key (DEK) は、Keychain に安全に保管されなければならない暗号化 key (KEK) で暗号化することができる。暗号化された DEK は、NSUserDefaults に保存するか、ファイルに書き込むことができる。必要なときに、アプリケーションは KEK を読み取り、DEK を復号化する。暗号キーの暗号化について詳しくは、[OWASP Cryptographic Storage Cheat Sheet](#) を参照する。

Keychain

暗号化は、秘密鍵が安全な設定で Keychain に保存されるように実装する必要があり、理想的には kSecAttrAccessibleWhenPasscodeSetThisDeviceOnly を使用する。これにより、ハードウェアでバックアップされたストレージ機構を使用することが保証される。AccessControlFlags が、Keychain 内のキーのセキュリティポリシーに従って設定されていることを確認する。

Keychain を使用してデータを保存、更新、削除する一般的な例は、Apple の公式文書に記載されている。また、Apple の公式ドキュメントには、Touch ID とパスコードで保護された key を使用する例も含まれている。

キーを作成するために使用できる Swift のサンプルコードである (kSecAttrTokenID が String: kSecAttrTokenIDSecureEnclave であることに注目すること: これは Secure Enclave を直接使用したいことを示す。):

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

(次のページに続く)

(前のページからの続き)

```
// 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, //なお、kSecAttrKeyTypeEC は現在非推奨。
    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) //なお、← SecKeyGeneratePair は現在非推奨。

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

iOS アプリに安全でないデータストレージがないか確認する場合は、デフォルトではデータを暗号化しない以降のデータ保存方法を考慮すること。

参考資料

- 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

ルールブック

- *Keychain Services API* を使用してセキュアに値を保存する（必須）

2.1.3 NSUserDefaults

`NSUserDefaults` クラスは、デフォルトシステムと連携するためのプログラムインターフェースを提供する。デフォルトシステムでは、アプリケーションがユーザ設定に従って動作をカスタマイズできる。`NSUserDefaults` によって保存されたデータは、アプリケーションバンドルで表示することができる。このクラスは、plist ファイルにデータを保存するが、少量のデータで使用することを想定している。

参考資料

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

ルールブック

- データ量による保存先の選定（推奨）

2.1.4 ファイルシステム

2.1.4.1 NSData

NSData は静的なデータオブジェクトを作成し、NSMutableData は動的なデータオブジェクトを作成する。NSData と NSMutableData は通常、データストレージとして使用されるが、データオブジェクトに含まれるデータをアプリケーション間でコピーしたり移動したりする分散オブジェクトアプリケーションにも有効である。以下は NSData オブジェクトの書き込みに使用されるメソッドである。

- NSDataWritingWithoutOverwriting
- NSDataWritingFileProtectionNone
- NSDataWritingFileProtectionComplete
- NSDataWritingFileProtectionCompleteUnlessOpen
- NSDataWritingFileProtectionCompleteUntilFirstUserAuthentication

参考資料

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

ルールブック

- iOS Data Protection API を活用して、フラッシュメモリに保存されたユーザデータにアクセス制御を実装する（必須）

2.1.4.2 writeToFile

データを NSData クラスの一部として保存する。

参考資料

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

2.1.4.3 ファイルパスを管理する

NSSearchPathForDirectoriesInDomains, NSTemporaryDirectory: ファイルパスの管理に使用する。

参考資料

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

2.1.4.4 NSFileManager

NSFileManager を使用することで、ファイルシステムの内容を調べたり変更したりすることができる。また、createFileAtPath を使用すると、ファイルを作成し、そのファイルに書き込むことができる。

次の例は、FileManager クラスを使用して完全な暗号化ファイルを作成する方法を示している。詳細については、Apple Developer Documentation "Encrypting Your App's Files" を参照すること。

Swift での例：

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

Objective-C での例:

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

参考資料

- owasp-mastg Testing Local Data Storage (MSTG-STORAGE-1 and MSTG-STORAGE-2) File system
- ルールブック
- *FileManager* クラスでの暗号化の使用方法（必須）

2.1.5 CoreData/SQLite データベース

2.1.5.1 Core Data

Core Data は、アプリケーション内のオブジェクトの Model Layer を管理するためのフレームワークである。オブジェクトのライフサイクルやオブジェクトグラフの管理に関する一般的なタスク（永続性を含む）に対する一般的な自動化されたソリューションを提供する。Core Data は永続的な保存場所として SQLite を使用できるが、フレームワーク自体はデータベースではない。

Core Data はデフォルトではデータを暗号化しない。MITRE Corporation の研究プロジェクト（iMAS）の一環として、オープンソースの iOS セキュリティコントロールに焦点を当て、Core Data に追加の暗号化レイヤーを追加することができるようになった。詳しくは GitHub Repo を参照する。

参考資料

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

2.1.5.2 SQLite Databases

アプリで SQLite を使用する場合は、SQLite 3 ライブラリをアプリに追加する必要がある。このライブラリは、SQLite コマンドの API を提供する C++ ラッパーである。

参考資料

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

2.1.5.3 Firebase Real-time Databases

Firebase は 15 を超える製品を備えた開発プラットフォームであり、そのうちの 1 つが Firebase Real-time Database である。アプリケーション開発者は、NoSQL クラウドでホストされたデータベースと、データの保存および同期をするために活用できる。データは JSON として保存され、接続されたすべてのクライアントとリアルタイムで同期され、アプリケーションがオフラインになっても引き続き利用できる。

誤った設定の Firebase インスタンスは、以下のネットワークコールを行うことで特定することができる。

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

firebaseProjectName は、プロパティリスト (.plist) ファイルから取得することができる。たとえば、PROJECT_ID key には、対応する Firebase プロジェクト名が GoogleService-Info.plist ファイルに格納されている。

また、アナリストは、以下に示すように、上記のタスクを自動化する Python スクリプトである Firebase Scanner を使用することもできる。

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

参考資料

- 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 と Realm Swift は Apple から提供されてはいないが、着目する必要がある。設定で暗号化が有効になっていない限り、すべて暗号化されずに保存される。

次の例は、Realm データベースで暗号化を使用する方法を示す。

```
// 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 {
```

(次のページに続く)

(前のページからの続き)

```
// If the encryption key is wrong, `error` will say that it's an invalid database
fatalError("Error opening realm: \\" + error + "\\")
}
```

参考資料

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

2.1.5.5 Couchbase Lite Databases

Couchbase Lite は、同期可能な軽量の組み込みドキュメント指向（NoSQL）データベースエンジンである。iOS および macOS 用にネイティブにコンパイルされる。

参考資料

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

2.1.5.6 YapDatabase

YapDatabase は、SQLite の上に構築されたキー/バリューの保管場所である。

参考資料

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

動的解析

iOS のネイティブ機能を利用せずに、機密情報（認証情報やキーなど）が安全でない状態で保存されているかどうかを判断する方法の 1 つが、アプリのデータディレクトリを分析することである。特定の機能がトリガーされた後にのみ、アプリが機密データを保存する可能性があるため、データを分析される前にすべてのアプリ機能をトリガーすることが重要である。その後、一般的なキーワードとアプリ固有のデータに従って、データダンプの静的解析を行うことができる。

以下の手順で、ジェイルブレイクされた iOS デバイスでアプリケーションがローカルにデータを保存する方法を確認することができる。

1. 潜在的な機密データを保存する機能をトリガーする。
2. iOS デバイスに接続し、その Bundle ディレクトリに移動する（iOS バージョン 8.0 以降に適用される）：/var/mobile/Containers/Data/Application/\$APP_ID/
3. 保存したデータで grep を実行する。例：grep -iRn "USERID"
4. 機密データがプレーンテキストで保存されている場合、アプリはこのテストに失敗する。

iMazing などのサードパーティアプリケーションを使用することで、ジェイルブレイクされていない iOS デバイスのアプリのデータディレクトリを解析することができる。

1. 潜在的に機密性の高いデータを格納する機能をトリガーする。

2. iOS デバイスをホストコンピュータに接続し、iMazing を起動する。
3. "Apps" を選択し、目的の iOS アプリケーションを右クリックし、"Extract App" を選択する。
4. 出力ディレクトリに移動し、\$APP_NAME.imazingを見つける。それを \$APP_NAME.zip にリネームする。
5. ZIP ファイルを解凍する。その後、アプリケーションデータを解析することができる。

iMazing のようなツールは、デバイスから直接データをコピーしないことに注意する。それらは作成したバックアップからデータを抽出しようとする。そのため、iOS デバイスに保存されている全てのアプリデータを取得することは不可能となる。全てのフォルダがバックアップに含まれるわけでは無い。ジェイルブレイクしたデバイスを使用するか、アプリを Frida で再パッケージ化し、Objection のようなツールを使用してすべてのデータとファイルにアクセスする。

Frida ライブラリをアプリに追加し、「Dynamic Analysis on Non-Jailbroken Devices」(「Tampering and Reverse Engineering on iOS」の章より) の説明に従って再パッケージ化した場合、「Basic Security Testing on iOS」の章の、「Host-Device Data Transfer」のセクションで説明したように、objection を使ってアプリのデータディレクトリから直接ファイルを転送したり、objection でファイルを読み取ったりすることが可能である。

Keychain のコンテンツは、動的解析中にダンプすることができる。ジェイルブレイクしたデバイスでは、"Basic Security Testing on iOS" の章で説明されているように、Keychain dumper を使用することができる。

Keychain のコンテンツは動的解析中に吐き出すことができる。ジェイルブレイクしたデバイスでは、「Basic Security Testing on iOS」の章で説明したように、Keychain dumper を使用することができる。

Keychain ファイルのパスは以下の通り。

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

ジェイルブレイクされていないデバイスでは、Objection を使用して、アプリによって作成および保存された Keychain アイテムをダンプことができる。

参考資料

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

2.1.6 Xcode と iOS シミュレータによる動的解析

このテストは、Xcode と iOS シミュレータが必要なため、macOS でのみ利用可能である。

ローカルストレージをテストし、その中にどのようなデータが保存されているかを確認するために、iOS デバイスを持っていることは必須ではない。ソースコードと Xcode にアクセスすることで、アプリをビルドして iOS シミュレータにデプロイすることができる。iOS シミュレータの現在のデバイスのファイルシステムは、

~/Library/Developer/CoreSimulator/Devices.

で利用できる。

iOS シミュレータでアプリが起動したら、以下のコマンドで開始された最新のシミュレーターのディレクトリに移動できる

```
$ 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
```

上記のコマンドは、開始された最新のシミュレーターの UUID を自動的に検出する。ここで、アプリ名やアプリのキーワードを grep する必要がある。これにより、アプリの UUID が表示される。

```
grep -iRn keyword .
```

そして、アプリのファイルシステムの変更を監視・検証し、アプリの使用中にファイル内に機密情報が保存されているいかどうかを調査することができる。

参考資料

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

2.1.7 Objection を用いた動的解析

Objection runtime mobile exploration toolkit を使用することで、アプリケーションのデータ保存メカニズムによって引き起こされた脆弱性を見つけることができる。Objection はジェイルブレイクデバイスでなくても使用できるが、iOS アプリケーションにパッチを適用する必要がある。

参考資料

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

2.1.7.1 Keychain の読み取り

Objection を使って Keychain を読み取るには、以下のコマンドを実行する。

```
...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 WhenUnlocked      None    Password
→ keychainValue   com.highaltitudehacks.DVIAswiftv2.develop
→ mysecretpass123
```

参考資料

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

2.1.7.2 Binary Cookies の検索

iOS アプリケーションは、アプリケーションサンドボックスに Binary Cookie ファイルを保存することがよくある。Cookie は、アプリケーションの WebView の Cookie データを含むバイナリファイルである。objection を使って、これらのファイルを JSON 形式に変換し、データを検査することができる。

```
...itudehacks.DVIA swiftv2.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"  
  }  
]
```

参考資料

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

2.1.7.3 プロパティリストファイルの検索

iOS アプリケーションは、多くの場合、アプリケーションサンドボックスと IPA パッケージの両方に保存されているプロパティリスト (plist) ファイルにデータを保存する。これらのファイルには、ユーザ名やパスワードなどの機密情報が含まれていることがある。したがって、iOS 評価の際には、これらのファイルの内容を検査する必要がある。plist ファイルを検査するには、ios plist cat plistFileName.plist コマンドを使用する。

userInfo.plist というファイルを見つけるには、env コマンドを使用する。アプリケーションの Library,Caches,Documents の各ディレクトリの場所が出力される。

```
...itudehacks.DVIA swiftv2.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
```

(次のページに続く)

(前のページからの続き)

```
DocumentDirectory /var/mobile/Containers/Data/Application/264C23B8-07B5-4B5D-8701-
↳ C020C301C151/Documents
LibraryDirectory /var/mobile/Containers/Data/Application/264C23B8-07B5-4B5D-8701-
↳ C020C301C151/Library
```

参考資料

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

Documents ディレクトリに移動し、ls を使用してすべてのファイルをリストアップする。

```
...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.
↳ 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
```

ios plist cat コマンドを実行し、userInfo.plist ファイルの内容を確認する。

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

2.1.7.4 SQLite データベースの検索

iOS アプリケーションは通常、SQLite データベースを使用して、アプリケーションで必要なデータを保存する。テスターは、これらのファイルのデータ保護値とその内容に機密データがないか確認する必要がある。Objection には、SQLite データベースと対話するためのモジュールが含まれている。これは、スキーマとそのテーブルをダンプし、レコードをクエリすることができる。

```
...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       |
+-----+-----+-----+-----+
1 row in set
Time: 0.013s
```

参考資料

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

2.1.7.5 キャッシュのデータベースを検索

デフォルトでは、NSURLSession は HTTP リクエストやレスポンスなどのデータを Cache.db データベースに保存する。トークン、ユーザ名、その他の機密情報がキャッシュされている場合、このデータベースには機密情報が含まれている可能性がある。キャッシュされた情報を見つけるには、アプリのデータディレクトリ (/var/mobile/Containers/Data/Application/<UUID>) を開き、/Library/Caches/<Bundle Identifier>に移動する。WebKit のキャッシュも Cache.db ファイルに保存されている。Objection では、通常の SQLite データベースであるため、sqlite connect Cache.db というコマンドでデータベースを開いてやり取りすることができる。

リクエストやレスポンスに機密情報が含まれている可能性があるため、このデータのキャッシュを無効にすることを推奨する。以下のリストは、これを実現するためのさまざまな方法を示している。

1. ログアウト後にキャッシュされたレスポンスを削除することをお勧めする。これは Apple が提供する `removeAllCachedResponses` というメソッドで行うことができる。このメソッドは以下のように呼び出す。

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

このメソッドは、Cache.db ファイルからすべてのキャッシュされた要求と応答を削除する。

2. Cookie の利点を使用する必要がない場合は、NSURLSession の `.ephemeral` 設定プロパティを使用することを推奨する。これにより、Cookie とキャッシュの保存が無効になる。

以下は Apple のドキュメント：

ephemeral session 設定オブジェクトは対応するセッションオブジェクトがキャッシュ、資格情報ストア、またはセッション関連のデータをディスクに保存しないことを除いて、デフォルトセッション構成(デフォルトを参照)に似ている。その代わり、セッション関連のデータは RAM に保存される。ephemeral session がディスクにデータを書き込むのは、URL の内容をファイルに書き込むように指示したときだけである。

3. また、Cache Policy を `.notAllowed` に設定することで、Cache を無効にすることもできる。これは、Cache をメモリやディスク上にキャッシュを保存することを無効にする。

参考資料

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

ルールブック

- キャッシュの仕様と Cache.db に保存されるデータを確認する (必須)

2.1.8 ルールブック

1. *iOS Data Protection API* を活用して、フラッシュメモリに保存されたユーザデータにアクセス制御を実装する（必須）
2. インストール後の初回起動の *keychain* のデータを消去する（必須）
3. アカウントログアウト時の *keychain* データを消去する（必須）
4. *Keychain Services API* を使用してセキュアに値を保存する（必須）
5. データ量による保存先の選定（推奨）
6. *FileManager* クラスでの暗号化の使用方法（必須）
7. キャッシュの仕様と *Cache.db* に保存されるデータを確認する（必須）

2.1.8.1 iOS Data Protection API を活用して、フラッシュメモリに保存されたユーザデータにアクセス制御を実装する（必須）

iOS Data Protection API は大きなデータを扱う場合のファイルを利用することになる。iOS では Data Protection というファイル保護の仕組みが備わっている。iOS では、下記のデータ保護属性が定義されている。

- NSFileProtectionNone
- NSFileProtectionComplete
- NSFileProtectionCompleteUnlessOpen
- NSFileProtectionCompleteUntilFirstUserAuthentication

ファイル作成時にデフォルトでどの属性が適用されるかは、フレームワークや API によって異なる。例えば `Data#write(to: options: error:)` では特に指定が無い場合は `NSFileProtectionComplete` が適用されるが、`NSPersistentStoreCoordinator` で永続化した Core Data のデータベースファイルは `NSFileProtectionCompleteUntilFirstUserAuthentication` が適用される。

下記は `Data` クラス生成の一例。

```
import UIKit

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

        let text: String = "Hello."

        // 文字列を Data 型に変換。
        guard let data = text.data(using: .utf8) else {
            return nil
        }

        return data
    }
}
```

(次のページに続く)

(前のページからの続き)

```

    }
}

```

保護属性 Complete Protection (NSFileProtectionComplete)

デバイスがロック解除されている間のみファイルにアクセスできるようにするオプション。

システムはファイルを暗号化された形式で保存し、アプリはデバイスのロックが解除されている間のみファイルの読み取りまたは書き込みを行うことができる。それ以外の場合は、アプリがファイルを読み書きしようとしても失敗する。

以下サンプルコードは、オプション「保護属性 Complete Protection」の使用例。

```

import UIKit

class ViewController: UIViewController {

    func writeFileDataCompleteFileProtection(data: Data, fileURL: URL) {
        do {
            // data バッファーの内容を URL の箇所に書き込む。
            // デバイスがロック解除されている間のみファイルにアクセスできるようにするオプション。
            try data.write(to: fileURL, options: .completeFileProtection)

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

```

保護属性 Protected Unless Open (NSFileProtectionCompleteUnlessOpen)

デバイスのロックが解除されているか、ファイルが既に開いているときにファイルにアクセスできるようにするオプション。

デバイスがロックされている場合、アプリはファイルを開いて読み書きすることはできないが、新しいファイルを作成できる。デバイスがロックされているときにファイルが開いている場合、アプリは開いているファイルを読み書きできる。

以下サンプルコードは、オプション「保護属性 Protected Unless Open」の使用例。

```

import UIKit

class ViewController: UIViewController {

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

(次のページに続く)

(前のページからの続き)

```
// data バッファーの内容を URL の箇所に書き込む。
// デバイスのロックが解除されているか、ファイルが既に開いているときにファイルにアクセスできる
// ようにするオプション。
try data.write(to: fileURL, options: .completeFileProtectionUnlessOpen)

} catch {
    // exception write failed.
}

}

}
```

保護属性 Protected Until First User Authentication (NSFileProtectionCompleteUntilFirstUserAuthentication)

ユーザが最初にデバイスのロックを解除した後にファイルにアクセスできるようにするオプション。

デバイスのロックが解除されている間、アプリはファイルの読み取りまたは書き込みを行うことができるが、アプリによって開かれているファイルは Complete Protection と同等の保護状態となる。

以下サンプルコードは、オプション「保護属性 Protected Until First User Authentication」の使用例。

```
import UIKit

class ViewController: UIViewController {

    func writeFileDataCompleteFileProtectionUntilFirstUserAuthentication(data:_
→Data, fileURL: URL) {

        do {

            // data バッファーの内容を URL の箇所に書き込む。
            // ユーザが最初にデバイスのロックを解除した後にファイルにアクセスできるようにするオプション
            try data.write(to: fileURL, options: .
→completeFileProtectionUntilFirstUserAuthentication)

        } catch {
            // exception write failed.
        }

    }
}
```

保護属性 No Protection (NSFileProtectionNone)

ファイルを書き出すときに保護クラスキーを UID でのみ作成する安全でない（保護なし）オプション。

システムはファイルを暗号化するが保護クラスキーが UID でのみ作成されるので結果として安全ではなく、アプリは常にこのファイルにアクセスできる。

以下サンプルコードは、オプション「保護属性 No Protection」の使用例。

```

import UIKit

class ViewController: UIViewController {

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

        do {

            // data バッファーの内容を URL の箇所に書き込む。
            // ファイルを書き出すときにファイルを暗号化しないオプション。
            try data.write(to: fileURL, options: .noFileProtection)

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

```

適切な保護レベルが設定

ファイルの保護レベルによっては、ファイルのコンテンツの読み書きの際に、続いてユーザがデバイスをロックすると、読み書きできなくなることがある。アプリがファイルに確実にアクセスできるようにするには、以下に従うこと。

- アクセス先のファイルに「完全な保護」レベルを割り当てるのは、アプリがフォアグラウンドにあるときだけにする。(Complete Protection)
- アプリが位置情報の更新処理などのバックグラウンド機能に対応している場合は、バックグラウンドにあるときにアクセス可能なように、別の保護レベルをファイルに割り当てる。(Protected Unless Open)

ユーザの個人情報を含むファイルや、ユーザが直接作成したファイルには常に最高レベルの保護が必要。

参考資料

- Encrypting Your App's Files

これに違反する場合、以下の可能性がある。

- 適切な保護レベルが設定されていない場合、意図しないファイルへのアクセスが実施される可能性がある。

2.1.8.2 インストール後の初回起動の keychain のデータを消去する（必須）

インストール後にアプリケーションを初めて起動するときに、アプリケーションに関連付けられているすべての Keychain データを消去する。これにより、デバイスの 2 番目のユーザが前のユーザのアカウントに誤ってアクセスするのを防ぐことができる。

次の Swift の例は、この消去手順の基本的なデモである。

```
let userDefaults = UserDefaults.standard
```

(次のページに続く)

(前のページからの続き)

```
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
}
```

これに違反する場合、以下の可能性がある。

- デバイスの 2 番目のユーザが前のユーザのアカウントに誤ってアクセスする可能性がある。

2.1.8.3 アカウントログアウト時の keychain データを消去する（必須）

iOS アプリケーションのログアウト機能を開発する時は、アカウントログアウトの一部として Keychain データが消去されることを確認する。これにより、ユーザはアプリケーションをアンインストールする前にアカウントをクリアできる。

これに違反する場合、以下の可能性がある。

- 別のユーザにより Keychain データを使用される可能性がある。

2.1.8.4 Keychain Services API を使用してセキュアに値を保存する（必須）

Keychain Service API は、Keychain と呼ばれる暗号化されたデータベースにデータを保存することができる。これは、機密性の高いデータ（パスワード、クレジットカード情報、Certificate, Key, and Trust Services で管理する暗号化キーと証明書など）を保存するのに適している。

Keychain アイテムは、表のキー（メタデータ）と行ごとのキー（秘密鍵）という 2 つの異なる AES-256-GCM キーを使用して暗号化される。Keychain のメタデータ（kSecValue 以外のすべての属性）は検索速度を高めるためにメタデータキーで暗号化され、秘密値（kSecValueData）は対応する秘密鍵で暗号化される。メタデータキーは Secure Enclave によって保護されているが、高速化するためにアプリケーションプロセッサにキャッシュされる。秘密鍵は常に、Secure Enclave を介してやりとりする必要がある。

Secure Enclave を使用するためにはハードウェアサポートが必要である。（A7 以降のプロセッサを搭載した iOS デバイス）

Keychain は SQLite データベース形式で実装され、ファイルシステムに保存される。データベースは 1 つしかなく、各プロセスやアプリがアクセスできる Keychain アイテムは、securityd デーモンによって決定される。

Keychain 項目の共有は、同じデベロッパによるアプリ間でのみ可能となる。Keychain アイテムを共有するために、他社製アプリはアプリケーショングループの Apple Developer Program を通じて割り当てられたプレフィックスに基づくアクセスグループを使用する。プレフィックス要件とアプリケーショングループの一意性は、コード署名、プロビジョニングプロファイル、および Apple Developer Program によって実現される。

Keychain データは、ファイルのデータ保護で使用されるものと同様のクラス構造で保護され、ファイルのデータ保護の各クラスと同じように動作する。

利用できるタイミング	ファイルのデータ保護	Keychain のデータ保護
ロック解除時	NSFileProtectionComplete	kSecAttrAccessibleWhenUnlocked
ロック中	NSFileProtectionCompleteUnlessOpen	不可
初回ロック解除後	NSFileProtectionCompleteUntilFirstUserAuthentication	kSecAttrAccessibleAfterFirstUnlock
常時	NSFileProtectionNone	kSecAttrAccessibleAlways (なお、この項目は現在非推奨。)
パスコードが有効なとき	不可	kSecAttrAccessibleWhenPasscodeSetThisDeviceOnly

バックグラウンド更新サービスを利用するアプリは、アクセスする必要がある Keychain アイテムに kSecAttrAccessibleAfterFirstUnlock を使用する。SecAttrAccessibleWhenPasscodeSetThisDeviceOnly の動作は kSecAttrAccessibleWhenUnlocked と同じだが、利用できるのはデバイスにパスコードが構成されているときのみ。

以下サンプルコードは、keychain からデータを検索し、データが存在しなければ keychain へ保存する処理の一例。

```
import Foundation

class Keychain {

    let group: String = "group_1" // グループ
    var id: String = "id"

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

    func addKeychain(data: Data) {

        // API を実行する際の引数設定
        let dic: [String: Any] = [kSecClass as String: kSecClassGenericPassword, // ← クラス: パスワードクラス
            kSecAttrGeneric as String: group, // 自由項目
            kSecAttrAccount as String: id, // アカウント (ログイン ID)
            kSecValueData as String: data, // パスワードなどの保存情報
            kSecAttrService as String: "key"] // サービス名

        // 検索用の Dictionary
        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]

        // keychain からデータを検索
        findKeychainItem(attributes: search as CFDictionary, { status, item in
            // 変更後の処理
        })
    }
}
```

(次のページに続く)

(前のページからの続き)

```

switch status {
    case errSecItemNotFound: // keychain にデータが存在しない
        // keychain に保存
        _ = SecItemAdd(dic as CFDictionary, nil)
    default:
        break
}
}

}

/// keychain からアイテムが存在するかを検索する
/// - Parameters:
///   - attrs: 検索用のデータ
///   - completion: 検索結果
func findKeychainItem(attributes attrs: CFDictionary, _ completion: @escaping ↴(OSStatus, CFTypeRef?) -> Void) {

    //呼び出しがスレッドをブロックするため、メインスレッドから呼び出された場合にアプリの UI がハングする可能性がある。
    //別スレッドで実行することを推奨。
    backgroundQueue.async {
        var item: CFTypeRef?
        let result = SecItemCopyMatching(attrs, &item)
        completion(result, item)
    }
}

}

```

keychain への保存 SecItemAdd

SecItemAdd により Keychain に 1 つ以上のアイテムを追加する。一度に複数のアイテムを Keychain に追加するには、ディクショナリの配列を値として持つディクショナリのキーを使用する。(パスワードの項目のみ対象外)

パフォーマンスに関する考慮事項として、スレッドをブロックするため、メインスレッドから呼び出された場合にアプリの UI がハングする可能性がある。メインスレッド以外での実行を推奨する。

以下サンプルコードは、keychain への保存処理の一例。

```

import Foundation

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

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

```

(次のページに続く)

(前のページからの続き)

```

    }
}
}
```

keychain の更新 SecItemUpdate

SecItemUpdate により検索クエリに一致するアイテムを更新する。値を変更する属性と新しい値を含むディクショナリを渡すことで該当のアイテムを更新する。

検索クエリの作成方法については、SecItemCopyMatching 関数を参照

パフォーマンスに関する考慮事項として、スレッドをブロックするため、メインスレッドから呼び出された場合にアプリの UI がハングする可能性がある。メインスレッド以外での実行を推奨する。

以下サンプルコードは、keychain の更新処理の一例。

```

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,           // 自由項目 (グループ)
                 kSecAttrAccount as String: loginId,       // アカウント (ログイン ID など)
                 kSecValueData as String: data             // 保存情報
                ] as CFDictionary

            // keychain の Matching 実行
            let matchingStatus = SecItemCopyMatching(attrs, nil)

            // Matching 結果のステータス
            switch matchingStatus {
            case errSecSuccess: // 検索成功
                // 更新
                let result = SecItemUpdate(attrs, updateAttrs)
                completion(result)
            default:
                debugPrint("failed updateKeychainItem status. (\(matchingStatus))")
                completion(matchingStatus)
            }
        }
    }
}
```

keychain のデータ検索 SecItemCopyMatching

SecItemCopyMatching により検索クエリに一致する 1 つ以上の Keychain アイテムを返すか、特定の Keychain

アイテムの属性をコピーする。

デフォルトでは、この関数は Keychain 内のアイテムを検索する。(kSecReturnData の指定と同じ) Keychain 検索を特定の Keychain または複数の Keychain に限定するには、検索キーを指定し、項目タイプの項目を含むオブジェクトをその値としてディクショナリを渡す。

検索結果の返却方法の指定変更

- kSecReturnData アイテムのデータをオブジェクトの形式(1項目)を返す。(デフォルト)
- kSecReturnAttributes アイテムのデータをディクショナリの形式(複数項目)を返す。
- kSecReturnRef アイテムの参照を返す。
- kSecReturnPersistentRef アイテムの参照を返す。通常の参照とは異なり、永続的な参照はディスクに保存されるか、プロセス間で使用する参照を返す。

パフォーマンスに関する考慮事項として、スレッドをブロックするため、メインスレッドから呼び出された場合にアプリの UI がハングする可能性がある。メインスレッド以外での実行を推奨する。

以下サンプルコードは、keychain のデータ検索処理の一例。

```
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 のデータ削除 SecItemDelete

SecItemDelete により検索クエリに一致するアイテムを削除する。

パフォーマンスに関する考慮事項として、スレッドをブロックするため、メインスレッドから呼び出された場合にアプリの UI がハングする可能性がある。メインスレッド以外での実行を推奨する。

以下サンプルコードは、keychain のデータ削除処理の一例。

```

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)
        }
    }
}

```

keychain ヘアクセスできる条件の設定

Keychain アイテムのデータに対してアプリからいつアクセスできるのかを設定することができる。

デフォルトでは、デバイスのロックが解除されている場合にのみ Keychain アイテムにアクセスできる。ただし、パスコードを設定していないデバイスを考慮し、パスコードで保護されたデバイスからのみアイテムにアクセスできるようにする必要がある場合がある。または、アクセス制限を緩和して、デバイスがロックされているときにバックグラウンドプロセスからアイテムにアクセス可能にすることもできる。

Keychain サービスは、ユーザからの入力と組み合わせて、デバイスの状態に従って個々の Keychain アイテムのアクセシビリティを管理する方法を提供している。

アイテムの作成時にアイテムの属性を設定することにより、デバイスの状態に関連する Keychain アイテムへのアプリのアクセスを制御する。アクセシビリティ値の設定の仕方として、既定のアクセシビリティを使用するクエリディクショナリは次のように指定する。

以下サンプルコードは、既定のアクセシビリティを使用するクエリディクショナリの指定方法の一例。

```

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)
    }
}

```

(次のページに続く)

(前のページからの続き)

{
}

アクセシビリティに設定できる値は次のとおり。

kSecAttrAccessibleAlways

Keychain アイテムのデータは、デバイスがロックされているかどうかに関係なく、常にアクセスできる。暗号化されたバックアップを使用すると、この属性を持つアイテムが新しいデバイスに移行する。なお、この項目は現在非推奨。

kSecAttrAccessibleAlwaysThisDeviceOnly

Keychain アイテムのデータは、デバイスがロックされているかどうかに関係なく、常にアクセスできる。この属性を持つアイテムは、新しいデバイスに移行されない。したがって、別のデバイスのバックアップから復元では、これらのアイテムは存在しなくなる。なお、この項目は現在非推奨。

kSecAttrAccessibleAfterFirstUnlock

ユーザがデバイスのロックを一度解除するまで、再起動後に Keychain アイテムのデータにアクセスすることはできない。最初のロック解除後、次の再起動までデータにアクセスできる。これは、バックグラウンドアプリケーションからアクセスする必要がある場合に推奨される。暗号化されたバックアップを使用すると、この属性を持つアイテムが新しいデバイスに移行される。

kSecAttrAccessibleAfterFirstUnlockThisDeviceOnly

ユーザがデバイスのロックを一度解除するまで、再起動後に Keychain アイテムのデータにアクセスすることはできない。最初のロック解除後、次の再起動までデータにアクセスできる。これは、バックグラウンドアプリケーションからアクセスする必要がある場合に推奨される。この属性を持つアイテムは、新しいデバイスに移行されない。したがって、別のデバイスのバックアップから復元では、これらのアイテムは存在しなくなる。

kSecAttrAccessibleWhenUnlocked

Keychain アイテムのデータは、ユーザがデバイスのロックを解除している間のみアクセスできる。これは、アプリケーションがフォアグラウンドにある間のみアクセス可能にする必要があるアイテムに推奨される。暗号化されたバックアップを使用すると、この属性を持つアイテムが新しいデバイスに移行される。

これは、アクセシビリティ定数を明示的に設定せずに追加された Keychain アイテムのデフォルト値。

kSecAttrAccessibleWhenUnlockedThisDeviceOnly

Keychain アイテムのデータは、ユーザがデバイスのロックを解除している間のみアクセスできる。これは、アプリケーションがフォアグラウンドにある間のみアクセス可能にする必要があるアイテムに推奨される。この属性を持つアイテムは、新しいデバイスに移行されない。したがって、別のデバイスのバックアップから復元では、これらのアイテムは存在しなくなる。

kSecAttrAccessibleWhenPasscodeSetThisDeviceOnly

Keychain 内のデータは、デバイスのロックが解除されている場合にのみアクセスできる。デバイスにパスコー

ドが設定されている場合にのみ使用できる。これは、アプリケーションがフォアグラウンドにある間のみアクセス可能にする必要があるアイテムに推奨される。この属性を持つアイテムが新しいデバイスに移行されることはない。バックアップが新しいデバイスに復元されると、これらのアイテムが失われる。パスコードのないデバイスでは、このクラスにアイテムを保存できない。デバイスパスコードを無効にすると、このクラスのすべてのアイテムが削除される。

システムがそのアイテムを可能な限り保護できるように、アプリに適した最も制限の厳しいオプションを常に使用すること。iCloud に保存したくない非常に機密性の高いデータの場合は、kSecAttrAccessibleWhenPasscodeSetThisDeviceOnly を使用する。

また、Keychain を使用する場合は以下ルールに従うこと。

- Keychain を使用する場合、AccessControlFlags の設定が、Keychain 内のキーのセキュリティポリシーに従っていること
- Keychain を使用する場合、設定するアクセシビリティ値は、kSecAttrAccessibleWhenPasscodeSetThisDeviceOnly を推奨

アクセシビリティでは、デバイス状態によるアクセス制限をしていたが、デバイスがロック解除されている場合にのみアクセスを許可することは、すべての場合において十分に安全であるとは限らない。アプリで銀行口座を直接制御できる場合は、Keychain からログイン資格情報を取得する直前に、もう一度許可されたユーザの確認を行うべきである。これにより、ユーザがデバイスをロック解除状態で他の人に渡した場合でも、アカウントを保護できる。

Keychain アイテムを作成するときに、属性の値としてインスタンス（SecAccessControlCreateFlags）を指定することにより、この制限を追加できる。その場合にアクセスコントロールを使用する。

以下サンプルコードは、アクセスコントロール内のアクセス制御フラグの指定方法の一例。

```
import Foundation

class KeyChainSample {

    func flagsSample() {
        // flags (kSecAccessControlBiometryAny |  

        ↪kSecAccessControlApplicationPassword)  

        // (Swift のビット集合型である OptionSet 使用)  

        let flags: SecAccessControlCreateFlags = [.biometryAny, .  

        ↪applicationPassword]  

        let access = SecAccessControlCreateWithFlags(nil,  

        ↪kSecAttrAccessibleWhenPasscodeSetThisDeviceOnly,  

        flags, // 「.biometryAny」として直接  

接指定も可能  

nil)  

    }
}
```

以下サンプルコードは、アクセスコントロールを使用してクエリディクショナリを指定する方法の一例。

```
import UIKit

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

        // Create SecAccessControl Flags
        guard let access = SecAccessControlCreateWithFlags(nil, // Use the
        ↪default allocator.

        ↪kSecAttrAccessibleWhenUnlocked, // accessible
            .userPresence,
            nil
        ) else {
            //異常処理
            return
        }

        // kSecAttrAccessible と kSecAttrAccessControl は共存できない。どちらかを指定する。
        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)
    }
}
```

アクセスコントロールに設定できるオプションは次のとおり。

kSecAccessControlDevicePasscode

パスコードでアイテムにアクセスするための制約。

kSecAccessControlBiometryAny

登録されている指の Touch ID または Face ID でアイテムにアクセスするための制約。Touch ID が利用可能で、少なくとも 1 本の指で登録されている必要がある。または、Face ID が利用可能で登録されている必要がある。指が追加または削除された場合は Touch ID で、ユーザが再登録された場合は Face ID でアイテムに引き続きアクセスできる。

kSecAccessControlBiometryCurrentSet

現在登録されている指の Touch ID を使用して、または現在登録されているユーザの Face ID からアイテムにアクセスするための制約。Touch ID が利用可能で、少なくとも 1 本の指で登録されているか、Face ID が利用可能で登録されている必要がある。Touch ID に指を追加または削除した場合、またはユーザが Face ID に再登録した場合、アイテムは無効になる。

kSecAccessControlUserPresence

バイオメトリまたはパスコードのいずれかを使用してアイテムにアクセスするための制約。バイオメトリは、利用可能または登録されている必要はない。指が追加または削除された場合でも Touch ID によって、またはユーザが再登録された場合は Face ID によってアイテムにアクセスできる。

さらに、特定のデバイスの状態とユーザの存在を要求するだけでなく、アプリケーション固有のパスワードを要求できる。これは、デバイスのロックを解除するパスコードとは異なるため、特定の Keychain アイテムを明確に保護することが可能である。

これを行うには、アクセスコントロール内のアクセス制御フラグを定義するときにフラグを含める。

以下サンプルコードは、アクセスコントロール内のアクセス制御フラグ定義時にフラグを含め、クエリディクショナリを指定する方法の一例。

```
import UIKit

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

        // Create SecAccessControl Flags

        // flags (kSecAccessControlBiometryAny | ←
        ↪kSecAccessControlApplicationPassword)
        // (Swift のビット集合型である OptionSet 使用)
        let flags: SecAccessControlCreateFlags = [.biometryAny, .←
        ↪applicationPassword]
        guard let access = SecAccessControlCreateWithFlags(nil, ←
        ↪
        ↪kSecAttrAccessibleWhenPasscodeSetThisDeviceOnly,
                                            flags,
                                            nil
        ) else {
            //異常処理
            return
        }
        // kSecAttrAccessible と kSecAttrAccessControl は共存できない。どちらかを指定する。
        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)
    }
}
```

このフラグを追加すると、アイテムの作成時にユーザにパスワードの入力を求めるプロンプトが表示され、ア

アイテムを取得する前に再度パスワードが要求される。アイテムは、他の条件が満たされているかどうかに関係なく、ユーザがパスワードを正常に入力した場合にのみ取得できる。

また、アクセスコントロールを複数設定した場合に、すべての制約をパスする必要がある（`kSecAccessControlAnd`）もしくは、制約の1つをパスすれば条件を満たすことになる（`kSecAccessControlOr`）条件を追加することが可能。ビット演算でアクセス制御フラグに追加する。

以下サンプルコードは、アクセスコントロールのビット指定方法の一例。

```
import Foundation

class KeyChainTest {

    func flagsTest() {
        // アクセス コントロールのビット指定 (Swift のビット集合型である OptionSet 使用)
        let flags: SecAccessControlCreateFlags = [.biometryAny, .devicePasscode, .
        ↪or]

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

kSecAccessControlApplicationPassword

これは、制約に加えて指定できる。

kSecAccessControlPrivateKeyUsage

このオプションは、他の任意のアクセス制御フラグと組み合わせることができる。

通常、この制約は、キーペアを作成し、秘密鍵をデバイスのセキュアエンクレーブの内部に保存する際に使用する（`kSecAttrTokenID` 属性を `kSecAttrTokenIDSecureEnclave` の値で指定することにより）。これにより、秘密鍵は、`SecKeyRawSign` および `SecKeyRawVerify` 関数の呼び出しによってセキュアエンクレーブの内部で行われる署名および検証タスクで使用できるようになる。セキュアエンクレーブの外部で鍵ペアを生成する際にこの制約を使用しようとすると、失敗する。同様に、セキュアエンクレーブの内部でこの制約を使用せずに生成された秘密鍵でブロックに署名しようとすると、失敗する。

独自の暗号化キーを作成する場合のアルゴリズムを kSecAttrKeyType で指定する

暗号化キーは、セキュリティを強化するために特殊な数学演算で他のデータと組み合わせるバイト文字列。多くの場合、ID、証明書、または Keychain からキーを取得する。ただし、独自のキーを作成する必要がある場合もある。

非対称鍵の作成

非対称暗号化鍵は、一緒に生成される公開キーと秘密キーで構成される。公開キーは自由に配布できるが、秘密キーは非公開にする。作成した秘密キーを Keychain に保管することができる。

以下サンプルコードは、属性ディクショナリを作成して、非対称鍵を作成する方法の一例。

```
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 アルゴリズム指定
                                         kSecAttrKeySizeInBits as String: 2048, // 2048bit
                                         kSecPrivateKeyAttrs as String: [
                                             kSecAttrIsPermanent as String:true,
                                             kSecAttrApplicationTag as String: tag]]

        // 秘密鍵作成
        var error: Unmanaged<CFError>?
        guard let privateKey = SecKeyCreateRandomKey(attributes as CFDictionary, &
            error) else {
            // error 处理
            return
        }

        // 公開鍵作成
        let generatedPublicKey = SecKeyCopyPublicKey(privateKey)

    }
}
```

上記の例は 2048 ビットの RSA キーを示しているが、他のオプションも利用できる。（サービスによって適宜どのアルゴリズムを指定するかを決める）

アルゴリズム キータイプの値

- kSecAttrKeyTypeDSA
 - DSA アルゴリズム。
- kSecAttrKeyTypeAES
 - AES アルゴリズム。
- kSecAttrKeyTypeCAST
 - CAST アルゴリズム。
- kSecAttrKeyTypeECSECPrimeRandom
 - 楕円曲線アルゴリズム。

属性ディクショナリに指定する tag データは、逆引き DNS 表記を使用して文字列から作成されるが、一意のタグであれば何でも構わない。同一のタグが付けられた複数のキーを生成しないように注意。他の検索可能な特性が異なる場合を除き、検索中に区別するのができなくなる。代わりに、キー生成操作ごとに一意のタグを

使用するか、特定のタグを使用している古いキーを削除してから、そのタグを使用して新しいキーを作成する必要がある。

秘密キーを Keychain に保存するには、kSecAttrIsPermanent に true を指定して秘密キーを作成する。

以下サンプルコードは、kSecAttrIsPermanent に true を指定して秘密キーを作成する方法の一例。

```
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], // 秘密鍵(privateKey)をKeychainに格納するフラグ
                                         kSecAttrApplicationTag as String: tag]
    }
}
```

Objective-C では、これらの主要な参照が完了したら、関連するメモリを解放する責任がある。

以下サンプルコードは、キーに関連するメモリを解放処理の一例。

```
#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];
    [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);
}
```

(次のページに続く)

(前のページからの続き)

```

// Add persistent version of the key to system keychain
NSData *data = [[NSData alloc] initWithBase64EncodedString:@"ssssssssss" ↴
options:NSDataBase64DecodingIgnoreUnknownCharacters];
[publicKey setObject:(__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

```

参考資料

- KeychainServices
- Keychain のデータ保護
- Secure Enclave
- SecItemAdd
- SecItemUpdate
- SecItemCopyMatching
- SecItemDelete
- Restricting Keychain Item Accessibility
- 新しい暗号鍵の生成
- Keychain へのキーの保存

これに違反する場合、以下の可能性がある。

- 機密データが漏洩する可能性がある。

2.1.8.5 データ量による保存先の選定（推奨）

データを保存する際にそのデータ量の多寡によって保存する場所を設計することが重要になる。例えば NSUserDefaults によって保存されたデータは、アプリケーションバンドルで表示することができる。このクラスは、plist ファイルにデータを保存するが、少量のデータを想定されている。

iOS 13 で NSUserDefaults に保存できるサイズとして 4194304 bytes の制限が追加され、それ以上を保存する際に警告が出るようになったことがある (Apple の公式な通告は存在しない)。今後の OS バージョンによっては保存できなくなる可能性があるため、使用するデータ量にて保存箇所を変更することを推奨する

大容量の場合

- サーバ
- ローカル DB

少量の場合

- UserDefaults

これに注意しない場合、以下の可能性がある。

- 保存先を適切に設計できていない場合、データの保存取得が出来なくなる可能性がある。

2.1.8.6 FileManager クラスでの暗号化の使用方法（必須）

FileManager クラスでファイル作成する際に、保護属性を指定することで暗号化された形式でディスクにファイルを保存することができる。適切な保護属性を設定すること。以下のサンプルコードは保護属性 complete を指定する例。

Swift での例：

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

Objective-C での例:

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

保護属性については以下のルールブックの記載を参照。

- *iOS Data Protection API を活用して、フラッシュメモリに保存されたユーザデータにアクセス制御を実装する（必須）*

これに違反する場合、以下の可能性がある。

- ファイルが流出した際に非暗号化のデータが簡単に判明されてしまう。

2.1.8.7 キャッシュの仕様と Cache.db に保存されるデータを確認する（必須）

iOS アプリケーションで Web サーバと通信する際、サーバ側でキャッシュコントロールがされない場合、Cache.db ファイルに Web サーバから返ってきたレスポンスがキャッシュとして保存される。そのため、iOS アプリケーションでキャッシュコントロールをしていないと機密情報が保存されてしまう。Cache.db にキャッシュとして保存されたファイルは平文で残るため、機密情報を含むキャッシュが保存されている場合、攻撃者にキャッシュを窃取される可能性がある。

キャッシュコントロールを行うには、いくつかの方法がある。適切なキャッシュコントロールをする必要がある。

1. キャッシュを使用している場合は、ログアウト後にキャッシュされたレスポンスを削除することをお勧めする。これは Apple が提供する `removeAllCachedResponses` というメソッドで行うことができる。なおこのメソッドは、Cache.db ファイルからすべてのキャッシュを削除する。
2. キャッシュを使用する必要がない場合は、`NSURLSession` の `.ephemeral` 設定プロパティを使用することを推奨する。これにより、該当するセッションについて Cookie とキャッシュがディスクに保存されなくなる。
3. または、Cache Policy を `.notAllowed` に設定することで、キャッシュを無効にすることもできる。この場合、メモリやディスク上にキャッシュが保存されなくなる。

これに違反する場合、以下の可能性がある。

- 適切なキャッシュコントロールが設定されていない場合、ユーザの個人情報が漏えいし悪用される可能性がある。

2.2 MSTG-STORAGE-2

機密データはアプリコンテナまたはシステムの資格情報保存機能の外部に保存されていない。

※ *MSTG-STORAGE-1* へ同一内容を記載しているため、本章への記載を省略。

2.3 MSTG-STORAGE-3

機密データはアプリケーションログに書き込まれていない。

2.3.1 ログ出力

モバイルデバイスにログファイルを作成する正当な理由はたくさんある。例えば、クラッシュやエラーを追跡するため（デバイスがオフラインの時にローカルに保存されることで、オンラインになった時にアプリ開発者に送信できる）や、使用統計を蓄積するためなど。ただし、クレジットカード番号やセッション IDなどの機密データを記録すると攻撃者や悪意のあるアプリケーションにデータが公開される可能性がある。ログファイルはさまざまな方法で作成される。以下のリストは iOS で利用できるメカニズムを示している。

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

以下のキーワードを使用して、アプリのソースコードに定義済みおよびカスタムのロギングステートメントがあるかどうかを確認する。

- 定義済み関数と組み込み関数の場合
 - NSLog
 - NSAssert
 - NSCAssert
 - fprintf
- カスタム機能の場合
 - Logging
 - Logfile

ルールブック

- 機密データがアプリケーションログを介して公開されないようにする（必須）

2.3.2 define による Debug ログ無効

`define` を使用して、開発およびデバッグ時に `NSLog` ステートメントを有効にし、ソフトウェアを出荷する前にそれらを無効にする。これは、適切な `PREFIX_HEADER (*.pch)` ファイルに次のコードを追加することで可能である。

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

参考資料

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

ルールブック

- 機密データがアプリケーションログを介して公開されないようにする（必須）

2.3.3 動的解析

「iOS Basic Security Testing」の章の、「Monitoring System Logs」セクションで、デバイスのログを確認するための様々な方法が説明されている。機密性の高いユーザの機密情報を入力するフィールドを表示する画面に移動する。

いずれかの方法を開始したら、入力フィールドに入力する。機密データが出力に表示される場合、アプリはこのテストに失敗する。

参考資料

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

2.3.4 ルールブック

1. 機密データがアプリケーションログを介して公開されないようにする（必須）

2.3.4.1 機密データがアプリケーションログを介して公開されないようにする（必須）

ログ出力をする場合は、出力内容に機密情報が含まれていないことを確認する必要がある。

一般的なログ出力用クラスとしては、以下が存在する。

Log

- print
- NSLog

print 項目のテキスト表現を標準出力に書き込む関数。Xcode (Run) をした際に Xcode 内のコンソールに表示する。

```
import Foundation

func examplePrint(message: String) {
    print(message)
}
```

NSLog エラーメッセージを Apple システムログ機能に記録する関数。Xcode (Run) をした際に Xcode 内のコンソールに表示する。Console.app にも NSLog で記述したログメッセージを出力する。

```
import Foundation

func exampleLog(message: String) {
    NSLog(message)
}
```

また、**define** を使用して、開発およびデバッグ時に NSLog ステートメントを有効できるが、ソフトウェアを出荷する前にそれらを無効にする必要がある。これは、適切な PREFIX_HEADER (*.pch) ファイルに次のコードを追加することで可能である。

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

これに違反する場合、以下の可能性がある。

- 機密データを記録すると攻撃者や悪意のあるアプリケーションにデータが公開される可能性がある。

2.4 MSTG-STORAGE-4

機密情報は、いくつかの手段で第三者に漏れる可能性がある。iOS では、通常、アプリに組み込まれたサードパーティサービスを介して行われる。

2.4.1 サードパーティサービスへのデータの共有

サードパーティサービスが提供する機能には、アプリ使用中のユーザの行動を監視するためのトラッキングサービス、バナー広告の販売、またはユーザエクスペリエンスの向上が含まれる。

欠点として、通常開発者は利用するサードパーティライブラリを介して実行されるコードの詳細を把握することができない。そのため、必要以上の情報をサービスに送信したり、機密情報を開示すべきではない。

サードパーティサービスの多くは、以下の 2 つの方法で実装されている。

- スタンドアローンライブラリを使用
- full SDK を使用

静的解析 サードパーティライブラリが提供する API 呼び出しと関数がベストプラクティスに従って使用されているかどうかを判断するには、ソースコードを確認し、アクセス許可を要求し、既知の脆弱性が存在しないかを確認する（「[サードパーティライブラリ](#)」を参照）。

サードパーティサービスに送信される全てのデータは、サードパーティがユーザアカウントを識別できるようになる PII（個人識別情報）の公開を防ぐために、匿名化する必要がある。その他のデータ（ユーザアカウントまたはセッションにマッピングできる ID など）をサードパーティに送信しないようにすること。

動的解析 機密情報が埋め込まれていないか、外部サービスへのすべてのリクエストを確認する。クライアントとサーバ間のトラフィックを傍受するには、[Burp Suite Professional](#) または [OWASP ZAP](#) を使用して中間者（MITM）攻撃を開始することにより、動的分析を実行できる。傍受プロキシを介してトラフィックをルーティングすることで、アプリとサーバの間を通過するトラフィックを傍受できる。主要な機能がホストされているサーバに直接送信されない全てのアプリリクエストは、トラッカーや広告サービスの PII などの機密情報についてチェックする必要がある。

参考資料

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

ルールブック

- サードパーティライブラリへ必要のない機密情報を共有しない（必須）
- 使用するサードパーティライブラリに既知の脆弱性が存在しないかを確認する（必須）
- サードパーティサービスに送信される全てのデータは匿名化する（必須）

2.4.2 ルールブック

1. サードパーティライブラリへ必要のない機密情報を共有しない（必須）
2. 使用するサードパーティライブラリに既知の脆弱性が存在しないかを確認する（必須）
3. サードパーティサービスに送信される全てのデータは匿名化する（必須）

2.4.2.1 サードパーティライブラリへ必要のない機密情報を共有しない（必須）

開発者は利用するサードパーティライブラリを介して実行されるコードの詳細を把握することができない。そのため、必要以上の情報をサービスに送信したり、機密情報を開示するべきではない。

機密情報が埋め込まれていないか、外部サービスへのすべてのリクエストを確認する。クライアントとサーバ間のトラフィックを傍受するには、[Burp Suite Professional](#) または [OWASP ZAP](#) を使用して中間者（MITM）攻撃を開始することにより、動的分析を実行できる。

これに違反する場合、以下の可能性がある。

- 知的財産 (IP) を危険にさらす可能性がある。

2.4.2.2 使用するサードパーティライブラリに既知の脆弱性が存在しないかを確認する（必須）

サードパーティライブラリには、脆弱性、互換性のないライセンス、悪意のあるコンテンツが含まれている可能性がある。

アプリが使用するライブラリに脆弱性がないことを確認するためには、[CocoaPods](#) や [Carthage](#) によってインストールされた依存関係を確認するのが最適である。

- Carthage はオープンソースで、Swift と Objective-C のパッケージに使用することができる。Swift で書かれ、分散化されており、Cartfile ファイルを使用してプロジェクトの依存関係をドキュメント化し管理する。
- CocoaPods はオープンソースで、Swift と Objective-C のパッケージに使用することができる。Ruby で書かれており、公開パッケージと非公開パッケージのための集中型パッケージレジストリを利用し、プロジェクトの依存関係をドキュメント化し管理するために Podfile ファイルを使用する。

これに違反する場合、以下の可能性がある。

- アプリケーションに不正なコードや脆弱性のあるコードが含まれており、悪用される可能性がある。
- サードパーティライブラリに含まれるライセンスにより、アプリのソースコードの展開を求められる可能性がある。

2.4.2.3 サードパーティサービスに送信される全てのデータは匿名化する（必須）

サードパーティサービスに送信される全てのデータは、サードパーティがユーザアカウントを識別できるようする PII（個人識別情報）の公開を防ぐために、匿名化する必要がある。

個人を特定できる情報（PII）とは、個人を特定するために使用できるあらゆるデータのことである。個人に直接または間接的に結びつくすべての情報が PII とみなされる。氏名、電子メールアドレス、電話番号、銀行口座番号、政府発行の ID 番号などはすべて PII の例である。

これに違反する場合、以下の可能性がある。

- 個人情報の漏洩やその他の攻撃の被害者となる可能性がある。

2.5 MSTG-STORAGE-5

機密データを処理するテキスト入力では、キーボードキャッシュが無効にされている。

2.5.1 キーボードの予測変換入力

キーボード入力を簡略化するためのオプションがいくつか用意されている。これらのオプションには、自動修正とスペルチェックが含まれる。ほとんどのキーボード入力は、デフォルトで /private/var/mobile/Library/Keyboard/dynamic-text.dat にキャッシュされる。

キーボードキャッシュには、`UITextInputTraits protocol` が使用されている。UITextField、UITextView、UISearchBar クラスは自動的にこのプロトコルをサポートしており、以下のプロパティを提供する。

- `var autocorrectionType: UITextAutocorrectionType`: `UITextAutocorrectionType` は、入力中に自動修正が有効かどうかを決定する。自動修正が有効な場合、テキストオブジェクトは未知の単語を追跡して適切な置換を提案し、ユーザが置換を上書きしない限り、入力されたテキストを自動的に置き換える。このプロパティのデフォルト値は `UITextAutocorrectionTypeDefault` で、ほとんどの入力メソッドで自動修正が有効になっている。
- `var secureTextEntry: BOOL` は、UITextField に対してテキストコピーとテキストキャッシュを無効にして、入力中のテキストを非表示にするかどうかを決定する。このプロパティのデフォルト値は NO である。

静的解析

- 次のような実装がないか、ソースコードから検索する。

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

- Xcode の Interface Builder で xib ファイルと storyboard ファイルを開き、該当するオブジェクトの Attributes Inspector で Secure Text Entry と Correction の状態を確認する。アプリケーションは、テキストフィールドに入力された機密情報がキャッシュされないようにする必要がある。キャッシュを防ぐには、目的の UITextField、UITextView、UISearchBar で `textObject.autocorrectionType =`

UITextFieldAutocorrectionTypeNo directive を使用して、プログラム的に無効にすることができる。PIN やパスワードなどマスクすべきデータについては、textObject.secureTextEntry を YES に設定する。

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

動的解析

ジェイルブレイクした iPhone がある場合は、以下の手順を実行すること。

1. iOS デバイスのキーボードキャッシュをリセットするには、「設定」→「一般」→「リセット」→「キーボードの変換学習をリセット」を開く。
2. アプリケーションを使用し、ユーザが機密データを入力できる機能性を確認する。
3. キーボードキャッシュファイル dynamic-text.dat を以下のディレクトリに Dump する（iOS のバージョンが 8.0 より前の場合は異なる可能性がある）。/private/var/mobile/Library/Keyboard/
4. ユーザ名、パスワード、電子メールアドレス、クレジットカード番号などの機密データを探す。キーボードキャッシュファイルを経由して機密データを取得できる場合、そのアプリはこのテストに不合格となる。

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

ジェイルブレイクした iPhone がない場合は、以下の手順を実行すること。

1. キーボードキャッシュをリセットする。
2. すべての機密データをキー入力する。
3. アプリを再度使用し、オートコレクトが以前に入力された機密情報を示唆しているかどうかを判断する。

参考資料

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

ルールブック

- テキストフィールドに入力された機密情報をキャッシュしない（必須）

2.5.2 ルールブック

1. テキストフィールドに入力された機密情報をキャッシュしない（必須）

2.5.2.1 テキストフィールドに入力された機密情報をキャッシュしない（必須）

キーボードキャッシュには、`UITextInputTraits protocol` が使用されている。`UITextField`、`UITextView`、`UISearchBar` クラスは自動的にこのプロトコルをサポートしており、以下のプロパティを提供する。

- `var autocorrectionType: UITextAutocorrectionType`: `UITextAutocorrectionType` は、入力中に自動修正が有効かどうかを決定する。自動修正が有効な場合、テキストオブジェクトは未知の単語を追跡して適切な置換を提案し、ユーザが置換を上書きしない限り、入力されたテキストを自動的に置き換える。このプロパティのデフォルト値は `UITextAutocorrectionTypeDefault` で、ほとんどの入力メソッドで自動修正が有効になっている。
- `var secureTextEntry: BOOL` は、`UITextField` に対してテキストコピーとテキストキャッシュを無効にして、入力中のテキストを非表示にするかどうかを決定する。このプロパティのデフォルト値は `NO` である。

アプリケーションは、テキストフィールドに入力された機密情報がキャッシュされないようにする必要がある。キャッシュを防ぐには、目的の `UITextFields`、`UITextViews`、`UISearchBars` で `textObject.autocorrectionType = UITextAutocorrectionTypeNo` directive を使用して、プログラム的に無効にすることができる。PIN やパスワードなどマスクすべきデータについては、`textObject.secureTextEntry` を `YES` に設定する。

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

これに違反する場合、以下の可能性がある。

- 機密情報を盗まれて不正利用されてしまう危険性があります。

2.6 MSTG-STORAGE-6

機密データは IPC メカニズムを介して公開されていない。

2.6.1 XPC サービス

XPC は基本的なプロセス間通信を提供する構造化された非同期ライブラリである。`launchd` によって管理されている。これは iOS における IPC の最も安全で柔軟な実装であり、推奨される方法である。可能な限り最も制限された環境で実行される。つまり、サンドボックス化され、ルート権限のエスカレーションがなく、ファイルシステムへのアクセスやネットワークへのアクセスも最小限に抑えられている。`XPC サービス` では、以下の 2 つの異なる API が使用される。

- `NSXPCConnection API`
- `XPC Services API`

静的解析

以下に iOS ソースコード内の IPC 実装を識別するために探す必要があるキーワードをまとめる。

XPC サービス

NSXPConnection API を実装するために、以下のクラスが使用できる。

- NSXPConnection
- NSXPCIInterface
- NSXPCListener
- NSXPCListenerEndpoint

接続のためのセキュリティ属性を設定することができる。属性は確認する必要がある。

XPC Services API (C 言語ベース) の Xcode プロジェクトに、以下の 2 つのファイルがあるか確認する。

- `xpc.h`
- `connection.h`

参考資料

- [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 ポート

すべての IPC 通信は、最終的に Mach Kernel API に依存する。**Mach ポート** はローカル通信(デバイス内通信)のみを可能にする。これらはネイティブに、または Core Foundation(CFMachPort) および Foundation(NSMachPort) wrapper を介して実装できる。

静的解析

以下に iOS ソースコード内の IPC 実装を識別するために探す必要があるキーワードをまとめる。

低レベル実装で検索すべきキーワード

- `mach_port_t`
- `mach_msg_*`

高レベル実装 (Core Foundation および Foundation ラッパー) で検索すべきキーワード

- `CFMachPort`
- `CFMessagePort`
- `NSMachPort`
- `NSMessagePort`

参考資料

- [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

`NSFileCoordinator` クラスは、ローカルファイルシステム上で利用可能なファイルを介してアプリとの間でデータを管理し、様々なプロセスに送信するために使用することができる。`NSFileCoordinator` のメソッドは同期的に実行されるため、実行が停止するまでコードはブロックされる。これは非同期ブロックのコールバックを待つ必要がないため便利だが、同時にメソッドが実行中のスレッドをブロックすることを意味する。

静的解析

以下に iOS ソースコード内の IPC 実装を識別するために探す必要があるキーワードをまとめます。

検索すべきキーワード

- `NSFileCoordinator`

参考資料

- 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

パスワードや PIN などの機密データは、ユーザインターフェースを介して公開されていない。

2.7.1 ユザインターフェースでの機密データの公開

アカウント登録や支払いなど、多くのアプリを利用する上で、機密情報の入力は不可欠な要素である。このデータは、クレジットカード情報やユーザアカウントのパスワードなどの金融情報である場合がある。入力中にアプリが適切にマスクしなければ、データが漏洩する可能性がある。

情報漏洩を防ぎ、`shoulder surfing` のようなリスクを軽減するため、明示的な要求（例：パスワードの入力）をされない限り、ユーザインターフェースを通じて機密データが公開されないことを確認する必要がある。必要なデータについては、平文の代わりにアスタリスクやドットを表示するなどして、適切にマスクする必要がある。

そのようなデータを表示、あるいは入力として受け取るすべての UI コンポーネントを注意深くレビューする。機密データの痕跡を探し、それをマスクするか完全に削除するかを評価する。

参考資料

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

2.7.2 入力テキスト

入力をマスクするテキストフィールドは、以下の方法で構成されている。

Storyboard

iOS プロジェクトの Storyboard で、機密データを取得するテキストフィールドの設定オプションに移動する。オプション「Secure Text Entry」が選択されていることを確認する。このオプションが有効な場合、テキストフィールドにはテキスト入力の代わりにドットが表示される。

ソースコード

テキストフィールドがソースコードで定義されている場合、オプション `isSecureTextEntry` が「`true`」に設定されていることを確認する。このオプションは、ドットを表示することでテキスト入力を非表示にする。

```
sensitiveTextField.isSecureTextEntry = true
```

アプリケーションが機密データをユーザインターフェースに漏洩しているかを判断するには、アプリケーションを実行し、そのような情報を表示するコンポーネント、または入力として受け取るコンポーネントを特定する。

アスタリスクやドットなどで情報がマスクされている場合、そのアプリケーションはユーザインターフェースを通じて機密データが漏洩することを防いでいる。

参考資料

- 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

ルールブック

- パスワード入力をマスクする（必須）

2.7.3 ルールブック

1. パスワード入力をマスクする（必須）

2.7.3.1 パスワード入力をマスクする（必須）

iOS の UITextField では、機密情報の入力に対してテキストを非表示（マスク）を行い、テキスト オブジェクトのコピーを禁止するプロパティ Secure Text Entry が存在する。

設定の仕方は、Storyboard 上の GUI にてチェックを入れる方法と、ソースコードにてプロパティを設定する方法がある。

Storyboard

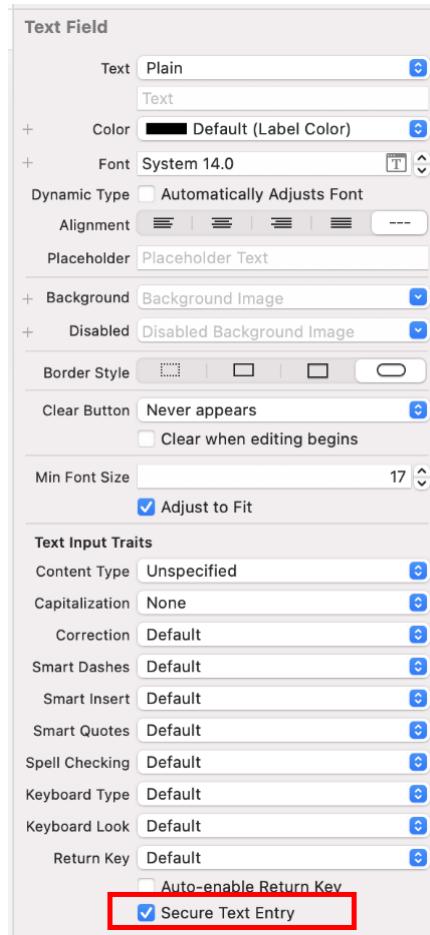


図 2.7.3.1.1 Storyboard Secure Text Entry の設定

ソースコード

```
import UIKit

class ViewController: UIViewController {

    @IBOutlet weak var passwordTextField: UITextField!

    override func viewDidLoad() {
        super.viewDidLoad()

        passwordTextField.isSecureTextEntry = true
    }
}
```

これに違反する場合、以下の可能性がある。

- 第三者に機密情報を読み取られる。

2.8 MSTG-STORAGE-12

アプリは処理される個人識別情報の種類をユーザに通知しており、同様にユーザがアプリを使用する際に従うべきセキュリティのベストプラクティスについて通知している。

2.8.1 アプリマーケットプレイスでのデータプライバシーに関するユーザ教育のテスト

現時点では、どのプライバシー関連情報が開発者によって開示されているかを知り、それが妥当であるかどうかを評価しようとしているだけである（アクセス許可をテストするときと同様）。

実際に収集または共有されている特定の情報を開発者が宣言していない可能性があるが、それはここでこのテストを拡張する別のトピックのためのものである。このテストの一環として、プライバシー違反の保証を提供することは想定されていない。

参考資料

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

2.8.2 静的解析

以下の手順で実行できる。

1. 対応するアプリマーケットプレイス（Google Play, App Store など）でアプリを検索する。
2. "Privacy Details" セクション（App Store）または "Safety Section" セクション（Google Play）に移動する。
3. 利用可能な情報があるかどうかを確認する。

開発者がアプリマーケットプレイスのガイドラインに従ってコンパイルし、必要なラベルと説明を含めた場合、テストは合格である。アプリマーケットプレイスから取得した情報を証拠として保存・提供し、後でそれを使用してプライバシーまたはデータ保護の潜在的な違反を評価できるようにする。

参考資料

- [owasp-mastg Testing User Education \(MSTG-STORAGE-12\) Static Analysis](#)

2.8.3 動的解析

オプションの手順として、このテストの一部として何らかの証拠を提供することもできる。例えば、iOS アプリをテストしている場合、アプリのアクティビティの記録を有効にして、写真、連絡先、カメラ、マイク、ネットワーク接続などのさまざまなリソースへの詳細なアプリアクセスを含むプライバシーレポートを簡単にエクスポートできる。

これを行うと、実際には他の MASVS カテゴリをテストする際に多くの利点がある。これは、[MASVS-NETWORK](#) でのネットワーク通信のテストや、[MASVS-PLATFORM](#) でのアプリのパーミッション

ンのテストに使用できる非常に役立つ情報を提供する。これらの他のカテゴリをテストしているときに、他のテストツールを使用して同様の測定を行った可能性がある。これをこのテストの証拠として提供することもできる。

理想的には、利用可能な情報を、アプリが実際に意図していることと比較する必要がある。ただし、リソースや自動化されたツールのサポートによっては、完了するまでに数日から数週間かかる可能性がある簡単なタスクではない。また、アプリの機能とコンテキストに大きく依存するため、理想的には、アプリ開発者と密接に連携するホワイトボックスセットアップで実行する必要がある。

参考資料

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

2.8.4 セキュリティのベストプラクティスに関するユーザ教育のテスト

このテストは、自動化を意図している場合は特に難しいかもしれない。アプリを広く使い、以下の質問に答えられるようにすることを推奨する。

- 指紋の使用：高リスクの取引／情報へのアクセスを提供する認証に指紋が使用される。アプリケーションは、デバイスに他の人の指紋が複数登録されている場合に起こりうる問題について、ユーザに通知しているか？
- Root 化 /Jailbreak：root 化またはジェイルブレイク検出が実装されている。アプリケーションは、特定のリスクの高いアクションがデバイスの ジェイルブレイク /root 化ステータスのために追加のリスクを伴うという事実をユーザに通知しているか？
- 特定の認証情報：ユーザがアプリケーションからリカバリコード、パスワード、PIN を取得する（または設定）。アプリケーションからリカバリコード、パスワード、PIN を取得（または設定）した場合、アプリケーションはこれを他のユーザと決して共有せず、アプリケーションのみが要求するようユーザに指示しているか？
- アプリケーションの配布：リスクの高いアプリケーションの場合、ユーザが危険なバージョンのアプリケーションをダウンロードするのを防ぐため。アプリケーションの製造元は、アプリケーションを配布する正式な方法（Google Play や App Store など）を適切に伝えているか？
- Prominent Disclosure：全てのケース。アプリケーションは、データへのアクセス、収集、使用、共有について、目立つように開示しているか？ 例えば、アプリケーションは、iOS 上で許可を求めるために App Tracking Transparency Framework を使用しているか。

参考資料

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

ルールブック

- アプリを広く使い、セキュリティのベストプラクティスに関する質問に答えられるようにする（推奨）

2.8.5 ルールブック

1. アプリを広く使い、セキュリティのベストプラクティスに関する質問に答えられるようにする（推奨）

2.8.5.1 アプリを広く使い、セキュリティのベストプラクティスに関する質問に答えられるようにする（推奨）

アプリを広く使い、セキュリティのベストプラクティスに関する以下の質問に答えられるようにすることを推奨する。

- 指紋の使用：高リスクの取引／情報へのアクセスを提供する認証に指紋が使用される。アプリケーションは、デバイスに他の人の指紋が複数登録されている場合に起こりうる問題について、ユーザに通知しているか？
- Root 化 / ジェイルブレイク：root 化またはジェイルブレイク検出が実装されている。アプリケーションは、特定のリスクの高いアクションがデバイスの ジェイルブレイク /root 化ステータスのために追加のリスクを伴うという事実をユーザに通知しているか？
- 特定の認証情報：ユーザがアプリケーションからリカバリーコード、パスワード、PIN を取得する（または設定）。アプリケーションからリカバリーコード、パスワード、PIN を取得（または設定）した場合、アプリケーションはこれを他のユーザと決して共有せず、アプリケーションのみが要求するよう指示しているか？
- アプリケーションの配布：リスクの高いアプリケーションの場合、ユーザが危険なバージョンのアプリケーションをダウンロードするのを防ぐため。アプリケーションの製造元は、アプリケーションを配布する正式な方法（Google Play や App Store など）を適切に伝えているか？
- Prominent Disclosure：全てのケース。アプリケーションは、データへのアクセス、収集、使用、共有について、目立つように開示しているか？ 例えば、アプリケーションは、iOS 上で許可を求めるために [App Tracking Transparency Framework](#) を使用しているか。

これに注意しない場合、以下の可能性がある。

- 機密情報が想定していない処理で利用される。
- 第三者に機密情報を読み取られる。

3

暗号化要件

3.1 MSTG-CRYPTO-1

アプリは暗号化の唯一の方法としてハードコードされた鍵による対称暗号化に依存していない。

3.1.1 問題のある暗号化構成

3.1.1.1 不十分なキーの長さ

最も安全な暗号化アルゴリズムであっても、不十分なキーサイズを使用すると、ブルートフォースアタックに 対して脆弱になる。

キーの長さが業界標準を満たしていることを確認する。なお日本国内においては「電子政府推奨暗号リスト」掲載の暗号仕様書一覧を確認する。

参考資料

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

ルールブック

- 業界標準を満たしたキーの長さを設定する（必須）

3.1.1.2 ハードコードされた暗号化鍵による対称暗号化

対称暗号化とキー付きハッシュ(MAC)のセキュリティは、キーの機密性に依存する。キーが公開されると、暗号化によって得られたセキュリティが失われる。これを防ぐには、作成に関与した暗号化データと同じ場所に秘密鍵を保存しないことである。よくある間違いは、静的なハードコードされた暗号化鍵を使用してローカルに保存されたデータを暗号化し、そのキーをアプリにコンパイルすることである。この場合、逆アセンブラーを使用できる人であれば誰でもそのキーにアクセスできるようになる。

ハードコードされた暗号化鍵とは、次のことを意味する。

- アプリケーションリソースの一部であること
- 既知の値から導出可能な値であること
- コードにハードコードされていること

まず、ソースコード内にキーやパスワードが保存されていないことを確認する。つまり、Objective-C/Swift をチェックする必要がある。難読化は動的インストルメンテーションによって容易にバイパスされるため、ハードコードされたキーはソースコードが難読化されても問題があることに注意する。

アプリが双方 TLS (サーバとクライアントの両方の証明書が検証される) を使用している場合、以下を確認する。

- クライアント証明書のパスワードがローカルに保存されていない、またはデバイスの Keychain にロックされていること。
- クライアント証明書は、すべてのインストール間で共有されていないこと。

アプリが、アプリのデータ内に保存され暗号化されたコンテナに依存する場合、暗号化鍵がどのように使用されるかを確認する。キーラップ方式を使用している場合、各ユーザのマスターシークレットが初期化されていること、またはコンテナが新しいキーで再暗号化されていることを確認する。マスターシークレットや以前のパスワードを使用してコンテナを復号できる場合、パスワードの変更がどのように処理されるかを確認する。

モバイルアプリで対称暗号化が使用される場合は常に秘密鍵を安全なデバイストレージに保存する必要がある。プラットフォーム固有の API の詳細については、「Data Storage on iOS」の章を参照する。

参考資料

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

ルールブック

- ソースコード内にキーやパスワードを保存しない (必須)
- クライアント証明書のパスワードをローカルに保存しない、またはデバイスの Keychain にロックする (必須)
- クライアント証明書はすべてのインストール間で共有しない (必須)
- コンテナに依存する場合、暗号化鍵がどのように使用されるかを確認する (必須)
- モバイルアプリで対称暗号化が使用される場合は常に秘密鍵を安全なデバイストレージに保存する (必須)

3.1.1.3 弱いキー生成関数

暗号化アルゴリズム(対称暗号化や一部の MAC など)は、特定のサイズの秘密の入力を想定している。例えば、AES はちょうど 16 バイトのキーを使用する。ネイティブな実装では、ユーザが提供したパスワードを直接入力キーとして使用することがある。ユーザが提供したパスワードを入力キーとして使用する場合、以下のような問題がある。

- パスワードがキーよりも小さい場合、完全なキースペースは使用されない。残りのスペースはパディングされる(パディングのためにスペースが使われることもある)。
- ユーザ提供のパスワードは、現実的には、ほとんどが表示・発音可能な文字で構成される。したがって、256 文字ある ASCII 文字の一部だけが使われ、エントロピーはおよそ 4 分の 1 に減少する。

パスワードが暗号化関数に直接渡されないようにする。代わりに、ユーザが提供したパスワードは、暗号化鍵を作成するために KDF に渡されるべきである。パスワード導出関数を使用する場合は、適切な反復回数を選択する。例えば、NIST は PBKDF2 の反復回数を少なくとも 10,000 回、ユーザが感じるパフォーマンスが重要な重要なキーの場合は少なくとも 10,000,000 回を推奨している。重要なキーについては、Argon2 のような Password Hashing Competition (PHC) で認められたアルゴリズムの実装を検討することが推奨される。

参考資料

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

ルールブック

- 暗号化アルゴリズム(対称暗号化や一部の MAC など)を使用する場合、想定されている特定のサイズの秘密の入力を使用する(必須)
- ユーザが提供したパスワードは、暗号鍵を作成するために KDF に渡す(必須)
- パスワード導出関数を使用する場合は、適切な反復回数を選択する(必須)

3.1.1.4 弱い乱数ジェネレーター

決定論的なデバイスで真の乱数を生成することは基本的に不可能である。擬似乱数ジェネレーター (RNG) は、擬似乱数のストリーム(あたかもランダムに発生したかのように見える数値のストリーム)を生成することでこれを補う。生成される数値の品質は、使用するアルゴリズムの種類によって異なる。暗号化的に安全な RNG は、統計的ランダム性テストに合格した乱数を生成し、予測攻撃に対して耐性がある。(例: 次に生成される数を予測することは統計的に不可能である)

Mobile SDK は、十分な人工的ランダム性を持つ数値を生成する RNG アルゴリズムの標準的な実装を提供している。利用可能な API については、iOS 固有のセクションで紹介する。

参考資料

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

ルールブック

- 十分な人工的ランダム性を持つ数値を生成する *RNG* アルゴリズムの標準的な実装を確認する (必須)

3.1.1.5 暗号化のカスタム実装

独自の暗号関数を開発することは、時間がかかり、困難であり、失敗する可能性が高い。その代わりに、安全性が高いと広く認められている、よく知られたアルゴリズムを使用することができる。モバイル OS は、これらのアルゴリズムを実装した標準的な暗号 API を提供している。

ソースコード内で使用されているすべての暗号化方式、特に機密データに直接適用されている暗号化方式を注意深く検査する。すべての暗号化操作では、iOS 標準の暗号化 API を使用する必要がある (これらについては、プラットフォーム固有の章で詳しく説明する)。既知のプロバイダが提供する標準的なルーチンを呼び出さない暗号化操作は、厳密に検査する必要がある。標準的なアルゴリズムが変更されている場合は、細心の注意を払う必要がある。エンコーディングは暗号化と同じではないことに注意する。XOR (排他的論理和) のようなビット操作の演算子を見つけたら、必ずさらに調査する。

暗号化のすべての実装で、以下のことが常に行われていることを確認する必要がある。

- ワーカーキー (AES/DES/Rijndael における中間鍵/派生鍵のようなもの) は、消費後またはエラー発生時にメモリから適切に削除される。
- 暗号の内部状態は、できるだけ早くメモリから削除する。

参考資料

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

ルールブック

- 暗号化に関するすべての実装では適切にメモリ状態を管理する (必須)
- OS が提供する業界標準の暗号 API を使用する (必須)

3.1.1.6 不適切な AES 構成

Advanced Encryption Standard (AES) は、モバイルアプリにおける対称暗号化の標準として広く受け入れられている。AES は、一連の連鎖的な数学演算に基づく反復型ブロック暗号である。AES は入力に対して可変回数のラウンドを実行し、各ラウンドでは入力ブロック内のバイトの置換と並べ替えが行われる。各ラウンドでは、オリジナルの AES キーから派生した 128 ビットのラウンドキーが使用される。

この記事の執筆時点では、AES に対する効率的な暗号解読攻撃は発見されていない。しかし、実装の詳細やブロック暗号モードなどの設定可能なパラメータには、エラーの余地がある。

弱いブロック暗号モード

ブロックベースの暗号化は、個々の入力ブロック (例: AES は 128 ビットブロック) に対して実行される。平文がブロックサイズより大きい場合、平文は内部で指定された入力サイズのブロックに分割され、各ブロックに対して暗号化が実行される。ブロック暗号操作モード (またはブロックモード) は、前のブロックの暗号化の結果が後続のブロックに影響を与えるかどうかを決定する。

ECB (Electronic Codebook) は、入力を一定サイズのブロックに分割し、同じキーを用いて個別に暗号化する。分割された複数のブロックに同じ平文が含まれている場合、それらは同一の暗号文ブロックに暗号化されるため、データのパターンを容易に特定することができる。また、状況によっては、攻撃者が暗号化されたデータを再生することも可能である。

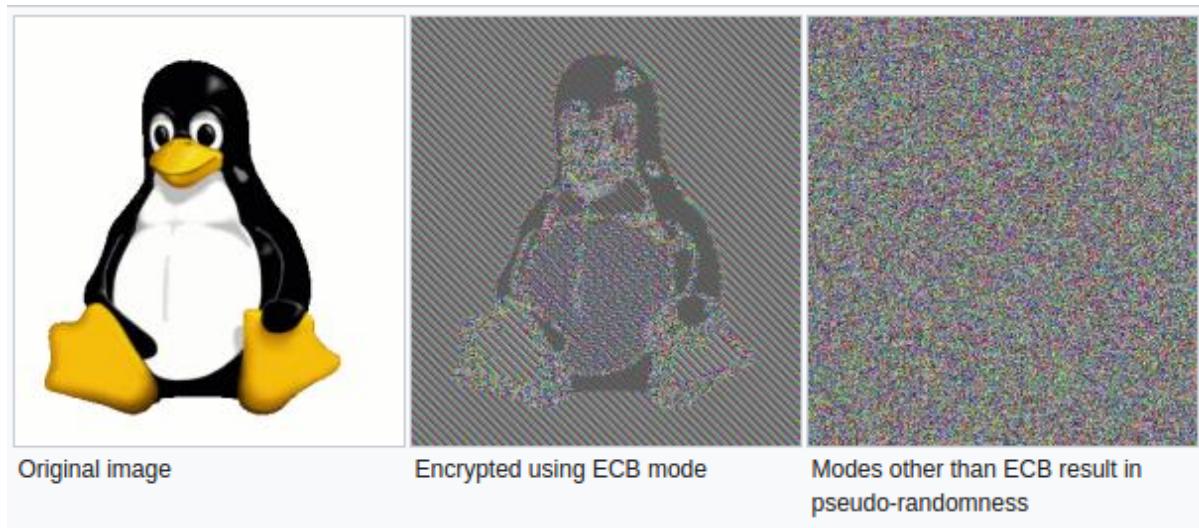


図 3.1.1.6.1 ECB の暗号化例

ECB の代わりに暗号ブロック連鎖 (CBC) モードが使用されていることを確認する。CBC モードでは、平文ブロックは直前の暗号文ブロックと XOR される。これにより、ブロックに同じ情報が含まれている場合でも、暗号化された各ブロックは一意であり、ランダムであることが保証される。CBC を HMAC と組み合わせたり、「パディングエラー」「MAC エラー」「復号失敗」などのエラーが出ないようにするために、パディングオラクル攻撃に対抗するために最善であることに注意してください。

暗号化されたデータを保存する場合、Galois/Counter Mode (GCM) のような、保存データの完全性も保護するブロックモードを使用することを推奨する。後者には、このアルゴリズムが各 TLSv1.2 の実装に必須であり、したがってすべての最新のプラットフォームで利用できるという利点もある。

有効なブロックモードの詳細については、ブロックモード選択に関する NIST のガイドラインを参照する。

予測可能な初期化ベクトル

CBC 、 OFB 、 CFB 、 PCBC 、 GCM モードでは、暗号への初期入力として、初期化ベクトル (IV) が必要である。IV は秘密にする必要はないが、予測可能であってはならない。暗号化されたメッセージごとにランダムで一意であり、非再現性である必要がある。IV は暗号的に安全な乱数ジェネレーターを用いて生成されていることを確認する。IV の詳細については、[Crypto Fail の初期化ベクトル](#)に関する記事を参照する。

コードで使用されている暗号化ライブラリに注意すること: 多くのオープンソースライブラリは、悪い習慣 (ハードコードされた IV の使用など) に従う可能性のあるドキュメントが提供されている。よくある間違いは、IV 値を変更せずにサンプルコードをコピーアンドペーストすることである。

ステートフル操作モードでの初期化ベクトル

初期化ベクトルがカウンター (CTR と nonce の組み合わせ) であることが多い CTR モードと GCM モードを使用する場合は、IV の使用方法が異なることに注意する。したがって、ここでは、独自のステートフルモデル

を持つ予測可能な IV を使用することが、まさに必要である。CTR では、新しいブロック操作のたびに、新しい nonce とカウンターを入力として使用する。例: 5120 ビット長の平文の場合では、20 個のブロックがあるため、nonce とカウンターで構成される 20 個の入力ベクトルが必要である。一方 GCM では、暗号化操作ごとに IV を 1 つだけ持ち、同じキーで繰り返さないようにする。IV の詳細と推奨事項については、GCM に関する NIST の資料の 8 項を参照する。

参考資料

- 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

ルールブック

- パディングオラクル攻撃に対抗するために、CBC を HMAC と組み合わせたりパディングエラー、MAC エラー、復号失敗などのエラーが出ないようにする（必須）
- 暗号化されたデータを保存する場合、Galois/Counter Mode (GCM) のような、保存データの完全性も保護するブロックモードを使用する（推奨）
- IV は暗号的に安全な乱数ジェネレーターを用いて生成する（必須）
- 初期化ベクトルがカウンターであることが多い CTR モードと GCM モードを使用する場合は、IV の使用方法が異なることに注意する（必須）

3.1.1.7 弱いパディングまたはブロック操作の実装によるパディングオラクル攻撃

以前は、非対称暗号を行う際のパディングメカニズムとして、PKCS1.5 パディング（コード: PKCS1Padding）が使われていた。このメカニズムは、パディングオラクル攻撃に対して脆弱である。そのため、PKCS#1 v2.0（コード: OAEPwithSHA-256andMGF1Padding、OAEPwithSHA-224andMGF1Padding、OAEPwithSHA-384andMGF1Padding、OAEPwithSHA-512andMGF1Padding）に取り込まれた OAEP（Optimal Asymmetric Encryption PaddingOAEPPadding）を使用するのが最適である。なお、OAEP を使用した場合でも、Kudelskisecurity のブログで紹介されている Mangers 攻撃としてよく知られている問題に遭遇する可能性があることに注意する。

注: PKCS #5 を使用する AES-CBC は、実装が「パディングエラー」、「MAC エラー」、または「復号に失敗しました」などの警告を表示するため、パディングオラクル攻撃に対しても脆弱であることが示されている。例として、The Padding Oracle Attack および The CBC Padding Oracle Problem を参照する。次に、平文を暗号化した後に HMAC を追加することが最善である。結局、MAC に失敗した暗号文は復号する必要がなく、破棄することができる。

参考資料

- 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

ルールブック

- 非対称暗号を行う際のパディングメカニズムとして PKCS#1 v2.0 に取り込まれた OAEP を使用する (必須)

3.1.1.8 ストレージおよびメモリ内のキーの扱い

メモリダンプが脅威モデルの一部である場合、キーがアクティブに使用された瞬間にキーにアクセスできる。メモリダンプには、ルートアクセス (ルート化されたデバイスやジェイルブレイクされたデバイスなど) が必要であるか、Frida でパッチされたアプリケーション (Fridump などのツールを使用できるように) が必要である。したがって、デバイスでキーがまだ必要な場合は、次のことを考慮するのが最善である。

- リモートサーバのキー: Amazon KMS や Azure Key Vault などのリモート Key Vault を使用することができる。一部のユースケースでは、アプリとリモートリソースの間にオーケストレーションレイヤーを開発することが適切なオプションとなる場合がある。例えば、Function as a Service (FaaS) システム (AWS Lambda や Google Cloud Functions など) 上で動作するサーバレス関数が、API キーやシークレットを取得するためのリクエストを転送するような場合である。その他の選択肢として、Amazon Cognito 、Google Identity Platform 、Azure Active Directory なども存在する。
- ハードウェアで保護された安全なストレージ内のキー: すべての暗号化アクションとキー自体が Secure Enclave (例: Keychain を使用) にあることを確認する。詳細については、iOS Data Storage の章を参照する。
- エンベロープ暗号化によって保護されたキー: キーが TEE/SE の外部に保存されている場合は、multi-layered 暗号化の使用を検討する。エンベロープ暗号化アプローチ (OWASP Cryptographic Storage Cheat Sheet 、 Google Cloud Key management guide 、 AWS Well-Architected Framework guide 参照) 、またはデータ暗号化鍵をキー暗号化する HPKE アプローチを使用する。
- メモリ内のキー: キーができるだけ短時間しかメモリに残さないようにし、暗号化操作に成功した後やエラー時にキーをゼロにし、無効化することを考慮する。一般的な暗号化のガイドラインについては、機密データのメモリの消去を参照する。より詳細な情報については、「 Testing Memory for Sensitive Data 」を参照する。

注: メモリダンプが容易になるため、署名の検証や暗号化に使用される公開鍵以外は、アカウントやデバイス間で同じキーを共有しない。

参考資料

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

ルールブック

- メモリダンプを考慮してキーを使用する (必須)

- アカウントやデバイス間で同じキーを共有しない（必須）

3.1.1.9 転送時のキーの扱い

キーをデバイス間で、またはアプリからバックエンドに転送する必要がある場合、トランスポート対称鍵や他のメカニズムによって、適切なキー保護が行われていることを確認する。多くの場合、キーは難読化された状態で共有されるため、簡単に元に戻すことができる。代わりに、非対称暗号化またはラッピングキーが使用されていることを確認する。例えば、対称鍵は非対称鍵の公開鍵で暗号化することができる。

参考資料

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

ルールブック

- トランスポート対称鍵や他のメカニズムによって、適切なキー保護を行う（必須）

3.1.2 ルールブック

1. 業界標準を満たしたキーの長さを設定する（必須）
2. ソースコード内にキーやパスワードを保存しない（必須）
3. クライアント証明書のパスワードをローカルに保存しない、またはデバイスの *Keychain* にロックする（必須）
4. クライアント証明書はすべてのインストール間で共有しない（必須）
5. コンテナに依存する場合、暗号化鍵がどのように使用されるかを確認する（必須）
6. モバイルアプリで対称暗号化が使用される場合は常に秘密鍵を安全なデバイストレージに保存する（必須）
7. 暗号化アルゴリズム（対称暗号化や一部の MAC など）を使用する場合、想定されている特定のサイズの秘密の入力を使用する（必須）
8. ユーザが提供したパスワードは、暗号鍵を作成するために KDF に渡す（必須）
9. パスワード導出関数を使用する場合は、適切な反復回数を選択する（必須）
10. 十分な人工的ランダム性を持つ数値を生成する RNG アルゴリズムの標準的な実装を確認する（必須）
11. 暗号化に関するすべての実装では適切にメモリ状態を管理する（必須）
12. OS が提供する業界標準の暗号 API を使用する（必須）
13. パディングオラクル攻撃に対抗するために、CBC を HMAC と組み合わせたりパディングエラー、MAC エラー、復号失敗などのエラーが出ないようにする（必須）

14. 暗号化されたデータを保存する場合、*Galois/Counter Mode (GCM)* のような、保存データの完全性も保護するブロックモードを使用する（推奨）
15. *IV* は暗号的に安全な乱数ジェネレーターを用いて生成する（必須）
16. 初期化ベクトルがカウンターであることが多い *CTR* モードと *GCM* モードを使用する場合は、*IV* の使用方法が異なることに注意する（必須）
17. 非対称暗号を行う際のパディングメカニズムとして *PKCS#1 v2.0* に取り込まれた *OAEP* を使用する（必須）
18. メモリダンプを考慮してキーを使用する（必須）
19. アカウントやデバイス間で同じキーを共有しない（必須）
20. トランスポート対称鍵や他のメカニズムによって、適切なキー保護を行う（必須）

3.1.2.1 業界標準を満たしたキーの長さを設定する（必須）

キーの長さが業界標準を満たしていることを確認する。なお日本国内においては「電子政府推奨暗号リスト」掲載の暗号仕様書一覧を確認する。最も安全な暗号化アルゴリズムであっても、不十分なキーサイズを使用すると、ブルートフォースアタックに対して脆弱になる。

※概念的なルールのため、サンプルコードはなし。

これに違反する場合、以下の可能性がある。

- ブルートフォースアタックに対して脆弱になる。

3.1.2.2 ソースコード内にキーやパスワードを保存しない（必須）

難読化は動的インストルメンテーションによって容易にバイパスされるため、ハードコードされたキーはソースコードが難読化されていても問題がある。そのため、ソースコード（Objective-C/Swift コード）内にキーやパスワードを保存しない。

※非推奨なルールのため、サンプルコードはなし。

これに違反する場合、以下の可能性がある。

- 難読化が動的インストルメンテーションによってバイパスされ、キーやパスワードが漏洩する。

3.1.2.3 クライアント証明書のパスワードをローカルに保存しない、またはデバイスの Keychain にロックする（必須）

アプリが双方向 TLS (サーバとクライアントの両方の証明書が検証される) を使用している場合、クライアント証明書のパスワードをローカルに保存しない。またはデバイスの Keychain にロックする。

サンプルコードは以下ルールブックを参照。

ルールブック

- *Keychain Services API* を使用してセキュアに値を保存する (必須)

これに違反する場合、以下の可能性がある。

- パスワードが他アプリに読み取られ悪用される。

3.1.2.4 クライアント証明書はすべてのインストール間で共有しない (必須)

アプリが双方向 TLS (サーバとクライアントの両方の証明書が検証される) を使用している場合、クライアント証明書はすべてのインストール間で共有しない。

※非推奨なルールのため、サンプルコードはなし。

これに違反する場合、以下の可能性がある。

- クライアントが攻撃者によってなりすまされる。

3.1.2.5 コンテナに依存する場合、暗号化鍵がどのように使用されるかを確認する (必須)

アプリが、アプリのデータ内に保存された暗号化されたコンテナに依存する場合、暗号化鍵がどのように使用されるかを確認する。

キーラップ方式を使用している場合

以下について確認する。

- 各ユーザのマスターシークレットが初期化されていること
- コンテナが新しいキーで再暗号化されていること

マスターシークレットや以前のパスワードを使用してコンテナを復号できる場合

パスワードの変更がどのように処理されるかを確認する。

※概念的なルールのため、サンプルコードはなし。

これに違反する場合、以下の可能性がある。

- パスワードやマスターシークレットが意図した目的以外で使用される。

3.1.2.6 モバイルアプリで対称暗号化が使用される場合は常に秘密鍵を安全なデバイストレージに保存する (必須)

モバイルアプリで対称暗号化が使用される場合は常に秘密鍵を安全なデバイストレージに保存する必要がある。iOS プラットフォームでの秘密鍵の保存方法については、「データ保護 API」を参照。

ルールブック

- *iOS Data Protection API* を活用して、フラッシュメモリに保存されたユーザデータにアクセス制御を実装する (必須)

これに違反する場合、以下の可能性がある。

- 秘密鍵が他アプリや第三者に読み取られる。

3.1.2.7 暗号化アルゴリズム(対称暗号化や一部の MAC など)を使用する場合、想定されている特定のサイズの秘密の入力を使用する(必須)

暗号化アルゴリズム(対称暗号化や一部の MAC など)を使用する場合、想定されている特定のサイズの秘密の入力を使用する必要がある。例えば、AES はちょうど 16 バイトのキーを使用する。

ネイティブな実装では、ユーザが提供したパスワードを直接入力キーとして使用することがある。ユーザが提供したパスワードを入力キーとして使用する場合、以下のような問題がある。

- パスワードがキーよりも小さい場合、完全なキースペースは使用されない。残りのスペースはパディングされる(パディングのためにスペースが使われることもある)。
- ユーザ提供のパスワードは、現実的には、ほとんどが表示・発音可能な文字で構成される。したがって、256 文字ある ASCII 文字の一部だけが使われ、エントロピーはおよそ 4 分の 1 に減少する。

※概念的なルールのため、サンプルコードはなし。

これに違反する場合、以下の可能性がある。

- 脆弱なキーが生成される。

3.1.2.8 ユーザが提供したパスワードは、暗号鍵を作成するために KDF に渡す(必須)

暗号化関数を使用する場合、ユーザが提供したパスワードは、暗号鍵を作成するために KDF に渡されるべきである。パスワードが暗号化関数に直接渡されないようにする。代わりに、ユーザが提供したパスワードは、暗号化鍵を作成するために KDF に渡されるべきである。パスワード導出関数を使用する場合は、適切な反復回数を選択する。

※概念的なルールのため、サンプルコードはなし。

これに違反する場合、以下の可能性がある。

- パスワードがキーよりも小さい場合、完全なキースペースは使用されない。残りのスペースはパディングされる。
- エントロピーがおよそ 4 分の 1 に減少する。

3.1.2.9 パスワード導出関数を使用する場合は、適切な反復回数を選択する(必須)

パスワード導出関数を使用する場合は、適切な反復回数を選択する必要がある。例えば、NIST は PBKDF2 の反復回数を少なくとも 10,000 回、ユーザが感じるパフォーマンスが重要でない重要なキーの場合は少なくとも 10,000,000 回を推奨している。重要なキーについては、Argon2 のような Password Hashing Competition (PHC) で認められたアルゴリズムの実装を検討することが推奨される。

※サーバ側のルールのため、サンプルコードはなし。

これに違反する場合、以下の可能性がある。

- 脆弱なキーが生成される。

3.1.2.10 十分な人工的ランダム性を持つ数値を生成する RNG アルゴリズムの標準的な実装を確認する（必須）

暗号化的に安全な RNG は、統計的ランダム性テストに合格した乱数を生成し、予測攻撃に対して耐性がある。安全な水準に満たない RNG アルゴリズムにより生成された乱数を使用すると、予測攻撃が成功する可能性が高まる。そのため、十分な人工的ランダム性を持つ数値を生成する RNG アルゴリズムを使用する必要がある。

iOS 標準で安全性の高い乱数を生成する API については以下を参照する。

ルールブック

- Randomization Services API を使用して安全な乱数を生成（推奨）*

これに違反する場合、以下の可能性がある。

- 予測攻撃が成功する可能性が高まる。

3.1.2.11 暗号化に関するすべての実装では適切にメモリ状態を管理する（必須）

暗号化に関するすべての実装では、ワーカーキー (AES/DES/Rijndael における中間鍵/派生鍵のようなもの) が消費後またはエラー発生時にメモリから適切に削除する必要がある。また暗号の内部状態も、できるだけ早くメモリから削除する必要がある。

AES 実装と実行後の解放：

```
import Foundation
import CryptoSwift

class CryptoAES {
    var aes: AES? = nil

    // 暗号化処理
    func encrypt(key: String, iv:String, text:String) -> String {

        do {
            // 関数最後に実行
            defer {
                aes = nil
            }
            // AES インスタンス化
            aes = try AES(key: key, iv: iv)
            guard let encrypt = try aes?.encrypt(Array(text.utf8)) else {
                return ""
            }

            // Data 型変換
        }
    }
}
```

(次のページに続く)

(前のページからの続き)

```

let data = Data( encrypt )
// base64 変換
let base64Data = data.base64EncodedData()
// UTF-8 変換 nil 不可
guard let base64String =
    String(data: base64Data as Data, encoding: String.Encoding.utf8) else {
    return ""
}
// base64 文字列
return base64String

} catch {
    return ""
}
}

// 複合処理
func decrypt(key: String, iv:String, base64:String) -> String {

do {
    // 関数最後に実行
    defer {
        aes = nil
    }
    // AES インスタンス化
    aes = try AES(key: key, iv:iv)

    // base64 から Data型へ
    let byteData = base64.data(using: String.Encoding.utf8)! as Data
    // base64 デコード
    guard let data = Data(base64Encoded: byteData) else {
        return ""
    }

    // UInt8 配列の作成
    let aBuffer = Array<UInt8>(data)
    // AES 複合
    guard let decrypted = try aes?.decrypt(aBuffer) else {
        return ""
    }
    // UTF-8 変換
    guard let text = String(data: Data(decrypted), encoding: .utf8) else {
        return ""
    }

    return text
} catch {
    return ""
}
}

```

(次のページに続く)

(前のページからの続き)

```

    }
}

```

CryptoSwift の Pods 使用例：

```

target 'MyApp' do
  use_frameworks!
  # CryptoSwift ライブラリを追加する
  pod 'CryptoSwift'
end

```

これに違反する場合、以下の可能性がある。

- メモリに残された暗号化情報を意図しない処理で利用される。

3.1.2.12 OS が提供する業界標準の暗号 API を使用する（必須）

独自の暗号関数を開発することは、時間がかかり、困難であり、失敗する可能性が高い。その代わりに、安全性が高いと広く認められている、よく知られたアルゴリズムを使用することができる。モバイル OS は、これらのアルゴリズムを実装した標準的な暗号 API を提供しているため、安全な暗号化のためにはこちらを利用する必要がある。

サンプルコードについては、以下のルールブックを参照。

ルールブック

- iOS 暗号化アルゴリズム Apple CryptoKit の実装（推奨）

これに違反する場合、以下の可能性がある。

- 脆弱性を含む実装となる可能性がある。

3.1.2.13 パディングオラクル攻撃に対抗するために、CBC を HMAC と組み合わせたりパディングエラー、MAC エラー、復号失敗などのエラーが出ないようにする（必須）

CBC モードでは、平文ブロックは直前の暗号文ブロックと XOR される。これにより、ブロックに同じ情報が含まれている場合でも、暗号化された各ブロックは一意であり、ランダムであることが保証される。

CBC モード実装の例：

```

import Foundation
import CryptoSwift

class CryptoCBC {
    // 暗号化する平文の例
    let cleartext = "Hello CryptoSwift"

    // 暗号化処理
    func encrypt( text:String ) -> (String, String) {

```

(次のページに続く)

(前のページからの続き)

```
// 適当な 256 ビット長の鍵（文字列）
let key = "BDC171111B7285F67F035497EE9A081D"

// エンコード
let byteText = text.data(using: .utf8)!.bytes
let byteKey = key.data(using: .utf8)!.bytes

// IV（初期化ベクトル）、iv の型は [UInt8]
let iv = AES.randomIV(AES.blockSize)
do {
    // AES-256-CBC インスタンスの生成
    let aes = try AES(key: byteKey, blockMode: CBC(iv: iv))

    // 平文を暗号化
    // デフォルトのパディングが PKC7
    let encrypted = try aes.encrypt(byteText)

    // IV、encrypted を出力 Base64 文字列にエンコード
    let strIV = NSData(bytes: iv, length: iv.count).
    ↪base64EncodedString(options: .lineLength64Characters)
    print("IV: " + strIV) // 出力 -> IV: 1BMiK2GWEwrPgNdGfrJEig==
    let strEnc = NSData(bytes: encrypted, length: encrypted.count).
    ↪base64EncodedString(options: .lineLength64Characters)
    print("Encrypted: " + strEnc) // 出力 -> Encrypted: MHf5ZeUL/
    ↪gjviiztpZKJFuqppdTgEe+Ik1Dgg3N1fQ=
    return (strIV, strEnc)
} catch {
    print("Error")
}
return ("", "")
}
```

これに違反する場合、以下の可能性がある。

- パディングオラクル攻撃に対して脆弱になる。

3.1.2.14 暗号化されたデータを保存する場合、Galois/Counter Mode (GCM) のような、保存データの完全性も保護するブロックモードを使用する（推奨）

暗号化されたデータを保存する場合、Galois/Counter Mode (GCM) のような、保存データの完全性も保護するブロックモードを使用することを推奨する。後者には、このアルゴリズムが各 TLSv1.2 の実装に必須であり、したがってすべての最新のプラットフォームで利用できるという利点もある。

GCM モード実装の例：

```
import Foundation
import CryptoKit

class CryptoGCM {
```

(次のページに続く)

(前のページからの続き)

```
// 鍵の生成
let symmetricKey: SymmetricKey = SymmetricKey(size: .bits256)
/// 暗号化
/// - Parameter data: 暗号化するデータ
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
    }
}

/// 復号
/// - Parameter data: 復号するデータ
private func decrypt(data: Data) -> Data? {
    do {
        // GCM decrypt
        let sealedBox = try AES.GCM.SealedBox(combined: data)
        return try AES.GCM.open(sealedBox, using: symmetricKey)
    } catch _ {
        return nil
    }
}
}
```

これに違反する場合、以下の可能性がある。

- データのパターンを容易に特定される。

3.1.2.15 IV は暗号的に安全な乱数ジェネレーターを用いて生成する（必須）

CBC、OFB、CFB、PCBC、GCM モードでは、暗号への初期入力として、初期化ベクトル (IV) が必要である。IV は秘密にする必要はないが、予測可能であってはならない。暗号化されたメッセージごとにランダムで一意であり、非再現性である必要がある。そのため、IV は暗号的に安全な乱数ジェネレーターを用いて生成する必要がある。IV の詳細については、[Crypto Fail の初期化ベクトルに関する記事](#)を参照する。

サンプルコードは以下ルールブックを参照。

ルールブック

- [Randomization Services API を使用して安全な乱数を生成（推奨）](#)

これに違反する場合、以下の可能性がある。

- 予測可能な初期化ベクトルが生成される。

3.1.2.16 初期化ベクトルがカウンターであることが多い CTR モードと GCM モードを使用する場合は、IV の使用方法が異なることに注意する（必須）

初期化ベクトルがカウンター (CTR と nonce の組み合わせ) であることが多い CTR モードと GCM モードを使用する場合は、IV の使用方法が異なることに注意する。そのため、独自のステートフルモデルを持つ予測可能な IV を使用することが必要である。CTR では、新しいブロック操作のたびに、新しい nonce とカウンターを入力として使用する。例: 5120 ビット長の平文の場合では、20 個のブロックがあるため、nonce とカウンターで構成される 20 個の入力ベクトルが必要である。一方 GCM では、暗号化操作ごとに IV を 1 つだけ持ち、同じキーで繰り返さないようにする。IV の詳細と推奨事項については、[GCM](#) に関する NIST の資料の 8 項を参照する。

※概念的なルールのため、サンプルコードはなし。

これに違反する場合、以下の可能性がある。

- 各モードで必要とされる初期化ベクトルの要件を満たせない。

3.1.2.17 非対称暗号を行う際のパディングメカニズムとして PKCS#1 v2.0 に取り込まれた OAEP を使用する（必須）

以前は、非対称暗号を行う際のパディングメカニズムとして、PKCS1.5 パディング（コード: PKCS1Padding）が使われていた。このメカニズムは、パディングオラクル攻撃に対して脆弱である。そのため、PKCS#1 v2.0（コード: OAEWithSHA-256andMGF1Padding、OAEWithSHA-224andMGF1Padding、OAEWithSHA-384andMGF1Padding、OAEWithSHA-512andMGF1Padding）に取り込まれた OAEP（Optimal Asymmetric Encryption PaddingOAEPPadding）を使用するのが最適である。なお、OAEP を使用した場合でも、Kudelskisecurity のブログで紹介されている Mangers 攻撃としてよく知られている問題に遭遇する可能性があることに注意する。

以下のサンプルコードは、OAEP の使用方法である。

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

// main digest SHA256, MGF1 digest SHA1 で RsaOAEPPadding クラスを生成
let padding = RsaOAEPPadding(mainDigest: OAEPDigest.SHA256, mgf1Digest: OAEPDigest.
    ↪SHA1)

// OAEP Padding を計算 (RSA 鍵長は 2048bit = 256byte 前提)
let padded = try! padding.pad(plain: plainData, blockSize: 256);

// raw で暗号化
guard let cipherData = SecKeyCreateEncryptedData(publicKey, SecKeyAlgorithm.
    ↪rsaEncryptionRaw, padded as CFData, &error) else {
    // Error 处理
}
```

これに違反する場合、以下の可能性がある。

- パディングオラクル攻撃に対して脆弱になる。

3.1.2.18 メモリダンプを考慮してキーを使用する（必須）

メモリダンプが脅威モデルの一部である場合、キーがアクティブに使用された瞬間にキーにアクセスできる。メモリダンプには、ルートアクセス（ルート化されたデバイスやジェイルブレイクされたデバイスなど）が必要であるか、Frida でパッチされたアプリケーション（Fridump などのツールを使用できるように）が必要である。したがって、デバイスでキーがまだ必要な場合は、次のことを考慮するのが最善である。

- リモートサーバのキー: Amazon KMS や Azure Key Vault などのリモート Key Vault を使用することができる。一部のユースケースでは、アプリとリモートリソースの間にオーケストレーションレイヤーを開発することが適切なオプションとなる場合がある。例えば、Function as a Service (FaaS) システム（AWS Lambda や Google Cloud Functions など）上で動作するサーバレス関数が、API キーやシークレットを取得するためのリクエストを転送するような場合である。その他の選択肢として、Amazon Cognito、Google Identity Platform、Azure Active Directory なども存在する。
- ハードウェアで保護された安全なストレージ内のキー: すべての暗号化アクションとキー自体が Secure Enclave（例: Keychain を使用）にあることを確認する。詳細については、iOS Data Storage の章を参照する。
- エンベロープ暗号化によって保護されたキー: キーが TEE/SE の外部に保存されている場合は、multi-layered 暗号化の使用を検討する。エンベロープ暗号化アプローチ（OWASP Cryptographic Storage Cheat Sheet、Google Cloud Key management guide、AWS Well-Architected Framework guide 参照）、またはデータ暗号化鍵をキー暗号化する HPKE アプローチを使用する。
- メモリ内のキー: キーができるだけ短時間しかメモリに残さないようにし、暗号化操作に成功した後やエラー時にキーをゼロにし、無効化することを考慮する。一般的な暗号化のガイドラインについては、機密データのメモリの消去を参照する。より詳細な情報については、「Testing Memory for Sensitive Data」を参照する。

以下サンプルコードは、アプリでのメモリ内のキーの漏洩防止用の処理。

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

また、キーを全く保存しないことで、キーマテリアルがダンプされないことが保証される。これは、PBKDF-2 などのパスワードキー導出機能を使用することで実現できる。以下の例を参照する。

```
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? {
```

(次のページに続く)

(前のページからの続き)

```

    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)
}

```

- 出典：<https://stackoverflow.com/questions/8569555/pbkdf2-using-commoncrypto-on-ios> (Arcane ライブ ラリのテストスイートでテスト済み)

これに違反する場合、以下の可能性がある。

- メモリダンプが脅威モデルの一部である場合、キーがアクティブに使用された瞬間にキーにアクセスできる。

3.1.2.19 アカウントやデバイス間で同じキーを共有しない（必須）

メモリダンプが容易になるため、署名の検証や暗号化に使用される公開鍵以外は、アカウントやデバイス間で同じキーを共有しない。

※非推奨なルールのため、サンプルコードはなし。

これに違反する場合、以下の可能性がある。

- キーのメモリダンプが容易になる。

3.1.2.20 トランSPORT対称鍵や他のメカニズムによって、適切なキー保護を行う（必須）

キーをデバイス間で、またはアプリからバックエンドに転送する必要がある場合、トランSPORT対称鍵や他のメカニズムによって、適切なキー保護が行われていることを確認する。多くの場合、キーは難読化された状態で共有されるため、簡単に元に戻すことができる。代わりに、非対称暗号化またはラッピングキーが使用されていることを確認する。例えば、対称鍵は非対称鍵の公開鍵で暗号化することができる。

キーを適切に保護するには Keychain を使用する。Keychain によるキーの保管方法は以下ルールブックを参照。

ルールブック

- *Keychain Services API* を使用してセキュアに値を保存する（必須）

これに違反する場合、以下の可能性がある。

- キーを元に戻され読み取られる。

3.2 MSTG-CRYPTO-2

アプリは実績のある暗号化プリミティブの実装を使用している。

3.2.1 問題のある暗号化構成

※問題のある暗号化構成については、「MSTG-CRYPTO-1 3.1.1. 問題のある暗号化構成」の内容を確認すること。

3.2.2 暗号化標準アルゴリズムの構成

Apple は、最も一般的な暗号化アルゴリズムの実装を含むライブラリを提供している。Apple の Cryptographic Services Guide が参考になる。これには、標準ライブラリを使用して暗号化プリミティブを初期化および使用する方法の一般化されたドキュメント、ソースコード分析に役立つ情報が含まれている。

参照資料

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

3.2.2.1 CryptoKit

Apple CryptoKit は iOS 13 でリリースされ、FIPS 140-2 検証済みの Apple のネイティブ暗号化ライブラリ corecrypto をベースに構築されている。Swift フレームワークは、厳密に型指定された API インターフェースを提供し、効果的なメモリ管理を行い、equatable に準拠し、ジェネリックをサポートする。CryptoKit には、ハッシュ、対称鍵暗号化、および公開鍵暗号化のための安全なアルゴリズムが含まれている。フレームワークは、Secure Enclave のハードウェアベースのキーマネージャーも利用できる。

Apple CryptoKit には、以下のアルゴリズムが含まれている。

ハッシュ：

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

対称鍵：

- メッセージ認証コード（HMAC）
- 認証付き暗号化
 - AES-GCM
 - ChaCha20-Poly1305

公開鍵：

- キー共有
 - Curve25519
 - NIST P-256
 - NIST P-384
 - NIST P-512

使用例：

対称鍵の生成と公開：

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

SHA-2 512-bit digest の計算：

```
let rawString = "OWASP MTSG"
let rawData = Data(rawString.utf8)
let hash = SHA512.hash(data: rawData) // Compute the digest
```

(次のページに続く)

(前のページからの続き)

```
let textHash = String(describing: hash)
print(textHash) // Print hash text
```

Apple CryptoKit の詳細については、以下のリソースを参照。

- [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](#)

参考資料

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

ルールブック

- [iOS 暗号化アルゴリズム Apple CryptoKit の実装 \(推奨\)](#)

3.2.2.2 CommonCrypto、SecKey、Wrapper の各ライブラリ

暗号化操作に最も一般的に使用されるクラスは、iOS runtime にパッケージされている CommonCrypto である。CommonCrypto オブジェクトによって提供される機能は、ヘッダーファイルのソースコードを見ることで最もよく分析できる。

- Commoncryptor.h : 対称暗号操作のパラメータを提供する。
- CommonDigest.h : ハッシュアルゴリズムのパラメータを提供する。
- CommonHMAC.h : サポートされている HMAC 操作のパラメータを提供する。
- CommonKeyDerivation.h : サポートされている KDF 関数のパラメータを提供する。
- CommonSymmetricKeywrap.h : 対称鍵をキー暗号化鍵でラップするために使用される関数を提供する。

残念ながら、CommonCryptor の public API には、次のようないくつかのタイプの操作が存在しない。GCM モードは、private API でのみ使用できる。ソースコードを参照。このためには、追加のバインディングヘッダーが必要である。または、他の wrapper ライブラリを使用することもできる。

次に、非対称な操作のために、Apple は SecKey を提供している。Apple は Developer Documentation の中で、この使い方を丁寧に説明している。

前に述べたように、利便性を提供するために、いくつかの wrapper ライブラリが両方に存在する。使用される典型的なライブラリは、以下のようなものである。

- IDZSwiftCommonCrypto
- Heimdall

- SwiftyRSA
- RNCryptor
- Arcane

参考資料

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

ルールブック

- iOS 暗号化アルゴリズム Apple CryptoKit の実装 (推奨)

3.2.2.3 サードパーティライブラリ

次のようなさまざまなサードパーティライブラリが利用できる。

- CJOSE : JWE の台頭と、AES GCM の公的サポートの不足により、CJOSE などの他のライブラリが登場した。CJOSE は C/C++ 実装のみを提供するため、より高いレベルの wrapping が必要である。
- CryptoSwift : GitHub 公開されている Swift のライブラリである。このライブラリは、さまざまなハッシュ関数、MAC 関数、CRC 関数、対称暗号、およびパスワードベースのキー導出関数をサポートしている。これは wrapper ではなく、各暗号を完全に自己実装したものである。関数の効果的な実装を検証することが重要である。
- OpenSSL : OpenSSL は、C で記述された TLS に使用されるツールキットライブラリである。その暗号化関数のほとんどは、(H) MAC、署名、対称および非対称暗号、ハッシュなどの作成など、必要なさまざまな暗号化アクションを実行するために使用できる。OpenSSL や MIHCrypto など、さまざまな wrapper がある。
- LibSodium : Sodium は、暗号化、復号、署名、パスワードハッシュなどのための最新の使いやすいソフトウェアライブラリである。これは、移植可能で、クロスコンパイル可能で、インストール可能で、パッケージ化可能な NaCl の fork であり、互換性のある API と使いやすさをさらに向上させる拡張 API を備えている。詳細については、LibSodium のドキュメントを参照。Swift-sodium, NAClchloride, libsodium-ios などの wrapper ライブラリがいくつかある。
- Tink : Google による新しい暗号化ライブラリである。Google は、セキュリティブログでライブラリの背後にある理由を説明している。ソースは Tinks GitHub リポジトリにある。
- Themis : Swift, Obj-C, Android/Java, C++, JS, Python, Ruby, PHP, Go のストレージとメッセージング用の暗号化ライブラリである。Themis は依存関係として LibreSSL/OpenSSL エンジン libcrypto を使用する。キー生成、安全なメッセージング（ペイロードの暗号化と署名など）、安全なストレージ、および安全なセッションの設定のために、Objective-C と Swift をサポートする。詳細については、関係者の wiki を参照。
- Others : CocoaSecurity, Objective-C-RSA, aerogear-ios-crypto など、他にも多くのライブラリがある。これら的一部はもはやメンテナンスされておらず、セキュリティレビューが行われていない可能性がある。いつものように、サポートおよび保守されているライブラリを探すことが必要。

- DIY：暗号や暗号関数の実装を独自に作成する開発者が増えている。このような行為は非常に推奨されておらず、使用する場合は暗号化の専門家に十分に吟味してもらう必要がある。

参考資料

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

3.2.2.4 静的解析

非推奨のアルゴリズムと暗号化構成については、「Cryptography for Mobile Apps」セクションで多くのことが述べられている。当然ながらこれらは、この章で言及されているライブラリごとに検証する必要がある。キーを保持するデータ構造と平文のデータ構造の削除方法がどのように定義されているかに注意すること。キーワード "let" を使用すると、メモリから消去するのがより困難な不变の構造が作成される。メモリから簡単に削除できる親構造体の一部であることを確認すること（例：一時的に存在する構造体）。

CommonCryptor アプリが Apple が提供する標準の暗号化実装を使用している場合、関連するアルゴリズムのステータスを判断する最も簡単な方法は、CCCrypt や CCCryptorCreate などの CommonCryptor からの関数への呼び出しを確認することである。ソースコードには、CommonCryptor.h のすべての関数の署名が含まれている。例えば、CCCryptorCreate には次の署名がある。

```
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 */
```

次に、すべての列挙型を比較して、どのアルゴリズム、パディング、およびキーマテリアルが使用されているかを判断できる。キーマテリアルに注意すること。キーは、キー導出関数または乱数生成関数を使用して安全に生成する必要がある。「Cryptography for Mobile Apps」の章で非推奨として記載されている関数は、プログラムで引き続きサポートされていることに注意すること。これらの関数は使用しない。

サードパーティライブラリすべてのサードパーティライブラリが継続的に進化していることを考えると、静的解析の観点から各ライブラリを評価する場所であってはならない。それでも注意点がいくつかある：

- 使用されているライブラリを見つけること。これは、次の方法を使用して実行できる。
 - Carthage が使用されている場合は、`cartfile` を確認する。
 - Cocoapods が使用されている場合は、`podfile` を確認する。
 - リンクされたライブラリを確認する：`xcodeproj` ファイルを開き、プロジェクトのプロパティを確認する。[Build Phases] タブに移動し、[Link Binary With Libraries] のエントリでライブラリのいずれかを確認する。[MobSF](#) を使用して同様の情報を取得する方法については、前のセクションを参照。
 - コピー&ペーストされたソースの場合：ヘッダーファイル（Objective-C を使用している場合）を検索し、それ以外の場合は Swift ファイルで既知のライブラリの既知のメソッド名を検索する。

- 使用されているバージョンを特定する：使用しているライブラリのバージョンを常に確認し、潜在的な脆弱性や欠点にパッチが適用された新しいバージョンが利用可能かどうかを確認する。ライブラリの新しいバージョンがなくても、暗号化機能がまだレビューされていない場合がある。したがって、検証済みのライブラリを使用するか、自分で検証を行う能力、知識、および経験があることを確認することを推奨する。
- 手動：独自の暗号を開発したり、既知の暗号機能を自分で実装したりしないことを推奨する。

参考資料

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

3.2.3 ルールブック

1. iOS 暗号化アルゴリズム Apple CryptoKit の実装（推奨）

3.2.3.1 iOS 暗号化アルゴリズム Apple CryptoKit の実装（推奨）

Apple CryptoKit を使用して、以下の一般的な暗号化操作を実行する。

- 暗号的に安全なダイジェストを計算して比較する。
- 公開鍵暗号を使用して、デジタル署名を作成および評価し、鍵交換を実行する。メモリに保存されたキーを操作するだけでなく、Secure Enclave に保存され管理されている秘密鍵を使用することも可能。
- 対称鍵を生成し、メッセージ認証や暗号化などの操作で使用する。

低レベルのインターフェースよりも CryptoKit の使用を推奨する。CryptoKit は、生ポインターの管理からアプリを解放し、メモリの割り当て解除中に機密データを上書きするなど、アプリをより安全にするタスクを自動的に処理している。

CryptoKit を使ったハッシュ値生成

CryptoKit を使用すると、以下のハッシュを生成できる。

暗号的に安全なハッシュ

- struct SHA512
 - 512 ビット ダイジェストによる Secure Hashing Algorithm 2 (SHA-2) ハッシュの実装。
- struct SHA384
 - 384 ビット ダイジェストによる Secure Hashing Algorithm 2 (SHA-2) ハッシュの実装。
- struct SHA256
 - 256 ビット ダイジェストによる Secure Hashing Algorithm 2 (SHA-2) ハッシュの実装。

暗号的に安全でないハッシュ

- Insecure. struct MD5
 - MD5 ハッシュの実装。
- Insecure. struct SHA1
 - SHA1 ハッシュの実装。

```
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 古い、暗号的に安全でないアルゴリズムのコンテナ
    let hashed = Insecure.MD5.hash(data: data)

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

func createSha256Hashing() {
    // sha256 strings generated
    let hash = sha256Hash(str: "test") //_
    ↵9f86d081884c7d659a2feaa0c55ad015a3bf4f1b2b0b822cd15d6c15b0f00a08
}

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

CryptoKit を使った デジタル署名 (Cryptographic Signature)

CryptoKit を使用すると、以下の公開鍵暗号で署名と検証が可能

公開鍵暗号

- enum Curve25519
 - X25519 鍵合意と ed25519 署名を可能にする楕円曲線。
- enum P521
 - NIST P-521 署名と鍵合意を可能にする楕円曲線。
- enum P384

- NIST P-384 署名と鍵合意を可能にする楕円曲線。
- enum P256
 - NIST P-256 署名と鍵合意を可能にする楕円曲線。

```

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.
↪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")!
    }
}

```

(次のページに続く)

(前のページからの続き)

```

if cryptoSigning.isValid(rawPublicKey: rawPublicKey, signature:_  

↳signedSignature) {
    // 検証 OK
}
}
}

```

CryptoKit を使った 対象鍵暗号 (Symmetric Encryption)

CryptoKit を使用すると、以下の対称鍵暗号化方式で操作ができる。

暗号

- enum AES
 - Advanced Encryption Standard (AES) 暗号のコンテナ。
 - GCM モードを使用するには AES コンテナにある GCM を使用する。
- enum ChaChaPoly
 - ChaCha20-Poly1305 暗号の実装。

```

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)
    }
}

class ChaChaPolyEncryptionSample {
    func execChaChaPolyEncryption() {
}

```

(次のページに続く)

(前のページからの続き)

```

let encryption = ChaChaPolyEncryption()

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

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

}
}

```

CommonCrypto と SecKey による暗号化の実装

CryptoKit が誕生する以前は CommonCrypto や SecKey が利用されていた。現在 Apple は CryptoKit 利用を推奨しているが、OS バージョンなどの理由から利用できない場合は、OS 標準の API としてこちらを利用することができる。

以下のサンプルコードへ CommonCrypto の利用方法を示す。

```

#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)
    }

    /// バイナリ Data を 16進数文字列に変換する
    ///
    /// - Parameter binaryData: バイナリの入った Data
    /// - Returns: 16進数文字列
    public static func convetHexString(frombinary data: Data) -> String {

```

(次のページに続く)

(前のページからの続き)

```

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

}

public class AES {
    /// 暗号
    public static func encrypt(plainString: String, sharedKey: String, iv: String) throws -> Data {
        guard let initializeVector = (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)
        }

        // 暗号化後のデータのサイズを計算
        let cryptLength = size_t(Int(ceil(Double(data.count / kCCBlockSizeAES128)) + 1.0) * kCCBlockSizeAES128)

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

        // 暗号化
        let cryptStatus = cryptData.withUnsafeMutableBytes {cryptBytes in
            initializeVector.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
    }
}

```

(次のページに続く)

(前のページからの続き)

```

    }

    /// 復号
    public static func decrypt(encryptedData: Data, sharedKey: String, iv:_
→String) throws -> String {
        guard let initializeVector = (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

        // 復号
        let cryptStatus = clearData.withUnsafeMutableBytes {clearBytes in
            initializeVector.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)
        }

        // パディングされていた文字数分のデータを捨てて文字列変換を行う
        guard let decryptedStr = String(data: clearData.
→prefix(numBytesEncrypted), encoding: .utf8) else {
            throw Chiper.AESError.decryptFailed("PKSC Unpad Failed", clearData)
        }
        return decryptedStr
    }

    /// ランダム IV 生成
    public static func generateRandomIV() throws -> String {
        // CSPRNG から乱数取得
    }
}

```

(次のページに続く)

(前のページからの続き)

```

var randData = Data(count: 8)
let result = randData.withUnsafeMutableBytes {mutableBytes in
    SecRandomCopyBytes(kSecRandomDefault, 16, mutableBytes)
}
if result != errSecSuccess {
    // SecRandomCopyBytes に失敗 (本来はあり得ない)
    throw Chiper.AESError.otherFailed("SecRandomCopyBytes Failed"
→GenerateRandom IV", result)
}
// 16進数文字列化
let ivStr = Chiper.convertHexString(frombinary: randData)
return ivStr
}
}
}
}

```

以下のサンプルコードへ SecKey の利用方法を示す。

```

import Foundation

public class SecKeyHelper {

    /// base64pem 形式の鍵文字列を iOS で利用する SecKey 形式に変換する
    ///
    /// - Parameters:
    ///   - argBase64Key: base64pem 形式の公開鍵 あるいは秘密鍵
    ///   - keyType: kSecAttrKeyClassPublic/kSecAttrKeyClassPrivate
    /// - Returns: SecKey 形式の鍵データ
    /// - 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
    }
}

```

(次のページに続く)

(前のページからの続き)

```

    /// 公開鍵で暗号化を行う
    ///
    /// - Parameters:
    ///   - argBody: 対象文字列
    ///   - argBase64PublicKey: 公開鍵文字列 (base64)
    /// - Returns: 暗号データ
    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))
            // Crypto 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 {
            // エラー処理
        }

        return Data()
    }
}

```

これに注意しない場合、以下の可能性がある。

- 脆弱な暗号化実装となる可能性がある。

3.3 MSTG-CRYPTO-3

アプリは特定のユースケースに適した暗号化プリミティブを使用している。業界のベストプラクティスに基づくパラメータで構成されている。

3.3.1 問題のある暗号化構成

※問題のある暗号化構成については、「MSTG-CRYPTO-1 3.1.1. 問題のある暗号化構成」の内容を確認すること。

3.3.2 暗号化標準アルゴリズムの構成

※暗号化標準アルゴリズムの構成については、「MSTG-CRYPTO-2 3.2.2. 暗号化標準アルゴリズムの構成」の内容を確認すること。

3.4 MSTG-CRYPTO-4

アプリはセキュリティ上の目的で広く非推奨と考えられる暗号プロトコルやアルゴリズムを使用していない。

3.4.1 セキュアでない、または非推奨な暗号化アルゴリズム

モバイルアプリを評価する際には、重大な既知の弱点を持つ暗号アルゴリズムやプロトコルを使用していないこと、あるいは最新のセキュリティ要件に対して不十分な点がないことを確認する必要がある。過去に安全とされていたアルゴリズムも、時間の経過とともに安全でなくなる可能性がある。したがって、現在のベストプラクティスを定期的にチェックし、それに応じて設定を調整することが重要である。

暗号化アルゴリズムが最新のものであり、業界標準に準拠していることを確認する。脆弱なアルゴリズムには、旧式のブロック暗号（DES、3DESなど）、ストリーム暗号（RC4など）、ハッシュ関数（MD5、SHA1など）、壊れた乱数ジェネレーター（Dual_EC_DRBG、SHA1PRNGなど）が含まれる。認証されているアルゴリズム（NISTなど）でも、時間の経過とともに安全でなくなる可能性があることに注意する。認証は、アルゴリズムの健全性を定期的に検証することに取って代わるものではない。既知の弱点を持つアルゴリズムは、より安全な代替手段に置き換える必要がある。さらに、暗号化に使用されるアルゴリズムは標準化され、検証可能である必要がある。未知のアルゴリズムや独自のアルゴリズムを使ってデータを暗号化すると、アプリケーションがさまざまな暗号攻撃にさらされ、平文が復元される可能性がある。

アプリのソースコードを調査し、以下のような脆弱性が知られている暗号化アルゴリズムのインスタンスを特定する。

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

暗号化 API の名前は、特定のモバイルプラットフォームによって異なる。

次のことを確認する。

- 暗号化アルゴリズムは最新で、業界標準に準拠している。これには、時代遅れのブロック暗号 (DES など) ストリーム暗号 (RC4 など)、ハッシュ関数 (MD5 など)、Dual_EC_DRBG などの破損した乱数ジェネレーター (NIST 認定であっても) が含まれますが、これらに限定されない。これらはすべて安全でないものとしてマークし、使用せず、アプリケーションとサーバから削除する必要がある。
- キーの長さは業界標準に準拠しており、十分な時間の保護を提供する。ムーアの法則を考慮したさまざまなキーの長さとその保護性能の比較は、[オンライン](#)で確認可能である。
- 暗号化手段は互いに混合されていない: 例えば、公開鍵で署名したり、署名に使用した対称鍵を暗号化に再利用しない。
- 暗号化パラメータが合理的な範囲で適切に定義されている。これには、ハッシュ関数出力と少なくとも同じ長さである必要がある暗号ソルト、パスワード導出関数と反復回数の適切な選択 (例: PBKDF2、scrypt、bcrypt)、IV はランダムでユニークであること、目的に合ったブロック暗号化モード (例: ECB は特定の場合を除き使用しない)、キー管理が適切に行われているか (例: 3DES は 3 つの独立したキーを持つべきである) などが含まれるが、これらに限定されない。

参考資料

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

ルールブック

- セキュアでない、または非推奨な暗号化アルゴリズムは使用しない (必須)

3.4.2 ルールブック

1. セキュアでない、または非推奨な暗号化アルゴリズムは使用しない (必須)

3.4.2.1 セキュアでない、または非推奨な暗号化アルゴリズムは使用しない (必須)

業界標準に準拠した最新の暗号化アルゴリズムで実装すること。

具体的な観点としては以下内容に準拠して実装する。

- 暗号化アルゴリズムは最新で、業界標準に準拠している。業界標準については「[業界標準を満たしたキーの長さを設定する \(必須\)](#)」を参照。
- キーの長さは業界標準に準拠し、十分な時間の保護を提供する。ムーアの法則を考慮したさまざまなキーの長さとその保護性能の比較は、[オンライン](#)で確認可能である。
- 暗号化手段を互いに混合しない: 例えば、公開鍵で署名したり、署名に使用した対称鍵を暗号化に再利用しない。
- 暗号化パラメータを合理的な範囲で適切に定義する。これには、ハッシュ関数出力と少なくとも同じ長さである必要がある暗号ソルト、パスワード導出関数と反復回数の適切な選択 (例: PBKDF2、scrypt、

`bcrypt`)、IV はランダムでユニークであること、目的に合ったブロック暗号化モード (例: ECB は特定の場合を除き使用しない)、キー管理が適切に行われているか (例: 3DES は 3 つの独立したキーを持つべきである) などが含まれるが、これらに限定されない。

サンプルコードは、以下ルールブックを参照。

ルールブック

- *iOS 暗号化アルゴリズム Apple CryptoKit の実装 (推奨)*

これに違反する場合、以下の可能性がある。

- 脆弱な暗号化処理となる可能性がある。

3.5 MSTG-CRYPTO-5

アプリは複数の目的のために同じ暗号化鍵を再利用していない。

参考資料

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

3.5.1 キー管理の検証

端末にキーを保存する方法は様々である。キーを全く保存しないことで、キーマテリアルがダンプされないことが保証される。これは、PKBDF-2 などのパスワードキー導出機能を使用することで実現できる。以下の例を参照する。

```
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)!
```

(次のページに続く)

(前のページからの続き)

```

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)
}

```

- 出典：<https://stackoverflow.com/questions/8569555/pbkdf2-using-commoncrypto-on-ios> (Arcane ライブ テストスイートでテスト済み)

キーを保存する必要がある場合、選択した保護クラスが kSecAttrAccessibleAlways でない限り、Keychain を使用することが推奨される。NSUserDefaults、プロパティリストファイル、または Core Data や Realm からの他のシンクによって、他の場所にキーを保管することは、通常 Keychain を使用するよりも安全ではない。Core Data や Realm の同期が NSFileProtectionComplete データ保護クラスで保護されている場合でも、Keychain を使用することを推奨する。詳しくは「[iOS Data Storage](#)」を参照する。

Keychain は 2 種類の保存メカニズムをサポートしている。キーはセキュアエンクレーブに保存された暗号キーによって保護されるか、キー自体がセキュアエンクレーブ内にあるかのどちらかである。後者は、ECDH 署名キーを使用する場合のみ有効である。その実装の詳細については、[Apple Documentation](#) を参照する。

最後の 3 つのオプションは、ソースコードにハードコードされた暗号キーを使用する、安定した属性に基づいて予測可能なキー生成関数を持つ、生成されたキーを他のアプリケーションと共有する場所に保存することから構成される。ハードコードされた暗号化鍵を使用することは、アプリケーションのすべてのインスタンスが

同じ暗号化鍵を使用することを意味するため、明らかに推奨される方法ではない。攻撃者は、ソースコード（ネイティブに保存されているか、Objective-C/Swift で保存されているかに問わらず）からキーを抽出するために、一度だけ作業を行う必要がある。その結果、攻撃者は、アプリケーションによって暗号化された他のすべてのデータを解読することができる。次に、他のアプリケーションからアクセス可能な識別子に基づく予測可能なキー導出関数がある場合、攻撃者はキーを見つけるために、KDF を見つけてデバイスに適用するだけでよい。最後に、対称暗号化キーの公開保存は強く推奨されない。

暗号に関して忘れるべきでないもう 2 つの概念がある。

- 常に公開鍵で暗号化・検証し、秘密鍵で復号・署名すること。
- 別の目的で対称鍵を再利用しないこと。これにより、キーに関する情報が漏洩する可能性がある。署名用の対称鍵と暗号化用の対称鍵を別々に用意すること。

ルールブック

- メモリダンプを考慮してキーを使用する（必須）
- ソースコード内にキーやパスワードを保存しない（必須）
- 暗号に関して忘れるべきでない概念の遵守（必須）

データストレージとプライバシー要件ルールブック

- Keychain Services API* を使用してセキュアに値を保存する（必須）

3.5.1.1 静的解析

探すキーワードは様々である。「問題のある暗号化構成」の概要と静的解析で紹介したライブラリで、キーの保存方法についてどのキーワードを確認するのがベストなのか、確認する。

確認事項

- 高リスクのデータを保護するために使用する場合、キーはデバイス間で同期させないこと。
- キーは追加保護なしで保存されること。
- キーがハードコードされていないこと。
- キーは、デバイスの安定した機能から派生したものではないこと。
- キーが低水準言語（C/C++ など）の使用により隠されていないこと。
- キーが安全でない場所からインポートされないこと。

静的解析に関する推奨事項のほとんどは、「Testing Data Storage for iOS」の章にすでに記載されている。次に、以下のページで読み解くことができる。

- [Apple Developer Documentation: Certificates and keys](#)
- [Apple Developer Documentation: Generating new keys](#)
- [Apple Developer Documentation: Key generation attributes](#)

ルールブック

- ソースコード内にキーやパスワードを保存しない（必須）
- クライアント証明書のパスワードをローカルに保存しない、またはデバイスの *Keychain* にロックする（必須）
- クライアント証明書はすべてのインストール間で共有しない（必須）
- モバイルアプリで対称暗号化が使用される場合は常に秘密鍵を安全なデバイストレージに保存する（必須）

データストレージとプライバシー要件ルールブック

- Keychain Services API* を使用してセキュアに値を保存する（必須）

3.5.1.2 動的解析

暗号化方式をフックし、使用されているキーを分析する。暗号化処理中のファイルシステムのアクセスを監視し、キーの書き込み、読み出しを評価する。

3.5.2 ルールブック

1. 暗号に関して忘れるべきでない概念の遵守（必須）

3.5.2.1 暗号に関して忘れるべきでない概念の遵守（必須）

暗号に関して忘れるべきでないもう 2 つの概念がある。以下の概念を遵守する必要がある。

- 常に公開鍵で暗号化・検証し、秘密鍵で復号・署名すること。
- 別の目的で対称鍵を再利用しないこと。これにより、キーに関する情報が漏洩する可能性がある。署名用の対称鍵と暗号化用の対称鍵を別々に用意すること。

これに違反する場合、以下の可能性がある。

- キーに関する情報が漏洩する可能性がある。

3.6 MSTG-CRYPTO-6

すべての乱数値は十分にセキュアな乱数ジェネレーターを用いて生成されている。

参考資料

- owasp-mastg Testing Random Number Generation (MSTG-CRYPTO-6)

3.6.1 亂数ジェネレーターの選択

Apple は、暗号的に安全な乱数を生成する Randomization Services API を提供している。

Randomization Services API は、`SecRandomCopyBytes` 関数を使用して数値を生成する。これは、`/dev/random` デバイスファイルの wrapper 関数であり、0 から 255 までの暗号的に安全な疑似乱数値を提供する。すべての乱数がこの API で生成されていることを確認する。開発者が別のものを使う理由はない。

3.6.1.1 静的解析

`SecRandomCopyBytes` API は次のように定義されている。

Swift :

```
func SecRandomCopyBytes(_ rnd: SecRandomRef?,
                       _ count: Int,
                       _ bytes: UnsafeMutablePointer<UInt8>) -> Int32
```

Object-C version :

```
int SecRandomCopyBytes(SecRandomRef rnd, size_t count, uint8_t *bytes);
```

API の使用例を以下に示す。

```
int result = SecRandomCopyBytes(kSecRandomDefault, 16, randomBytes);
```

注意：もし他のメカニズムが乱数のために使われているならば、それらが上記の API の wrapper であるか、安全な乱数であるかどうかを検証する。多くの場合、これは難しそうなため、上記の実装に固執するのが最善である。

ルールブック

- *Randomization Services API を使用して安全な乱数を生成（推奨）*

3.6.1.2 動的解析

ランダム性をテストしたい場合は、大きな数値をキャプチャしてみて、Burp Sequencer のプラグインでランダム性の品質を確認できる。

3.6.2 ルールブック

1. *Randomization Services API を使用して安全な乱数を生成（推奨）*

3.6.2.1 Randomization Services API を使用して安全な乱数を生成（推奨）

鍵の生成やパスワード文字列を生成などの強度は、文字列の文字が完全にランダムである（そして隠されている）場合、攻撃者はブルート フォース攻撃で考えられるすべての組み合わせを 1 つずつ試すしかなくなる。十分に長い文字列の場合、解析は実行不可能になる。

ただし、ランダム化の品質に依存する。明確に定義されたルールに従って境界セットからのソフトウェア命令が実行されるような決定論的システムでは、真のランダム化は不可能である。十分にランダム化するサービスとして Apple では Randomization Services API を使用して、暗号的に安全な乱数を生成することができる。

```
import UIKit

class GenerateBitSample {
    func generate10bitRandom() -> [Int8?] {
        var bytes = [Int8](repeating: 0, count: 10)
        let status = SecRandomCopyBytes(kSecRandomDefault, bytes.count, &bytes)

        if status == errSecSuccess { // Always test the status.
            return bytes
        }

        return []
    }
}
```

これに注意しない場合、以下の可能性がある。

- 安全性の低い乱数が生成される。

4

認証とセッション管理要件

4.1 MSTG-AUTH-1

アプリがユーザにリモートサービスへのアクセスを提供する場合、ユーザ名/パスワード認証など何らかの形態の認証がリモートエンドポイントで実行されている。

※本章での適切な認証対応については「[MSTG-ARCH-2 の 1.2.1.1. 適切な認証対応](#)」を参照する。

参考資料

- [owasp-mastg Verifying that Appropriate Authentication is in Place \(MSTG-ARCH-2 and MSTG-AUTH-1\)](#)
- [owasp-mastg Testing OAuth 2.0 Flows \(MSTG-AUTH-1 and MSTG-AUTH-3\)](#)

4.2 MSTG-AUTH-2

ステートフルなセッション管理を使用する場合、リモートエンドポイントはランダムに生成されたセッション識別子を使用し、ユーザの資格情報を送信せずにクライアント要求を認証している。

4.2.1 セッション情報の管理

ステートフル(または「セッションベース」)認証は、クライアントとサーバの両方に認証レコードがあることが特徴である。認証の流れは以下の通りである。

1. アプリはユーザの認証情報を含む要求をバックエンドサーバに送信する。
2. サーバは認証情報を確認する。認証情報が有効な場合、サーバはランダムなセッション IDとともに新しいセッションを作成する。
3. サーバはセッション ID を含む応答をクライアントに送信する。
4. クライアントは以降のすべての要求でセッション ID を送信する。サーバはセッション ID を検証し、関連するセッションレコードを取得する。

5. ユーザがログアウトした後、サーバ側のセッションレコードは破棄され、クライアントはセッション ID を破棄する。

セッションの管理が不適切な場合、様々な攻撃に対して脆弱となり、正規のユーザのセッションが侵害され、攻撃者がそのユーザになりますことが可能になる。その結果、データの損失、機密性の低下、不正な操作が生じる可能性がある。

参考資料

- [owasp-mastg Testing Stateful Session Management \(MSTG-AUTH-2\)](#)

4.2.1.1 セッション管理のベストプラクティス

機密情報や機能を提供するサーバ側のエンドポイントを探し出し、一貫した認可の実施を検証する。バックエンドサービスは、ユーザのセッション ID またはトークンを検証し、ユーザがリソースにアクセスするための十分な権限を持っていることを確認する必要がある。セッション ID またはトークンがない、または無効な場合、要求を拒否する必要がある。

確認事項

- セッション ID は、サーバ側でランダムに生成される。
- ID は簡単に推測できないようにする。(適切な長さとエントロピーを使用する)
- セッション ID は常に安全な接続（例：HTTPS）で交換される。
- モバイルアプリはセッション ID を永続的なストレージに保存しない。
- サーバは、ユーザが権限を要するアプリケーションにアクセスするたびに、セッションを検証する。
(セッション ID は有効で、適切な認証レベルに対応している必要がある)
- セッションはサーバ側で終了し、タイムアウトまたはユーザがログアウトした後にモバイルアプリ内でセッション情報が削除される。

認証は実装するべきでなく、実績のあるフレームワークの上に構築する必要がある。多くの一般的なフレームワークは、既製の認証およびセッション管理機能を提供している。アプリが認証のためにフレームワークの API を使用する場合、ベストプラクティスのためにフレームワークのセキュリティドキュメントを確認する。一般的なフレームワークのセキュリティガイドは、以下のリンクで入手できる。

- [Spring \(Java \)](#)
- [Struts \(Java \)](#)
- [Laravel \(PHP \)](#)
- [Ruby on Rails](#)

サーバ側の認証を検証するための最適なリソースは、OWASP Web Testing Guide、特に、「Testing Authentication」と「Testing Session Management」の章に掲載されている。

参考資料

- owasp-mastg Testing Stateful Session Management (MSTG-AUTH-2) Session Management Best Practices

ルールブック

- モバイルアプリはセッション ID を永続的なストレージに保存しないことを確認する（必須）

4.2.2 ルールブック

1. モバイルアプリはセッション ID を永続的なストレージに保存しないことを確認する（必須）

4.2.2.1 モバイルアプリはセッション ID を永続的なストレージに保存しないことを確認する（必須）

永続的なストレージにセッション ID を保存すると、ユーザにより読み書きされたり、第三者により利用されたりする恐れがある。そのため、セッション ID はこのようなストレージには保存しないようにする必要がある。

※非推奨なルールのため、サンプルコードはなし。

これに違反する場合、以下の可能性がある。

- セッション ID をユーザにより読み書きされたり、第三者により利用されたりする恐れがある。

4.3 MSTG-AUTH-3

ステートレスなトークンベースの認証を使用する場合、サーバはセキュアなアルゴリズムを使用して署名されたトークンを提供している。

4.3.1 トークンの管理

トークンベースの認証は、HTTP 要求ごとに署名されたトークン（サーバによって検証される）を送信することによって実装される。最も一般的に使用されているトークン形式は、RFC7519 で定義されている JSON Web Token である。JWT は、完全なセッション状態を JSON オブジェクトとしてエンコードすることができる。そのため、サーバはセッションデータや認証情報を保存する必要がない。

JWT トークンは、ドットで区切られた 3 つの Base64Url エンコード部分から構成されている。トークンの構造は以下の通りである。

```
base64UrlEncode(header).base64UrlEncode(payload).base64UrlEncode(signature)
```

次の例は、Base64Url でエンコードされた JSON Web Token を示している。

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydW9.TJVA95OrM7E2cBab30RMHrHDcEfxfjoYZgeFONFh7HgQ
```

ヘッダーは通常 2 つの部分から構成される。トークンタイプ（JWT）と、署名を計算するために使用されるハッシュアルゴリズムである。上の例では、ヘッダーは以下のようにデコードされる。

```
{ "alg": "HS256", "typ": "JWT" }
```

トークンの 2 番目の部分はペイロードで、いわゆるクレームを含んでいる。クレームは、あるエンティティ（通常はユーザ）に関するステートメントと、追加のメタデータである。例を以下に示す。

```
{ "sub": "1234567890", "name": "John Doe", "admin": true }
```

署名は、符号化されたヘッダー、符号化されたペイロード、および秘密値に対して、JWT ヘッダーで指定されたアルゴリズムを適用することによって作成される。例えば、HMAC SHA256 アルゴリズムを使用する場合、署名は以下のように作成される。

```
HMACSHA256(base64UrlEncode(header) + "." + base64UrlEncode(payload), secret)
```

この秘密情報は、認証サーバとバックエンドサービスの間で共有され、クライアントはそれを認識しないことに注意する。これは、トークンが正当な認証サービスから取得されたことを証明するものである。また、クライアントがトークンに含まれるクレームを改ざんすることも防ぐことができる。

参考資料

- [owasp-mastg Testing Stateless \(Token-Based\) Authentication \(MSTG-AUTH-3\)](#)

4.3.2 静的解析

サーバとクライアントが使用している JWT ライブラリを特定する。使用している JWT ライブラリに既知の脆弱性があるかどうかを確認する。

実装が JWT の [ベストプラクティス](#) を遵守していることを確認する。

- トークンを含むすべての受信要求について、HMAC が検証されることを確認する。
- 秘密署名鍵または HMAC 秘密鍵の場所を確認する。キーはサーバに保管し、決してクライアントと共有してはならない。発行者と検証者のみが利用可能であるべきである。
- JWT に個人を特定できる情報のような機密データが埋め込まれていないことを確認する。何らかの理由でトークン内にそのような情報を送信する必要があるアーキテクチャの場合、ペイロードの暗号化が適用されていることを確認する。[OWASP JWT Cheat Sheet](#) にある Java 実装のサンプルを参照する。
- JWT に一意な識別子を与える jti (JWT ID) クレームでリプレイ攻撃に対処していることを確認する。
- クロスサービスリレー攻撃には、トークンがどのアプリケーションのためのものかを定義する aud (audience) クレームで対処していることを確認する。
- トークンは、Keychain (iOS) により、モバイルデバイス上に安全に保管されていることを確認する。

参考資料

- [owasp-mastg Testing Stateless \(Token-Based\) Authentication \(MSTG-AUTH-3\) Static Analysis](#)

ルールブック

- 使用している JWT ライブラリの特定及び既知の脆弱性があるかどうかを確認する（必須）
- トークンは、Keychain (iOS) により、モバイルデバイス上に安全に保管されていることを確認する（必須）

4.3.2.1 ハッシュアルゴリズムの強制

攻撃者はトークンを変更し、「none」キーワードを使って署名アルゴリズムを変更し、トークンの完全性がすでに検証されていることを示すことによってこれを実行する。ライブラリによっては、「none」アルゴリズムで署名されたトークンを、検証済みの署名を持つ有効なトークンであるかのように扱い、アプリケーションは変更されたトークンの要求を信頼する可能性がある。

参考資料

- owasp-mastg Testing Stateless (Token-Based) Authentication (MSTG-AUTH-3) Enforcing the Hashing Algorithm

ルールブック

- 期待するハッシュアルゴリズムを要求する必要がある（必須）

4.3.2.2 トークンの有効期限

署名されたステートレス認証トークンは、署名キーが変更されない限り永久に有効である。トークンの有効期限を制限する一般的な方法は、有効期限を設定することである。トークンに "exp" expiration claim が含まれ、バックエンドが期限切れのトークンを処理しないことを確認する。

トークンを付与する一般的な方法は、アクセストークンとリフレッシュトークンを組み合わせることである。ユーザがログインすると、バックエンドサービスは有効期間の短いアクセストークンと有効期間の長いリフレッシュトークンを発行する。アクセストークンの有効期限が切れた場合、アプリケーションはリフレッシュトークンを使って新しいアクセストークンを取得することができる。

機密データを扱うアプリの場合、リフレッシュトークンの有効期限が適切な期間であることを確認する必要がある。次のコード例では、リフレッシュトークンの発行日をチェックするリフレッシュトークン API を紹介する。トークンが 14 日以上経過していない場合、新しいアクセストークンが発行される。それ以外の場合は、アクセスが拒否され、再度ログインするよう促される。

```
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
})
```

(次のページに続く)

(前のページからの続き)

```
    var renewed_access_token = jwt.sign(profile, secret, { expiresInMinutes: 60*5 })
  );
  res.json({ token: renewed_access_token });
});
```

参考資料

- owasp-mastg Testing Stateless (Token-Based) Authentication (MSTG-AUTH-3) Token Expiration

ルールブック

- リフレッシュトークンの有効期限に適切な期間を設定する（必須）

4.3.3 動的解析

動的解析を行いながら、以下の JWT の脆弱性を調査する。

- クライアント上のトークン保存場所
 - JWT を使用するモバイルアプリの場合、トークンの保管場所を確認する必要がある。
- 署名キーのクラック
 - トークン署名は、サーバ上の秘密キーを介して作成される。JWT を取得した後、オフラインで秘密鍵をブルートフォースするためのツールを選択する。
- 情報の開示
 - Base64Url でエンコードされた JWT をデコードし、それがどのようなデータを送信しているか、そのデータが暗号化されているかどうかを調べることができる。
- ハッシュアルゴリズムの改ざん
 - 非対称型アルゴリズムの使用。JWT は RSA や ECDSA のような非対称アルゴリズムをいくつか提供している。これらのアルゴリズムが使用される場合、トークンは秘密鍵で署名され、検証には公開鍵が使用される。もしサーバが非対称アルゴリズムで署名されたトークンを期待し、HMAC で署名されたトークンを受け取った場合、サーバは公開鍵を HMAC 密钥として扱う。そして、公開鍵はトークンに署名するために HMAC 密钥として採用され、悪用される可能性がある。
 - トークンヘッダの alg 属性を修正し、HS256 を削除して none に設定し、空の署名（例えば、signature = ""）を使用する。このトークンを使用し、要求で再生する。いくつかのライブラリは、none アルゴリズムで署名されたトークンを、検証済みの署名を持つ有効なトークンとして扱う。これにより、攻撃者は自分自身の「署名付き」トークンを作成することができる。

上記のような脆弱性をテストするために、2種類の Burp プラグインが用意されている。

- JSON Web Token Attacker
- JSON Web Tokens

また、その他の情報については、OWASP JWT Cheat Sheet を必ず確認すること。

参考資料

- owasp-mastg Testing Stateless (Token-Based) Authentication (MSTG-AUTH-3) Dynamic Analysis

4.3.4 OAuth 2.0 のフローテスト

OAuth 2.0 は、API やウェブ対応アプリケーションのネットワーク上で認可の決定を伝えるための委任プロトコルを定義している。ユーザ認証アプリケーションなど、さまざまなアプリケーションで使用されている。

OAuth2 の一般的な用途は以下の通りである。

- ユーザのアカウントを使用してオンラインサービスにアクセスする許可をユーザから取得する。
- ユーザに代わってオンラインサービスに対する認証を行う。
- 認証エラーの処理。

OAuth 2.0 によると、ユーザのリソースにアクセスしようとするモバイルクライアントは、まずユーザに認証サーバに対する認証を要求する必要がある。ユーザの承認後、認証サーバはユーザに代わってアプリが動作することを許可するトークンを発行する。OAuth2 仕様では、特定の種類の認証やアクセストークンの形式は定義されていないことに注意する。

OAuth 2.0 では、4 つの役割が定義されている。

- リソースオーナー：アカウントの所有者
- クライアント：アクセストークンを使ってユーザのアカウントにアクセスしようとするアプリケーション
- リソースサーバ：ユーザアカウントをホストする
- 認可サーバ：ユーザの身元を確認し、アプリケーションにアクセストークンを発行する

注：API はリソースオーナーと認可サーバの両方の役割を果たす。したがって、ここでは両方を API と呼ぶことにする。

抽象的なプロトコルのフロー

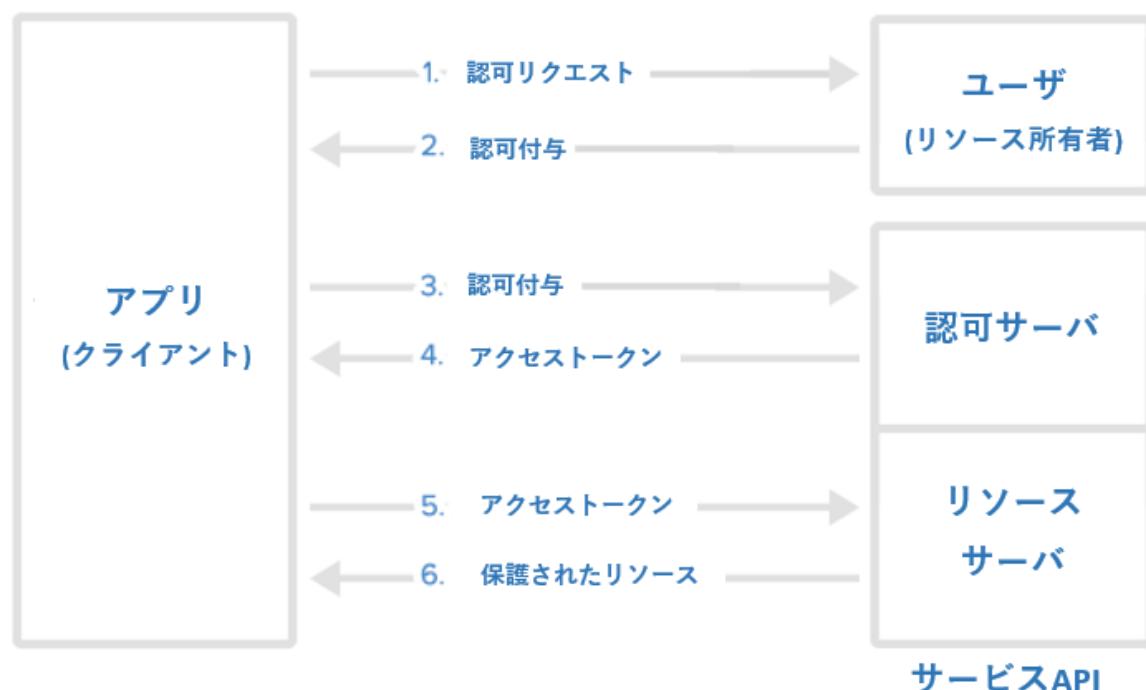


図 4.3.4.1 抽象的なプロトコルのフロー

図中の各ステップについて、以下でより詳細に説明する。

1. アプリケーションは、サービス・リソースにアクセスするためのユーザ認証を要求する。
2. ユーザが要求を承認すると、アプリケーションは認証付与を受け取る。認可付与は、いくつかの形式（明示的、暗黙的など）を取ることができる。
3. アプリケーションは、認可サーバ（API）にアクセストークンを要求する際に、認証付与と一緒に自身の ID の認証を提示する。
4. アプリケーションの ID が認証され、認証付与が有効である場合、認可サーバ（API）はアプリケーションにアクセストークンを発行し、認可プロセスを完了する。アクセストークンには、リフレッシュトークンが付随している場合がある。
5. アプリケーションはリソースサーバ（API）にリソースを要求し、認証のためにアクセストークンを提示する。アクセストークンは、いくつかの方法で使用することができる（ベアートークンなど）。
6. アクセストークンが有効であれば、リソースサーバ（API）はアプリケーションにリソースを提供する。

参考資料

- owasp-mastg Testing Stateless (Token-Based) Authentication (MSTG-AUTH-3)

4.3.5 OAuth 2.0 のベストプラクティス

以下のベストプラクティスが守られていることを確認する。

ユーザエージェント

- ユーザは信頼を視覚的に確認する方法（例：TLS(Transport Layer Security) 確認、ウェブサイトの仕組み）を持つべきである。
- 中間者攻撃を防ぐために、クライアントはサーバの完全修飾ドメイン名を、接続確立時にサーバが提示した公開鍵で検証する必要がある。

付与の種類

- ネイティブアプリでは、暗黙的な付与ではなく、コード付与を使用する必要がある。
- コード付与を使用する場合、コード付与を保護するためにPKCE (Proof Key for Code Exchange) を実装する必要があります。サーバ側でも実装されていることを確認する。
- 認可「コード」は有効期限が短く、受信後すぐに使用されるべきである。認可コードが一時的なメモリ上にのみ存在し、保存やログに記録されないことを確認する。

クライアントシークレット

- クライアントはなりすまされる可能性があるため、クライアントのIDを証明するために共有シークレットを使うべきではない（「client_id」がすでに証明の役目を果たしている）。もし、クライアントシークレットを使用する場合は、安全なローカルストレージに保存されていることを確認する。

エンドユーザの認証情報

- TLSなどのトランSPORT層方式でエンドユーザ認証情報の送信を保護する。

トークン

- アクセストークンは一時的なメモリに保存する。
- アクセストークンは、暗号化された接続で送信する必要がある。
- エンドツーエンドの機密性が保証されない場合や、トークンが機密情報や取引へのアクセスを提供する場合は、アクセストークンの範囲と期間を縮小する。
- アプリケーションがアクセストークンをペアラートークンとして使用し、クライアントを識別する他の方法がない場合、トークンを盗んだ攻撃者はそのスコープとそれに関連するすべてのリソースにアクセスできることに留意する。
- リフレッシュトークンは、安全なローカルストレージに保管する。

参考資料

- [owasp-mastg Testing OAuth 2.0 Flows \(MSTG-AUTH-1 and MSTG-AUTH-3\) OAUTH 2.0 Best Practices](#)

ルールブック

- [OAuth 2.0 のベストプラクティスの遵守（必須）](#)

4.3.5.1 外部ユーザエージェントと組み込みユーザエージェント

OAuth2 認証は、外部のユーザエージェント（Chrome や Safari など）を介して行うことも、アプリ自体（アプリに埋め込まれた WebView や認証ライブラリなど）を介して行うこともできる。どちらのモードが本質的に「優れている」ということはなく、どのモードを選択するかはコンテキストに依存する。

外部ユーザエージェントを使用する方法は、ソーシャルメディアアカウント（Facebook、Twitter など）と対話する必要があるアプリケーションで選択される方法である。この方法のメリットは以下の通りである。

- ユーザの認証情報は、決してアプリに直接公開されることはない。このため、ログイン時にアプリが認証情報を取得することはできない（「認証情報のフィッシング」）。
- アプリ自体に認証ロジックをほとんど追加する必要がないため、コーディングミスを防ぐことができる。

マイナス面としては、ブラウザの動作を制御する方法（証明書のピン留めを有効にする等）がないことである。

閉じたエコシステムの中で動作するアプリの場合、埋め込み認証の方が良い選択となる。例えば、OAuth2 を使って銀行の認証サーバからアクセストークンを取得し、そのアクセストークンを使って多くのマイクロサービスにアクセスする銀行アプリを考える。この場合、クレデンシャルフィッシングは実行可能なシナリオではない。認証プロセスは、外部のコンポーネントを信頼するのではなく、（できれば）慎重に保護されたバンкиングアプリの中で行なうことが望ましい。

参考資料

- [owasp-mastg Testing OAuth 2.0 Flows \(MSTG-AUTH-1 and MSTG-AUTH-3\) External User Agent vs. Embedded User Agent](#)

ルールブック

- *OAuth2 認証で使用される主な推奨ライブラリ（推奨）*
- *OAuth2 認証で外部ユーザエージェントを使用する場合はメリット、デメリットを理解して使用する（推奨）*

4.3.6 その他の OAuth 2.0 のベストプラクティス

その他のベストプラクティスや詳細情報については、以下のソースドキュメントを参照する。

- [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 \(January 2013\)](#)

参考資料

- [owasp-mastg Testing OAuth 2.0 Flows \(MSTG-AUTH-1 and MSTG-AUTH-3\) Other OAuth2 Best Practices](#)

4.3.7 ルールブック

1. 使用している JWT ライブラリの特定及び既知の脆弱性があるかどうかを確認する（必須）
2. トークンは、Keychain (iOS) により、モバイルデバイス上に安全に保管されていることを確認する（必須）
3. 期待するハッシュアルゴリズムを要求する必要がある（必須）
4. リフレッシュトークンの有効期限に適切な期間を設定する（必須）
5. OAuth 2.0 のベストプラクティスの遵守（必須）
6. OAuth2 認証で使用される主な推奨ライブラリ（推奨）
7. OAuth2 認証で外部ユーザエージェントを使用する場合はメリット、デメリットを理解して使用する（推奨）

4.3.7.1 使用している JWT ライブラリの特定及び既知の脆弱性があるかどうかを確認する（必須）

以下の内容を確認して、脆弱性があるか確認する必要がある。

- 使用している JWT ライブラリを特定し、既知の脆弱性があるかどうかを確認する。
- トークンを含むすべての受信要求について、HMAC が検証されることを確認する。
- 秘密署名鍵または HMAC 秘密鍵の場所を確認する。キーはサーバに保管し、決してクライアントと共有してはならない。発行者と検証者のみが利用可能であるべきである。
- JWT に個人を特定できる情報のような機密データが埋め込まれていないことを確認する。何らかの理由でトークン内にそのような情報を送信する必要があるアーキテクチャの場合、ペイロードの暗号化が適用されていることを確認する。OWASP JWT Cheat Sheet にある Java 実装のサンプルを参照する。
- JWT に一意な識別子を与える jti (JWT ID) クレームでリプレイ攻撃に対処していることを確認する。
- クロスサービスリレー攻撃には、トークンがどのアプリケーションのためのものかを定義する aud (audience) クレームで対処していることを確認する。
- トークンは、Keychain (iOS) により、モバイルデバイス上に安全に保管されていることを確認する。

以下サンプルコードは、Keychain からトークンを検索し、トークンが Keychain に存在しない場合は Keychain へ保存する処理の一例。

```
import Foundation

class Keychain {
    let group: String = "group_1" // グループ
    var id: String = "id"

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

    func addKeychain(token: Data) {
        // トークンを検索する
        if let existingToken = tokenDataForId(id) {
            // トークンが存在する場合、更新する
            updateToken(token: token, id: id)
        } else {
            // トークンが存在しない場合、新規登録する
            saveToken(token: token, id: id)
        }
    }

    func tokenDataForId(_ id: String) -> Data? {
        // Keychain からトークンを検索する実装
    }

    func updateToken(token: Data, id: String) {
        // トークンを更新する実装
    }

    func saveToken(token: Data, id: String) {
        // トークンを新規登録する実装
    }
}
```

(次のページに続く)

(前のページからの続き)

```

// API を実行する際の引数設定
let dic: [String: Any] = [kSecClass as String: kSecClassGenericPassword, // → クラス: パスワードクラス
    kSecAttrGeneric as String: group,                                     // 自由項目
    kSecAttrAccount as String: id,                                         // アカウント (ログイン ID)
    kSecValueData as String: token,                                         // パスワードなどの保存情報
    kSecAttrService as String: "key"]                                       // サービス名

// 検索用の Dictionary
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]

// keychain からトークンを検索
findKeychainItem(attributes: search as CFDictionary, { status, item in
    switch status {
        case errSecItemNotFound: // keychain にトークンが存在しない
            // keychain に保存
            _ = SecItemAdd(dic as CFDictionary, nil)
        default:
            break
    }
})}

/// keychain からアイテムが存在するかを検索する
/// - Parameters:
///   - attrs: 検索用のトークン
///   - completion: 検索結果
func findKeychainItem(attrs: CFDictionary, _ completion: @escaping _ OSStatus, CFTypeRef?) -> Void) {

    //呼び出しスレッドをブロックするため、メイン スレッドから呼び出された場合にアプリの UI がハングする可能性がある。
    // 別スレッドで実行することを推奨。
    backgroundQueue.async {
        var item: CFTypeRef?
        let result = SecItemCopyMatching(attrs, &item)
        completion(result, item)
    }
}
}

```

これに違反する場合、以下の可能性がある。

- 使用する JWT ライブラリに関する脆弱性を悪用される。

4.3.7.2 トークンは、Keychain (iOS) により、モバイルデバイス上に安全に保管されていることを確認する（必須）

トークンは Keychain によりモバイルデバイス上に安全に保管する必要がある。これを怠った場合、第三者によるトークンを利用したサーバ認証が可能になる恐れがある。

Keychain の使用方法は以下ルールを参照する。

データストレージとプライバシー要件ルールブック

- *Keychain Services API を使用してセキュアに値を保存する（必須）*

これに違反する場合、以下の可能性がある。

- 第三者によるトークンを利用したサーバ認証が可能になる

4.3.7.3 期待するハッシュアルゴリズムを要求する必要がある（必須）

攻撃者はトークンを変更し、「none」キーワードを使って署名アルゴリズムを変更し、トークンの完全性がすでに検証されていることを示す場合がある。以下は JWTDecode.swift を使って JWT をデコードする実装である。

```
import JWTDecode

class JWTSample {
    func jwtSample() {
        // JWT
        let jwtText = "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
        →eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6Ikpvag4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.
        →SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c"

        // ヘッダー、ボディの値を取り出す
        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!
```

(次のページに続く)

(前のページからの続き)

```
# Pods for SampleApp

target 'SampleAppTests' do
    # Pods for testing
end

pod 'JWTDecode'

end
```

これに違反する場合、以下の可能性がある。

- ・「none」アルゴリズムで署名されたトークンを、検証済みの署名を持つ有効なトークンであるかのように扱い、アプリケーションは変更されたトークンの要求を信頼する可能性がある。

4.3.7.4 リフレッシュトークンの有効期限に適切な期間を設定する（必須）

リフレッシュトークンの有効期限が適切な期間であることを確認する必要がある。リフレッシュトークンの有効期限内のは、新しいアクセストークンの発行をし、有効期限外の場合は、アクセスを拒否してユーザに再ログインを促す必要性がある。

これに違反する場合、以下の可能性がある。

- ・有効期限が不適切な場合はログインユーザを第三者が利用できる可能性がある。

4.3.7.5 OAuth 2.0 のベストプラクティスの遵守（必須）

以下の OAuth 2.0 のベストプラクティスを遵守すること。

ユーザエージェント

- ・ユーザは信頼を視覚的に確認する方法（例：TLS(Transport Layer Security) 確認、ウェブサイトの仕組み）を持つべきである。
- ・中間者攻撃を防ぐために、クライアントはサーバの完全修飾ドメイン名を、接続確立時にサーバが提示した公開鍵で検証する必要がある。

付与の種類

- ・ネイティブアプリでは、暗黙的な付与ではなく、コード付与を使用する必要がある。
- ・コード付与を使用する場合、コード付与を保護するために PKCE (Proof Key for Code Exchange) を実装する必要があります。サーバ側でも実装されていることを確認する。
- ・認可「コード」は有効期限が短く、受信後すぐに使用されるべきである。認可コードが一時的なメモリ上にのみ存在し、保存やログに記録されないことを確認する。

クライアントシークレット

- クライアントはなりすまされる可能性があるため、クライアントの ID を証明するために共有シークレットを使うべきではない（「client_id」がすでに証明の役目を果たしている）。もし、クライアントシークレットを使用する場合は、安全なローカルストレージに保存されていることを確認する。

エンドユーザの認証情報

- TLS などのトランSPORT層方式でエンドユーザ認証情報の送信を保護する。

トークン

- アクセストークンは一時的なメモリに保存する。
- アクセストークンは、暗号化された接続で送信する必要がある。
- エンドツーエンドの機密性が保証されない場合や、トークンが機密情報や取引へのアクセスを提供する場合は、アクセストークンの範囲と期間を縮小する。
- アプリケーションがアクセストークンをペアラートークンとして使用し、クライアントを識別する他の方法がない場合、トークンを盗んだ攻撃者はそのスコープとそれに関連するすべてのリソースにアクセスできることに留意する。
- リフレッシュトークンは、安全なローカルストレージに保管する。

これに違反する場合、以下の可能性がある。

- 脆弱な OAuth 2.0 の実装となる可能性がある。

4.3.7.6 OAuth2 認証で使用される主な推奨ライブラリ（推奨）

OAuth2 認証の実装を独自開発しないことが重要となる。

OAuth2 認証を実現するために利用されるライブラリは主に以下である。

- GTMAppAuth (Google)

gtm-oauth2 (Google) は現在非推奨となっている。代替に GTMAppAuth (Google) を使用となる。

Podfile の指定:

```
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
```

認証フロー

認可リクエストを開始するには authStateByPresentingAuthorizationRequest 関数を使用する。この関数内で OAuth トークンの交換が自動的に実行され、すべてが PKCE で保護される動作となる。(サーバサポートしている場合)。この関数の戻り値で OIDExternalUserAgentSession インスタンスを取得する。

認証フローではカスタム URL スキーマで承認用のリダイレクト URI(認証応答 URL)が渡ってくる。認証フロー中で取得した OIDExternalUserAgentSession インスタンスに認証応答 URL を設定する必要がある。

以下サンプルコードは、gtm-oauth2 での OAuth2 認証の一例。

```
- (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 *_Nonnull 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]];
        }
    }];
}
```

(次のページに続く)

(前のページからの続き)

```

    }];
}

// 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;
}

```

authorization インスタンスの Keychain 保存

GTMAuthFetcherAuthorization 含まれている機能を使用して、OIDExternalUserAgentSession インスタンスを Keychain に簡単に保存できる。保存したインスタンスをリストアすることも簡単にできる。

Keychain の保存と削除:

```

- (void)saveState {

    if (_authorization.canAuthorize) {
        // save
        [GTMAuthFetcherAuthorization saveAuthorization:_authorization
        toKeychainForName:kAuthorizerKey];

    } else {
        // remove
        [GTMAuthFetcherAuthorization
        removeAuthorizationFromKeychainForName:kAuthorizerKey];
    }
}

```

Keychain から読み込み:

```

- (void)loadState {

    // Restore from Keychain
    GTMAuthFetcherAuthorization *authorization =

```

(次のページに続く)

(前のページからの続き)

```

↳ [GTMAuthFetcherAuthorization authorizationFromKeychainForName:kAuthzrKey];
    [self setGtmAuthorization:authorization];
}

```

実装全体の例

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 "GTMAuthExampleViewController.h"

@implementation AppDelegate

@synthesize window = _window;

- (BOOL)application:(UIApplication *)application
    didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {
    UIWindow *window = [[UIWindow alloc] initWithFrame:[UIScreen mainScreen].bounds];
    UIViewController *mainViewController =
        [[GTMAuthExampleViewController 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
    // an authorization response.
    if ([_currentAuthorizationFlow resumeExternalUserAgentFlowWithURL:url]) {
}

```

(次のページに続く)

(前のページからの続き)

```

    _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 "GTMAuthExampleViewController.h"
#import <QuartzCore/QuartzCore.h>

#import AppAuth;
#import GTMSessionFetcher;
#import GTMAuth;

#import "AppDelegate.h"

static NSString *const kIssuer = @"https://accounts.google.com";

static NSString *const kClientID = @"xxxxxx.apps.googleusercontent.com";

static NSString *const kRedirectURI =
    @"com.googleusercontent.apps.YOUR_CLIENT:/oauthredirect";

static NSString *const kAuthorizerKey = @"authorizationkey";

@interface GTMAuthExampleViewController () <OIDAuthStateChangeDelegate,
OIDAuthStateErrorDelegate>
@end

@implementation GTMAuthExampleViewController

- (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];
}

```

(次のページに続く)

(前のページからの続き)

```

}

- (void)setGtmAuthorization:(GTMAuthFetcherAuthorization*)authorization {
    if ([_authorization isEqual:authorization]) {
        return;
    }

    _authorization = authorization;
    [self stateChanged];
}

- (void)stateChanged {
    [self saveState];
}

- (void)didChangeState:(OIDAuthState *)state {
    [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;
}

```

(次のページに続く)

(前のページからの続き)

```

 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"];

    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 isEqualToString: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;
    };
}

```

(次のページに続く)

(前のページからの続き)

```

// 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

```

これに注意しない場合、以下の可能性がある。

- 脆弱な OAuth 2.0 の実装となる可能性がある。

4.3.7.7 OAuth2 認証で外部ユーザエージェントを使用する場合はメリット、デメリットを理解して使用する (推奨)

OAuth2 認証は、外部のユーザエージェント（Chrome や Safari など）を介して行うことできる。外部ユーザエージェントを利用時のメリット、デメリットは以下である。

メリット

- ユーザの認証情報は、決してアプリに直接公開されることはない。このため、ログイン時にアプリが認証情報を取得することはできない（「認証情報のフィッシング」）。
- アプリ自体に認証ロジックをほとんど追加する必要がないため、コーディングミスを防ぐことができる。

デメリット

- ・ブラウザの動作を制御する方法（証明書のピン留めを有効にする等）がない。

これに注意しない場合、以下の可能性がある。

- ・証明書のピン留めを有効にする必要がある際に、利用することができない。

4.4 MSTG-AUTH-4

ユーザがログアウトする際に、リモートエンドポイントは既存のセッションを終了している。

4.4.1 リモートセッション情報の破棄

このテストケースの目的は、ログアウト機能を検証し、それがクライアントとサーバの両方で効果的にセッションを終了させ、ステートレストークンを無効にするかどうかを判断することである。

サーバ側セッションの破棄に失敗することは、ログアウト機能の実装で最もよくあるエラーの一つである。このエラーは、ユーザがアプリケーションからログアウトした後も、セッションやトークンを保持し続ける。有効な認証情報を取得した攻撃者は、その情報を使い続け、ユーザのアカウントを乗っ取ることが可能である。

多くのモバイルアプリは、ユーザを自動的にログアウトさせない。顧客にとって不便だから、あるいはステートレス認証の実装時に決定されたからなど、さまざまな理由が考えられる。しかし、アプリケーションにはログアウト機能が必要であり、ベストプラクティスに従って実装し、ローカルに保存されたトークンやセッション識別子をすべて破棄する必要がある。セッション情報がサーバに保存されている場合、そのサーバにログアウト要求を送信することによっても破棄されなければならない。リスクの高いアプリケーションの場合、トークンを無効にする必要がある。トークンやセッション識別子を破棄しないと、トークンが流出した場合にアプリケーションに不正にアクセスされる可能性がある。なお、適切に消去されない情報は、デバイスのバックアップ時など、後で流出する可能性があるため、他の機密情報も同様に破棄する必要がある。

4.4.2 静的解析

サーバコードが利用可能な場合は、ログアウト機能によってセッションが正しく終了することを確認する。この検証は、テクノロジーによって異なる。以下は、サーバ側のログアウトが正しく行われるためのセッション終了の様々な例である。

- ・ Spring (Java)
- ・ Ruby on Rails
- ・ PHP

ステートレス認証でアクセストークンやリフレッシュトークンを使用する場合は、モバイルデバイスから破棄する必要がある。リフレッシュトークンはサーバ上で無効化する必要がある。

4.4.3 動的解析

動的アプリケーション解析用のインターフェンスプロキシを使用し、以下の手順を実行して、ログアウトが正しく実装されているかどうかを確認する。

1. アプリケーションにログインする。
2. 認証を必要とするリソースにアクセスする。通常は、アカウントに属する個人情報の要求である。
3. アプリケーションからログアウトする。
4. 手順 2 の要求を再送信して、再度データにアクセスする。

サーバ上でログアウトが正しく実装されている場合は、エラーメッセージまたはログインページへのリダイレクトがクライアントに返される。一方、手順 2 で得たのと同じレスポンスを受け取る場合、トークンあるいはセッション ID はまだ有効で、サーバ上で正しく終了していない。OWASP ウェブテストガイド (WSTG-SESS-06) には、詳しい説明とより多くのテストケースが含まれている。

参考資料

- [owasp-mastg Testing User Logout \(MSTG-AUTH-4\)](#)

ルールブック

- アプリにログイン機能が存在する場合のリモートセッション情報の破棄のベストプラクティス（推奨）

4.4.4 ルールブック

1. アプリにログイン機能が存在する場合のリモートセッション情報の破棄のベストプラクティス（推奨）

4.4.4.1 アプリにログイン機能が存在する場合のリモートセッション情報の破棄のベストプラクティス（推奨）

アプリにログイン機能が存在する場合、以下のベストプラクティスに従いリモートセッション情報を破棄する。

- アプリケーションにログアウト機能を用意する。
- ログアウト時にローカルに保存されたトークンやセッション識別子をすべて破棄する。
- セッション情報がサーバに保存されている場合、サーバにログアウト要求を送信して破棄する。
- リスクの高いアプリの場合、トークンを無効にする。

トークンやセッション識別子を破棄しないと、トークンが流出した場合にアプリケーションに不正にアクセスされる可能性がある。破棄されない情報は、デバイスのバックアップ時など、後で流出する可能性があるため、他の機密情報も同様に破棄する必要がある。

これに注意しない場合、以下の可能性がある。

- ログイン情報がメモリ内に残り、第三者に利用される。

4.5 MSTG-AUTH-5

パスワードポリシーが存在し、リモートエンドポイントで実施されている。

4.5.1 パスワードポリシー遵守

パスワードの強度は、認証にパスワードが使用される場合の重要な懸念事項である。パスワードポリシーは、エンドユーザが遵守すべき要件を定義するものである。パスワードポリシーは通常、パスワードの長さ、パスワードの複雑さ、およびパスワードのトポロジーを指定する。「強力な」パスワードポリシーは、手動または自動によるパスワードクラッキングを困難または不可能にする。以下のセクションでは、パスワードのベストプラクティスに関するさまざまな領域をカバーする。より詳細な情報は、[OWASP Authentication Cheat Sheet](#) を参照する。

参考資料

- [owasp-mastg Testing Best Practices for Passwords \(MSTG-AUTH-5 and MSTG-AUTH-6\)](#)

4.5.2 静的解析

パスワードポリシーの存在を確認し、長さと無制限の文字セットに焦点を当てた [OWASP Authentication Cheat Sheet](#) に従って、実装されたパスワードの複雑さの要件を検証すること。ソースコードに含まれるパスワード関連の関数をすべて特定し、それぞれで検証チェックが行われていることを確認する。パスワード検証機能を確認し、パスワードポリシーに違反するパスワードが拒否されることを確認する。

参考資料

- [owasp-mastg Testing Best Practices for Passwords \(MSTG-AUTH-5 and MSTG-AUTH-6\) Static Analysis](#)

ルールブック

- 「強力な」パスワードポリシー（推奨）

4.5.2.1 zxcvbn

`zxcvbn` は、パスワードクラッカーにヒントを得て、パスワードの強度を推定するために使用できる一般的なライブラリである。JavaScript で利用可能であるが、サーバ側では他の多くのプログラミング言語でも利用可能である。インストール方法は様々なので、Github のリポジトリを確認する。インストールすると、`zxcvbn` を使って、パスワードの複雑さと解読に必要な推測回数を計算することができる。

HTML ページに `zxcvbn` JavaScript ライブラリを追加した後、ブラウザのコンソールで `zxcvbn` コマンドを実行すると、スコアを含むパスワードのクラックの可能性に関する詳細情報を取得できる。

```
> zxcvbn('ThisShouldBeVeryHardToCrack!')
< {password: "ThisShouldBeVeryHardToCrack!", guesses: 9.71881e+21, guesses_log10: 21.98761309187359, sequence: Array
(5), calc_time: 14, ...}
  calc_time: 14
  crack_times_display: {online_throttling_100_per_hour: "centuries", online_no_throttling_10_per_second: "centuri...
  crack_times_seconds: {online_throttling_100_per_hour: 3.4987716e+23, online_no_throttling_10_per_second: 971881...
  feedback: {warning: "", suggestions: Array(0)}
  guesses:
  9.71881e+21
  guesses_log10: 21.98761309187359
  password: "ThisShouldBeVeryHardToCrack!"
  score: 4
  sequence: (5) [...], [...], [...], [...], [...]
  __proto__: Object
```

図 4.5.2.1.1 zxcvbn コマンド例

スコアは以下のように定義され、例えばパスワードの強度バーに使用することができる。

```
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)
```

なお、zxcvbn はアプリ開発者が Java（または他の）実装を使って実装することもでき、ユーザに強力なパスワードを作成するように導くことができる。

参考資料

- owasp-mastg Testing Best Practices for Passwords (MSTG-AUTH-5 and MSTG-AUTH-6) zxcvbn

4.5.3 Have I Been Pwned : PwnedPasswords

一要素認証スキーム（パスワードのみなど）に対する辞書攻撃の成功の可能性をさらに低くするために、パスワードがデータ漏洩で流出したかどうかを検証することができる。これは、Troy Hunt 氏による Pwned Passwords API をベースにしたサービス（api.pwnedpasswords.com で入手可能）を使って行うことができる。例えば、コンパニオンウェブサイトの「Have I been pwned？」である。この API は、パスワード候補の SHA-1 ハッシュをもとに、指定されたパスワードのハッシュが、このサービスが収集したさまざまなデータ漏洩の事例の中で見つかった回数を返す。ワークフローは以下のステップを実行する。

- ユーザ入力を UTF-8 にエンコードする。（例：the password test）
- ステップ 1 の結果の SHA-1 ハッシュを取得する。（例：test のハッシュは A94A8FE5CC...）
- 最初の 5 文字（ハッシュプレフィックス）をコピーし、次の API を用いて範囲検索に使用する：http GET https://api.pwnedpasswords.com/range/A94A8

- 結果を繰り返し、ハッシュの残りを探す。(たとえば、FE5CC... が返されたリストの一部であるかどうか) もしそれが返されたリストの一部でなければ、与えられたハッシュのパスワードは見つからない。そうでなければ、FE5CC... のように、それが違反で見つかった回数を示すカウンターを返す。(例: FE5CC... : 76479)

Pwned Passwords API に関する詳細なドキュメントは、[オンライン](#)で確認できる。

なお、この API は、ユーザが登録やパスワードの入力を行う際に、推奨パスワードかどうかを確認するために、アプリ開発者が使用するのが最適である。

参考資料

- [owasp-mastg Testing Best Practices for Passwords \(MSTG-AUTH-5 and MSTG-AUTH-6\) Have I Been Pwned: PwnedPasswords](#)

ルールブック

- [推奨パスワードかの確認 \(推奨\)](#)

4.5.3.1 ログイン制限

スロットリングの手順をソースコードで確認する：

指定されたユーザ名で短時間にログインを試行した場合のカウンタと、最大試行回数に達した後にログインを試行しない方法を提供する。認証されたログイン試行後は、エラーカウンターをリセットすること。

ブルートフォース対策として、以下のベストプラクティスを遵守する。

- ログインに数回失敗したら、対象のアカウントをロックし（一時的または恒久的に）、それ以降のログイン試行を拒否すべき。
- 一時的なアカウントロックには、5 分間のアカウントロックが一般的に使用される。
- クライアント側の制御は簡単にバイパスされるため、制御はサーバ上で行う必要がある。
- 不正なログインの試行は、特定のセッションではなく、対象となるアカウントについて集計する必要がある。

その他のブルートフォース軽減技術については、OWASP の[ブルートフォース攻撃の阻止](#)のページに記載されている。

参考資料

- [owasp-mastg Testing Best Practices for Passwords \(MSTG-AUTH-5 and MSTG-AUTH-6\) Login Throttling](#)

ルールブック

- [ブルートフォース対策として、以下のベストプラクティスを遵守する（必須）](#)

4.5.4 ルールブック

1. 「強力な」 パスワードポリシー（推奨）
2. 推奨パスワードかの確認（推奨）
3. ブルートフォース対策として、以下のベストプラクティスを遵守する（必須）

4.5.4.1 「強力な」 パスワードポリシー（推奨）

以下は「強力な」 パスワードを設定するためのポリシーの一例である。

- パスワードの長さ
 - 最小：8 文字未満のパスワードは脆弱であると見なされる。
 - 最大：一般的な最大長は 64 文字である。long password Denial of Service attacks による攻撃を防ぐために最大長を設定することは必要である。
- パスワードをユーザに通知なしで切り捨てない。
- パスワードの構成規則 Unicode と空白を含むすべての文字の使用を許可する。
- 資格情報のローテーションパスワードの漏洩が発生、または特定されたときに資格情報のローテーションを行う必要がある。
- パスワード強度メーターユーザが複雑なパスワードを作成する、過去に特定されたパスワードの設定をロックするためにパスワード強度メーターを用意する。

以下サンプルコードは、iOS アプリで対応すべき最大入力制限と、バリデーションチェックの処理の一例。

最大入力制限

```
import UIKit

class ClassSample {
    func textField(_ textField: UITextField, shouldChangeCharactersIn range: NSRange, replacementString string: String) -> Bool {

        if let text = textField.text,
           let textRange = Range(range, in: text) {
            let updatedText = text.replacingCharacters(in: textRange, with: string)
            if updatedText.count > 64{
                return false
            }
        }
        return true
    }
}
```

バリデーションチェック

```

import Foundation

class ValidateSample {
    func validate(text: String, with regex: String) -> Bool {
        // 長さチェック
        if text.count <= 64 || text.count >= 8 {
            return true
        }
        return false
    }
}

```

参考資料

- OWASP CheatSheetSeries Implement Proper Password Strength Controls

これに注意しない場合、以下の可能性がある。

- 脆弱なパスワードとなり予測される。

4.5.4.2 推奨パスワードかの確認（推奨）

Pwned Passwords API を使用することで、ユーザが登録やパスワードの入力を行う際に、推奨パスワードかどうかを確認することができる。

これに注意しない場合、以下の可能性がある。

- 脆弱なパスワードとなり予測される。

4.5.4.3 ブルートフォース対策として、以下のベストプラクティスを遵守する（必須）

ブルートフォース対策として、以下のベストプラクティスを遵守する。

- ログインに数回失敗したら、対象のアカウントをロックし（一時的または恒久的に）、それ以降のログイン試行を拒否すべき。
- 一時的なアカウントロックには、5分間のアカウントロックが一般的に使用される。
- クライアント側の制御は簡単にバイパスされるため、制御はサーバ上で行う必要がある。
- 不正なログインの試行は、特定のセッションではなく、対象となるアカウントについて集計する必要がある。

その他のブルートフォース軽減技術については、OWASP のブルートフォース攻撃の阻止のページに記載されている。

※サーバ側のルールのため、サンプルコードはなし。

これに違反する場合、以下の可能性がある。

- ブルートフォース攻撃に脆弱になる。

4.6 MSTG-AUTH-6

リモートエンドポイントは過度な資格情報の送信に対する保護を実装している。

※リモートサービス側での対応に関する章であるため、本資料ではガイド記載を省略

参考資料

- owasp-mastg Dynamic Testing (MSTG-AUTH-6)

4.7 MSTG-AUTH-7

事前に定義された非アクティブ期間およびアクセストークンの有効期限が切れた後に、セッションはリモートエンドポイントで無効にしている。

※リモートサービス側での対応に関する章であるため、本資料ではガイド記載を省略

参考資料

- owasp-mastg Testing Session Timeout (MSTG-AUTH-7)

4.8 MSTG-AUTH-12

認可モデルはリモートエンドポイントで定義および実施されている。

※リモートサービス側での対応に関する章であるため、本資料ではガイド記載を省略

5

ネットワーク通信要件

5.1 MSTG-NETWORK-1

データはネットワーク上で TLS を使用して暗号化されている。セキュアチャネルがアプリ全体を通して一貫して使用されている。

5.1.1 安全なネットワーク通信の設定

提示されたすべてのケースは、全体として慎重に分析する必要がある。例えば、アプリが Info.plist で平文トラフィックを許可していない場合でも、実際には HTTP トラフィックを送信している可能性がある。低レベルの API (ATS が無視される) を使用している場合や、クロスプラットフォームフレームワークの構成が不十分な場合は、このようなケースになる可能性がある。

重要：これらのテストは、アプリのメインコードだけでなく、アプリ内に組み込まれたアプリの拡張機能、フレームワーク、Watch アプリにも適用する必要がある。

詳しくは、Apple Developer Documentation の記事 「[Preventing Insecure Network Connections](#)」 と 「[Fine-tune your App Transport Security settings](#)」 を参照する。

参考資料

- [owasp-mastg Testing Data Encryption on the Network \(MSTG-NETWORK-1\)](#)

ルールブック

- [App Transport Security \(ATS\) を使用する \(必須\)](#)

5.1.2 静的解析

5.1.2.1 セキュアプロトコルでのネットワーク要求の検証

まず、ソースコードですべてのネットワーク要求を識別し、平文の HTTP URL が使用されていないことを確認する必要がある。[URLSession](#) (iOS の標準的な [URL Loading System](#) を使用) または [Network](#) (TLS と TCP および UDP へのアクセスを使用したソケットレベルの通信用) を使用して、機密情報が安全なチャネルを介して送信されることを確認する。

参考資料

- [owasp-mastg Testing Data Encryption on the Network \(MSTG-NETWORK-1\) Testing Network Requests over Secure Protocols](#)

ルールブック

- [URLSession を使用する \(推奨\)](#)
- [Network Framework を使用する \(推奨\)](#)

5.1.2.2 低レベルのネットワーキング API の使用状況を確認

アプリが使用するネットワーク API を特定し、低レベルのネットワーク API を使用しているかどうかを確認する。

Apple の推奨：アプリに高レベルのフレームワークを優先する「ATS は、アプリケーションが Network framework や CFNetwork のように低レベルのネットワーキングインターフェースを呼び出す場合には適用されない。このような場合、接続のセキュリティを確保する責任は、ユーザ自身にある。この方法で安全な接続を構築することはできるが、間違いが起こりやすく、コストもかかる。代わりに URL Loading System に頼るのが一般的に最も安全である。」(ソース参照)

アプリが [Network](#) や [CFNetwork](#) などの低レベルの API を使用している場合、それらが安全に使用されているかどうかを慎重に調査する必要がある。クロスプラットフォームフレームワーク (Flutter、Xamarin など) やサードパーティフレームワーク (Alamofire など) を使用しているアプリの場合、それらがベストプラクティスに従って安全に設定され使用されているかどうかを分析する必要がある。

アプリを確認する：

- サーバの信頼性評価を行う際に、チャレンジの種類とホスト名および認証情報を検証する。
- TLS エラーを無視しない。
- 安全でない TLS 設定を使用していない。(「TLS 設定の検証 (MSTG-NETWORK-2)」を参照)

これらの確認はあくまで参考であり、アプリごとに異なるフレームワークを使用している可能性があるため、特定の API を挙げることはできない。コードを調査する際の参考情報とすること。

参考資料

- [owasp-mastg Testing Data Encryption on the Network \(MSTG-NETWORK-1\) Check for Low-Level Networking API Usage](#)

ルールブック

- *App Transport Security (ATS) を使用する (必須)*
- *CFNetwork を使用する (非推奨)*
- *Network Framework を使用する (推奨)*
- *低レベルのネットワーキング API はベストプラクティスに従って安全に使用する (必須)*

5.1.2.3 平文トラフィックの検証

アプリが平文の HTTP トラフィックを許可していないことを確認する。iOS 9.0 以降では、平文の HTTP トラフィックはデフォルトでブロックされる（App Transport Security（ATS）による）が、アプリケーションが平文の HTTP を送信する方法は複数存在する。

- アプリの Info.plist にある NSAppTransportSecurity の NSAllowsArbitraryLoads 属性を true（または YES）に設定し、ATS が平文トラフィックを有効にするように設定する。
- Info.plist を取得する
- NSAllowsArbitraryLoads が、どのドメインでも、グローバルで true に設定されていないことを確認する。
- アプリケーションが WebView でサードパーティの Web サイトを開く場合、iOS 10 以降、NSAllowsArbitraryLoadsInWebContent を使用すると、WebView に読み込まれるコンテンツの ATS 制限を無効にすることができる。

Apple の警告：ATS を無効にすると、安全でない HTTP 接続が許可される。HTTPS 接続も許可され、デフォルトのサーバ信頼性評価が適用される。ただし、TLS（Transport Layer Security）プロトコルの最低バージョンを要求するなどの拡張セキュリティチェックは無効となる。ATS を使用しない場合、「[手動サーバ信頼性認証の実行](#)」で説明するように、デフォルトのサーバ信頼性要件を自由に緩和することもできる。

以下のスニペットは、アプリが ATS の制限をグローバルに無効化する脆弱性のある例である。

```
<key>NSAppTransportSecurity</key>
<dict>
    <key>NSAllowsArbitraryLoads</key>
    <true/>
</dict>
```

ATS は、アプリケーションのコンテキストを考慮して検討する必要がある。アプリケーションは、その意図する目的を果たすために、ATS の例外を定義する場合がある。例えば、Firefox の iOS アプリケーションは、グローバルに ATS を無効にしている。そうでなければ、ATS の要件をすべて満たしていない HTTP ウェブサイトに接続できないため、この例外は許容される。場合によっては、アプリケーションはグローバルに ATS を無効にするが、特定のドメインではメタデータを安全に読み込んだり、安全なログインを可能にするなどの目的で有効にすることがある。

ATS は、このための正的な理由を示す記述を含める必要がある。（例：「このアプリは、安全な接続をサポートしない別のエンティティによって管理されるサーバに接続する必要がある」）

参考資料

- owasp-mastg Testing Data Encryption on the Network (MSTG-NETWORK-1) Testing for Cleartext Traffic ルールブック
 - *App Transport Security (ATS)* を使用する (必須)

5.1.3 動的解析

テスト対象のアプリの送受信ネットワークトラフィックを傍受し、このトラフィックが暗号化されていることを確認する。ネットワークトラフィックの傍受は、以下のいずれかの方法で行うことができる。

- OWASP ZAP や Burp Suite のようなインターフェンスプロキシですべての HTTP(S) と WebSocket のトラフィックを取得し、すべての要求が HTTP ではなく HTTPS で行われることを確認する。
- Burp や OWASP ZAP のようなインターフェンスプロキシは、HTTP(S) トラフィックのみを表示する。しかし、Burp-non-HTTP-Extension などの Burp プラグインや mitm-relay というツールを使えば、XMPP やその他のプロトコルによる通信をデコードして可視化することが可能である。

一部のアプリケーションは、証明書のピン留めにより、Burp や OWASP ZAP のようなプロキシで動作しない場合があります。そのような場合は、「[Testing Custom Certificate Stores and Certificate Pinning](#)」を確認すること。

詳細は以下参照：

- 「[Testing Network Communication](#)」の章から「Intercepting Traffic on the Network Layer」
- 「[iOS Basic Security Testing](#)」の章から「Setting up a Network Testing Environment」

参考資料

- owasp-mastg Testing Data Encryption on the Network (MSTG-NETWORK-1) Dynamic Analysis ルールブック
 - *App Transport Security (ATS)* を使用する (必須)

5.1.4 ルールブック

1. *App Transport Security (ATS)* を使用する (必須)
2. *URLSession* を使用する (推奨)
3. *CFNetwork* を使用する (非推奨)
4. *Network Framework* を使用する (推奨)
5. 低レベルのネットワーキング API はベストプラクティスに従って安全に使用する (必須)

5.1.4.1 App Transport Security (ATS) を使用する (必須)

ATS では、すべての HTTP 接続に HTTPS を使用する必要がある。さらに、Transport Layer Security (TLS) プロトコルによって規定されたデフォルトのサーバ信頼評価を補完する拡張セキュリティチェックを課す。ATS は、最低限のセキュリティ仕様を満たさない接続をブロックする。

アプリケーション内で利用されているデータを可能な限りセキュアにするために、現時点でセキュアでない接続をしているのかどうかをまず把握することが重要である。

その確認のため、アクティブな ATS の例外をすべて無効にするために Info.plist の App Transport Security Settings の Allow Arbitrary Loads 属性に「NO」を設定する。アプリケーションがセキュアでない接続を行った場合、それぞれの接続に対し Xcode 上でランタイムエラーが発生する。

ATS が有効になるのは以下の場合である。

- Info.plist に App Transport Security Settings が未指定の場合。
- Info.plist で、Allow Arbitrary Loads 属性、Allows Arbitrary Loads for Media 属性、Allows Arbitrary Loads in Web Content 属性、NSExceptionAllowsInsecureHTTPLoads 属性に NO が指定され、NSExceptionRequiresForwardSecrecy 属性に YES が指定され、NSExceptionMinimumTLSVersion に 1.2 以上が指定されている場合。

Xcode 上での info.plist 設定:

App Transport Security Settings	Dictionary	(2 items)
Allows Arbitrary Loads for Media	Boolean	NO
Exception Domains	Dictionary	(1 item)
example.com	Dictionary	(1 item)
NSExceptionAllowsInsecureHTTPLoads	Boolean	0

図 5.1.4.1.1 ATS で除外する例

これに違反する場合、以下の可能性がある。

- セキュリティ仕様を満たさない接続を実施する可能性がある。

5.1.4.2 URLSession を使用する (推奨)

URL を操作し、標準のインターネットプロトコルを使用してサーバと通信するために、iOS には URL Loading System が用意されている。これを利用するために URLSession を使用する。

```
import UIKit

class Fetch {
    func getDataAPI(completion: @escaping (Any) -> Void) {
        let requestUrl = URL(string: "http://xxxxx")!
        // URLSession の datatask を使い、data 取得。
        let task = URLSession.shared.dataTask(with: requestUrl) { data, response, error in
            // エラーが返ってきたときの処理。
            if let error = error {

```

(次のページに続く)

(前のページからの続き)

```

        completion(error)
    } else if let data = data {
        completion(data)
    }
}
task.resume()
}
}

```

これに注意しない場合、以下の可能性がある。

- 機密情報が安全ではないチャネルを介して送信される可能性がある。

5.1.4.3 CFNetwork を使用する（非推奨）

CFNetwork は、iOS 2.0 で用意された BSD ソケットの操作、HTTP および FTP サーバ通信を行う API。

現在はそのほとんどが非推奨になっている。ATS を有効に機能させるためには URLSession（iOS の標準的な URL Loading System を使用）を推奨する。

これが非推奨である理由は以下である。

- 機密情報が安全ではないチャネルを介して送信される可能性がある。

5.1.4.4 Network Framework を使用する（推奨）

TLS、TCP、UDP や独自のアプリケーションプロトコルに直接アクセスする必要がある場合に使用する。HTTP (S) や URL ベースのリソースの読み込みの場合はこれまでと変わらず URLSession を使用する。

```

import Foundation
import Network

class NetworkUDP {
    // 定数
    let networkType = "_myApp._udp."
    let networkDomain = "local"

    private func startListener(name: String) {
        // using に udp で listener を作成
        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")

        // 新しいコネクション受診時の処理
        listener.newConnectionHandler = { [unowned self] (connection:_  
→NWConnection) in

```

(次のページに続く)

(前のページからの続き)

```

connection.start(queue: listnerQueue)
self.receive(on: connection)
}

// Listener 開始
listener.start(queue: listnerQueue)

}

private func receive(on connection: NWConnection) {
    /* データ受信 */
    connection.receive(minimumIncompleteLength: 0,
                        maximumLength: 65535,
                        completion:{(data, context, flag, error) in
        if let error = error {
            NSLog("\(#function), \(error)")
        } else {
            if data != nil {
                /* 受信データのデシリアライズ */

            }
            else {
                NSLog("receiveMessage data nil")
            }
        }
    })
}
}
}

```

これに注意しない場合、以下の可能性がある。

- ・機密情報が安全ではないチャネルを介して送信される可能性がある。

5.1.4.5 低レベルのネットワーキング API はベストプラクティスに従って安全に使用する（必須）

アプリが低レベルの API を使用している場合、それらが安全に使用されているかどうかを慎重に調査する必要がある。クロスプラットフォームフレームワーク（Flutter、Xamarin など）やサードパーティフレームワーク（Alamofire など）を使用しているアプリの場合、それらがベストプラクティスに従って安全に設定され使用されているかどうかを分析する必要がある。

アプリが以下に準拠していることを確認する：

- ・サーバの信頼性評価を行う際に、チャレンジの種類とホスト名および認証情報を検証する。
- ・TLS エラーを無視しない。
- ・安全でない TLS 設定を使用していない。（「TLS 設定の検証（MSTG-NETWORK-2）」を参照）

これらの確認はあくまで参考であり、アプリごとに異なるフレームワークを使用している可能性があるため、特定の API を挙げることはできない。コードを調査する際の参考情報とすること。

これに違反する場合、以下の可能性がある。

- アプリが安全性の低い通信を実施する可能性がある。

5.2 MSTG-NETWORK-2

TLS 設定は現在のベストプラクティスと一致している。モバイルオペレーティングシステムが推奨される標準規格をサポートしていない場合には可能な限り近い状態である。

5.2.1 推奨される TLS 設定

サーバ側で適切な TLS 設定を行うことも重要である。SSL プロトコルは非推奨であり、もはや使用するべきではない。また、TLS v1.0 と TLS v1.1 には既知の脆弱性があり、2020 年までにすべての主要なブラウザでその使用が非推奨となった。TLS v1.2 および TLS v1.3 は、安全なデータ通信のためのベストプラクティスと考えられている。

クライアントとサーバの両方が同じ組織で管理され、互いに通信するためだけに使用されている場合、設定を強化することでセキュリティを強化できる。

モバイルアプリケーションが特定のサーバに接続する場合、そのネットワークスタックを調整することで、サーバの構成に対して可能な限り高いセキュリティレベルを確保することができる。オペレーティングシステムのサポートが不十分な場合、モバイルアプリケーションはより弱い構成を使用せざるを得なくなる可能性がある。

アプリの目的の一部である可能性があるため、対応する正当な理由の確認を忘れないこと。

以下へ検討する対象となる正当な理由の一例を示す。

- アプリが、安全な接続をサポートしていない別のエンティティによって管理されているサーバに接続する必要がある。
- アプリが、安全な接続を使用するバージョンへアップグレードできず、パブリックホスト名を使用してアクセスする必要があるデバイスへの接続をサポートする必要がある。
- アプリが、さまざまなソースからの埋め込まれた Web コンテンツを表示する必要があるが、Web コンテンツの例外でサポートされているクラスを使用することはできない。
- アプリが、暗号化され個人情報を含まないメディアコンテンツを読み込む。

アプリを App Store に提出する際は、App Store が既定で安全な接続を確立できない理由を判断できるように、十分な情報を提供すること。

特定のエンドポイントに通信する際に、どの ATS 設定が使用できるかを確認することができる。macOS では、コマンドラインユーティリティの `nscurl` を使用することができる。指定されたエンドポイントに対して、異なる設定の並べ替えが実行され、検証される。ATS のデフォルトのセキュアな接続テストにパスしていれば、ATS はデフォルトのセキュアな設定で使用することができる。`nscurl` の出力に失敗がある場合は、クライアント側の ATS の設定を脆弱化するのではなく、サーバ側の TLS の設定をよりセキュアになるように変更する。詳細は [Apple Developer Documentation](#) の記事「Identifying the Source of Blocked Connections」を参照。

詳細は、「Testing Network Communication」の章の「Verifying the TLS Settings」の項を参照する。

参考資料

- owasp-mastg Verifying the TLS Settings (MSTG-NETWORK-2) Recommended TLS Settings
- owasp-mastg Testing the TLS Settings (MSTG-NETWORK-2)

ルールブック

- 安全な通信プロトコル（必須）
- App Transport Security (ATS) を使用する（必須）*

5.2.2 推奨される暗号スイート

暗号スイートの構造は以下の通りである。

Protocol_KeyExchangeAlgorithm_WITH_BlockCipher_IntegrityCheckAlgorithm

この構造には以下が含まれる。

- 暗号化で使用されるプロトコル
- TLS ハンドシェイク中にサーバとクライアントが認証に使用する鍵交換アルゴリズム
- メッセージストリームの暗号化に使用されるブロック暗号
- メッセージの認証に使用される完全性保証チェックアルゴリズム

例：TLS_RSA_WITH_3DES_EDE_CBC_SHA

上記の例では、以下の暗号化スイートが使用されている。

- プロトコルとしての TLS
- 認証のための RSA 非対称暗号化
- EDE_CBC モードによる対称暗号化のための 3DES
- 完全性のための SHA ハッシュアルゴリズム

TLSv1.3 では、鍵交換アルゴリズムは暗号スイートの一部ではなく、TLS ハンドシェイク中に決定されることに注意する。

次のリストでは、暗号スイートの各部分のさまざまなアルゴリズムを紹介する。

プロトコル：

- SSL v1
- SSL v2 - RFC 6176
- SSL v3 - RFC 6101

- TLS v1.0 - [RFC 2246](#)
- TLS v1.1 - [RFC 4346](#)
- TLS v1.2 - [RFC 5246](#)
- TLS v1.3 - [RFC 8446](#)

鍵交換アルゴリズム :

- 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](#)

ブロック暗号 :

- 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

完全性チェックアルゴリズム：

- MD5 - RFC 6151
- SHA - RFC 6234
- SHA256 - RFC 6234
- SHA384 - RFC 6234

暗号スイートの効率は、そのアルゴリズムの効率に依存することに注意する必要がある

以下のリソースは、TLS で使用するために推奨される最新の暗号スイートが含まれている。

- IANA が推奨する暗号スイートは、[TLS Cipher Suites](#) に記載されている。
- OWASP が推奨する暗号スイートは、[TLS Cipher String Cheat Sheet](#) に記載されている。

iOS の一部バージョンでは、推奨する暗号スイートに対応していないものもあるため、互換性のために、iOS のバージョンでサポートされている暗号スイートを確認し、上位の暗号スイートを選択することが可能である。

サーバが適切な暗号スイートをサポートしているかどうかを確認する場合は、さまざまなツールを使用できる。

- nscurl - 詳細については、[iOS ネットワーク通信](#) を参照。
- testssl.sh は、「TLS/SSL 暗号、プロトコル、およびいくつかの暗号の欠陥のサポートについて、任意のポートでサーバのサービスをチェックする無料のコマンドラインツール」である。

最後に、HTTPS 接続が終了するサーバまたは終了プロキシが、ベストプラクティスに従って設定されていることを確認する。[OWASP Transport Layer Protection cheat sheet](#) および [Qualys SSL/TLS Deployment Best Practices](#) を参照する。

参考資料

- [owasp-mastg Verifying the TLS Settings \(MSTG-NETWORK-2\) Cipher Suites Terminology](#)

ルールブック

- [TLS で推奨される暗号化スイート（推奨）](#)

5.2.3 ルールブック

1. 安全な通信プロトコル（必須）
2. [TLS で推奨される暗号化スイート（推奨）](#)

5.2.3.1 安全な通信プロトコル（必須）

サーバ側で適切な TLS 設定を行うことも重要である。SSL プロトコルは非推奨であり、もはや使用するべきではない。

非推奨プロトコル

- SSL
- TLS v1.0
- TLS v1.1

TLS v1.0 と TLS v1.1 については、2020 年までにすべての主要なブラウザでその使用が非推奨となった。

推奨プロトコル

- TLS v1.2
- TLS v1.3

これに違反する場合、以下の可能性がある。

- セキュリティエクスプロイトに対して脆弱である。

5.2.3.2 TLS で推奨される暗号化スイート（推奨）

以下は推奨される暗号化スイートの一例。（TLS Cipher Suites で推奨されている暗号化スイートの中で、iOS で定義されているものを記載。）

- 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

これに注意しない場合、以下の可能性がある。

- 脆弱な暗号化スイートを使用する可能性がある。

5.3 MSTG-NETWORK-3

セキュアチャネルが確立されたときに、アプリはリモートエンドポイントの X.509 証明書を検証している。信頼された CA により署名された証明書のみが受け入れられている。

※証明書検証に関する記載は「[安全なネットワーク通信の設定](#)」へ纏めて記載するため、本章での記載を省略

6

プラットフォーム連携要件

6.1 MSTG-PLATFORM-1

アプリは必要となる最低限のパーミッションのみを要求している。

iOS はすべてのサードパーティアプリを非特権モバイルユーザの下で実行させる。各アプリは固有のホームディレクトリを持ち、サンドボックス化されているため、保護されたシステムリソースや、システムまたは他のアプリによって保存されたファイルにアクセスすることができない。これらの制限は、サンドボックスポリシー（別名：プロファイル）を介して実装され、カーネル拡張を介して [Trusted BSD \(MAC\) Mandatory Access Control Framework](#) によって強制される。iOS は、コンテナと呼ばれるすべてのサードパーティアプリに汎用サンドボックスプロファイルを適用する。保護されたリソースやデータ（一部はアプリの機能としても知られている）へのアクセスは可能ですが、[entitlements](#) として知られる特別な許可によって厳密に制御されている。

一部の権限は、アプリの開発者が設定でき（データ保護や Keychain シェアなど）、インストール後に直接有効になる。しかし、他のものについては、アプリが保護されたリソースに初めてアクセスしようとしたときなどに、ユーザに明示的に問い合わせることになる。例えば：

- Bluetooth 周辺機器
- カレンダーデータ
- カメラ
- 連絡先
- Health sharing
- Health updating
- ホームキット
- 位置情報
- マイク
- モーション

- 音楽とメディアライブラリ
- 写真
- リマインダー
- Siri
- 音声認識
- TV プロバイダー

Apple はユーザのプライバシーを保護し、[パーミッションを求める方法を明確にする](#)よう促しているが、それでも、アプリが明白でない理由で多くのパーミッションを要求することがある。

カメラ、写真、カレンダーデータ、モーション、連絡先、音声認識などの一部の権限は、アプリがそのタスクを実行するために必要であるかどうかが明らかであるため、確認するのはかなり簡単なはずである。許可された場合、アプリは「カメラロール」(写真を保存するための iOS のシステム全体のデフォルトの場所) 内のすべてのユーザの写真にアクセスすることができる写真権限について、次の例のとおりとする。

- 典型的な QR コードスキャンアプリは、機能するために明らかにカメラを必要とするが、同様に写真の許可を要求している可能性がある。ストレージが明示的に必要な場合、撮影される写真の機密性によつては、これらのアプリはアプリのサンドボックスストレージを使用して、他のアプリ（写真の許可を持つ）がアクセスするのを避ける方がよい。機密データの保存に関する詳細については、「Data Storage on iOS」の章を参照してください。
- アプリによっては、写真のアップロードが必要なものもある（プロフィール写真など）。最近の iOS のバージョンでは、[UIImagePickerController](#) (iOS 11+) やそれに代わる [PHPickerViewController](#) (iOS 14+) などの新しい API が導入されている。これらの API はアプリとは別のプロセスで実行される。これらを使用すると、アプリは「カメラロール」全体ではなく、ユーザが選択した画像にのみ読み取り専用でアクセスすることができるようになる。これは、不要なパーミッションのリクエストを避けるためのベストプラクティスと考えられている。

Bluetooth や位置情報のようなその他の許可は、より詳細な検証手順を必要とする。これらはアプリが適切に機能するために必要かもしれません、これらのタスクで扱われるデータは適切に保護されていないかもしれない。詳細といいくつかの例については、以下の「静的解析」セクションの「ソースコードインスペクション」および「動的解析」セクションを参照。

機密性の高いデータを収集したり、単純に扱う場合（例えばキャッシュ）、アプリは、ユーザがそのデータを制御できるように、例えばアクセスを取り消したり削除できるような適切なメカニズムを提供する必要がある。しかし、機密データは保存やキャッシュされるだけでなく、ネットワーク経由で送信されることもある。どちらの場合も、アプリが適切なベストプラクティス（この場合、適切なデータ保護と転送セキュリティを実装すること）に従っていることを確認する必要がある。このようなデータを保護する方法についての詳細は、「Network APIs」の章を参照する。

このように、アプリの機能と権限の使用は、ほとんどが個人データの取り扱いを伴うため、ユーザのプライバシーを保護することが重要である。詳しくは、Apple Developer Documentation の "Protecting the User's Privacy" と "Accessing Protected Resources" の記事を参照。

参考資料

- owasp-mastg Testing App Permissions (MSTG-PLATFORM-1)

ルールブック

- *permission* のリクエスト方法 (必須)
- 写真にアクセスするための設定 (必須)
- 適切なデータ保護と転送セキュリティを実装 (必須)

6.1.1 Device Capabilities

Device capabilities は、互換性のあるデバイスのみが一覧表示され、アプリのダウンロードが許可されるようにするために、App Store によって使用される。この機能は、アプリの Info.plist ファイルの `UIRequiredDeviceCapabilities` キーで指定される。

```
<key>UIRequiredDeviceCapabilities</key>
<array>
    <string>armv7</string>
</array>
```

典型的に armv7 は、アプリが armv7 命令セット専用にコンパイルされていること、または 32/64 ビットのユニバーサルアプリであることを意味する。

例えば、あるアプリが動作するために完全に NFC に依存している場合がある（例えば、「NFC タグリーダー」アプリ）。アーカイブされた iOS Device Compatibility Reference によると、NFC は iPhone 7 (および iOS11) からしか利用できません。開発者は、nfc device capability を設定することで、すべての非互換デバイスを除外したいと思うかもしれません。

テストに関しては、`UIRequiredDeviceCapabilities` は、アプリがいくつかの特定のリソースを使用していることを示す単なる表示と考えることができる。app capabilities に関する entitlements とは異なり、Device Capabilities は、保護されたリソースに対するいかなる権利またはアクセス権も付与しない。そのためには、各機能に追加の設定手順が必要になる場合がある。

例えば、BLE がアプリのコア機能である場合、Apple の Core Bluetooth Programming Guide は、考慮すべきさまざまな事柄を説明している。

- BLE 非対応端末のアプリダウンロードを制限するために、Bluetooth-le device capability を設定することができる。
- BLE のバックグラウンド処理 が必要な場合は、bluetooth-peripheral や bluetooth-central (いずれも UIBackgroundModes)などのアプリ機能を追加する必要がある。

しかし、これだけではまだアプリが Bluetooth 周辺機器にアクセスするには不十分で、`NSBluetoothPeripheralUsageDescription` キーが Info.plist ファイルに含まれていなければならず、つまりユーザが積極的に許可を与える必要がある。詳細は、以下の「Info.plist ファイル内の目的文字列」を参照。

参考資料

- owasp-mastg Testing App Permissions (MSTG-PLATFORM-1) Overview Device Capabilities

6.1.2 Entitlements

Apple の iOS セキュリティガイドによると

entitlements とは、アプリに署名されたキーと値のペアで、UNIX user ID のような実行時の要因を超えた認証を可能にする。entitlements はデジタル署名されているため、変更することはできない。entitlements は、システムアプリやデーモンで広く使用されており、他のプロセスでは root として実行する必要がある特定の特権操作を実行することができる。これは、侵害されたシステムアプリやデーモンによる特権昇格の可能性を大幅に低減する。

多くの entitlements は、Xcode target editor の "Summary" タブを使用して設定することができる。他の entitlements は、ターゲットの entitlements プロパティリストファイルを編集する必要があるか、アプリを実行するために使用される iOS プロビジョニングプロファイルから継承される。

Entitlement Sources:

1. Entitlement Sources : アプリのコード署名に使用されるプロビジョニングプロファイルに埋め込まれた entitlements で、以下のもので構成されている。
 - Xcode プロジェクトの target Capabilities タブで定義された機能 and/or 。
 - 証明書、ID、およびプロファイルのウェブサイトの Identifiers セクションで構成される、アプリの App ID で有効なサービス。
 - プロファイル生成サービスによって注入されるその他の entitlements。
2. signing entitlements ファイルからの Entitlements。

Entitlement Destinations:

1. アプリの署名
2. アプリに埋め込まれたプロビジョニングプロファイル

Apple Developer Documentation にも説明がある。

- コード署名の間、アプリの有効な Capabilities/Services に対応する entitlement は、Xcode がアプリに署名するために選んだプロビジョニングプロファイルから、アプリの署名に転送される。
- プロビジョニングプロファイルは、ビルト中にアプリのバンドルに埋め込まれる (embedded.mobileprovision)。
- Xcode の "Build Settings" タブの "Code Signing Entitlements" セクションの Entitlements は、アプリの署名に転送される。

例えば、「 Default Data Protection 」 capability を設定したい場合、Xcode の「 Capabilities 」 タブで、Data Protection を有効にする必要がある。これは、デフォルト値 NSFileProtectionComplete を持つ com.apple.developer.default-data-protection entitlement として、Xcode によって <appname>.entitlements ファイルに直接書き込まれる。IPA では、embedded.mobileprovision に次のように記述されている。

```

<key>Entitlements</key>
<dict>
  ...
    <key>com.apple.developer.default-data-protection</key>
    <string>NSFileProtectionComplete</string>
</dict>

```

HealthKit のような他の機能については、ユーザに許可を求める必要があるため、entitlement を追加するだけでは不十分で、アプリの Info.plist ファイルに特別なキーと文字列を追加する必要がある。

次のセクションでは、これらのファイルについて詳しく説明し、それらを使用して静的および動的解析を実行する方法について説明する。

参考資料

- [owasp-mastg Testing App Permissions \(MSTG-PLATFORM-1\) Overview Entitlements](#)

ルールブック

- [iOS 10 以降、パーミッションの記載が必要な項目（必須）](#)

6.1.3 静的解析

iOS 10 以降、パーミッションの検査が必要なのは主にこの部分である。

- Info.plist ファイル内の目的文字列
- Code Signing Entitlements ファイル
- 埋め込み型プロビジョニングプロファイル
- コンパイルされたアプリバイナリに埋め込まれた Entitlements
- ソースコードの検査

参考資料

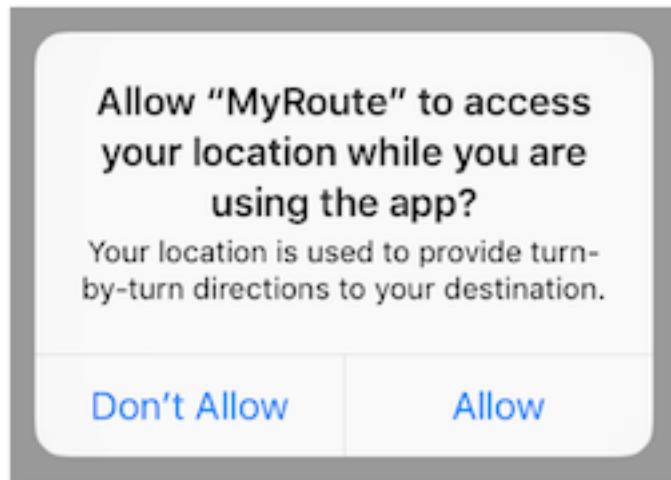
- [owasp-mastg Testing App Permissions \(MSTG-PLATFORM-1\) Static Analysis](#)

ルールブック

- [iOS 10 以降、パーミッションの記載が必要な項目（必須）](#)

6.1.3.1 Info.plist ファイル内の目的文字列

目的文字列または`_useDescription`は、保護されたデータまたはリソースへのアクセス許可を要求する際に、システムの許可要求アラートでユーザに提示されるカスタムテキストである。



iOS 10 以降でリンクする場合、開発者はアプリの Info.plist ファイルに目的文字列を含めることが要求される。そうしないと、アプリが保護されたデータやリソースにアクセスしようとしたときに、対応する目的文字列がない場合、アクセスに失敗し、アプリがクラッシュする可能性もある。

オリジナルのソースコードがある場合は、Info.plist ファイルに含まれるパーミッションを確認することができる。

- Xcode でプロジェクトを開く。
- Info.plist ファイルをデフォルトのエディタで開き、"Privacy -"で始まるキーを検索してください。

右クリックして「Show Raw Keys/Values」を選択すると、生の値を表示するように切り替えることができる（この方法では、例えば「Privacy - Location When In Use Usage Description」は「NSLocationWhenInUseUsageDescription」に変わる）。

Key	Type	Value
▼ Information Property List	Dictionary	(15 items)
NSLocationWhenInUseUsageDescription	String	◆ Your location is used to provide turn-by-turn directions to your destination.
CFBundleDevelopmentRegion	String	◆ \$(DEVELOPMENT_LANGUAGE)
CFBundleExecutable	String	◆ \$(EXECUTABLE_NAME)
CFBundleIdentifier	String	◆ \$(PRODUCT_BUNDLE_IDENTIFIER)
CFBundleInfoDictionaryVersion	String	◆ 0

IPA のみを持っている場合。

- IPA を解凍する。
- Info.plist は、Payload /<アプリ名>.app/ Info.plist にある。
- 必要に応じて変換する必要がある。（例：plutil -convert xml1 Info.plist）「iOS 基本セキュリティテスト」の章の「Info.plist ファイル」の項で詳しく説明されている。
- すべての目的の文字列 Info.plist のキー、通常は `UsageDescription` で終わっているものを検査する。

```
<plist version="1.0">
<dict>
    <key>NSLocationWhenInUseUsageDescription</key>
    <string>Your location is used to provide turn-by-turn directions to your
↳destination.</string>
```

CocoaKeys リファレンス で各キーの完全な説明を参照できる。

これらのガイドラインに従うと、許可が意味をなすかどうかをチェックするために、Info.plist ファイルの各エントリを評価するのが比較的簡単になる。

例えば、次の行がソリティアゲームで使用される Info.plist ファイルから抽出されたとする。

```
<key>NSHealthClinicalHealthRecordsShareUsageDescription</key>
<string>Share your health data with us!</string>
<key>NSCameraUsageDescription</key>
<string>We want to access your camera</string>
```

通常のソリティアゲームがこのようなリソースアクセスを要求するのは、おそらくカメラやユーザの健康記録にアクセスする必要がないため、疑わしいと考えるべきである。

パーミッションが意味をなすかどうかのチェックとは別に、目的文字列の分析からさらなる分析手順が導き出されるかもしない（たとえば、それらが機密データの保存に関連している場合など）。例えば、`NSPhotoLibraryUsageDescription` は、アプリのサンドボックスの外側にあり、他のアプリによってアクセス可能かもしれないファイルへのアクセスを与えるストレージ権限を考えることができる。この場合、機密データ（この場合は写真）がそこに保存されていないことをテストする必要がある。`NSLocationAlwaysUsageDescription` のような他の目的の文字列については、アプリがこのデータを安全に保存しているかどうかも考慮する必要がある。機密データを安全に保存するための詳細な情報とベストプラクティスについては、「Testing Data Storage」の章を参照。

參考資料

- owasp-mastg Testing App Permissions (MSTG-PLATFORM-1) Static Analysis Purpose Strings in the Info.plist File

6.1.3.2 Code Signing Entitlements ファイル

特定の機能は、code signing entitlements ファイル（.entitlements）を必要とする。これは Xcode によって自動的に生成されるが、開発者が手動で編集および/または拡張することもできる。

以下は、App Groups Entitlements（application-groups）を含む、オープンソースアプリ Telegram の entitlements ファイルの例である。

```
<?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>
...

```

(前のページからの続き)

```

<key>com.apple.security.application-groups</key>
<array>
    <string>group.ph.telegra.Telegraph</string>
</array>
</dict>
...
</plist>

```

上記の entitlements では、ユーザからの追加のパーミッションは必要ありません。しかし、アプリがユーザに過剰な権限を要求し、それによって情報が漏洩する可能性があるため、すべての権限を確認することは推奨する。

[Apple Developer Documentation](#) に記載されているように、App Groups の権限は、IPC または共有ファイルコンテナを通じて異なるアプリ間で情報を共有するために必要であり、これはデータがアプリ間で直接デバイス上で共有されることを意味する。この資格は、[拡張アプリ](#)がそのアプリと情報を共有する必要がある場合にも必要がある。

共有するデータによっては、ユーザ自身による改ざんを避けるため、データを検証できるバックエンドなど、別の方法で共有する方が適切な場合がある。

参考資料

- [owasp-mastg Testing App Permissions \(MSTG-PLATFORM-1\) Static Analysis Code Signing Entitlements File](#)

6.1.3.3 組み込み型プロビジョニングプロファイルファイル

ソースコードがない場合、IPA を分析し、通常、ルートアプリバンドルフォルダ (Payload/<アプリ名>.app/) に embedded.mobileprovision という名前で存在する埋め込みプロビジョニングプロファイルを内部から検索する必要がある。

このファイルは .plist ではなく、Cryptographic Message Syntax を使用してエンコードされている。macOS では、次のコマンドを使用して、埋め込みプロビジョニングプロファイルの entitlements を検査することができる。

```
security cms -D -i embedded.mobileprovision
```

そして、Entitlements キーリージョン (<key>Entitlements</key>) を検索する。

参考資料

- [owasp-mastg Testing App Permissions \(MSTG-PLATFORM-1\) Static Analysis Embedded Provisioning Profile File](#)

6.1.3.4 コンパイルされたアプリのバイナリに埋め込まれた entitlements

アプリの IPA や、ジェイルブレイクしたデバイスにインストールされたアプリしか持っていない場合、通常 entitlements ファイルを見つけることができない。これは、embedded.mobileprovision ファイルの場合もある。それでも、アプリのバイナリから entitlements のプロパティリストを自分で抽出することができるはずである（これは、「iOS Basic Security Testing」の章の「Acquiring the App Binary」のセクションで説明したように、以前に取得したことがある）。

以下の手順は、暗号化されたバイナリをターゲットにしている場合でも動作するはず。何らかの理由でうまくいかない場合、例えば Clutch（iOS のバージョンと互換性がある場合）、frida-ios-dump などを使ってアプリを復号し、抽出する必要がある。

- アプリのバイナリから Entitlements Plist を抽出する

アプリのバイナリがコンピュータにある場合、binwalk を使ってすべての XML ファイル (-y=xml) を抽出 (-e) するというのも一つの方法である。

```
$ binwalk -e -y=xml ./Telegram\ X

DECIMAL      HEXADECIMAL      DESCRIPTION
-----      -----
1430180      0x15D2A4      XML document, version: "1.0"
1458814      0x16427E      XML document, version: "1.0"
```

あるいは、radare2（-qc で静かに 1 コマンド実行して終了）を使って、アプリバイナリー（izz）上の「PropertyList」を含むすべての文字列（~PropertyList）を検索することができる。

```
$ 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<array>
\n\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>...
```

どちらの場合も（binwalk または radare2）同じ 2 つの plist ファイルを抽出することができた。最初のファイル（0x0015d2a4）を見てみると、Telegram から元の entitlements ファイルを完全に復元できていることがわかる。

注意：strings コマンドはこの情報を見つけることができないので、ここでは役に立ちません。バイナリ上で直接 grep を -a フラグ付きで使うか、radare2 (izz)/rabin2 (-zz) を使うのがよいでしょう。

ジェイルブレイクしたデバイスのアプリバイナリにアクセスする場合（例：SSH 経由）、-a、--text フラグで grep を使用できる（すべてのファイルを ASCII テキストとして扱う）。

```
$ 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>
        ...
    </array>
</dict>

```

-A num, --after-context=num のフラグで、表示する行数を増やしたり減らしたりできる。上記で紹介したようなツールも、ジェイルブレイクした iOS デバイスにインストールされているのであれば、利用してもよい。

この方法は、アプリのバイナリがまだ暗号化されている場合でも動作する可能性がある（いくつかの App Store アプリに対してテストされている）。

参考資料

- owasp-mastg Testing App Permissions (MSTG-PLATFORM-1) Static Analysis Entitlements Embedded in the Compiled App Binary

6.1.3.5 ソースコードインスペクション

<appname>.entitlements ファイルと Info.plist ファイルをチェックした後、要求されたパーミッションと割り当てられた機能がどのように使用されるかを確認する。これには、ソースコードのレビューで十分である。ただし、オリジナルのソースコードを持っていない場合、アプリをリバースエンジニアリングする必要があるかもしれませんため、パーミッションの使用を検証することは特別に難しいかもしれない。進め方の詳細については、「動的解析」を参照。

ソースコードレビューを行う際には、以下の点に注意する必要がある。

- Info.plist ファイル内の目的文字列とプログラム上の実装が一致しているかどうか。
- 登録された機能が機密情報を漏らさないように使用されているかどうか。

ユーザは「設定」によっていつでもパーミッションを付与したり取り消したりすることができるため、通常、アプリは機能にアクセスする前にその権限の状態を確認する。これは、保護されたリソースへのアクセスを提供する多くのシステムフレームワークで利用可能な専用の API を使用して行うことができる。

Apple Developer Documentation を参考にできる。例えば

- Bluetooth: CBCentralManager クラスの state プロパティは、Bluetooth 周辺機器を使用するためのシステム認証の状態を確認するために使用される。
- 位置情報：CLLocationManager のメソッドを検索する。例：locationServicesEnabled。

```
func checkForLocationServices() {
    if CLLocationManager.locationServicesEnabled() {
        // Location services are available, so query the user's location.
```

(次のページに続く)

(前のページからの続き)

```

} else {
    // Update your app's UI to show that the location is unavailable.
}
}

```

一覧は、"Determining a Availability of Location Services" (Apple Developer Documentation) の表 1 を参照。

これらの API を使用するアプリケーションを検索し、API から取得されるかもしれない機密データがどうなるかをチェックする。例えば、ネットワーク上で保存または転送される可能性があり、その場合、適切なデータ保護と転送セキュリティを追加で検証する必要がある。

参考資料

- owasp-mastg Testing App Permissions (MSTG-PLATFORM-1) Static Analysis Source Code Inspection

6.1.4 動的解析

静的解析の助けを借りて、すでに使用されている許可とアプリ機能のリストがあるはず。しかし、「ソースコードインスペクション」で述べたように、オリジナルのソースコードがない場合、これらの権限やアプリの機能に関連する機密データや API を発見することは困難な作業となる可能性がある。動的解析は、静的解析を反復するためのインプットを得るために役立つ。

以下に紹介するようなアプローチに従うと、前述の機密データや API を発見するのに役立つはず。

1. 静的解析で特定された権限／能力のリスト（例：NSLocationWhenInUseUsageDescription）を検討する。
2. それらを、対応するシステムフレームワーク（例：Core Location）で利用可能な専用 API にマッピングする。これには、Apple Developer Documentation を使用することができる。
3. それらの API（例えば CLLocationManager）のクラスや特定のメソッドを、例えば frida-trace を使用してトレースする。
4. 関連する機能（例えば「位置情報を共有する」）にアクセスしている間、アプリによって実際に使用されているメソッドを特定する。
5. それらのメソッドのバックトレースを取得し、コールグラフを作成する。

すべてのメソッドが特定されたら、この知識を使ってアプリをリバースエンジニアリングし、データがどのように処理されるかを調べることができる。そうしているうちに、プロセスに関わる新しいメソッドが見つかるかもしれない。それを再び上記のステップ 3 に送り、静的解析と動的解析を繰り返し行う。

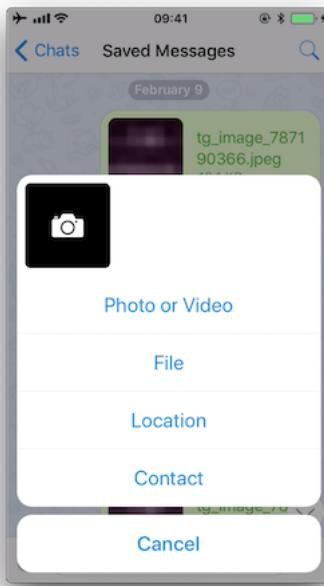
次の例では、Telegram を使ってチャットから共有ダイアログを開き、frida-trace でどのメソッドが呼び出されているかを確認する。

まず、Telegram を起動し、文字列 "authorizationStatus" にマッチするすべてのメソッドのトレースを開始する（CLLocationManager 以外の多くのクラスがこのメソッドを実装しているため、これは一般的なアプローチとなる）。

```
frida-trace -U "Telegram" -m "*[* *authorizationStatus*]"
```

-U は、USB デバイスに接続する。-m は、Objective-C のメソッドをトレースに含める。グローブパターンを使用することができる（例えば、「*」ワイルドカードを使用して、-m "*[* *authorizationStatus*]" は、「『 authorizationStatus 』を含む任意のクラスの任意の Objective-C メソッドを含める」ことを意味する）。詳細は frida-trace -h と入力する。

ここで、共有ダイアログを開く。



以下の方法が表示される。

```
1942 ms +[PHPhotoLibrary authorizationStatus]
1959 ms +[TGMediaAssetsLibrary authorizationStatusSignal]
1959 ms | +[TGMediaAssetsModernLibrary authorizationStatusSignal]
```

Location をクリックすると、別のメソッドがトレースされる。

```
11186 ms +[CLLocationManager authorizationStatus]
11186 ms | +[CLLocationManager _authorizationStatus]
11186 ms | | +[CLLocationManager _authorizationStatusForBundleIdentifier:bundle:]
11186 ms | | authorizationStatusForBundleIdentifier:0x0 bundle:0x0
```

戻り値やバックトレースなど、より多くの情報を得るには、frida-trace の自動生成されたタブを使用する。以下の JavaScript ファイルに以下の修正を加える（パスはカレントディレクトリからの相対パス）。

```
// __handlers__/_CLLocationManager_authorizationStatus_.js
```

```
onEnter: function (log, args, state) {
    log("+[CLLocationManager authorizationStatus]");
```

(次のページに続く)

(前のページからの続き)

```

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());
}
}

```

位置情報をもう一度クリックすると、さらに詳細な情報が表示される。

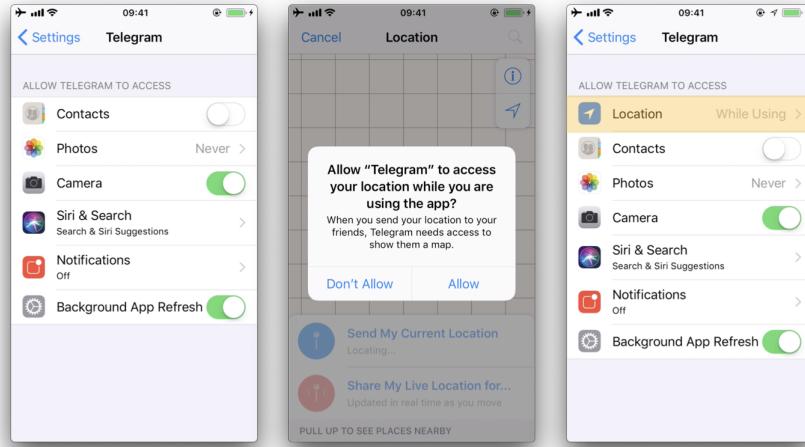
```

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

```

`CLLocationManager authorizationStatus]` が `0x4` (`CLAuthorizationStatus.authorizedWhenInUse` を返し、`+[TGLocationUtils requestWhenInUserLocationAuthorizationWithLocationManager:]` から呼び出されたことが分かる。前述したように、アプリをリバースエンジニアリングする際に、このような情報をエントリポイントとして使用し、そこから動的解析のための入力（クラスやメソッドの名前など）を取得し続けることができるかもしれません。

次に、iPhone/iPad を使用する際、「設定」を開き、興味のあるアプリが見つかるまでスクロールダウンすることで、いくつかのアプリのパーミッションの状態を視覚的に調査する方法がある。それをクリックすると、「ALLOW APP_NAME TO ACCESS」画面が表示される。しかし、すべての許可がまだ表示されていない可能性がある。その画面に表示されるためには、それらをトリガーする必要がある。



例えば、先ほどの例では、初めてアクセス許可ダイアログを表示するまで、「ロケーション」の項目は表示されない。一度許可すると、アクセスを許可したかどうかに関係なく、「ロケーション」の項目が表示されるようになる。

参考資料

- owasp-mastg Testing App Permissions (MSTG-PLATFORM-1) Dynamic Analysis

6.1.5 ルールブック

1. *permission* のリクエスト方法（必須）
2. 写真にアクセスするための設定（必須）
3. 適切なデータ保護と転送セキュリティを実装（必須）
4. iOS 10 以降、パーミッションの記載が必要な項目（必須）

6.1.5.1 permission のリクエスト方法（必須）

ユーザのリソースへアクセスするためには、ユーザの許可を得なければならない。そのための手段として、*permission* リクエスト時に OS 標準のアラートに説明文を乗せることが必要になる。説明文は plist に記載し、適切なタイミング（カメラを使用する画面の直前など）で許可のリクエストを処理するソースを書く必要がある。アプリ開始直後で不需要に許可をリクエストすることはしない。

permission の plist 記載：

Key	Type	Value
Information Property List	Dictionary	(3 items)
Privacy - Camera Usage Description	String	It is necessary for shooting with the camera and for video distribution.
> App Transport Security Settings	Dictionary	(2 items)
> Application Scene Manifest	Dictionary	(2 items)

図 6.1.5.1.1 カメラの permission 追加

plist の自動生成 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>
                    <key>UISceneStoryboardFile</key>
                    <string>Main</string>
                </dict>
            </array>
        </dict>
    </dict>
</dict>
</plist>
```

カメラの許可チェックとリクエスト処理:

```
import Foundation
import AVFoundation

class CameraHelper {
```

(次のページに続く)

(前のページからの続き)

```
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
            ()
    }
}
```

これに違反する場合、以下の可能性がある。

- 不要なパーミッションを要求すると、意図しないデータが漏洩する可能性がある。

6.1.5.2 写真にアクセスするための設定（必須）

アルバムから写真を取得する際には PhotosUI ライブラリの中にある PHPickerViewController を使用すると、permission の確認を行わずにユーザが画像選択する画面が表示できる。ユーザが選んだ写真以外の取得が必要でない限りこちらを選択すべきである。

アルバムに PHPickerViewController 経由で画像選択する方法:

```
import PhotosUI

class PViewController: UIViewController, PHPickerViewControllerDelegate {

    var selectedImages: [UIImage] = []

    override func viewDidLoad() {
        super.viewDidLoad()
    }

    func makeViewContoroller() {
        var configuration = PHPickerConfiguration()
        configuration.selectionLimit = 36 // 選択上限。0 にすると無制限に。
        configuration.filter = .images // 取得できるメディアの種類。
    }
}
```

(次のページに続く)

(前のページからの続き)

```

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)
        }
    }
}
}

```

これに違反する場合、以下の可能性がある。

- 不要なパーミッションを要求すると、意図しないデータが漏洩する可能性がある。

6.1.5.3 適切なデータ保護と転送セキュリティを実装（必須）

ネットワーク経由でデータを送信する場合。iOS の ATS を有効にすること。

ルールブック

- App Transport Security (ATS) を使用する（必須）*

これに違反する場合、以下の可能性がある。

- 意図しないデータが漏洩する可能性がある。

6.1.5.4 iOS 10 以降、パーミッションの記載が必要な項目（必須）

plist にパーミッションとして記載が必要になる操作と、それに紐づくキーを以下に示す。

- NSAppleMusicUsageDescription メディアライブラリへのアクセス
- NSCalendarsUsageDescription カレンダーへのアクセス
- NSContactsUsageDescription 連絡先へのアクセス

- NSPhotoLibraryUsageDescription フォトライブラリへのアクセス
- NSRemindersUsageDescription リマインダーへのアクセス
- NSCameraUsageDescription カメラへのアクセス
- NSMicrophoneUsageDescription マイクへのアクセス
- NSMotionUsageDescription 加速度計へのアクセス
- NSHealthShareUsageDescription ヘルスデータへのアクセス
- NSHealthUpdateUsageDescription ヘルスデータの変更
- NSHomeKitUsageDescription HomeKit の設定データへのアクセス
- NSSiriUsageDescription Siri ヘユーザデータ送信
- NSSpeechRecognitionUsageDescription Speech Recognition Server へのユーザデータ送信
- NSLocationWhenInUseUsageDescription 位置情報へのアクセス (使用中のみ許可)
- NFCReaderUsageDescription デバイスの NFC ハードウェアへのアクセス
- NSFaceIDUsageDescription Face ID で認証する
- NSPhotoLibraryAddUsageDescription フォトライブラリへの追加専用アクセス
- NSLocationAlwaysAndWhenInUseUsageDescription 位置情報へのアクセス (常時許可)
- NSHealthClinicalHealthRecordsShareUsageDescription 臨床記録の読み取り許可を要求
- NSHealthRequiredReadAuthorizationTypeIdentifiers 読み取り権限を取得しなければならない臨床記録データの種類
- NSBluetoothAlwaysUsageDescription Bluetooth へのアクセス
- NSLocationTemporaryUsageDescriptionDictionary 位置情報へのアクセス (1 度だけ許可)
- NSWidgetWantsLocation ウィジェットが位置情報を使用する
- NSLocationDefaultAccuracyReduced デフォルトで位置精度の低下を要求する
- NSLocalNetworkUsageDescription ローカルネットワークへのアクセス
- NSUserTrackingUsageDescription ユーザやデバイスを追跡するためのデータ使用許可
- NSSensorKitUsageDescription 調査研究の目的を簡単に説明する
- NSGKFriendListUsageDescription Game Center のフレンドリストへのアクセス
- NSNearbyInteractionUsageDescription 近くの機器とのインタラクションセッションを開始する
- NSIdentityUsageDescription ID 情報を要求する
- NSSensorKitPrivacyPolicyURL プライバシーポリシーを表示するウェブページへのハイパーリンク

- `UIRequiresPersistentWiFi` Wi-Fi 接続を必要とするか
- `NSSensorKitUsageDetail` アプリが収集する特定の情報のキーを含む辞書
 - `SRSSensorUsageKeyboardMetrics` キーボード操作を監視する
 - `SRSSensorUsageDeviceUsage` デバイスが起動する頻度を観察する
 - `SRSSensorUsageWristDetection` 時計をどのように装着しているかを観察する
 - `SRSSensorUsagePhoneUsage` 電話操作を観察する
 - `SRSSensorUsageMessageUsage` メッセージ内のユーザの活動を監視する
 - `SRSSensorUsageVisits` 頻繁に訪れる場所を観察する
 - `SRSSensorUsagePedometer` ステップ情報を観察する
 - `SRSSensorUsageMotion` モーションデータを観測する
 - `SRSSensorUsageSpeechMetrics` ユーザーの音声を分析する
 - `SRSSensorUsageAmbientLightSensor` 環境における光量を監視する

※ `NSLocationAlwaysUsageDescription` は iOS 11 から非推奨となつたため `NSLocationAlwaysAndWhenInUseUsageDescription` を使用すること。

※ `NSNearbyInteractionAllowOnceUsageDescription` は iOS 15 から非推奨となつたため `NSNearbyInteractionUsageDescription` を使用すること。

※ Bluetooth 周辺機器にアクセスする API を使用し、iOS 13 より前のデプロイメントターゲットを持つ場合、`NSBluetoothPeripheralUsageDescription` が必要。

これに違反する場合、以下の可能性がある。

- 指定の機能へのアクセスが出来ない。

6.2 MSTG-PLATFORM-2

外部ソースおよびユーザからの入力はすべて検証されており、必要に応じてサニタイズされている。これには UI、インテントやカスタム URL などの IPC メカニズム、ネットワークソースを介して受信したデータを含んでいる。

6.2.1 クロスサイトスクリプティングの問題

クロスサイトスクリプティング（XSS）の問題は、攻撃者がユーザが閲覧するウェブページにクライアント側のスクリプトを埋め込むことができるようにするものである。このタイプの脆弱性は、ウェブアプリケーションに広く存在する。ユーザが埋め込まれたスクリプトをブラウザで閲覧すると、攻撃者は同一オリジンポリシーをバイパスする権限を取得し、様々な悪用（セッションクッキーの窃盗、キー押下のログ、任意のアクションの実行など）が可能になる。

ネイティブアプリの場合、Web ブラウザに依存しないため、XSS のリスクははるかに低くなる。しかし、iOS の WKWebView や非推奨の UIWebView、Android の WebView などの WebView コンポーネントを使用したアプリは、このような攻撃に対して潜在的な脆弱性を持っている。

古い例だが、よく知られているのは、Phil Purviance が最初に発見した、iOS 用 Skype アプリのローカル XSS 問題である。この Skype アプリは、メッセージの送信者名を適切にエンコードしておらず、攻撃者は、ユーザがメッセージを閲覧した際に実行される悪意のある JavaScript を埋め込むことが可能であった。Phil は概念実証の中で、この問題を悪用し、ユーザのアドレス帳を盗む方法を示した。

参考資料

- [owasp-mastg Cross-Site Scripting Flaws \(MSTG-PLATFORM-2\)](#)

6.2.1.1 静的解析

WebView を確認して、アプリからレンダリングされた信頼できない入力がないか調査する。

WebView によって開かれた URL の一部がユーザの入力によって決定される場合、XSS 問題が存在する可能性がある。WebView を使用してリモート Web サイトを表示する場合、HTML をエスケープする負担はサーバ側に移行する。Web サーバに XSS の欠陥が存在する場合、これをを利用して WebView のコンテキストでスクリプトを実行することができる。そのため、Web アプリケーションのソースコードに対して静的解析を行うことが重要である。

以下のベストプラクティスに従っていることを確認する。

- HTML、JavaScript、その他の解釈されるコンテキストでは、絶対に必要な場合を除き、信頼できないデータはレンダリングされない。
- エスケープ文字には、HTML のエンティティエンコーディングなど、適切なエンコーディングが適用される。注：HTML が他のコードにネストされている場合、エスケープのルールは複雑になる。例えば、JavaScript ブロックの中にある URL を表示させる場合などである。

レスポンスでデータがどのようにレンダリングされるかを検討する。例えば、データが HTML コンテキストでレンダリングされる場合、エスケープする必要がある 6 つの制御文字が存在する。

表 6.2.1.1.1 エスケープする必要がある制御文字一覧

Character	Escaped
&	&
<	<
>	>
"	"
'	'
/	/

エスケープルールやその他の防止策の包括的なリストについては、OWASP XSS Prevention Cheat Sheet を参照する。

参考資料

- owasp-mastg Cross-Site Scripting Flaws (MSTG-PLATFORM-2) Static Analysis

ルールブック

- WebView を確認して、アプリからレンダリングされた信頼できない入力がないか調査する（必須）

6.2.1.2 動的解析

HTML タグや特殊文字を利用可能なすべての入力フィールドに注入し、ウェブアプリケーションが無効な入力を拒否するか、出力中の HTML メタ文字をエスケープするかを検証することで、XSS 問題を最もよく検出することができる。

reflected XSS 攻撃とは、悪意のあるリンクを経由して悪意のあるコードが挿入されるエクスプロイトを指す。このような攻撃をテストするためには、自動入力ファジングが有効な方法と考えられている。例えば、BURP Scanner は、reflected XSS 攻撃の脆弱性を特定するのに非常に有効である。自動解析の場合と同様に、テストパラメータを手動で確認し、すべての入力ベクトルがカバーされていることを確認する。

参考資料

- owasp-mastg Cross-Site Scripting Flaws (MSTG-PLATFORM-2) Dynamic Analysis

6.2.2 ルールブック

1. *WebView* を確認して、アプリからレンダリングされた信頼できない入力がないか調査する（必須）

6.2.2.1 **WebView** を確認して、アプリからレンダリングされた信頼できない入力がないか調査する（必須）

以下のベストプラクティスに従っていることを確認する。

- HTML、JavaScript、その他の解釈されるコンテキストでは、絶対に必要な場合を除き、信頼できないデータはレンダリングされない。
- エスケープ文字には、HTML のエンティティエンコーディングなど、適切なエンコーディングが適用される。注：HTML が他のコードにネストされている場合、エスケープのルールは複雑になる。例えば、JavaScript ブロックの中にある URL を表示させる場合などである。

レスポンスでデータがどのようにレンダリングされるかを検討する。例えば、データが HTML コンテキストでレンダリングされる場合、エスケープする必要がある 6 つの制御文字が存在する。

表 6.2.2.1.1 エスケープする必要がある制御文字一覧

Character	Escaped
&	&
<	<
>	>
"	"
'	'
/	

エスケープルールやその他の防止策の包括的なリストについては、OWASP XSS Prevention Cheat Sheet を参照する。

これに違反する場合、以下の可能性がある。

- XSS 問題が存在する可能性がある。

6.3 MSTG-PLATFORM-3

アプリはメカニズムが適切に保護されていない限り、カスタム URL スキームを介して機密な機能をエクスポートしていない。

6.3.1 カスタム URL スキーム

カスタム URL スキームにより、アプリはカスタムプロトコルを介して通信することができる。アプリは、このスキームをサポートすることを宣言し、そのスキームを使用する受信 URL を取り扱う必要がある。

Apple は、[Apple Developer Documentation](#)において、カスタム URL スキームの不適切な使用について警告している。

URL スキームは、アプリへの潜在的な侵入経路となるため、すべての URL パラメータを検証し、不正な URL は破棄すること。さらに、ユーザのデータを危険にさらすことのないよう、操作可能な項目を制限すること。例えば、他のアプリがコンテンツを直接削除したり、ユーザに関する機密情報にアクセスすることを許可しないようにする。URL 処理コードを検証する際には、テストケースに不適切な形式の URL が含まれていることを確認する。

また、ディープリンクの実装を目的とする場合は、ユニバーサルリンクの使用を推奨している。

カスタム URL スキームはディープリンクの一形態として認められるが、ユニバーサルリンクはベストプラクティスとして強く推奨される。

カスタム URL スキームのサポートは、以下の方法で行う。

- アプリの URL の形式を定義する。
- システムからアプリに適切な URL を割り当てるために、スキームを登録する。
- アプリが受け取る URL を処理する。

アプリが URL スキームの呼び出しを処理する際に、URL とその引数を適切に検証しない場合や、重要な処理を実行する前にユーザに同意を求めない場合、セキュリティ上の問題が発生する。

一例として、2010 年に発見された Skype Mobile アプリの次のような不具合がある。Skype アプリは skype:// プロトコルハンドラを登録し、別のアプリが他の Skype ユーザの電話番号へ通話を発信する前にユーザに許可を求めなかったため、どのアプリもユーザが知らないうちに任意の番号に通話することができた。攻撃者はこの脆弱性を利用して、目に見えない <iframe src="skype://xxx?call"></iframe> (xxx はプレミアム番号に置き換えられる) を設置し、悪意のある Web サイトを閲覧した Skype ユーザがプレミアム番号に通話できるようにしている。

開発者としては、URL を呼び出す前に、慎重に検証する必要がある。登録されたプロトコルハンドラを介して起動される特定のアプリケーションのみを許可することができる。また、URL から呼び出された操作を確認するようユーザに促すことも有効な制御方法である。

すべての URL は、アプリの起動時、アプリの実行中、またはバックグラウンドで、アプリのデリゲートに渡される。受け取った URL を処理するために、デリゲートは以下のようなメソッドを実装する必要がある。

- URL の情報を取得し、開くかどうかを決定する。
- URL で指定されたリソースを開く。

詳細は、過去の [App Programming Guide for iOS](#) と [Apple Secure Coding Guide](#) に記載されている。

さらに、アプリは他のアプリに URL リクエスト（別名：クエリ）を送信することもできる。これは次のようにして行われる。

- アプリが要求するアプリケーションクエリスキームを登録する。
- 任意で他のアプリに照会し、特定の URL を聞くことができるかどうかを確認する。
- URL リクエストを送信する。

これらはすべて、静的および動的解析のセクションで取り上げる、広い攻撃対象領域を示している。

参考資料

- owasp-mastg Testing Custom URL Schemes (MSTG-PLATFORM-3)

ルールブック

- ディープリンクの実装を目的とする場合は、ユニバーサルリンクの使用をする（推奨）
- カスタム URL スキームの使い方（必須）
- アプリが URL スキームの呼び出し処理では、URL とその引数を適切に検証し、重要な処理を実行する前にユーザに同意を求める（必須）
- ユニバーサルリンクの使い方（推奨）
- ディープリンクの URL とその引数を検証する（必須）

6.3.2 静的解析

静的解析でできることはいくつかある。次の章では、次のようなことを見していく。

- カスタム URL スキーム登録の検証
- アプリケーションクエリスキーマの登録検証
- URL の処理と検証の検証
- 他のアプリへの URL リクエストの検証
- 非推奨のメソッドに関する検証

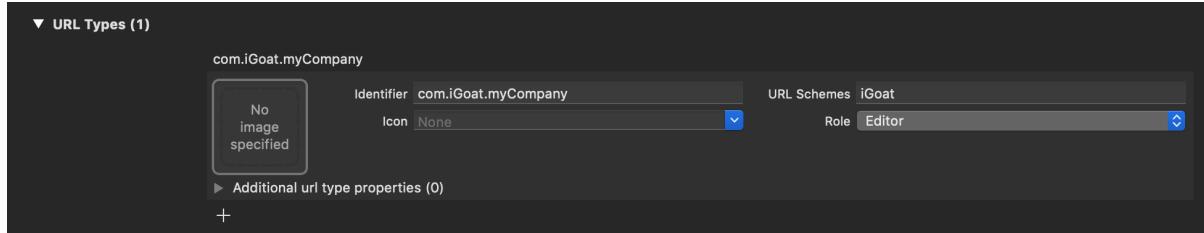
参考資料

- owasp-mastg Testing Custom URL Schemes (MSTG-PLATFORM-3) Static Analysis

6.3.2.1 カスタム URL スキーム登録

カスタム URL スキームを検証する最初の手順は、アプリケーションが何らかのプロトコルハンドラを登録しているかどうかを確認することである。

もし、元のソースコードを手元に持っていて、登録されているプロトコルハンドラを確認したい場合は、以下のスクリーンショットのように、Xcode でプロジェクトを開き、Info タブの URL Types 部分を開く。



また、Xcode では、アプリの Info.plist ファイル内の CFBundleURLTypes キーを検索することで見つけることができる。（iGoat-Swift の例）

```
<key>CFBundleURLTypes</key>
<array>
  <dict>
    <key>CFBundleURLName</key>
    <string>com.iGoat.myCompany</string>
    <key>CFBundleURLSchemes</key>
    <array>
      <string>iGoat</string>
    </array>
  </dict>
</array>
```

コンパイル済みのアプリケーション（または IPA）において、登録されたプロトコルハンドラは、アプリバンドル内のルートフォルダにある Info.plist ファイルにある。それを開いて、CFBundleURLSchemes キーを探す。もし存在すれば、文字列の配列が含まれているはずである。（iGoat-Swift の例）

```
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>
```

URL スキームが登録されると、他のアプリはスキームを登録したアプリを開き、適切な URL を作成し、UIApplication openURL:options:completionHandler: メソッドで開くことにより、パラメータを渡すことができる。

iOS 用アプリプログラミングガイドからの注記。

複数のサードパーティアプリが同じ URL スキームを扱うために登録した場合、どのアプリにそのスキームが与えられるかを決定するプロセスは、現在のところ存在しない。

これは、URL スキームハイジャック攻撃 ([#thiel2] の 136 ページを参照) につながる可能性がある。

参考資料

- owasp-mastg Testing Custom URL Schemes (MSTG-PLATFORM-3) Static Analysis Testing Custom URL Schemes Registration

ルールブック

- カスタム URL スキームの使い方（必須）
- ユニバーサルリンクの使い方（推奨）
- ディープリンクの URL とその引数を検証する（必須）

6.3.2.2 アプリケーションクエリスキーム登録

`openURL:options:completionHandler:` メソッドを呼び出す前に、アプリは `canOpenURL:` を呼び出して対象のアプリが利用可能かどうかを確認することができる。しかし、このメソッドはインストールされているアプリを列挙する手段として悪意のあるアプリに利用されていたため、iOS 9.0 からは、アプリの Info.plist ファイルに `LSApplicationQueriesSchemes` キーを追加して最大 50 個の URL スキームの配列を追加し、これに渡す URL スキームを宣言する必要がある。

```
<key>LSApplicationQueriesSchemes</key>
<array>
  <string>url_scheme1</string>
  <string>url_scheme2</string>
</array>
```

`canOpenURL` は、適切なアプリがインストールされているかどうかにかかわらず、未宣言のスキームに対しては常に NO を返す。ただし、この制限は `canOpenURL` にのみ適用される。

`openURL:options:completionHandler:` メソッドは、たとえ `LSApplicationQueriesSchemes` 配列が宣言されても、任意の URL スキームを開き、その結果に応じて YES / NO を返す。

例えば、Telegram は `Info.plist` で以下のような Queries Schemes を宣言している。

```
<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>
  ...

```

参考資料

- owasp-mastg Testing Custom URL Schemes (MSTG-PLATFORM-3) Static Analysis Testing Application Query Schemes Registration

6.3.2.3 URL の処理と検証

URL Path がどのように構築され、検証されているかを調べるには、元のソースコードがある場合、以下の方法で検索することができる。

- application:didFinishLaunchingWithOptions: メソッドまたは application:willFinishLaunchingWithOptions:: がどのように決定され、URL に関する情報がどのように取得されるかを検証する。
- application:openURL:options:: リソースがどのように開かれているか、すなわちどのようにデータが解析されているか、オプションを検証し、特に呼び出し側のアプリ（sourceApplication）によるアクセスが許可または拒否されるべきかどうかを検証する。また、カスタム URL スキームを使用する場合、アプリにはユーザ許可が必要な場合がある。

Telegram では、4 種類の方式が使われている。

```
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  
}
```

ここで、いくつか注意すべき点がある。

- このアプリは、application:handleOpenURL: や application:openURL:sourceApplication:annotation: といった非推奨のメソッドも実装している。
- これらのメソッドのいずれにおいても、ソースアプリケーションは検証されていない。
- これらのメソッドはすべて、非公開の openUrl メソッドを呼び出している。それを調べることで、URL リクエストがどのように処理されるかについて詳しく知ることができる。

参考資料

- owasp-mastg Testing Custom URL Schemes (MSTG-PLATFORM-3) Static Analysis Testing URL Handling and Validation

ルールブック

- カスタム URL スキームの使い方（必須）
- ユニバーサルリンクの使い方（推奨）
- ディープリンクの URL とその引数を検証する（必須）

6.3.2.4 他のアプリへの URL リクエスト

UIApplication の openURL:options:completionHandler: メソッドと非推奨の openURL: メソッドは、現在のアプリにローカルな、あるいは別のアプリが提供しなければならない URL を開く（つまり、他のアプリに要求を送る／問い合わせを行う）役割を担っている。元のソースコードがあれば、これらのメソッドの使用法を直接検索することができる。

さらに、アプリが特定のサービスやアプリを照会しているかどうか、また、アプリが有名かどうかを知りたい場合は、一般的な URL スキームをオンラインで検索し、それを grep に含めることもできる。例えば、Google で検索すると、以下のようなことがわかる。

```
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://
...
```

この方法を Telegram のソースコードから、今回は Xcode を使わずに egrep だけで検索してみる。

```
$ 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
```

結果を見ると、openURL:options:completionHandler: が実際にはユニバーサルリンクに使われていることがわかるため、検索を続ける必要がある。例えば、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=\1")
                                         )
./AppDelegate.swift:423:        UIApplication.shared.openURL(url)
./AppDelegate.swift:538:        UIApplication.shared.openURL(parsedUrl)
...
...
```

これらの内容を見ると、この方法は「設定」を開いたり、「App Store」を開いたりするのにも使われていることがわかる。

//だけを検索すると、次のようになる。

```
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\(appId)") {
        } else if let url = url as? String, url.lowercased().hasPrefix("tg://") {
            [[WKExtension sharedExtension] openSystemURL:[NSURL URLWithString:[NSString stringWithFormat:@"tel://%@", userHandle.data]]];
        }
    }
}
```

両方の検索結果を組み合わせて、ソースコードを慎重に調査した結果、次のようなコードが見つかった。

```
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)
```

URL を開く前にスキームを検証し、必要に応じて "https" を追加し、"tg" スキームの URL は開かない。準備ができたら、非推奨の openURL メソッドを使用する。

コンパイルされたアプリケーション（IPA）しか持っていない場合でも、他のアプリケーションへの問い合わせに使用されている URL スキームを特定することは可能である。

- LSApplicationQueriesSchemes が宣言されているかどうかを確認するか、一般的な URL スキームを検索する。

- また、アプリがスキームを宣言していない可能性があるため、文字列 `://` を使用するか、正規表現を構築して URL を一致させる。

そのためには、まずアプリのバイナリにその文字列が含まれているかどうかを unix の strings コマンドなどで確認する。

```
strings <yourapp> | grep "someURLscheme://"
```

あるいは、radare2 の iz/izz コマンドや rafind2 を使うと、unix の string コマンドでは見つけられないような文字列を見つけることができる。

iGoat-Swift の例：

```
$ r2 -qc izz~iGoat:// iGoat-Swift
37436 0x001ee610 0x001ee610 23 24 (4._TEXT.__cstring) ascii iGoat://?
↪contactNumber=
```

参考資料

- owasp-mastg Testing Custom URL Schemes (MSTG-PLATFORM-3) Static Analysis Testing URL Requests to Other Apps

6.3.2.5 非推奨メソッド

次のような Deprecated されたメソッドを検索する。

- application:handleOpenURL:
- openURL:
- application:openURL:sourceApplication:annotation:

例えば、ここではこの 3 つのメソッドを見つけることができる。

```
$ 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:
0x1000df772 35 34 openURL:options:completionHandler:
...
```

参考資料

- owasp-mastg Testing Custom URL Schemes (MSTG-PLATFORM-3) Static Analysis Testing for Deprecated Methods

6.3.3 動的解析

アプリが登録したカスタム URL スキームを特定したら、それを以下 の方法で検証する。

- URL リクエストの実行
- URL ハンドラメソッドの特定とフック化
- URL スキームのソース検証
- URL スキームのファジング

参考資料

- owasp-mastg Testing Custom URL Schemes (MSTG-PLATFORM-3) Dynamic Analysis

6.3.3.1 URL リクエストの実行

Safari の使用

1 つの URL スキームを素早く検証するには、Safari で URL を開き、アプリの動作を確認する。例えば、Safari のアドレスバーに tel:/123456789 と入力すると、電話番号と「発信」「キャンセル」のオプションがあるポップアップが表示される。発信を押すと、電話アプリが起動し、直接電話をかけることができる。

また、カスタム URL スキームをトリガーとするページについては、そのページへ移動するだけで、カスタム URL スキームを見つけたときに Safari が自動的に問い合わせるようになっている。

参考資料

- owasp-mastg Testing Custom URL Schemes (MSTG-PLATFORM-3) Dynamic Analysis Performing URL Requests Using Safari

メモアプリの使用

「ユニバーサルリンクのトリガー」で説明したように、メモアプリを使用し、作成したリンクを長押しして、カスタム URL スキームを検証することができる。リンクを開くには、編集モードを終了する必要がある。カスタム URL スキームを含むリンクをクリックまたは長押しできるのは、アプリがインストールされている場合のみで、インストールされていない場合はクリック可能なリンクとして認識されないため注意する。

参考資料

- owasp-mastg Testing Custom URL Schemes (MSTG-PLATFORM-3) Dynamic Analysis Performing URL Requests Using the Notes App

Frida の使用

単に URL スキームを開きたいだけなら、Frida を使用する。

```
$ frida -U iGoat-Swift
[iPhone:::iGoat-Swift] -> function openURL(url) {
    var UIApplication = ObjC.classes.UIApplication.
```

(次のページに続く)

(前のページからの続き)

```

↳sharedApplication();
    var toOpen = ObjC.classes.NSURL.URLWithString_(url);
    return UIApplication.openURL_(toOpen);
}
[iPhone::iGoat-Swift] -> openURL("tel://234234234")
true

```

Frida CodeShare のこの例では、著者は非公開 API の LSApplicationWorkspace.openSensitiveURL:withOptions: を使って（SpringBoard アプリから）URL を開いている。

```

function openURL(url) {
    var w = ObjC.classes.LSApplicationWorkspace.defaultWorkspace();
    var toOpen = ObjC.classes.NSURL.URLWithString_(url);
    return w.openSensitiveURL_withOptions_(toOpen, null);
}

```

なお、App Store では非公開の API の使用は許可されていない。そのため、これらの検証は行っていないが、動的解析のために使用することは許可されている。

参考資料

- owasp-mastg Testing Custom URL Schemes (MSTG-PLATFORM-3) Dynamic Analysis Performing URL Requests Using Frida

6.3.3.2 URL ハンドラメソッドの特定とフック

元のソースコードを見ることができない場合、アプリが受け取る URL スキーム要求を処理するためにどのメソッドを使用しているか自分で見つける必要がある。それが Objective-C のメソッドか Swift のメソッドか、あるいはアプリが非推奨のメソッドを使用しているかどうかは不明である。

参考資料

- owasp-mastg Testing Custom URL Schemes (MSTG-PLATFORM-3) Dynamic Analysis Performing URL Requests Identifying and Hooking the URL Handler Method

リンクを作成し Safari で開く

Frida CodeShare の ObjC メソッドオブザーバーを使用する。これは非常に便利なスクリプトで、単純なパターンを指定するだけでメソッドやクラスの集まりを簡単に確認することができる。

この場合、「openURL」を含むすべてのメソッドに注目するため、パターンは *[*openURL*] となる。

- 最初のアスタリスクは、すべてのインスタンス - およびクラス + メソッドに合致する。
- 2 番目のアスタリスクは、すべての Objective-C クラスに合致する。
- 3 番目と 4 番目は、文字列 openURL を含むすべてのメソッドに合致する。

```
$ frida -U iGoat-Swift --codeshare mrmacete/objc-method-observer
```

(次のページに続く)

(前のページからの続き)

```
[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:]
...
...
```

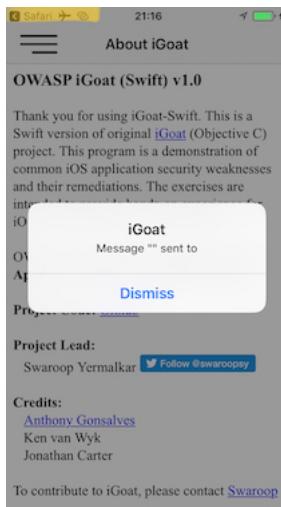
この一覧は非常に長く、すでに説明した方法も含まれる。今ある URL スキーム、例えば Safari から「igoat://」を起動して、アプリで開くことを許可すると、次のように表示される。

```
[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
```

既知事項

- メソッド -[iGoat_Swift.AppDelegate application:openURL:options:] が呼び出される。以前見たように、これは推奨される方法であり、非推奨ではない。
- このメソッドは、パラメータとして URL(igoat://) を受け取る。
- また、ソースアプリケーションである com.apple.mobilesafari を確認することができる。
- [UIApplication _applicationOpenURLAction:payload:origin:] から予想されるように、どこから呼び出されたかも確認することができる。
- このメソッドは 0x1 を返す。これは YES (デリゲートが要求を正常に処理したこと) を意味する。

呼び出しが成功し、iGoat のアプリが開かれたことが確認できる。



スクリーンショットの左上を見ると、呼び出し元（ソースアプリケーション）が Safari であることも確認できる。

参考資料

- owasp-mastg Testing Custom URL Schemes (MSTG-PLATFORM-3) Dynamic Analysis Performing URL Requests Crafting the Link Yourself and Letting Safari Open It

アプリ自体からリンクを動的に聞く

また、途中で他のどのメソッドが呼び出されるかを調べてみるのも興味深い。結果を少し変えるために、同じ URL スキームを iGoat アプリ自体から呼び出すことにする。ここでも ObjC のメソッドオブザーバと Frida REPL を使用する。

```
$ 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
...
```

(次のページに続く)

(前のページからの続き)

```

RET: nil

...
(0x101d0fad0) -[UIApplication openURL:]
openURL: iGoat://?contactNumber=123456789&message=hola
0x10a610044
...
RET: 0x1

true
(0x1c4038280) -[iGoat_SwiftAppDelegate 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

```

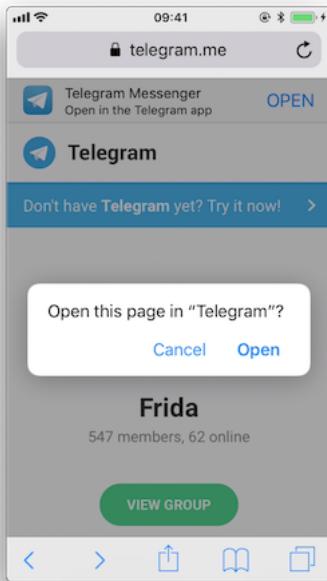
出力は読みやすくするために切り捨てている。今回は、UIApplicationOpenURLOptionsSourceApplicationKey が OWASP.iGoat-Swift に変更されていることがわかるが、これは理にかなっている。さらに、openURL のようなメソッドの長い一覧が呼び出されている。これらを考慮することは、次の手順、例えばどのメソッドをフックするか、あるいは改ざんするか、を決めるのに役立つため、ある状況においては非常に有用である。

参考資料

- owasp-mastg Testing Custom URL Schemes (MSTG-PLATFORM-3) Dynamic Analysis Performing URL Requests Dynamically Opening the Link from the App Itself

ページに移動し Safari でリンクを開く

ページに含まれるリンクをクリックしたときに、同じ状況を検証できるようになった。Safari は URL スキームを識別して処理し、どの動作を実行するかを選択する。このリンク「<https://telegram.me/fridadotre>」を開くと、以下の動作が発生する。



まず最初に、frida-trace にスタブを生成させる。

```
$ 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 |
↪$S10TelegramUI15openExternalUrl7account7context3url05forceD016presentationData
|
↪$S18applicationContext20navigationController12dismissInputy0A4Core7AccountC_
↪AA14Open
|
↪$SURLContext0SSbAA012PresentationK0CAA0a11ApplicationM0C7Display010NavigationO0CSgyyctF()
```

これで、目的のスタブを手動で修正するだけだよ。

- Objective-C のメソッド application:openURL:options:

```
// __handlers__/_AppDelegate_application_openUR_3679fad.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());
},
```

- Swift のメソッド \$S10TelegramUI15openExternalUrl...

```
// __handlers__/_TelegramUI/_S10TelegramUI15openExternalUrl17_b1a3234e.js

onEnter: function (log, args, state) {

    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());
},
}
```

次に実行すると、以下のような出力が得られる。

```
$ frida-trace -U Telegram -m "* [* restorationHandler]" -i "*open*Url*"
-m "*[* application*URL*]" -m "*[* openURL]*"

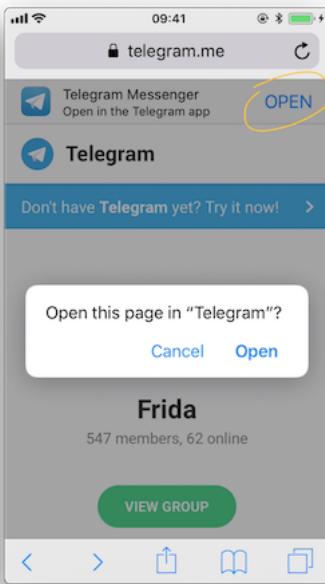
8144 ms -[UIApplication _applicationOpenURLAction: 0x1c44ff900 payload: ↵
0x10c5ee4c0 origin: 0x0]
8145 ms | -[AppDelegate application: 0x105a59980 openURL: 0x1c46ebb80 ↵
options: 0x1c0e222c0]
8145 ms | application: <Application: 0x105a59980>
8145 ms | openURL: tg://resolve?domain=fridakotre
8145 ms | 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=fridakotre
8269 ms | | presentationData: 0x1c4c51741
8269 ms | | applicationContext: nil
8269 ms | | navigationController: TelegramUI.PresentationData
8274 ms | -[UIApplication applicationOpenURL:0x1c46ebb80]
```

ここでは、次のようなことを確認することができる。

- 期待通り app delegate から application:openURL:options: を呼び出している。
- ソースアプリケーションは Safari ("com.apple.mobilesafari") である。
- application:openURL:options: は URL を処理するが、URL は開かない。
- 開かれる URL は、tg://resolve?domain=fridakotre である。
- これは、Telegram の tg:// カスタム URL スキームを使用している。

「<https://telegram.me/fridakotre>」に再度遷移し、キャンセルをクリックしてから、ページが提供するリンク

("Open in Telegram app") をクリックすると、カスタム URL スキーム経由で開くのではなく、ユニバーサルリンク経由で開くのは興味深い点である。



それぞれの方式をトレースしながら試してみる。

```
$ 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=fridakotre
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/fridakotre
406575 ms         activityType:NSUserActivityTypeBrowsingWeb
406575 ms         userInfo:{}
}
406575 ms     restorationHandler:<__NSStackBlock__: 0x16f27a898>
```

参考資料

- 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

非推奨メソッドの検証

非推奨のメソッドを検索する。

- application:handleOpenURL:
- openURL:
- application:openURL:sourceApplication:annotation:

これらのメソッドが使用されているかどうかを確認するために、frida-trace を使用することができる。

参考資料

- owasp-mastg Testing Custom URL Schemes (MSTG-PLATFORM-3) Dynamic Analysis Performing URL Requests Testing for Deprecated Methods

6.3.3.3 URL スキームのテストソース検証

バリデーションの破棄や確認の方法は、それに使用されそうな典型的なメソッドをフックすることで実現する可能性がある。(例 : 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)
  }
});
```

このフックを適用して、再度 URL スキームを呼び出す。

```
$ 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")
```

(次のページに続く)

(前のページからの続き)

```
true
nil
```

何も起こらない。フックとツイートのテキストの間に OWASP.iGoat-Swift や com.apple.mobilesafari などのアプリパッケージらしき文字列が見つからないことから、このメソッドはそのために使用されていないことが分かる。しかし、今回は 1 つの方法を調査しているだけで、アプリは他の方法で比較している可能性があることを考慮する必要がある。

参考資料

- [owasp-mastg Testing Custom URL Schemes \(MSTG-PLATFORM-3\) Dynamic Analysis Testing URL Schemes Source Validation](#)

6.3.3.4 URL スキームのファジング

アプリが URL の一部を解析する場合、入力ファジングを実行してメモリ破損のバグを検出することも可能である。

上記で学んだことを利用し、選択した言語で独自のファザーを構築できる。Python で、Frida の RPC を使用して openURL を呼び出す。そのファザーは次のことを行う必要がある。

- ペイロードを生成する。
- それぞれに対して openURL を呼び出す。
- アプリがクラッシュレポート (.ips) を /private/var/mobile/Library/Logs/CrashReporter に生成しているかどうかを確認する。

FuzzDB プロジェクトは、ペイロードとして使用できるファジング辞書を提供している。

参考資料

- [owasp-mastg Testing Custom URL Schemes \(MSTG-PLATFORM-3\) Dynamic Analysis Fuzzing URL Schemes](#)

Frida の使用

Frida でこれを行うのは非常に簡単である。iGoat-Swift アプリ（iOS 11.1.2 で動作）をファジングした例については、この[ブログ記事](#)を参照する。

ファザーを実行する前に、URL スキームの入力が必要である。静的解析から、iGoat-Swift アプリは次の URL スキームとパラメータを提供していることが分かっている：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
```

(次のページに続く)

(前のページからの続き)

スクリプトは、クラッシュが発生したかどうかを検出する。今回の実行でクラッシュは検出されなかったが、他のアプリではこのようなことが起こる可能性がある。クラッシュレポートは /private/var/mobile/Library/Logs/CrashReporter または /tmp (スクリプトによって移動された場合) で確認することができる。

参考資料

- owasp-mastg Testing Custom URL Schemes (MSTG-PLATFORM-3) Fuzzing URL Schemes Using Frida

6.3.4 ルールブック

1. ディープリンクの実装を目的とする場合は、ユニバーサルリンクの使用をする（推奨）
2. カスタム URL スキームの使い方（必須）
3. アプリが URL スキームの呼び出し処理では、URL とその引数を適切に検証し、重要な処理を実行する前にユーザに同意を求める（必須）
4. ユニバーサルリンクの使い方（推奨）
5. ディープリンクの URL とその引数を検証する（必須）

6.3.4.1 ディープリンクの実装を目的とする場合は、ユニバーサルリンクの使用をする（推奨）

ディープリンクの実装を目的とする場合は、ユニバーサルリンクの使用を推奨している。

apple-app-site-association 設定:

```
{
  "applinks": {
    "apps": [],
    "details": [
      {
        "appID": "9JA89QQLNQ.com.apple.wwdc",
        "paths": [ "/wwdc/news/", "/videos/wwdc/2015/*" ]
      },
      {
        "appID": "ABCD1234.com.apple.wwdc",
        "paths": [ "*" ]
      }
    ]
  }
}
```

Swift AppDelegate で Handling Universal Links 設定:

```
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, _)
```

(次のページに続く)

(前のページからの続き)

```

    ↪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
    }
}
}

```

これに注意しない場合、以下の可能性がある。

- URL スキームハイジャック攻撃や不正な URL にアクセスしてしまう可能性がある。

6.3.4.2 カスタム URL スキームの使い方（必須）

別のアプリがカスタムスキームを含む URL を開くと、システムは必要に応じてアプリを起動しフォアグラウンドに移動する。システムは、アプリのデリゲートメソッドを呼び出して、URL をアプリにデータを送信する。application メソッドにコードを追加して、URL のコンテンツを解析し、適切なアクションを実行する。

カスタム URL スキームの plist 記載：

Key	Type	Value
Information Property List	Dictionary	(3 items)
URL types	Array	(1 item)
Item 0 (Editor)	Dictionary	(3 items)
Document Role	String	Editor
URL identifier	String	jp.co.HelloWorld3
URL Schemes	Array	(1 item)
Item 0	String	cm-app
App Transport Security Settings	Dictionary	(2 items)
Application Scene Manifest	Dictionary	(2 items)

図 6.3.4.2.1 カスタム URL スキームの追加

plist の自動生成 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>

```

(次のページに続く)

(前のページからの続き)

```

<key>CFBundleURLTypes</key>
<array>
  <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>

```

カスタム URL スキーム経由でアプリの実行:

```

import UIKit

@main
class AppDelegate: UIResponder, UIApplicationDelegate {

  func application(_ application: UIApplication,

```

(次のページに続く)

(前のページからの続き)

```

        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,
          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
}
}

```

これに違反する場合、以下の可能性がある。

- 意図しないデータをアプリへ送信する可能性がある。

6.3.4.3 アプリが URL スキームの呼び出し処理では、URL とその引数を適切に検証し、重要な処理を実行する前にユーザに同意を求める（必須）

重要な処理を実行する前にユーザに同意を求める場合、セキュリティ上の問題が発生する。

一例として、2010 年に発見された Skype Mobile アプリの次のような不具合がある。Skype アプリは skype:// プロトコルハンドラを登録し、別のアプリが他の Skype ユーザの電話番号へ通話を発信する前にユーザに許可を求めなかったため、どのアプリもユーザが知らないうちに任意の番号に通話することができた。攻撃者はこの脆弱性を利用して、目に見えない <iframe src="skype://xxx?call"></iframe> (xxx はプレミアム番号に置き換えられる) を設置し、悪意のある Web サイトを閲覧した Skype ユーザがプレミアム番号に通話できるようにしている。

開発者としては、URL を呼び出す前に、慎重に検証する必要がある。登録されたプロトコルハンドラを介して起動される特定のアプリケーションのみを許可することができる。また、URL から呼び出された操作を確認するようユーザに促すことも有効な制御方法である。

引数の検証についてはディープリンクの URL とその引数を検証する（必須）を参照する。

すべての URL は、アプリの起動時、アプリの実行中、またはバックグラウンドで、アプリのデリゲートに渡される。受け取った URL を処理するために、デリゲートは以下のようなメソッドを実装する必要がある。

- URL の情報を取得し、開くかどうかを決定する。
- URL で指定されたリソースを開く。

詳細は、過去の App Programming Guide for iOS と Apple Secure Coding Guide に記載されている。

さらに、アプリは他のアプリに URL リクエスト（別名：クエリ）を送信することもできる。これは次のようにして行われる。

- アプリが要求するアプリケーションクエリスキームを登録する。
- 任意で他のアプリに照会し、特定の URL を開くことができるかどうかを確認する。
- URL リクエストを送信する。

これらはすべて、静的および動的解析のセクションで取り上げる、広い攻撃対象領域を示している。

参考資料

- owasp-mastg Testing Custom URL Schemes (MSTG-PLATFORM-3)

これに違反する場合、以下の可能性がある。

- ユーザの個人情報をユーザ自身気づかないうちに外部に送信する危険性がある。

6.3.4.4 ユニバーサルリンクの使い方（推奨）

ユーザがユニバーサルリンクをタップすると、システムは Safari や Web サイトを経由せずに、リンクをインストールしたアプリに直接リダイレクトできる。ユーザがアプリをインストールしていない場合、システムは Safari で URL を開き、Web サイトで処理できるようになる。この際、どのアプリにリダイレクトするかの指定を apple-app-site-accosiation ファイルで定義し、Xcode の Associated Domains でファイルの Domain を設定することで可能になる。

関連ファイルの指定 web サーバに配置 (AWS S3 などアクセス可能ならどこでもいい)

- apple-app-site-accosiation ファイルを web サーバに配置
 - appID は team 名 + bundleID

関連ファイルの json 記載方法:

```
{
  "webcredentials": {
    "apps": [ "${TeamID}.${BundleID}" ]
}
```

(次のページに続く)

(前のページからの続き)

```

},
"applinks": {
    "apps": [],
    "details": [
        "appID": "${TeamID}.${BundleID}",
        "paths": ["NOT /tests/*",
                  "NOT /settings/*",
                  "/*"]
    ]
}
}

```

iOS14 以降の apple-app-site-association 取得上の注意。iOS14 から取得経路が変わり、Apple の CDN を経由して apple-app-site-association が取得される。この変更による問題点は、IP 制限を行なっているサーバに apple-app-site-association ファイルを配置してしまうと、Apple の CDN が取得できず、UniversalLinks が機能しない。

Xcode での設定

- Xcode 上で Capabilities を設定。「Associated Domains」を追加し、Domains を設定
 - Domain は apple-app-site-association ファイルを配置したサイトのドメイン

Associated Domains 記載：



図 6.3.4.4.1 Associated Domains の追加

ユニバーサルリンク経由のソース:

```

import UIKit

@main
class AppDelegate: UIResponder, UIApplicationDelegate {

    func application(application: UIApplication, continueUserActivity_
    ↪userActivity: NSUserActivity, restorationHandler: ([AnyObject]?) -> Void) ->_
    ↪Bool {

        if userActivity.activityType == NSUserActivityTypeBrowsingWeb {
            // ユニバーサルリンクの処理
            print(userActivity.webpageURL!)
        }

        return true
    }
}

```

これに注意しない場合、以下の可能性がある。

- 意図しないアプリへリダイレクトする可能性がある。

6.3.4.5 ディープリンクの URL とその引数を検証する（必須）

ディープリンクで送信された query parameter に対して、本当にアプリが期待したものなのかの検証を行う。これは、アプリで用意した全てのパラメータに対して行うことが重要である。該当しない場合は ディープリンクの関数で false を返却する。

ディープリンクのパラメータ引数検証:

```
import UIKit

@main
class AppDelegate: UIResponder, UIApplicationDelegate {

    func application(application: UIApplication, continueUserActivity userActivity: NSUserActivity, restorationHandler: ([AnyObject]?) -> Void) -> Bool {

        if userActivity.activityType == NSUserActivityTypeBrowsingWeb {
            // ユニバーサルリンクの処理
            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.

        let sendingAppID = options[.sourceApplication]
        print("source application = \(sendingAppID ?? "Unknown")")
    }
}
```

(次のページに続く)

(前のページからの続き)

```

// 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
    }
}
}

```

これに違反する場合、以下の可能性がある。

- アプリが期待しない query parameter が送信される可能性がある。
- パラメーターをそのまま WebView に利用した際に不正な認証入力サイトなどを表示させる可能性がある。

6.4 MSTG-PLATFORM-4

アプリはメカニズムが適切に保護されていない限り、IPC 機構を通じて機密な機能をエクスポートしていない。

モバイルアプリケーションの実装では、開発者は IPC のための伝統的な技術(共有ファイルやネットワークソケットの使用など)を適用することができる。モバイルアプリケーションプラットフォームが提供する IPC システム機能は、従来の技術よりもはるかに成熟しているため、これを使用する必要がある。セキュリティを考慮せずに IPC メカニズムを使用すると、アプリケーションから機密データが漏洩したり、公開されたりする可能性がある。

iOS ではアプリ間の通信にかなり制限されたオプションが用意されている。実際、アプリが直接通信する方法は存在しない。この章では、iOS が提供するさまざまな種類の間接通信と、その検証方法について説明する。以下はその概要である。

- カスタム URL スキーム
- ユニバーサルリンク
- UIActivity の共有
- アプリ拡張
- UIPasteboard

参考資料

- owasp-mastg Testing for Sensitive Functionality Exposure Through IPC (MSTG-PLATFORM-4)

6.4.1 カスタム URL スキーム

カスタム URL スキームとは何か、どのように検証するかについては、「[カスタム URL スキームの検証](#)」の章を参照する。

参考資料

- owasp-mastg Testing for Sensitive Functionality Exposure Through IPC (MSTG-PLATFORM-4) Custom URL Schemes

ルールブック

- カスタム URL スキームの使い方（必須）

6.4.2 ユニバーサルリンク

6.4.2.1 概要

ユニバーサルリンクは、Android のアプリリンク（別名：デジタルアセットリンク）に相当する iOS で、ディープリンクのために使用される。ユニバーサルリンク（アプリのウェブサイトへのリンク）をタップすると、ユーザは Safari を経由することなく、対応するインストール済みのアプリに自動的にリダイレクトされる。アプリがインストールされていない場合、リンクは Safari で開かれる。

ユニバーサルリンクは、標準的な Web リンク（HTTP/HTTPS）であり、元々ディープリンクにも使用されていたカスタム URL スキームと混同されないようにする必要がある。

例えば、Telegram アプリは、カスタム URL スキームとユニバーサルリンクの両方をサポートしている。

- tg://resolve?domain=fridakotre はカスタム URL スキームで、tg:// スキームを使用する。
- https://telegram.me/fridakotre はユニバーサルリンクであり、https:// スキームを使用する。

どちらも同じ動作になり、ユーザは Telegram の指定されたチャット（この場合は "fridakotre"）にリダイレクトされる。しかし、ユニバーサルリンクには、カスタム URL スキームを使用した場合には適用できないいくつかの重要な利点があり、[Apple Developer Documentation](#)によれば、ディープリンクの実装方法として推奨されている。具体的には、ユニバーサルリンクは次のとおりである。

- **Unique:** カスタム URL スキームとは異なり、ユニバーサルリンクは、アプリの Web サイトへの標準的な HTTP または HTTPS リンクを使用するため、他のアプリによって要求されることはない。ユニバーサルリンクは、URL スキームハイジャック攻撃（元のアプリの後にインストールされたアプリが同じスキームを宣言し、システムが最後にインストールされたアプリへのすべての新しい要求を対象とする可能性がある）を防ぐ方法として導入された
- **Secure:** ユーザがアプリをインストールすると、iOS は Web サーバにアップロードされたファイル（Apple App Site Association または AASA）をダウンロードしてチェックし、Web サイトがアプリの代わりに URL を開くことを許可しているかどうかを確認する。URL の正当な所有者だけがこのファイルをアップロードできるため、そのウェブサイトとアプリの関連付けは安全に行われる。
- **Flexible:** ユニバーサルリンクは、アプリがインストールされていないときでも機能する。ウェブサイトへのリンクをタップすると、ユーザが期待するように、Safari でその内容が開かれる。
- **Simple:** 1 つの URL で Web サイトとアプリの両方に対応している。
- **Private:** 他のアプリは、アプリがインストールされているかどうかを確認することなく、通信することができる。

参考資料

- [owasp-mastg Testing for Sensitive Functionality Exposure Through IPC \(MSTG-PLATFORM-4\) Universal Links](#)

6.4.2.2 静的解析

静的なアプローチでのユニバーサルリンクのテストには、以下のようなものがある。

- 関連するドメインの entitlements の確認
- Apple App Site Association ファイルの取得
- リンク受信メソッドの確認
- データハンドラーメソッドの確認
- 他のアプリのユニバーサルリンクを呼び出しているかどうかの確認

参考資料

- [owasp-mastg Testing for Sensitive Functionality Exposure Through IPC \(MSTG-PLATFORM-4\) Universal Links Static Analysis](#)

関連するドメインの entitlements の確認

ユニバーサルリンクの場合、開発者は Associated Domains entitlements を追加し、その中にアプリがサポートするドメインのリストを含める必要がある。

Xcode で、**Capabilities** タブに移動し、**Associated Domains** を検索する。また、.entitlements ファイルで com.apple.developer.associated-domains を検索することもできる。各ドメインには、applinks:www.mywebsite.com のように、applinks: を先頭に付ける必要がある。

Telegram の .entitlements ファイルの例である。

```
<key>com.apple.developer.associated-domains</key>
<array>
  <string>applinks:telegram.me</string>
  <string>applinks:t.me</string>
</array>
```

より詳細な情報は、アーカイブされた Apple Developer Documentation に記載されている。

元のソースコードがない場合でも、「コンパイルされたアプリのバイナリに埋め込まれた entitlements」で説明するように、検索することができる。

参考資料

- owasp-mastg Testing for Sensitive Functionality Exposure Through IPC (MSTG-PLATFORM-4) Universal Links Static Analysis Checking the Associated Domains Entitlement

Apple App Site Association File の取得

前のステップで取得した関連ドメインを使用して、サーバから apple-app-site-association ファイルを取得することを試みる。このファイルは、リダイレクトなしの HTTPS 経由で <https://<domain>/apple-app-site-association> または <https://<domain>/.well-known/apple-app-site-association> にアクセスできることが前提である。

ブラウザを使って <https://<domain>/apple-app-site-association> や <https://<domain>/.well-known/apple-app-site-association> に移動するか、Apple の CDN を使って <https://app-site-association.cdn-apple.com/a/v1/<domain>> から自分で取得することが可能である。

または、[Apple App Site Association \(AASA \) Validator](#) を使用することもできる。ドメインを入力した後、ファイルを表示し、あなたに代わって検証し、結果を表示する。(例えば、HTTPS で適切に提供されていない場合など) [apple.com https://www.apple.com/.well-known/apple-app-site-association](https://www.apple.com/.well-known/apple-app-site-association) の次の例を参照する。

 **apple.com** -- This domain validates, JSON format is valid, and the Bundle and Apple App Prefixes match (if provided).
Below you'll find a list of tests that were run and a copy of your apple-app-site-association file:

Your domain is valid (valid DNS).

Your file is served over HTTPS.

Your server does not return error status codes greater than 400.

Your file's 'content-type' header was found :)

Your JSON is validated.

```
{
  "activitycontinuation": {
    "apps": [
      "W74U47NE8E.com.apple.store.Jolly"
    ]
  }
}
```

(次のページに続く)

(前のページからの続き)

```

        ],
    },
    "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",
                ]
            }
        ]
    }
}

```

applinks 内の "details" キーは、1つまたは複数のアプリを含むかもしれない配列の JSON 表現を含んでいる。 "appID" は、アプリの entitlements にある「application-identifier」キーと一致させる必要がある。次に、"paths" キーを使用して、開発者はアプリごとに処理される特定のパスを指定することができる。Telegram のような一部のアプリでは、すべての可能なパスを許可するために、単独の * ("paths": ["*"]) を使用する。ウェブサイトの特定の領域があるアプリで処理してはいけない場合に限り、開発者は対応するパスの前に「NOT」(T の後の空白に注意) を付けることで、それらを除外してアクセスを制限することができる。また、システムは配列内の辞書の順番に従って一致するものを探す。(最初に一致したものが優先される)

このパス除外メカニズムは、セキュリティ機能としてではなく、むしろ開発者がどのアプリがどのリンクを開くかを指定するために使用するフィルタと見なす。デフォルトでは、iOS は検証されていないリンクは開かない。

ユニバーサルリンクの検証は、インストール時に行われることに注意する。iOS は com.apple.developer.associated-domains entitlements で宣言されたドメイン (applinks) 用の AASA ファイルを取得する。iOS は、検証が成功しなかった場合、それらのリンクを開くことが拒否される。検証に失敗する理由としては、以下のようなものが考えられる。

- AASA ファイルは HTTPS で提供されない。
- AASA が利用できない。
- appID が一致しない。(これは悪意のあるアプリの場合) iOS は、想定されるハイジャック攻撃をうまく防ぐことができる。

参考資料

- owasp-mastg Testing for Sensitive Functionality Exposure Through IPC (MSTG-PLATFORM-4) Universal Links Static Analysis Retrieving the Apple App Site Association File

リンク受信メソッドの確認

リンクを受け取って適切に処理するためには、`application:continueUserActivity:restorationHandler:` を `app delegate` に実装する必要がある。もし、元のプロジェクトを持っていたら、このメソッドを検索してみる。

アプリのウェブサイトへのユニバーサルリンクを開くために `openURL:options:completionHandler:` を使用する場合、リンクはアプリで開かないことに注意する。呼び出し元がアプリであるため、ユニバーサルリンクとして処理されない。

Apple Docs より引用：ユーザがユニバーサルリンクをタップした後に iOS がアプリを起動すると、`activityType` 値が `NSUserActivityTypeBrowsingWeb` である `NSUserActivity` オブジェクトが受け取られる。アクティビティオブジェクトの `webpageURL` プロパティには、ユーザがアクセスしている URL が含まれている。`WebpageURL` プロパティには、常に HTTP または HTTPS URL が含まれ、`NSURLComponents` API を使用して URL のコンポーネントを操作することができる。ユーザのプライバシーとセキュリティを保護するために、データを転送する必要がある場合は HTTP を使用せず、HTTPS などの安全な転送プロトコルを使用する。

上記の注釈から、それを強調することができる。

- 上記のメソッドで見られるように、言及された `NSUserActivity` オブジェクトは `continueUserActivity` パラメータから来るものである。
- `webpageURL` のスキームは HTTP または HTTPS である必要がある。（その他のスキームの場合は例外が発生する）`URLComponents` / `NSURLComponents` の `scheme` インスタンスプロパティを使用して、これを確認することができる。

元のソースコードがない場合は、radare2 や rabin2 を使ってリンク受信メソッドのバイナリ文字列を検索できる。

```
$ rabin2 -zq Telegram\ X.app/Telegram\ X | grep restorationHan
0x1000deea9 53 52 application:continueUserActivity:restorationHandler:
```

参考資料

- owasp-mastg Testing for Sensitive Functionality Exposure Through IPC (MSTG-PLATFORM-4) Universal Links Static Analysis Checking the Link Receiver Method

データハンドラメソッドの確認

受信したデータがどのように検証されるかを確認する必要がある。Apple はこのことについて明確に警告している。

ユニバーサルリンクは、アプリへの潜在的な侵入経路となるため、すべての URL パラメータを検証し、不正な URL は破棄する必要がある。さらに、利用可能な機能をユーザのデータを危険にさらすことのないものに限定する。たとえば、コンテンツを直接削除したり、ユーザの機密情報にアクセスしたりするためのユニバーサルリンクを許可しないようにする。URL 処理コードを検証する際には、テストケースに不適切な形式の URL が含まれていることを確認する。

Apple Developer Documentation に記載されているように、iOS がユニバーサルリンクの結果としてアプリを開くと、アプリは `activityType` 値が `NSUserActivityTypeBrowsingWeb` である `NSUserActivity` オブジェクトを受

け取る。アクティビティオブジェクトの webpageURL プロパティには、ユーザがアクセスした HTTP または HTTPS URL が含まれる。Swift の次の例では、URL を開く前に厳重に検証している。

```
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
}
```

さらに、URL にパラメータが含まれている場合、慎重にサニタイズと検証を行うまでは、（信頼できるドメインからのものであっても）信頼しない方がよいことを覚えておく。たとえば、攻撃者によってなりすまされたり、不正なデータが含まれている可能性がある。このような場合、URL 全体、したがってユニバーサルリンク要求は破棄されなければならない。

NSURLComponents API を使用して、URL のコンポーネントを解析および操作することができる。これは application:continueUserActivity:restorationHandler: メソッドの一部であることもあれば、そこから呼び出される別のメソッドで発生することもある。次の例は、これを示している。

```
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
    }
}
```

最後に、上記のように、URL によって引き起こされる動作が、機密情報を公開したり、ユーザのデータを何ら

かの形で危険にさらすがないことを必ず確認すること。

参考資料

- owasp-mastg Testing for Sensitive Functionality Exposure Through IPC (MSTG-PLATFORM-4) Universal Links Static Analysis Checking the Data Handler Method

ルールブック

- ユニバーサルリンクの結果として受信したデータの検証方法（必須）
- ユーザのプライバシーとセキュリティを保護するために安全な転送プロトコルを使用する（必須）
- URL にパラメータが含まれている場合、慎重にサニタイズと検証を行うまでは URL を信頼しない（推奨）

アプリが他のアプリのユニバーサルリンクを呼び出しているかどうかを確認する

アプリがユニバーサルリンクを介して他のアプリを呼び出すのは、単に何らかの動作を引き起こすため、あるいは情報を転送するためかもしれないが、その場合、機密情報を漏洩していないことを確認する必要がある。

元のソースコードがあれば、openURL:options: completionHandler: メソッドを検索して、処理されるデータを確認することができる。

openURL:options:completionHandler: メソッドは、ユニバーサルリンクを開くためだけでなく、カスタム URL スキームを呼び出すためにも使用されることに注意すること。

Telegram アプリの例：

```
}, 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],
                completionHandler: { value in completion(value) })
        }
    }
}
```

アプリが URL を開く前にスキームを "https" に適応させ、オプション UIApplicationOpenURLOptionUniversalLinksOnly: true を使用して、URL が有効なユニバーサルリンクであり、その URL を開くことができるインストール済みのアプリがある場合にのみ URL を開いていることに注目する。

元のソースコードがない場合は、シンボルの中やアプリのバイナリの文字列の中で検索する。例えば、"openURL" を含む Objective-C のメソッドを検索する。

```
$ 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:
```

予想通り、openURL:options:completionHandler: が検出された。（アプリがカスタム URL スキームを開くため、これも存在する可能性があることを覚えておく）次に、機密情報を漏洩していないことを確認するために、動的解析を実行し、送信されるデータを検査する必要がある。このメソッドのフックとトレースに関するいくつかの例については、「カスタム URL スキーム」の「動的解析」章の「[URL ハンドラメソッドの特定とフック](#)」を参照する。

参考資料

- [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](#)

ルールブック

- **ユニバーサルリンクを介して他のアプリを呼び出す場合、機密情報を漏洩していないことを確認する（必須）**

6.4.2.3 動的解析

アプリにユニバーサルリンクが実装されている場合、静的解析から以下のような出力が得られるはずである。

- 関連するドメイン
- Apple App Site Association ファイル
- リンク受信メソッド
- データハンドラーメソッド

これで動的にテストすることが可能である。

- **ユニバーサルリンクの発動**
- **有効なユニバーサルリンクの特定**
- **リンクの受信方法の追跡**
- **リンクの開き方の確認**

参考資料

- [owasp-mastg Testing for Sensitive Functionality Exposure Through IPC \(MSTG-PLATFORM-4\) Universal Links Dynamic Analysis](#)

ユニバーサルリンクのトリガー

カスタム URL スキームとは異なり、残念ながら Safari からユニバーサルリンクを直接検索バーに入力してテストすることは Apple によって許可されていないためできない。しかし、メモアプリのような他のアプリを使っていつでも検証することができる。

- メモアプリを開き、新しいメモを作成する。
- ドメインを含むリンクを書き込む。
- メモアプリで編集モードを終了する。
- リンクを長押しして開く。（標準のクリックではデフォルトの選択肢が起動することに注意する）

Safari からこれを行うには、一度クリックするとユニバーサルリンクとして認識される Web サイト上の既存のリンクを見つける必要がある。これは少し時間がかかるかもしれない。

また、Frida の使用も可能である。詳しくは「[URL リクエストの実行](#)」の章を参照する。

参考資料

- owasp-mastg Testing for Sensitive Functionality Exposure Through IPC (MSTG-PLATFORM-4) Universal Links Dynamic Analysis Triggering Universal Links

有効なユニバーサルリンクの特定

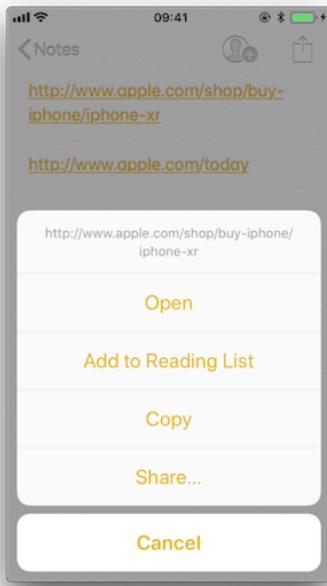
まず、許可されたユニバーサルリンクを開くか、許可すべきでないリンクを開くかの違いについて確認する。

上で見た apple.com の apple-app-site-association から、以下のパスを選択した。

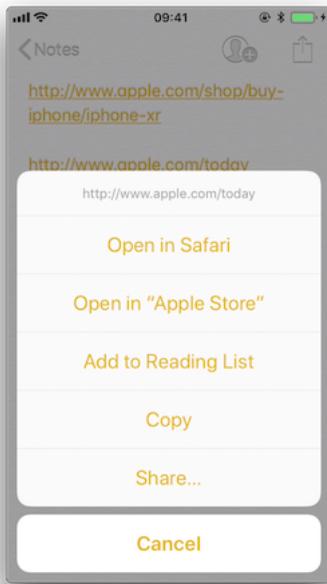
```
"paths": [
    "NOT /shop/buy-iphone/*",
    ...
    "/today",
```

一方は「アプリで開く」オプションを提供し、もう一方は提供しないようにする。

最初のもの (<http://www.apple.com/shop/buy-iphone/iphone-xr>) を長押しすると、（ブラウザで）開くというオプションだけが表示される。



2つ目（<http://www.apple.com/today>）を長押しすると、Safari で開くか、「Apple Store」で開くかのオプションが表示される。



クリックと長押しには違いがあることに注意する。一度リンクを長押しして「Safari で開く」などのオプションを選択すると、再度長押しして別のオプションを選択するまで、次回以降のクリックはすべてこのオプションが選択された状態になる。

`application:continueUserActivity: restorationHandler:` メソッドに対してフックまたはトレースで処理を繰り返すと、許可されたユニバーサルリンクを開くとすぐにそれが呼び出される様子が分かる。このために、例えば `frida-trace` を使うことができる。

```
frida-trace -U "Apple Store" -m "*[* *restorationHandler*]"
```

参考資料

- owasp-mastg Testing for Sensitive Functionality Exposure Through IPC (MSTG-PLATFORM-4) Universal Links Dynamic Analysis Identifying Valid Universal Links

リンク受信メソッドをトレースする

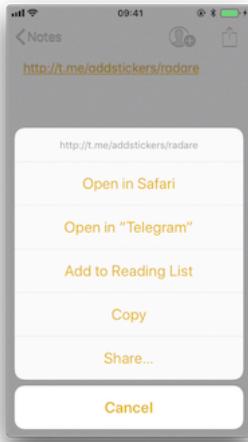
リンク受信メソッドをトレースし、追加情報を取得する方法について説明する。この例では、apple-app-site-association ファイルに制約がないため、Telegram を使用することにする。

```
{
  "applinks": {
    "apps": [],
    "details": [
      {
        "appID": "X834Q8SBVP.org.telegram.TelegramEnterprise",
        "paths": [
          "*"
        ]
      },
      {
        "appID": "C67CF9S4VU.ph.telegra.Telegraph",
        "paths": [
          "*"
        ]
      },
      {
        "appID": "X834Q8SBVP.org.telegram.Telegram-iOS",
        "paths": [
          "*"
        ]
      }
    ]
  }
}
```

また、リンクを開くために、以下の手順でメモアプリと frida-trace を使用する。

```
frida-trace -U Telegram -m "*[* *restorationHandler*]"
```

<https://t.me/addstickers/radare> (インターネットで調べて見つけた) を書き、メモアプリから開く。



まず、frida-trace に __handlers__ / にスタブを生成させる。

```
$ frida-trace -U Telegram -m "*[* *restorationHandler*]"
Instrumenting functions...
-[AppDelegate application:continueUserActivity:restorationHandler:]
```

1 つの関数だけが見つかり、それが計測されていることがわかる。ユニバーサルリンクを起動し、そのトレースを確認する。

```
298382 ms  -[AppDelegate application:0x10556b3c0 continueUserActivity:0x1c4237780
           restorationHandler:0x16f27a898]
```

実際に関数が呼び出されていることが確認できる。これで、__handlers__ / のスタブにコードを追加して、より詳細を取得することができる。

```
// __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\ webpageURL: " + ObjC.Object(args[3]).webpageURL().toString());
    log("\t\ tactivityType: " + ObjC.Object(args[3]).activityType().toString());
    log("\t\ tuserInfo: " + ObjC.Object(args[3]).userInfo().toString());
    log("\t\ trestorationHandler: " + ObjC.Object(args[4]).toString());
},
```

新しい出力

```
298382 ms  -[AppDelegate application:0x10556b3c0 continueUserActivity:0x1c4237780
           restorationHandler:0x16f27a898]
298382 ms  application:<Application: 0x10556b3c0>
298382 ms  continueUserActivity:<NSUserActivity: 0x1c4237780>
```

(次のページに続く)

(前のページからの続き)

```

298382 ms      webpageURL:http://t.me/addstickers/radare
298382 ms      activityType:NSUserActivityTypeBrowsingWeb
298382 ms      userInfo:{}
}
298382 ms      restorationHandler:<__NSStackBlock__: 0x16f27a898>

```

関数パラメータとは別に、より詳細な情報を得るために、関数パラメータからいくつかのメソッドを呼び出して、この場合は NSUserActivity について情報を追加している。[Apple Developer Documentation](#) を見ると、このオブジェクトから他に何を呼び出すことができるかがわかる。

参考資料

- owasp-mastg Testing for Sensitive Functionality Exposure Through IPC (MSTG-PLATFORM-4) Universal Links Dynamic Analysis Tracing the Link Receiver Method

リンクの開き方の確認

どの関数が実際に URL を開き、データが実際にどのように処理されているのかについて詳しく知りたい場合は、調査を続ける必要がある。

先ほどのコマンドを拡張して、URL を開くために必要な他の機能があるかどうかを調べる。

```
frida-trace -U Telegram -m "*[* *restorationHandler*]" -i "*open*Url*"
```

-i は任意のメソッドをインクルードする。ここでグローブパターンを使うこともできる。(例えば -i "*open*Url*" は「'open'、'Url'、その他の何かを含むすべての関数を含める」という意味である)

ここでも、まず frida-trace に __handlers__ / のスタブを生成させる。

```

$ frida-trace -U Telegram -m "*[* *restorationHandler*]" -i "*open*Url*"
Instrumenting functions...
-[AppDelegate application:continueUserActivity:restorationHandler:]
$S10TelegramUI0A19ApplicationBindingsC16openUniversalUrlyySS_
↪AA0ac4OpenG10Completion...

↪$S10TelegramUI15openExternalUrl7account7context3url05forceD016presentationData18application.
↪...
↪$S10TelegramUI31AuthorizationSequenceControllerC7account7strings7openUrl5apiId0J4HashAC0A4Core1
↪...
...

```

長い関数の一覧が表示されたが、どれが呼び出されるかはまだ不明である。ユニバーサルリンクをもう一度起動し、そのトレースを確認する。

```

/* TID 0x303 */
298382 ms  -[AppDelegate application:0x10556b3c0 continueUserActivity:0x1c4237780
            restorationHandler:0x16f27a898]
298619 ms  |
↪$S10TelegramUI15openExternalUrl7account7context3url05forceD016presentationData

```

(次のページに続く)

(前のページからの続き)

```

→18applicationContext20navigationController12dismissInputy0A4Core7AccountC_AA
→14OpenURLContextOSSSbAA012PresentationK0CAA0a11ApplicationM0C7Display0
    10NavigationO0CSgyyctF()

```

Objective-C の方法とは別に、今度は Swift の関数で注目すべきものが 1 つある。

その Swift 関数のドキュメントはおそらくないが、`xcrun` 経由で `swift-demangle` を使ってそのシンボルをデマングルする。

`xcrun` は、Xcode 開発者ツールをパスに入れずに、コマンドラインから呼び出すために使われる。この場合、`Swift-demangle`、Swift のシンボルを分離する Xcode ツールを探して実行する。

```

$ xcrun swift-demangle
→S10TelegramUI15openExternalUrl17account7context3url05forceD016presentationData
18applicationContext20navigationController12dismissInputy0A4Core7AccountC_
→AA14OpenURLContextOSSSbAA0
12PresentationK0CAA0a11ApplicationM0C7Display010NavigationO0CSgyyctF

```

結果

```

---> TelegramUI.openExternalUrl(
    account: TelegramCore.Account, context: TelegramUI.OpenURLContext, url: Swift.
    →String,
    forceExternal: Swift.Bool, presentationData: TelegramUI.PresentationData,
    applicationContext: TelegramUI.TelegramApplicationContext,
    navigationController: Display.NavigationController?, dismissInput: () -> () ->
    →()

```

これは、メソッドのクラス（またはモジュール）、名前、パラメータだけでなく、パラメータの型と戻り値の型も示しているため、より深く掘り下げる必要がある場合は、どこから始めればよいかわかる。

今回は、この情報を元にスタブファイルを編集して、パラメータを適切に出力する。

```

// __handlers__/_S10TelegramUI15openExternalUrl17_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]);
}

```

(次のページに続く)

(前のページからの続き)

```

log("\tapplicationContext: " + ObjC.Object(args[4]).toString());
log("\tnavigationController: " + ObjC.Object(args[5]).toString());
},

```

こうすることで、次に実行したときに、より詳細な出力が得られる。

```

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>

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

```

そこでは、次のようなことを確認することができる。

- 期待通り、application:continueUserActivity:restorationHandler: を App delegate から呼び出している。
- application:continueUserActivity:restorationHandler: は URL を処理するが、URL は開かないので、そのために TelegramUI.openExternalUrl が呼び出される。
- 開かれている URL は、<https://t.me/addstickers/radare> である。

続けて、データがどのように検証されているかを追跡し、検証することができる。例えば、ユニバーサルリンクで通信する 2 つのアプリがある場合、送信側のアプリが受信側のアプリでこれらのメソッドをフックすることによって機密データを漏洩していないかどうかを確認するために使用することができる。これは特にソースコードがない場合に便利で、ボタンをクリックしたり、何らかの機能を呼び出したりした結果である可能性があるため、他の方法では見ることができない完全な URL を取得することができる。

場合によっては、NSUserActivity オブジェクトの userInfo にデータが見つかる可能性がある。前の事例では、転送されるデータはなかったが、他の事例ではそうなる可能性がある。これを確認するには、userInfo プロパティをフックするか、フック内で continueUserActivity オブジェクトから直接アクセスする。(たとえば、次のような行を追加する log("userInfo:" + ObjC.Object(args[3]).userInfo().toString());)

参考資料

- owasp-mastg Testing for Sensitive Functionality Exposure Through IPC (MSTG-PLATFORM-4) Universal Links Dynamic Analysis Checking How the Links Are Opened

ユニバーサルリンクと Handoff に関する最終的な注意事項

ユニバーサルリンクと Apple の Handoff 機能は関連性がある。

- どちらもデータ受信時に同じ方式を採用している。

```
application:continueUserActivity:restorationHandler:
```

- ユニバーサルリンクと同様に、Handoff のアクティビティ継続は、com.apple.developer.associated-domains entitlements およびサーバの apple-app-site-association ファイルで（どちらの場合もキーワード「activitycontinuation」を介して）宣言する必要がある。例として、上記の "Apple App Site Association File の取得" を参照する。

前回の「リンクの開き方の確認」の例は、「Handoff Programming Guide」で説明した「Web Browser-to-Native App Handoff」の場合と非常によく似ている。

ユーザが発信端末でウェブブラウザを使用しており、受信端末が webpageURL プロパティのドメイン部分を主張するネイティブアプリを持つ iOS 端末の場合、iOS はネイティブアプリを起動し、activityType 値が NSUserActivityTypeBrowsingWeb である NSUserActivity オブジェクトを送信する。webpageURL プロパティにはユーザが閲覧していた URL が含まれ、userInfo ディクショナリは空になっている。

上記の詳細出力では、受け取った NSUserActivity オブジェクトが、まさに前述の条件を満たしていることが確認できる。

```
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>
```

この知識は、Handoff に対応したアプリを検証する際に役立つはずである。

参考資料

- owasp-mastg Testing for Sensitive Functionality Exposure Through IPC (MSTG-PLATFORM-4) Universal Links Dynamic Analysis Final Notes about Universal Links and Handoff

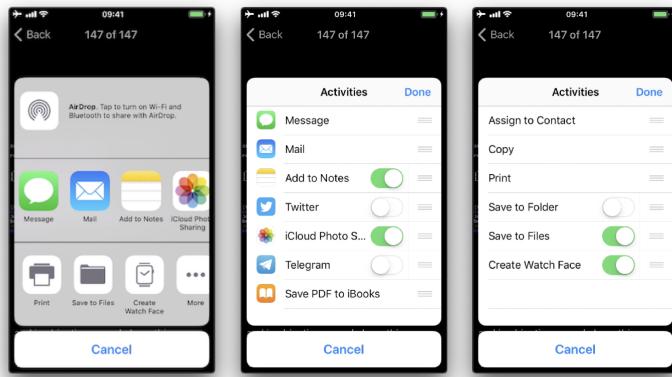
ルールブック

- ユニバーサルリンクと Handoff (必須)

6.4.3 UIActivity Sharing

6.4.3.1 概要

iOS6 以降、サードパーティアプリは、例えば AirDrop のような特定のメカニズムを介してデータ（アイテム）を共有することが可能である。ユーザから見ると、この機能は、「共有」ボタンをクリックした後に表示される、よく知られたシステム全体の「共有アクティビティシート」である。



組み込みの共有メカニズム（別名：アクティビティタイプ）には、以下のものがある。

- airDrop
- 連絡先への割り当て
- ペーストボードにコピー
- mail
- message
- Facebook に投稿
- Twitter に投稿

完全な一覧は、`UIActivity.ActivityType` に記載されている。アプリに適していないと思われる場合、開発者はこれらの共有メカニズムのいくつかを除外することができる。

参考資料

- owasp-mastg Testing for Sensitive Functionality Exposure Through IPC (MSTG-PLATFORM-4) UIActivity Sharing

6.4.3.2 静的解析

送信項目

UIActivity Sharing を検証する場合、特に注意しなければならないことがある。

- 共有されるデータ（項目）
- カスタムアクティビティ
- 除外されたアクティビティタイプ

UIActivity によるデータ共有は、UIActivityViewController を作成し、`init(activityItems: applicationActivities:)` で必要な項目（URL、テキスト、画像）を渡すことで機能する。

前に述べたように、コントローラの `excludedActivityTypes` プロパティによって、共有メカニズムの一部を除外することが可能である。除外できるアクティビティタイプの数が増える可能性があるため、iOS の最新バージョンを使用して検証を行うことを強く推奨する。開発者はこのことを認識し、アプリデータに適切でないものを明示的に除外する必要がある。アクティビティタイプの中には、「Create Watch Face」のようにドキュメント化されていないものもある可能性がある。

ソースコードを持っていたら、UIActivityViewController を参照する。

- `init(activityItems:applicationActivities:)` メソッドに渡されたアクティビティを確認する。
- カスタムアクティビティ（前のメソッドにも渡されている）が定義されているかどうかを確認する。
- `excludedActivityTypes` があれば、それを確認する。

コンパイル済み/インストール済みのアプリしかない場合は、例えば、前回のメソッドとプロパティを検索する。

```
$ rabin2 -zq Telegram\ X.app/Telegram\ X | grep -i activityItems
0x1000df034 45 44 initWithActivityItems:applicationActivities:
```

参考資料

- owasp-mastg Testing for Sensitive Functionality Exposure Through IPC (MSTG-PLATFORM-4) UIActivity Sharing Static Analysis Sending Items

ルールブック

- *UIActivity Sharing* が対象とするアクティビティタイプの明示的な除外（必須）

受信項目

受信の際には、確認が必要である。

- アプリがカスタムドキュメントタイプを宣言しているかどうかは、Exported/Imported UTI（Xcode プロジェクトの「Info」タブ）を調べることで確認できる。システムで宣言されたすべての UTI(Uniform Type Identifiers) のリストは、アーカイブされた Apple Developer Documentation で見ることができる。
- アプリが Document Types（Xcode プロジェクトの「Info」タブ）を調べることによって開くことができるドキュメントタイプを指定している場合。存在する場合、それらは名前とデータ型を表す 1 つ以上

の UTI (例: PNG ファイルの「public.png」) から構成される。iOS はこれを使用して、アプリが与えられたドキュメントを開く資格があるかどうかを判断する。（Exported/Imported UTI を指定するだけでは不十分である）

- App delegate の application:openURL:options:(またはその非推奨バージョン `UIApplicationDelegate` application:openURL:sourceApplication:annotation:) の実装を調べることによって、アプリケーションが受信データを適切に検証できるかどうかを確認する。

ソースコードがない場合でも、Info.plist ファイルを見て、次のように検索することができる。

- `UTEportedTypeDeclarations/UTImportedTypeDeclarations` は、アプリがエクスポート/インポートされたカスタムドキュメントタイプを宣言しているかどうかを確認するために使用する。
- アプリが開くことのできるドキュメントタイプを指定しているかどうかを確認するために、`CFBundleDocumentTypes` を使用する。

これらのキーの使用方法については、[Stackoverflow](#) に非常に詳しい説明が記載されている。

では、実際の例を見てみる。File Manager アプリを取り上げ、これらのキーについて見ていく。ここでは、Info.plist ファイルを読み込むために `objection` を使用した。

```
objection --gadget SomeFileManager run ios plist cat Info.plist
```

これは、電話から IPA を取得するか、例えば SSH 経由でアクセスし、IPA / アプリサンドボックス内の対応するフォルダに移動する場合と同じであることに注意する。しかし、Objection では、私たちは目的のコマンドを 1 つ実行するだけなので、これはまだ静的解析と考えることができる。

最初に注目したのは、アプリがインポートされたカスタムドキュメントタイプを宣言していないことである。しかし、エクスポートされたドキュメントタイプはいくつか見つけることができた。

```
UTEportedTypeDeclarations =      (
    {
    UTTtypeConformsTo =          (
        "public.data"
    );
    UTTtypeDescription = "SomeFileManager Files";
    UTTtypeIdentifier = "com.some.filemanager.custom";
    UTTtypeTagSpecification =      {
        "public.filename-extension" =      (
            ipa,
            deb,
            zip,
            rar,
            tar,
            gz,
            ...
            key,
            pem,
            p12,
            cer
        );
    };
}
```

(次のページに続く)

(前のページからの続き)

```
};  
}  
);
```

また、CFBundleDocumentTypes というキーから、アプリが開くドキュメントの種類を宣言している。

```
CFBundleDocumentTypes =  
{  
    ...  
    CFBundleTypeName = "SomeFileManager Files";  
    LSItemContentTypes =  
    {  
        "public.content",  
        "public.data",  
        "public.archive",  
        "public.item",  
        "public.database",  
        "public.calendar-event",  
        ...  
    };  
};
```

この File Manager は LSItemContentTypes に記載されている UTI のいずれかに適合するものを開こうとし、UTTypeTagSpecification/"public.filename-extension" に記載されている拡張子を持つファイルを開く準備ができたことがわかる。動的解析を行う際に、異なる種類のファイルを扱って脆弱性を検索したい場合に有効なので、参考にする。

参考資料

- owasp-mastg Testing for Sensitive Functionality Exposure Through IPC (MSTG-PLATFORM-4) UIActivity Sharing Static Analysis Receiving Items

ルールブック

- 他のアプリからファイルを受け取る場合は必ず確認する（必須）

6.4.3.3 動的解析

送信項目

動的計測を行うことで簡単に検証できることは、主に 3 つある。

- activityItems: 共有されるアイテムの配列。例えば、メッセージングアプリで共有する文字列と画像など、異なるタイプのものがある。
- applicationActivities: アプリのカスタムサービスを表す UIActivity オブジェクトの配列。
- excludedActivityTypes: サポートされていない Activity Types の配列。（例：postToFacebook）

これを実現するためには、2 つの方法がある。

- 静的解析で見たメソッド（`init(activityItems: applicationActivities:)`）をフックして、`activityItems` と `applicationActivities` を取得する。
- `excludedActivityTypes` プロパティをフックして、除外されたアクティビティを見つける。

Telegram を使って、画像とテキストファイルを共有する例を見てみる。まずフックを用意し、Frida REPL を使用して、このためのスクリプトを記述する。

```
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);
        }
    });

    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());
        }
    }
};
```

これを JavaScript ファイル（例：`inspect_send_activity_data.js`）として保存し、次のように読み込むことがで

きる。

```
frida -U Telegram -l inspect_send_activity_data.js
```

ここで、最初に写真を共有したときの出力を確認してみる。

```
[*] 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
```

上記を、テキストファイルで作成する。

```
[*] initWithActivityItems:applicationActivities: @ 0x18c130c07
initWithActivityItems: (
    "<QLActivityItemProvider: 0x1c4a30140>",
    "<UIPrintInfo: 0x1c0699a50>"
)
applicationActivities: (
)
RET @ 0x13c4bdc00:
<_UIDICActivityViewController: 0x13c4bdc00>

[*] excludedActivityTypes @ 0x18c0f8429
RET @ 0x1c001b1d0:
(
    "com.appleUIKit.activity.MarkupAsPDF"
)
```

以下のことがわかる。

- 画像については、アクティビティ項目は `UIImage` であり、除外されたアクティビティは存在しない。
- テキストファイルの場合、2つの異なるアクティビティ項目である `com.apple/UIKit.activity.MarkupAsPDF` と `MarkupAsPDF` は除外されている。

前の例では、カスタムの `applicationActivities` ではなく、除外されたアクティビティは1つだけである。しかし、他のアプリに期待できることをよりよく示すために、他のアプリを使用した画像を共有した。ここでは、アプリケーションアクティビティと除外されたアクティビティの集まりを見ることができる。（出力は、元のアプリの名前を隠すために編集されたものである）

```
[*] initWithActivityItems:applicationActivities: @ 0x18c130c07
initWithActivityItems: (
    "<SomeActivityItemProvider: 0x1c04bd580>"
```

(次のページに続く)

(前のページからの続き)

```
)
applicationActivities: (
    "<SomeActionItemActivityAdapter: 0x141de83b0>",
    "<SomeActionItemActivityAdapter: 0x147971cf0>",
    "<SomeOpenInSafariActivity: 0x1479f0030>",
    "<SomeOpenInChromeActivity: 0x1c0c8a500>"
)
RET @ 0x142138a00:
<SomeActivityViewController: 0x142138a00>

[*] excludedActivityTypes @ 0x18c0f8429
RET @ 0x14797c3e0:
(
    "com.appleUIKit.activity.Print",
    "com.apple/UIKit.activity.AssignToContact",
    "com.apple/UIKit.activity.SaveToCameraRoll",
    "com.apple/UIKit.activity.CopyToPasteboard",
)

```

受信項目

静的解析を行うと、アプリが開くことのできるドキュメントタイプと、カスタムドキュメントタイプを宣言しているかどうか、関連するメソッドの（一部）などが確認できる。これを利用して、受信項目を検証することができる。

- 他のアプリからアプリでファイルを共有するか、AirDrop や電子メールでファイルを送信する。「Open with...」ダイアログが表示されるようにファイルを選択する。（つまり、例えば PDF のようなファイルを開くデフォルトのアプリが存在しない）
- application:openURL:options: と、以前の静的解析で特定されたその他のメソッドをフックする。
- アプリの動作を確認する。
- さらに、特定の不正なファイルを送信したり、ファジング技術を使用することもできる。

これを例として説明するために、静的解析の章で紹介したのと同じ実際のファイルマネージャーアプリを選び、以下の手順を実施する。

- 他の Apple 製品（MacBook など）から **Airdrop** で PDF ファイルを送信する。
- AirDrop のポップアップが表示されるまで待ち、「Accept」をクリックする。
- ファイルを開くデフォルトのアプリがないため、「Open with...」ポップアップに切り替わる。そこで、ファイルを開くアプリを選択することができる。次のスクリーンショットはこれを示している。（アプリの実名を隠すために、Frida を使用して表示名を変更している）



4. SomeFileManager を選択すると、以下のようなになる。

```
(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
```

送信側のアプリケーションは com.apple.sharingd で、URL のスキームは file://。ファイルを開くアプリを選択すると、システムはすでにファイルを送信先、つまりアプリの受信トレイに移動していることに注

意する。アプリは受信箱の中のファイルを削除する役割を担っている。例えばこのアプリは、ファイルを /var/mobile/Documents/ に移動し、Inbox から削除している。

```
(0x1c002c760) -[XXFileManager moveItemAtPath:toPath:error:]
moveItemAtPath: /var/mobile/Library/Application Support/Containers
               /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
```

スタックトレースを見ると、application:openURL:options: が __handleOpenURL: を呼び出し、それが moveItemAtPath:toPath:error: を呼び出したことが分かる。ターゲットアプリのソースコードがなくても、この情報が得られることに注意する。最初にしなければならなかったことは、hook application:openURL:options: を削除することである。たとえば、「copy」、「move」、「remove」などの文字列を含むすべてのメソッドをトレースし、呼び出されたメソッドが moveItemAtPath:toPath:error: であることを確認する必要がある。

最後に、この受信ファイルの処理方法は、カスタム URL スキームでも同じであることに注目する必要がある。詳しくは、「[カスタム URL スキームの検証](#)」を参照する。

参考資料

- [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 概要

App Extensions とは

iOS 8 と同時に、Apple は App Extensions を導入した。[Apple App Extension Programming Guide](#) によると、アプリの拡張機能により、ユーザが他のアプリやシステムとやり取りしている間に、アプリがカスタム機能やコンテンツを提供できるようになる。例えば、ユーザが「共有」ボタンをクリックし、何らかのアプリや操作を選択した後に何が起こるかを定義したり、Today ウィジェットのコンテンツを提供したり、カスタムキーボードを有効にしたりといった、特定の適切な目的のための処理を実装する。

作業に応じて、アプリの拡張は特定のタイプ（1つだけ）、いわゆる拡張ポイントを持つことになる。注目すべきは、以下のようなものである。

- Custom Keyboard : iOS のシステムキーボードを、すべてのアプリケーションで使用できるカスタムキーボードに置き換える。
- Share : 共有サイトに投稿したり、他の人とコンテンツを共有したりする。

- Today : ウィジェットとも呼ばれ、通知センターの Today ビューでコンテンツを提供したり、クリックタスクを実行したりする。

参考資料

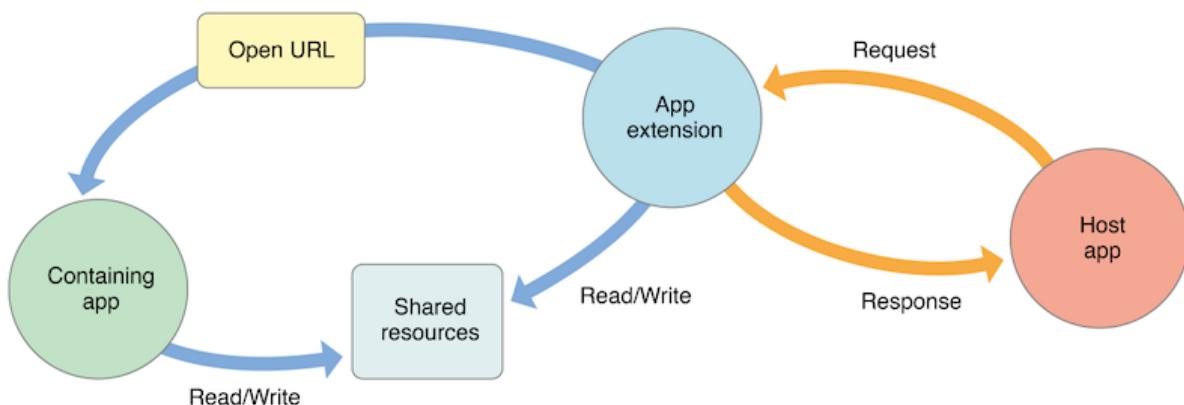
- owasp-mastg Testing for Sensitive Functionality Exposure Through IPC (MSTG-PLATFORM-4) App Extensions Overview What are app extensions

App Extensions は他のアプリとどのように相互作用するか

ここには 3 つの重要な要素がある。

- App extension : 標準アプリの中に含まれているもの。ホストアプリは、これと相互作用する。
- ホストアプリ : 他のアプリのアプリ拡張を実行する（サードパーティの）アプリである。
- 標準アプリ : 標準アプリに含まれている拡張機能を含むアプリである。

例えば、ユーザはホストアプリでテキストを選択し、「共有」ボタンをクリックし、一覧から 1 つの「アプリ」またはアクションを選択する。これにより、含まれているアプリの app extension が実行される。app extension は、ホストアプリのコンテキスト内でビューを表示し、ホストアプリが提供するアイテム（この場合は選択したテキスト）を使用して、特定のタスク（ソーシャルネットワークへの投稿など）を実行する。[Apple App Extension Programming Guide](#) に掲載されているこの図が、これをよく表している。



参考資料

- owasp-mastg Testing for Sensitive Functionality Exposure Through IPC (MSTG-PLATFORM-4) App Extensions Overview How do app extensions interact with other apps

セキュリティに関する配慮

セキュリティの観点から重要な点は、以下である。

- app extension は、そのアプリを含むアプリと直接通信することはない。（通常、app extension が実行されている間は、実行されない）
- app extension とホストアプリは、プロセス間通信を経由して通信する。
- app extension's の標準アプリとホストアプリは、全く通信しない。

- Today ウィジェット（および他のアプリ拡張タイプ）は、`NSExtensionContext` クラスの `openURL:completionHandler:` メソッドを呼び出して、システムにそのアプリを開くよう要求できる。
- 任意のアプリ拡張とそのアプリは、個人的に定義された共有コンテナ内の共有データにアクセスできる。

その他

- App Extensions は、HealthKit など、一部の API にアクセスすることができない。
- AirDrop を使用してデータを受信することはできないが、データを送信することは可能である。
- バックグラウンドでの長時間タスクは許可されないが、アップロードやダウンロードを開始することは可能である。
- App Extensions は、iOS デバイスのカメラやマイクにアクセスできない。（iMessage App Extensions を除く）

参考資料

- owasp-mastg Testing for Sensitive Functionality Exposure Through IPC (MSTG-PLATFORM-4) App Extensions Overview Security Considerations

6.4.4.2 静的解析

静的解析で対応する。

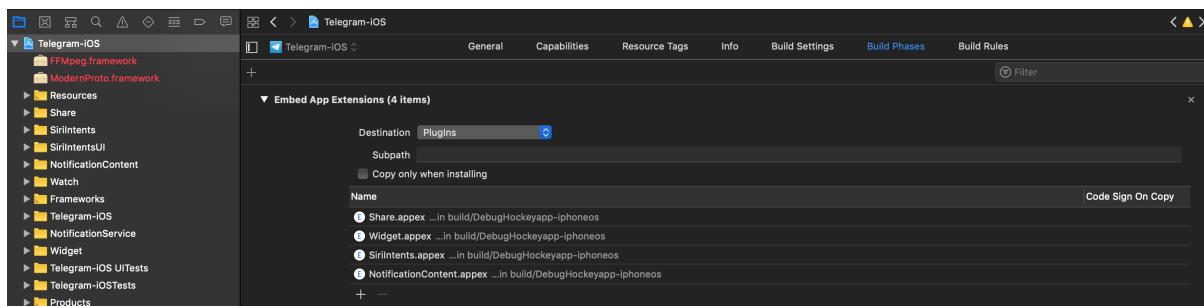
- アプリに App extensions が含まれているかどうかの確認
- 対応するデータ型の判定
- アプリとのデータ共有の確認
- アプリが拡張機能の利用を制限しているかどうかの確認

参考資料

- owasp-mastg Testing for Sensitive Functionality Exposure Through IPC (MSTG-PLATFORM-4) App extensions Static Analysis

アプリに App Extensions が含まれているかどうかの検証

もし元のソースコードがあれば、Xcode (cmd+shift+f) で `NSExtensionPointIdentifier` のすべての出現箇所を検索するか、「Build Phases / Embed App extensions」を確認してみる。



そこに、.appex の後に続く、すべての埋め込みアプリ拡張子の名前がある。これで、プロジェクト内の個々のアプリ拡張子に移動することができる。

元のソースコードを持っていない場合：

アプリバンドル（IPA またはインストール済みアプリ）内のすべてのファイルから NSExtensionPointIdentifier を Grep する。

```
$ 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
```

SSH でアクセスし、アプリバンドルを見つけて中の PlugIns を全てリスト表示する（デフォルトでそこに配置されている）か、Objection で行うこともできる。

```
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
```

Xcode で以前見たのと同じ 4 つの app extensions が表示されるようになった。

参考資料

- [owasp-mastg Testing for Sensitive Functionality Exposure Through IPC \(MSTG-PLATFORM-4\) App extensions Static Analysis Verifying if the App Contains App Extensions](#)

サポートするデータ型の決定

これは、ホストアプリと共有されるデータ（Share や Action Extensions など）にとって重要である。ユーザがホストアプリで何らかのデータ型を選択し、それがここで定義されたデータ型と一致する場合、ホストアプリは extension を提供する。UIActivity によるデータ共有では、UTI を使用してドキュメントタイプを定義する必要があったが、これとの違いに注目する必要がある。この場合、アプリが extension を持つ必要はない。UIActivity だけでデータ共有が可能である。

app extension's の Info.plist ファイルを確認し、NSExtensionActivationRule を検索する。このキーは、サポートされるデータ、およびサポートされる項目の最大値などを指定する。例を以下に示す。

```

<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>

```

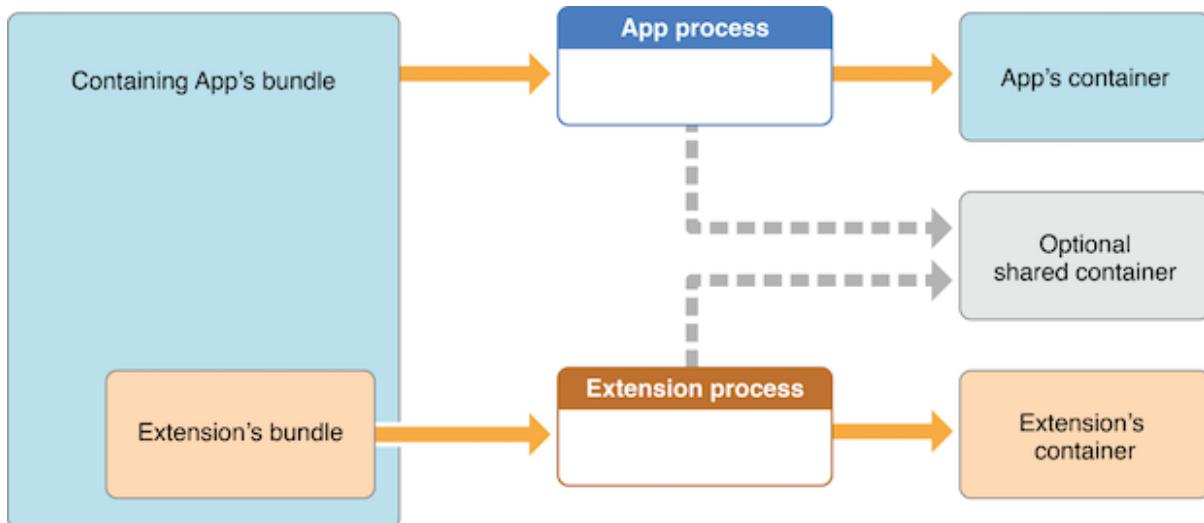
ここに存在し、MaxCount として 0 を持たないデータ型のみがサポートされる。しかし、与えられた UTI を評価するいわゆる predicate 文字列を使用することで、より複雑なフィルタリングを行うことが可能である。これに関するより詳細な情報については、[Apple App Extension Programming Guide](#) を参照する。

参考資料

- owasp-mastg Testing for Sensitive Functionality Exposure Through IPC (MSTG-PLATFORM-4) App extensions Static Analysis Determining the Supported Data Types

内包するアプリとのデータ共有の確認

app extensions とそれを含むアプリは、互いのコンテナに直接アクセスできないことに注意する。ただし、データの共有は可能である。これは、「[App Groups](#)」と NSUserDefaults API を介して行われる。[Apple App Extension Programming Guide](#) の図を参照する。



ガイドにも記載されているように、app extensions が NSURLSession クラスを使用してバックグラウンドでアップロードまたはダウンロードを実行する場合、アプリは共有コンテナを設定する必要があり、拡張機能とそれを含むアプリの両方が転送データにアクセスできるようにする。

参考資料

- 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 App Extensions の利用を制限しているかどうかの検証

以下の方法で、特定の種類の App Extensions を拒否することが可能である。

- application:shouldAllowExtensionPointIdentifier:

ただし、現状では「カスタムキーボード」app extensions でのみ可能である。（銀行アプリなど、キーボード経由で機密データを扱うアプリを検証する場合は、確認が必要）

参考資料

- 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 動的解析

動的解析の場合、ソースコードがなくても、以下のような方法で情報を取得することができる。

- 共有されるアイテムの確認
- 関係するアプリの拡張子の特定

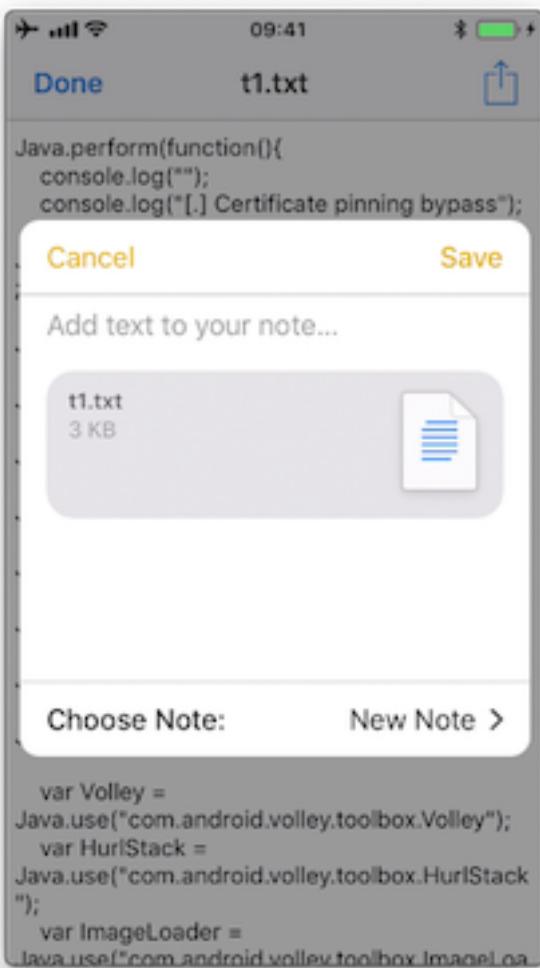
参考資料

- owasp-mastg Testing for Sensitive Functionality Exposure Through IPC (MSTG-PLATFORM-4) App extensions Dynamic Analysis

共有されるアイテムの確認

そのためには、データ発信元アプリの NSExtensionContext - inputItems をフックする必要がある。

前回の Telegram の例にならって、今度は（チャットで受け取った）テキストファイルの「共有」ボタンを使って、メモアプリにメモを作成する。



トレースを実行すると、次のような出力が得られる。

```
(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!-[NSXPConnection _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 )}"
    );
}"
```

ここでは、その様子を観察することができる。

- これは XPC を介して水面下で行われ、具体的には libxpc.dylib フレームワークを使用する NSXPConnection を介して実装されている。
- NSItemProvider に含まれる UTI は public.plain-text と public.file-url で、後者は Telegram の "Share Extension" の Info.plist から NSExtensionActivationRule に含まれている。

参考資料

- owasp-mastg Testing for Sensitive Functionality Exposure Through IPC (MSTG-PLATFORM-4) App extensions Dynamic Analysis Inspecting the Items Being Shared

関係する app extensions の特定

NSExtension - _plugIn をフックすることで、どの app extensions が要求と応答を引き受けているかを確認することができる。

もう一度同じ例を実行する。

```
(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>
```

見ての通り、2つの app extensions が関係している。

- Share.appex は、テキストファイル（public.plain-text と public.file-url）を送信している。
- com.apple.mobilenotes.SharingExtension.appex は、テキストファイルを受信して処理する。

XPC の内部で起こっていることをもっと知りたい場合は、「libxpc.dylib」からの内部呼び出しを参照することをお勧めする。例えば、frida-trace を使用し、自動生成されたスタブを拡張することによって、より詳細にメソッドを理解することができる。

参考資料

- 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 概要

入力フィールドにデータを入力する際、クリップボードを使用してデータをコピーすることができる。クリップボードはシステム全体からアクセスできるため、アプリによって共有される。この共有状態を悪意のあるアプリが悪用し、クリップボードに保存されている機密データを取得する可能性がある。

アプリを使用する際には、Facebook アプリのように、他のアプリがクリップボードを継続的に読み取っている可能性があることに注意する必要がある。iOS 9 以前は、悪意のあるアプリがバックグラウンドでペーストボードを監視し、定期的に [UIPasteboard generalPasteboard].string を取得する可能性があった。iOS 9 では、ペーストボードのコンテンツはフォアグラウンドのアプリからのみアクセスできるため、クリップボードからのパスワード漏洩のリスクは劇的に軽減される。それでも、パスワードのコピーペーストは、注意すべきセキュリティリスクであると同時に、アプリで解決することはできない。

- アプリの入力フィールドへの貼り付けを防止しても、ユーザが機密情報をコピーしてしまうことは防ぐことができない。ユーザが貼り付けられないことに気付く前に、すでに情報はコピーされているため、悪意のあるアプリはすぐにクリップボードを盗み見たことになる。
- パスワード欄への貼り付けが無効になっていると、ユーザは覚えていられる程度の弱いパスワードを選んでしまう可能性もあり、パスワードマネージャーを使えなくなってしまうため、アプリの安全性を高めるという本来の意図に反してしまうことになる。

UIPasteboard は、アプリ内でのデータ共有や、アプリから他のアプリへのデータ共有を可能にする。ペーストボードは 2 種類ある。

- **システム全体の一般的なペーストボード**：任意のアプリでデータを共有するためのものである。デバイスの再起動やアプリのアンインストールを経ても、デフォルトで持続する。（iOS 10 以降）
- **カスタム / 名前付きペーストボード**：別のアプリ（共有するアプリと同じチーム ID を持つ）またはアプリ自身とデータを共有するためのものである。（それらは作成するプロセスでのみ利用可能）デフォルトでは非永続的（iOS 10 以降）つまり、所有（作成）するアプリが終了するまでしか存在しない。

セキュリティに関するいくつかの注意点：

- ユーザは、アプリによるペーストボードの読み取りを許可または拒否することはできない。
- iOS 9 以降、アプリはバックグラウンドでペーストボードにアクセスできなくなったため、バックグラウンドでのペーストボードの監視が緩和された。しかし、悪意のあるアプリが再びフォアグラウンドになり、ペーストボードにデータが残っている場合、ユーザの知識や同意なしにプログラムによってデータを取得することができるようになる。
- Apple は、永続的な名前付きペーストボードについて警告し、その使用を推奨しない。代わりに、共有コンテナを使用する必要がある。
- iOS 10 からは、ユニバーサルクリップボードと呼ばれる新しい Handoff 機能があり、デフォルトで有効になっている。これは、一般的なペーストボードの内容をデバイス間で自動的に転送することを可能にする。この機能は開発者が選択すれば無効にすることができる、コピーしたデータの有効期限を設定することも可能である。

参考資料

- owasp-mastg Testing for Sensitive Functionality Exposure Through IPC (MSTG-PLATFORM-4) UIPasteboard

ルールブック

- 他のアプリがクリップボードを継続的に読み取っている可能性があることに注意する（必須）
- ペーストボードセキュリティに関する注意点（必須）

6.4.5.2 静的解析

システム全体の一般的なペーストボードは、generalPasteboard を使用して取得できる。このメソッドについては、ソースコードまたはコンパイル済みバイナリを検索する。システム全体の一般的なペーストボードの使用は、機密データを扱う場合には避けるべきである。

カスタムペーストボードは pasteboardWithName:create: または pasteboardWithUniqueName で作成することができる。iOS 10 からは非推奨なので、カスタムペーストボードを永続化するように設定しているかどうか確認する。代わりに、共有コンテナを使用する必要がある。

また、以下の調査も可能である。

- アプリのペーストボードを無効にする removePasteboardWithName: でペーストボードが削除されていないかチェックし、それによって使用されるすべてのリソースを解放する。（一般的なペーストボードには影響しない）
- 除外されたペーストボードがあるかどうか、UIPasteboardOptionLocalOnly オプションで setItems:options: を呼び出す必要があるかどうか、確認する必要がある。
- 期限切れのペーストボードがあるかどうか、UIPasteboardOptionExpirationDate オプションを持つ setItems:options: の呼び出しがあることを確認すること。
- アプリがバックグラウンドになるときや終了するときに、ペーストボードのデータを削除するかどうか確認する。これは、機密データの露出を制限しようとする一部のパスワード管理アプリによって行われる。

参考資料

- owasp-mastg Testing for Sensitive Functionality Exposure Through IPC (MSTG-PLATFORM-4) UIPasteboard Static Analysis

ルールブック

- ペーストボードセキュリティに関する注意点（必須）

6.4.5.3 動的解析

ペーストボードの使用状況を検知

以下をフックまたはトレースする。

- generalPasteboard は、システム全体の一般的なペーストボードのためのものである。
- pasteboardWithName:create と pasteboardWithUniqueName は、カスタムペーストボードのためのものである。

持続的なペーストボードの使用を検出

非推奨の `setPersistent:` メソッドをフックまたはトレースし、それが呼び出されているかどうかを検証する。

ペーストボードアイテムの監視と検証

ペーストボードを監視する際、いくつかの情報を動的に取得することができる。

- `pasteboardWithName:create:` をフックして入力パラメータを調べるか、`pasteboardWithUniqueName` をフックして戻り値を調べることで、`pasteboard` の名前を取得することができる。
- 最初に利用可能な `pasteboard` の項目を取得する: 例えば、文字列には `string` メソッドを使用する。あるいは、[標準的なデータ型](#)のための他のメソッドを使用する。
- `numberOfItems` でアイテムの数を取得する。
- `hasImages`, `hasStrings`, `hasURLs` (iOS 10 から) などの[便利なメソッド](#)で、標準的なデータ型の存在を確認することができる。
- `containsPasteboardTypes: inItemSet:` で他のデータ型（通常は UTI）を確認する。例えば、`public.png` や `public.tiff` のような画像 (UTI) や `com.mycompany.myapp.mytype` のようなカスタムデータなど、より具体的なデータ型に対して調査することができる。この場合、型に関する知識を宣言したアプリだけが、ペーストボードに書き込まれたデータを理解することができる。これは、「[UIActivity Sharing](#)」で見てきたことと同じである。`itemSetWithPasteboardTypes:` を使ってそれらを取得し、対応する UTI を設定する。
- `setItems:options:` をフックし、`UIPasteboardOptionLocalOnly` または `UIPasteboardOptionExpirationDate` のオプションを調査して、除外または期限切れの項目を確認する。

文字列を探すだけなら、[Objection](#) のコマンド `ios pasteboard monitor` を使用するとよい。

iOS の `UIPasteboard` クラスにフックし、5秒ごとに `generalPasteboard` をポーリングしてデータを取得する。もし新しいデータが見つかったら、前のポーリングと違って、そのデータはスクリーンにダンプされる。

また、上記のような特定の情報をモニターするペーストボードモニターを自作することもできる。

例えば、このスクリプト（[Objection](#) のペーストボードモニタの背後にあるスクリプトから影響を受けた）は、5秒ごとにペーストボードのアイテムを読み取り、何か新しいものがあれば、それを表示するものである。

```
const UIPasteboard = ObjC.classes.UIPasteboard;
const Pasteboard = UIPasteboard.generalPasteboard();
var items = "";
```

(次のページに続く)

(前のページからの続き)

```

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);

}, 1000 * 5);

```

出力では以下のようになる。

```

[* 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";
}
)

```

最初に "hola" という文字列を含むテキストがコピーされ、その後に URL がコピーされ、最後に画像がコピーされていることがわかる。これらのうちいくつかは、異なる UTI を介して利用可能である。他のアプリは、これらの UTI を考慮して、このデータの貼り付けを許可するかどうかを決定する。

参考資料

- owasp-mastg Testing for Sensitive Functionality Exposure Through IPC (MSTG-PLATFORM-4)
UIPasteboard Dynamic Analysis

6.4.6 ルールブック

1. ユニバーサルリンクの結果として受信したデータの検証方法（必須）
2. ユーザのプライバシーとセキュリティを保護するために安全な転送プロトコルを使用する（必須）
3. URL にパラメータが含まれている場合、慎重にサニタイズと検証を行うまでは URL を信頼しない（推奨）
4. ユニバーサルリンクを介して他のアプリを呼び出す場合、機密情報を漏洩していないことを確認する（必須）
5. *UIActivity Sharing* が対象とするアクティビティタイプの明示的な除外（必須）
6. 他のアプリからファイルを受け取る場合は必ず確認する（必須）
7. ユニバーサルリンクと *Handoff*（必須）
8. 他のアプリがクリップボードを継続的に読み取っている可能性があることに注意する（必須）
9. ペーストボードセキュリティに関する注意点（必須）

6.4.6.1 ユニバーサルリンクの結果として受信したデータの検証方法（必須）

ユニバーサルリンクは、アプリへの潜在的な侵入経路となるため、すべての URL パラメータを検証し、不正な URL は破棄する必要がある。

iOS がユニバーサルリンクの結果としてアプリを開くと、アプリは `activityType` 値が `NSUserActivityTypeBrowsingWeb` である `NSUserActivity` オブジェクトを受け取る。アクティビティオブジェクトの `webpageURL` プロパティには、ユーザがアクセスした HTTP または HTTPS URL が含まれる。Swift の次の例では、URL を開く前に厳重に検証している。

```
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
}
```

これに違反する場合、以下の可能性がある。

- URL スキームハイジャック攻撃に対して脆弱になる。

- ウェブサイトとアプリの関連付けが安全に行われない。

6.4.6.2 ユーザのプライバシーとセキュリティを保護するために安全な転送プロトコルを使用する（必須）

HTTP 通信では、Web サイトに SSL サーバ証明書が使われていないため、通信の内容が不正に取得・改ざんされるリスクがある。例えばオンラインサイトで名前や住所、クレジットカード番号などの個人情報を入力すると、悪意ある第三者に情報が漏れてしまう危険性がある。そのため HTTP は使用しないこと。

一方 HTTPS 通信では、SSL を使用して暗号化を行い Web サイトの安全性を高めている。App Transport Security (ATS) を使用すると HTTPS のみを通信するようになる。

ATS の有効化については、以下ルールブックのサンプルコードを参照。

ルールブック

- App Transport Security (ATS) を使用する（必須）*

これに違反する場合、以下の可能性がある。

- 中間者攻撃によって通信内容の盗聴や改ざんが行われる可能性がある。

6.4.6.3 URL にパラメータが含まれている場合、慎重にサニタイズと検証を行うまでは URL を信頼しない（推奨）

URL にパラメータが含まれている場合、URL が正しくてもパラメータが改ざんされている可能性があるため、慎重にサニタイズと検証を行うまでは URL を信頼しないことが推奨される。パラメータとはサーバに情報を送るために URL の末尾に付け足す変数のことである。

パラメータの基本構造

URL の後ろに「？」をつけキーと値を付与している。キーが複数ある場合は「&」で結合する。

例) `https://example.com?key=value&key=value&key=value`

そこで以下の検証を行い、安全な URL かどうかを判断する必要がある。

- 想定のパラメータ以外のキーが存在しないか
- パラメータ内の値が期待通りか（数値の上限下限以内、使用可能文字など）
 - 1 つのパラメータにつき、1 つの値が入っていること
 - 使用可能文字 (a-z A-Z 0-9 - _ . ! ' () *) 以外が使われていないこと

上記の検証に違反している場合は URL が想定通りであっても破棄する。

```
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, ✓
```

(次のページに続く)

(前のページからの続き)

```

    ↳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
    }
}
}

```

これに注意しない場合、以下の可能性がある。

- 意図した Web ページが表示されない。さらには攻撃者によってなりすまされ、意図せず不正アクセスしてしまう可能性がある。

6.4.6.4 ユニバーサルリンクを介して他のアプリを呼び出す場合、機密情報を漏洩していないことを確認する (必須)

アプリがユニバーサルリンクを介して他のアプリを呼び出すのは、単に何らかの動作を引き起こすため、あるいは情報を転送するためかもしれないが、その場合、機密情報を漏洩していないことを確認する必要がある。

元のソースコードがあれば、openURL:options:completionHandler: メソッドを検索して、処理されるデータを確認することができる。

openURL:options:completionHandler: メソッドは、ユニバーサルリンクを開くためだけでなく、カスタム URL スキームを呼び出すためにも使用されることに注意すること。

Telegram アプリの例：

```

}, 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,

```

(次のページに続く)

(前のページからの続き)

```
options: [UIApplicationOpenURLOptionUniversalLinksOnly:  
↪true as NSNumber],  
completionHandler: { value in completion(completion(value))  
})
```

アプリが URL を開く前にスキームを "https" に適応させ、オプション UIApplicationOpenURLOptionUniversalLinksOnly: true を使用して、URL が有効なユニバーサルリンクであり、その URL を開くことができるインストール済みのアプリがある場合にのみ URL を開いていることに注目する。

これに違反する場合、以下の可能性がある。

- 確認を怠った場合、機密情報を漏洩する可能性がある。

6.4.6.5 UIActivity Sharing が対象とするアクティビティタイプの明示的な除外（必須）

UIActivity でデータを共有することができるが、対象とするアクティビティタイプを明示的に除外する設定が可能。この指定を行うことで他のアプリへ不要なデータ共有を行わないようにすることができる。

ただし、除外できるアクティビティタイプが OS のバージョンアップにより増える可能性があるため、新規 OS では除外する項目を再度確認することを推奨する。

除外するコード:

```
import UIKit  
import Accounts  
  
class ShareViewController: UIViewController {  
  
    @IBAction func share(sender: AnyObject) {  
        // 共有する項目  
        let shareText = "Hello world"  
        let shareWebsite = NSURL(string: "https://www.apple.com/jp/watch/")!  
  
        let activityItems = [shareText, shareWebsite] as [Any]  
  
        // 初期化処理  
        let activityVC = UIActivityViewController(activityItems: activityItems,  
↪applicationActivities: nil)  
  
        // アクティビティタイプの除外設定  
        let excludedActivityTypes = [  
            UIActivity.ActivityType.postToFacebook,  
            UIActivity.ActivityType.postToTwitter,  
            UIActivity.ActivityType.saveToCameraRoll,  
            UIActivity.ActivityType.print  
        ]  
  
        activityVC.excludedActivityTypes = excludedActivityTypes
```

(次のページに続く)

(前のページからの続き)

```
// UIActivityViewController を表示
self.present(activityVC, animated: true, completion: nil)
}

override func viewDidLoad() {
    super.viewDidLoad()
}

}
```

リストで指定できるアクティビティタイプ

- UIActivityTypeAddToReadingList: URL を Safari の読み取りリストに追加
- UIActivityTypeAirDrop: コンテンツを AirDrop で利用
- UIActivityTypeAssignToContact: 画像を連絡先に割り当てる
- UIActivityTypeCollaborationCopyLink: ※ Developer に説明記載なし
- UIActivityTypeCollaborationInviteWithLink: ※ Developer に説明記載なし
- UIActivityTypeCopyToPasteboard: コンテンツをペーストボードに投稿する
- UIActivityTypeMail: コンテンツを新しい電子メール メッセージに投稿
- UIActivityTypeMarkupAsPDF: コンテンツを PDF ファイルとしてマークアップする
- UIActivityTypeMessage: コンテンツをメッセージ アプリに投稿
- UIActivityTypeOpenInIBooks: iBooks でコンテンツを開く
- UIActivityTypePostToFacebook: 提供されたコンテンツを Facebook 上のユーザのウォールに投稿する
- UIActivityTypePostToFlickr: 提供された画像をユーザの Flickr アカウントに投稿する
- UIActivityTypePostToTencentWeibo: 提供されたコンテンツをユーザの Tencent Weibo フィードに投稿する
- UIActivityTypePostToTwitter: 提供されたコンテンツをユーザの Twitter フィードに投稿する
- UIActivityTypePostToVimeo: 提供された動画をユーザの Vimeo アカウントに投稿する
- UIActivityTypePostToWeibo: 提供されたコンテンツをユーザの Weibo フィードに投稿する
- UIActivityTypePrint: 提供されたコンテンツを出力する
- UIActivityTypeSaveToCameraRoll: 画像またはビデオをユーザのカメラ ロールに割り当てる
- UIActivityTypeSharePlay: 提供されたコンテンツを SharePlay を通じて利用可能にする

これに違反する場合、以下の可能性がある。

- 想定外の共有アプリ側に機密情報などを送信する可能性がある。

6.4.6.6 他のアプリからファイルを受け取る場合は必ず確認する（必須）

受け取ったファイルのファイル名や中身の文字列データをそのまま DB のクエリなどにしていないことをソース上で確認する。

- ・アプリがカスタムドキュメントタイプを宣言しているかどうかは、Exported/Imported UTI（Xcode プロジェクトの「Info」タブ）を調べることで確認できる。システムで宣言されたすべての UTI(Uniform Type Identifiers) のリストは、[アーカイブされた Apple Developer Documentation](#) で見ることができる。
- ・アプリが Document Types（Xcode プロジェクトの「Info」タブ）を調べることによって開くことができるドキュメントタイプを指定している場合。存在する場合、それらは名前とデータ型を表す 1 つ以上の UTI（例：PNG ファイルの「public.png」）から構成される。iOS はこれを使用して、アプリが与えられたドキュメントを開く資格があるかどうかを判断する。（Exported/Imported UTI を指定するだけでは不十分である）
- ・App delegate の application:openURL:options: の実装を調べることによって、アプリケーションが受信データを適切に検証できるかどうかを確認する。

以下に sourceApplication で URL リクエストを送信したアプリの BundleId を判定するサンプルコードを示す。

```
import UIKit

@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {
{
    // 省略

    func application(_ app: UIApplication, open url: URL, options: [UIApplication.→OpenURLOptionsKey : Any] = [:]) -> Bool {
        guard let sourceApplication = options[.sourceApplication] as? String else {
            return false
        }

        if sourceApplication.hasPrefix("--他アプリの BundleId--") {
            // クエリパラメータを受けて色々処理
            // ...
        }

        return true
    }
}
```

ソースコードがない場合でも、Info.plist ファイルを見て、次のように検索することができる。

- ・UTEportedTypeDeclarations/UTImportedTypeDeclarations は、アプリがエクスポート/インポートされたカスタムドキュメントタイプを宣言しているかどうかを確認するために使用する。
- ・アプリが開くことのできるドキュメントタイプを指定しているかどうかを確認するために、CFBundleDocumentTypes を使用する。

これらのキーの使用方法については、[Stackoverflow](#) に非常に詳しい説明が記載されている。

では、実際の例を見てみる。File Manager アプリを取り上げ、これらのキーについて見ていく。ここでは、Info.plist ファイルを読み込むために `objection` を使用した。

```
objection --gadget SomeFileManager run ios plist cat Info.plist
```

これは、電話から IPA を取得するか、例えば SSH 経由でアクセスし、IPA / アプリサンドボックス内の対応するフォルダに移動する場合と同じであることに注意する。しかし、Objection では、私たちは目的のコマンドを 1 つ実行するだけなので、これはまだ静的解析と考えることができる。

最初に注目したのは、アプリがインポートされたカスタムドキュメントタイプを宣言していないことである。しかし、エクスポートされたドキュメントタイプはいくつか見つけることができた。

```
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
        );
    };
}
);
```

また、`CFBundleDocumentTypes` というキーから、アプリが開くドキュメントの種類を宣言している。

```
CFBundleDocumentTypes = (
{
    ...
    CFBundleTypeName = "SomeFileManager Files";
    LSItemContentTypes = (
        "public.content",
        "public.data",
        "public.archive",
        "public.item",
        "public.database",
        "public.calendar-event",
        ...
    );
}
```

(次のページに続く)

(前のページからの続き)

```
) ;
}

);
```

この File Manager は LSItemContentTypes に記載されている UTI のいずれかに適合するものを開こうとし、UTTypeTagSpecification/"public.filename-extension" に記載されている拡張子を持つファイルを開く準備ができたことがわかる。動的解析を行う際に、異なる種類のファイルを扱って脆弱性を検索したい場合に有効なので、参考にする。

これに違反する場合、以下の可能性がある。

- 文字列をそのまま使用してしまうと SQL インジェクションが起きる可能性がある。
- 文字列をそのまま WebView に利用した際に不正な認証入力サイトなどを表示させる可能性がある。

6.4.6.7 ユニバーサルリンクと Handoff (必須)

ユニバーサルリンクでアプリが起動された場合、AppDelegate の application:continueUserActivity:restorationHandler: メソッドが呼ばれるため、これを実装する。なお、このメソッドは Universal Links だけでなく、SearchAPI や Handoff によって起動された時も呼ばれるため activityType をチェックしておく必要がある。

以下のサンプルコードはユニバーサルリンクのチェック処理である。

```
import UIKit

extension AppDelegate {
    func application(application: UIApplication, continueUserActivity: NSUserActivity, restorationHandler: ([AnyObject]?) -> Void) -> Bool {
        if userActivity.activityType == NSUserActivityTypeBrowsingWeb {
            // アクティビティタイプを確認する
            // Universal Links の場合
            if UniversalLinkHandler.handleUniversalLink(userActivity.webpageURL) == false {
                UIApplication.sharedApplication().openURL(userActivity.webpageURL!)
                print(userActivity.activityType)
            }
        } else {
            print(userActivity.activityType)
        }
    }
    return true
}
```

以下のサンプルコードは Handoff のチェック処理である。

```
import UIKit
```

(次のページに続く)

(前のページからの続き)

```

extension AppDelegate {
    func handoffApplication(application: UIApplication, continueUserActivity_
    ↪userActivity: NSUserActivity, restorationHandler: ([AnyObject]?) -> Void) ->_
    ↪Bool {
        if userActivity.activityType == NSUserActivityTypeBrowsingWeb {
            // アクティビティタイプを確認する
            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
    }
}

```

これに違反する場合、以下の可能性がある。

- URL スキームハイジャック攻撃に対して脆弱になる。
- ウェブサイトとアプリの関連付けが安全に行われない。

6.4.6.8 他のアプリがクリップボードを継続的に読み取っている可能性があることに注意する（必須）

UIPasteboard を使用するとクリップボードにコピーまたはペーストができる。アプリ内でコピーした文字列を別アプリで取得することが可能。パスワードなどの機密の情報はコピーしないことが重要。

UIPasteboard でコピーする方法:

```
class UIPasteboardSample {

    func uiPasteboardSample() {
        // システム全体の一般的なペーストボード
        UIPasteboard.general.string = "コピー"

        // UIPasteboard.general.string に入っている文字列を取得（他アプリ共有）
        let parst = UIPasteboard.general.string

        // カスタム / 名前付きペーストボード
        guard let customPasteboard = UIPasteboard(name: UIPasteboard.
→Name(rawValue: "myData"), create: false) else {
            return
        }
        customPasteboard.string = "aaaa"
    }
}
```

これに違反する場合、以下の可能性がある。

- 共有状態を悪意のあるアプリが悪用し、クリップボードに保存されている機密データを取得する可能性がある。

6.4.6.9 ペーストボードセキュリティに関する注意点（必須）

Apple は、永続的な名前付きペーストボードについて警告し、その使用を推奨しない。

iOS14 から別のアプリからコピーしたテキストをクリップボードから取得すると、セキュリティの関係で画面上部にアラートが表示される。ユニバーサルクリップボードが導入された UIPasteboard では、セキュリティ上のリスクを下げるため、次の 2 つの機能が追加されている。

- 有効範囲をローカルのみに制限する
- 有効期限を設定する

これが利用できる条件は端末同士が近くにあり、それぞれが以下のように設定されている場合である。

- 各端末でそれぞれ同じ Apple ID を使って iCloud にサインインしている。
- 各端末で Bluetooth がオンになっている。
- 各端末で Wi-Fi がオンになっている。
- 各端末で Handoff がオンになっている。

有効範囲をローカルのみに制限する有効範囲をローカルのみに制限する例は、以下のようになる。

```
import UIKit

class Pasteboard: UIViewController {

    override func viewDidLoad() {
        super.viewDidLoad()

        let pasteboard = UIPasteboard.general
        pasteboard.setItems([["key" : "value"]], options: [UIPasteboard.OptionsKey.
↪localOnly : true])
    }
}
```

有効期限を設定する以下の例ではコピーしてから 24 時間経過したら、コピーデータが自動的に削除されるようになっている。

```
import UIKit

class Pasteboard: UIViewController {

    override func viewDidLoad() {
        super.viewDidLoad()

        let options: [UIPasteboard.OptionsKey : Any] = [.expirationDate: Date().
↪addingTimeInterval(60 * 60 * 24)]
        UIPasteboard.general.setItems([["key" : "value"]], options: options)
    }
}
```

これに違反する場合、以下の可能性がある。

- 共有状態を悪意のあるアプリが悪用し、クリップボードに保存されている機密データを取得する可能性がある。

6.5 MSTG-PLATFORM-5

明示的に必要でない限り WebView で JavaScript が無効化されている。

WebView は、インタラクティブな Web コンテンツを表示するためのアプリ内ブラウザコンポーネントである。アプリのユーザインターフェースに直接ウェブコンテンツを埋め込むために使用できる。iOS の WebView は、デフォルトで JavaScript の実行をサポートしているため、スクリプトインジェクションやクロスサイトスクリプティング攻撃の影響を受ける可能性がある。

参考資料

- owasp-mastg Testing iOS WebViews (MSTG-PLATFORM-5)

6.5.1 UIWebView

UIWebView は iOS 12 から非推奨となったため使用しないこと。Web コンテンツの埋め込みには、WKWebView か SFSafariViewController のどちらかを使用する必要がある。また、UIWebView は JavaScript を無効にすることができないため、これも使用を控えるべき理由のひとつである。

参考資料

- owasp-mastg Testing iOS WebViews (MSTG-PLATFORM-5) Overview UIWebView

ルールブック

- *UIWebView は iOS 12 から非推奨となったため使用しない (必須)*

6.5.2 WKWebView

WKWebView は iOS 8 で導入され、アプリの機能拡張、表示コンテンツの制御（ユーザが任意の URL に遷移することを防ぐなど）、カスタマイズに適した選択肢と言える。また、WKWebView は、Nitro JavaScript エンジン [#thiel2] を介して、WebView を使用しているアプリのパフォーマンスを大幅に向上させることができる。

WKWebView は、UIWebView と比較すると、セキュリティ面でいくつかの利点がある。

- JavaScript はデフォルトで有効だが、WKWebView の `javaScriptEnabled` プロパティの使用により、完全に無効化することが可能であり、全てのスクリプトインジェクションの欠陥を防止することができる。なお、この項目は現在非推奨。
- `JavaScriptCanOpenWindowsAutomatically` は、JavaScript がポップアップなどの新しいウィンドウを開くことを防ぐために使用することができる。
- `hasOnlySecureContent` プロパティは、WebView によって読み込まれたリソースが暗号化された接続を介して取得されることを確認するために使用できる。
- WKWebView はアウトオブプロセスレンダリングを実装しているため、メモリ破損のバグがメインアープリプロセスに影響することはない。

WKWebViews (および UIWebViews) を使用する場合、JavaScript Bridge を有効にすることが可能である。詳細については、以下の「[WebView での JavaScript とネイティブとの Bridge](#)」を参照する。

参考資料

- owasp-mastg Testing iOS WebViews (MSTG-PLATFORM-5) Overview WKWebView

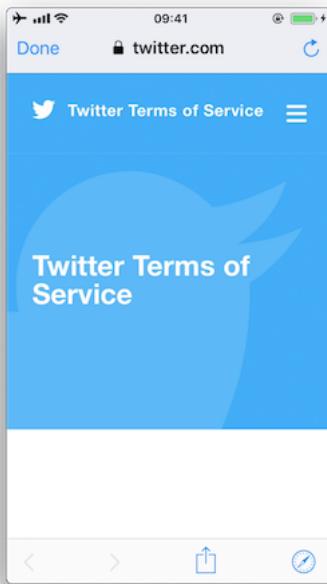
ルールブック

- *WKWebView の使用方法 (必須)*

6.5.3 SFSafariViewController

SFSafariViewController は iOS 9 から利用可能で、一般的なウェブ表示環境を提供するために使用する必要がある。これらの WebView は、以下の要素を含む特徴的なレイアウトを持つため、容易に見分けることができる。

- セキュリティインジケータ付きの読み取り専用アドレス入力欄。
- アクション（「共有」）ボタン。
- 完了ボタン、戻る・進むボタン、Safari で直接ページを開くための「Safari」ボタン。



いくつか注意点がある。

- SFSafariViewController では JavaScript を無効にすることができないため、アプリのユーザインターフェースの拡張を目的としている場合は、WKWebView の使用を推奨する。
- また、SFSafariViewController は Safari と Cookie やその他のウェブサイトデータを共有する。
- SFSafariViewController を使用するユーザのアクティビティとインタラクションは、アプリからは見えないため、AutoFill データ、閲覧履歴、ウェブサイトのデータにアクセスすることはできない。
- App Store 審査ガイドラインによると、SFSafariViewController は、他のビューやレイヤーによって隠されたり、見えなくされたりすることはない。

アプリの解析は以上で完結するため、SFSafariViewController は静的解析と動的解析の対象外である。

参考資料

- [owasp-mastg Testing iOS WebViews \(MSTG-PLATFORM-5\) Overview SFSafariViewController](#)

ルールブック

- *SFSafariViewController* の使用方法（必須）

6.5.4 Safari Web インスペクタ

iOS で Safari Web インスペクションを有効にすると、macOS 端末からリモートで WebView の内容を閲覧することができ、ジェイルブレイクされた iOS 端末は必要ない。Safari Web インスペクタを有効にすることは、ハイブリッドアプリケーションなど、JavaScript ブリッジを使用してネイティブ API を公開するアプリケーションで特に有効である。

Web インスペクションを有効にするには、以下の手順を実行する必要がある。

1. iOS 端末の場合、設定アプリを開く。**Safari** -> **詳細設定** を開き、「Web インスペクタ」をオンに切り替える。
2. macOS 端末の場合、Safari を開く。メニューバーで、**Safari** -> **環境設定** -> **詳細設定** と進み、「メニューバーに開発メニューを表示する」を有効にする。
3. iOS 端末を macOS 端末に接続し、ロックを解除する：開発メニューに iOS 端末名が表示される。
4. (まだ信頼されていない場合) macOS の Safari で、開発メニューを開き、iOS 端末名をクリック、「開発に使用」をクリックし、信頼を有効にする。

Web インスペクタを開いて、WebView をデバッグするには

1. iOS では、アプリを開き、WebView が含まれる画面に遷移する。
2. macOS の Safari では、**Developer** -> 「**iOS Device Name**」と進み、WebView ベースのコンテキストの名前が表示される。それをクリックすると、Web Inspector が表示される。

これで、デスクトップのブラウザで通常の Web ページと同じように、WebView をデバッグできるようになった。

参考資料

- [owasp-mastg Testing iOS WebViews \(MSTG-PLATFORM-5\) Overview Safari Web Inspector](#)

ルールブック

- *WKWebView* の使用方法（必須）
- *SFSafariViewController* の使用方法（必須）

6.5.5 静的解析

静的解析では、UIWebView と WKWebView を対象とし、主に以下の点に着目している。

- WebView の使用状況の確認
- JavaScript 設定の検証
- コンテンツが混在している場合の検証
- WebView の URI 操作の検証

参考資料

- owasp-mastg Testing iOS WebViews (MSTG-PLATFORM-5) Static Analysis

6.5.5.1 WebView の使用状況の確認

Xcode で検索して、上記の WebView クラスの使用例を探してみる。

コンパイルされたバイナリでは、次のようにシンボルや文字列を検索することができる。

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
```

また、これらの WebView クラスの既知のメソッドを検索することもできる。例えば、WKWebView を初期化するために使用されるメソッド (init(frame:configuration:)) を検索する。

```
$ rabin2 -zzq ./WheresMyBrowser | egrep "WKWebView.*frame"
0x5c3ac 77 76 __TOSo9WKWebViewCABSC6CGRectV5frame_
↪So0aB13ConfigurationC13configurationtcfC
0xd97a 79 78 __TOSo9WKWebViewCABSC6CGRectV5frame_
↪So0aB13ConfigurationC13configurationtcfctO
0xb5d5 77 76 __TOSo9WKWebViewCABSC6CGRectV5frame_
↪So0aB13ConfigurationC13configurationtcfC
0xc3fa 79 78 __TOSo9WKWebViewCABSC6CGRectV5frame_
↪So0aB13ConfigurationC13configurationtcfctO
```

また、デマングルも可能である。

```
$ xcrun swift-demangle __TSo9WKWebViewCABSC6CGRectV5frame_
↪So0aB13ConfigurationC13configurationtcfctO

---> @nonobjc __C.WKWebView.init(frame: __C_Synthesized.CGRect,
                                     configuration: __C.WKWebViewConfiguration) -> __C.
↪WKWebView
```

参考資料

- owasp-mastg Testing iOS WebViews (MSTG-PLATFORM-5) Static Analysis Identifying WebView Usage

ルールブック

- WKWebView の使用方法（必須）

6.5.5.2 JavaScript 設定の検証

まず、UIWebViews では、JavaScript を無効にできないことを覚えておく。

WKWebViews では、ベストプラクティスとして、JavaScript は明示的に必要とされない限り、無効化されるべきである。JavaScript が適切に無効化されていることを確認するために、プロジェクトで WKPreferences を使用し、`javaScriptEnabled` プロパティに `false` が設定されていることを確認する。なお、この項目は現在非推奨。

```
let webPreferences = WKPreferences()
webPreferences.javaScriptEnabled = false // なお、javaScriptEnabled は現在非推奨。
```

コンパイル済みのバイナリがあれば、その中で検索することができる。

```
$ rabin2 -zz ./WheresMyBrowser | grep -i "javasciptenabled" // なお、
↪javaScriptEnabled は現在非推奨。
391 0x0002f2c7 0x10002f2c7 17 18 (4.__TEXT.__objc_methname) ascii_
↪javaScriptEnabled
392 0x0002f2d9 0x10002f2d9 21 22 (4.__TEXT.__objc_methname) ascii_
↪setJavaScriptEnabled:
```

ユーザスクリプトが定義されている場合は、`javaScriptEnabled` プロパティが影響しないので、実行を継続する。なお、この項目は現在非推奨。WKWebViews にユーザスクリプトを組み込む方法については、`WKUserContentController` および `WKUserScript` を参照する。

参考資料

- owasp-mastg Testing iOS WebViews (MSTG-PLATFORM-5) Static Analysis Testing JavaScript Configuration

ルールブック

- WKWebView の使用方法（必須）
- JavaScript は明示的に必要とされない限り無効化する（推奨）

6.5.5.3 Mixed Content の場合の検証

UIWebViews とは対照的に、WKWebViews を使用する場合、mixed content (HTTPS ページから読み込まれた HTTP コンテンツ) を検出することが可能である。メソッド `hasOnlySecureContent` を使用することで、ページ上の全てのリソースが安全に暗号化された接続を介して読み込まれたかどうかを確認することができる。

コンパイル済みのバイナリでは

```
$ rabin2 -zz ./WheresMyBrowser | grep -i "hasonlysecurecontent"  
# nothing found
```

この場合、アプリはこれを利用しない。

元のソースコードまたは IPA があれば、埋め込まれた HTML ファイルを調査し、それらが mixed content を含んでいないことを確認することができる。ソースやタグの属性内に `http://` を検索する。ただし、例えばアンカータグ `<a>` の `href` 属性内に `http://` を含んでいても、必ずしも mixed content の問題があるとは限らないので、誤検出する可能性があることを覚えておく。

MDN Web Docs で混合コンテンツについて詳しく説明している。

参考資料

- [owasp-mastg Testing iOS WebViews \(MSTG-PLATFORM-5\) Static Analysis Testing for Mixed Content](#)

ルールブック

- [WKWebView の使用方法（必須）](#)

6.5.6 動的解析

動的解析では、静的解析と同じポイントを押さえる。

- `WebView` のインスタンスの列挙
- JavaScript が有効かどうかの検証
- 安全なコンテンツのみが許可されているかどうかの確認

動的な計測を行うことで、実行時に `WebView` を特定し、その全てのプロパティを取得することができる。これは、元のソースコードを持っていない場合に非常に有効である。

以下の例では、"Where's My Browser?" アプリと Frida REPL を引き続き使用する。

参考資料

- [owasp-mastg Testing iOS WebViews \(MSTG-PLATFORM-5\) Dynamic Analysis](#)

6.5.6.1 WebView のインスタンスの列挙

アプリ内の WebView を特定したら、ヒープを調査して、上記で見た WebView の 1 つまたは複数のインスタンスを見つけることができる。

例えば、Frida を使用する場合、"ObjC.choose()" を介してヒープを調査することで実現することができる。

```
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!');
    }
});
```

UIWebView と WKWebView の WebView については、念のため関連する URL も出力している。

ヒープ内の WebView のインスタンスを確実に見つけるために、まず見つけた WebView に移動する。そこで、Frida REPL にコピーして、上記のコードを実行する。

```
$ frida -U com.authenticationfailure.WheresMyBrowser

# copy the code and wait ...

onMatch: <UIWebView: 0x14fd25e50; frame = (0 126; 320 393);
           autosize = 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 =      (
```

(次のページに続く)

(前のページからの続き)

```
        "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, likeURL
        ↵Gecko) Mobile/...)"
    );
}
}
```

ここで、q を入力して終了し、別の WebView (この場合は WKWebView) を開く。前の手順を繰り返すと、これも検出される。

```
$ 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
```

次の章では、WebView からより多くの情報を取得するために、この例を拡張する。このコードを webviews_inspector.js などのファイルに保存して、次のように実行することを推奨する。

```
frida -U com.authenticationfailure.WheresMyBrowser -l webviews_inspector.js
```

參考資料

- owasp-mastg Testing iOS WebViews (MSTG-PLATFORM-5) Dynamic Analysis Enumerating WebView Instances

6.5.6.2 JavaScript が有効かどうかの検証

UIWebView が使用されている場合、JavaScript はデフォルトで有効になっており、無効にすることはできないことを覚えておく。

WKWebView の場合、JavaScript が有効になっているかどうかを確認する必要がある。これには、WKPreferences の `javaScriptEnabled` を使用する。なお、この項目は現在非推奨。

先ほどのスクリプトを以下の行で拡張する。

```
ObjC.choose(ObjC.classes['WKWebView'], {
  onMatch: function (wk) {
    console.log('onMatch: ', wk);
    console.log('javaScriptEnabled:', wk.configuration().preferences().javaScriptEnabled()); // なお、javaScriptEnabled は現在非推奨。
```

(次のページに続く)

(前のページからの続き)

```
//...
}
});
```

出力は、実際に JavaScript が有効であることを示している。

```
$ frida -U com.authenticationfailure.WheresMyBrowser -l webviews_inspector.js

onMatch: <WKWebView: 0x1508b1200; frame = (0 0; 320 393); layer = <CALayer:<
↪0x1c4238f20>>

javaScriptEnabled: true
```

参考資料

- owasp-mastg Testing iOS WebViews (MSTG-PLATFORM-5) Dynamic Analysis Checking if JavaScript is Enabled

ルールブック

- WKWebView の使用方法 (必須)

6.5.6.3 安全なコンテンツのみが許可されているかどうかの確認

UIWebView は、このためのメソッドを提供していない。しかし、各 UIWebView インスタンスの request メソッドを呼び出すことで、システムが "Upgrade-Insecure-Requests" CSP (Content Security Policy) ディレクティブを有効にしているかどうかを確認できる。("Upgrade-Insecure-Requests" は iOS 10 から有効になっている。このバージョンには iOS WebView を動かすブラウザエンジン、WebKit の新しいバージョンが含まれている。) 前のセクション「[WebView のインスタンスの列挙](#)」の例を参照する。

WKWebView の場合、ヒープで見つかった WKWebView のそれぞれに対して、hasOnlySecureContent メソッドを呼び出すことができる。WebView が読み込まれたら、この操作を忘れないこと。

先ほどのスクリプトを以下の行で拡張する。

```
ObjC.choose(ObjC.classes['WKWebView'], {
  onMatch: function (wk) {
    console.log('onMatch: ', wk);
    console.log('hasOnlySecureContent: ', wk.hasOnlySecureContent().toString());
    //...
  }
});
```

出力は、ページ上のリソースの一部が安全でない接続を介して読み込まれたことを示している。

```
$ frida -U com.authenticationfailure.WheresMyBrowser -l webviews_inspector.js

onMatch: <WKWebView: 0x1508b1200; frame = (0 0; 320 393); layer = <CALayer:<
↪0x1c4238f20>>
```

(次のページに続く)

(前のページからの続き)

```
hasOnlySecureContent: false
```

参考資料

- owasp-mastg Testing iOS WebViews (MSTG-PLATFORM-5) Dynamic Analysis Verifying that Only Secure Content is Allowed

ルールブック

- *WKWebView* の使用方法 (必須)
- *WKWebView* 読み込み後は、ロードされたリソースが暗号化された接続を通じて取得されたことを *hasOnlySecureContent* メソッドにより検証する (必須)

6.5.6.4 WebView の URI 操作の検証

WebView の URI をユーザが操作して、WebView の機能に必要なもの以外のリソースをロードできないようにすること。これは、WebView のコンテンツがローカルファイルシステムからロードされ、ユーザがアプリケーション内の他のリソースに移動できるようになる場合に特に危険である。

参考資料

- owasp-mastg Testing iOS WebViews (MSTG-PLATFORM-5) Dynamic Analysis Testing for WebView URI Manipulation

6.5.7 ルールブック

1. *UIWebView* は iOS 12 から非推奨となったため使用しない (必須)
2. *WKWebView* の使用方法 (必須)
3. *JavaScript* は明示的に必要とされない限り無効化する (推奨)
4. *WKWebView* 読み込み後は、ロードされたリソースが暗号化された接続を通じて取得されたことを *hasOnlySecureContent* メソッドにより検証する (必須)
5. *SFSafariViewController* の使用方法 (必須)

6.5.7.1 UIWebView は iOS 12 から非推奨となったため使用しない（必須）

iOS 12 から UIWebView は廃止されたため、WKWebView もしくは SFSafariViewController を利用する必要がある。

WKWebView と SFSafariViewController どちらを使用するかについては以下を参照する。

- *WKWebView の使用方法（必須）*
- *SFSafariViewController の使用方法（必須）*

※非推奨なルールのため、サンプルコードはなし。

これに違反する場合、以下の可能性がある。

- アプリ申請、App Store Connect へのアップロード、TestFlight も不可能になる。

6.5.7.2 WKWebView の使用方法（必須）

WKWebView は、WebKit フレームワークに含まれている。WKWebView を使用することで、ウェブコンテンツを App の UI にシームレスに統合が可能。既存の HTML、CSS、および JavaScript のコンテンツが機能するビューを読み込むことで、App 内で直接ウェブコンテンツの全体または一部分を表示できる。

SFSafariViewController では JavaScript を無効にすることができないため、アプリのユーザインターフェースの拡張を目的としている場合は、WKWebView の使用を推奨する。

WKWebView の実装:

```
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)
    }
}
```

これに違反する場合、以下の可能性がある。

- JavaScript の使用が有効な場合、スクリプトインジェクションに対して脆弱になる。

6.5.7.3 JavaScript は明示的に必要とされない限り無効化する（推奨）

WKWebView は UIWebView よりも JavaScript との連携が強化されている。WKWebView を使うことでアプリ側独自の JavaScript を既存 Web ページに読み込ませ、実行させるということが簡単にできるようになった。

`javaScriptEnabled` のデフォルト値は `true` である。このプロパティを `false` に設定すると、Web ページによってロードまたは実行される JavaScript が無効になる。この設定は、ユーザスクリプトには影響しない。なお、この項目は現在非推奨。

```
let webPreferences = WKPreferences()
webPreferences.javaScriptEnabled = false // なお、javaScriptEnabled は現在非推奨。
```

iOS 14 から `javaScriptEnabled` は廃止されたため、`allowsContentJavaScript` を使用する必要がある。このプロパティを `false` に設定すると、Web ページによってロードまたは実行される JavaScript が無効になる。

```
let webView = WKWebView()
webView.configuration.defaultWebpagePreferences.allowsContentJavaScript = false
```

これに注意しない場合、以下の可能性がある。

- Web ページによってロードまたは実行される JavaScript が有効になる。

6.5.7.4 WKWebView 読み込み後は、ロードされたリソースが暗号化された接続を通じて取得されたことを `hasOnlySecureContent` メソッドにより検証する（必須）

`hasOnlySecureContent` は WebView が、ページ上のすべてのリソースが安全に暗号化された接続を介してロードされているかどうかを示す布尔値である。

WebView が読み込まれたら、以下の操作を実行し、暗号化された接続を通じて取得されたことを確認する。

```
ObjC.choose(ObjC.classes['WKWebView'], {
  onMatch: function (wk) {
    console.log('onMatch: ', wk);
    console.log('hasOnlySecureContent: ', wk.hasOnlySecureContent().toString());
    //...
  }
});
```

これに違反する場合、以下の可能性がある。

- WebView によりロードされたリソースが安全でない接続を通じて取得された場合、それを確認できない。

6.5.7.5 SFSafariViewController の使用方法（必須）

SFSafariViewController は、SafariServices フレームワークの一部分。この API を使用することで、ユーザは App 内でウェブページまたはウェブサイトを閲覧することができる。また、パスワードの自動入力、リーダー、セキュアブラウジングなどの機能を含む Safari と同様の動きを実現できる。

SFSafariViewController の実装:

```
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)
    }
}
```

これに注意しない場合、以下の可能性がある。

- JavaScript の使用が無効に出来ないため、スクリプトインジェクションに対して脆弱になる。

※以下の場合、WKWebView でしか実装できないことに注意する。

- ページが読み込まれたタイミングで何かを実行したい。
- 読み込んだページや読み込もうとしたページの URL を知りたい。
- 表示中のページに対して JavaScript を実行したい。

6.6 MSTG-PLATFORM-6

WebView は最低限必要なプロトコルハンドラのセットのみを許可するよう構成されている（理想的には、https のみがサポートされている）。file, tel, app-id などの潜在的に危険なハンドラは無効化されている。

6.6.1 WebView のロード

iOS の WebView などで利用されている、デフォルトのスキームがいくつか用意されている。

- http(s)://
- file://
- tel://

WebView はエンドポイントからリモートコンテンツを読み込むことができるが、アプリデータディレクトリからローカルコンテンツを読み込むこともできる。ローカルコンテンツが読み込まれる場合、ユーザはファイル名やファイルの読み込みに使用されるパスに影響を与えることができないようにし、また読み込まれたファイルを編集することができないようにする必要がある。

多層防御の手段として、次のベストプラクティスを使用する。

- ローカルおよびリモートのウェブページと、読み込みを許可する URL スキームを定義したリストを作成する。
- ローカルの HTML/JavaScript ファイルのチェックサムを作成し、アプリの起動中にチェックする。
JavaScript ファイルを "Minification (programming)" (最小化) して、読みにくくする。

参考資料

- owasp-mastg Testing WebView Protocol Handlers (MSTG-PLATFORM-6) Overview

ルールブック

- *WebView* でローカルコンテンツを読み込む方法 (必須)

6.6.2 静的解析

- WebView のロード方法
- WebView ファイルアクセス
- 電話番号の検出を確認

6.6.2.1 WebView のロード方法

WebView がアプリのデータディレクトリからコンテンツをロードする場合、ユーザはファイル名やロード元のパスを変更できないようにし、ロードされたファイルを編集できないようにする必要がある。

これは特に、非推奨のメソッドである `loadHTMLString:baseURL:` や `loadData:MIMEType:textEncodingName:baseURL:` によって信頼できないコンテンツをロードする UIWebViews において問題となり、`baseURL` パラメータに `nil` または `file:` や `applewebdata.URL` または URL スキームを設定することができる。この場合、ローカルファイルへの不正アクセスを防止するために、代わりに `about:blank` に設定するのが最適な方法である。ただし、UIWebViews の使用は避け、代わりに WKWebViews を使用することを推奨する。

「Where's My Browser?」に掲載されている脆弱な UIWebView の例である。

```
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 {}
```

このページは、HTTP を使用してインターネットからリソースをロードし、潜在的な MITM が共有設定などのローカルファイルに含まれる機密情報を流出させることを可能にする。

WKWebViews を使用する場合、Apple はローカルの HTML ファイルをロードするために loadHTMLString:baseURL: または loadData:MIMEType:textEncodingName:baseURL: を、ウェブコンテンツのために loadRequest: を使うことを推奨している。通常、ローカルファイルの読み込みは、特に pathForResource ofType: 、 URLForResource:withExtension: 、 init(contentsOf:encoding:) といったメソッドと組み合わせて行われることが多い。

記載されているメソッドのソースコードを検索し、そのパラメータを調査する。

Objective-C の例

```
- (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 = 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];
}
```

Swift の例「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 {}
```

コンパイル済みのバイナリしかない場合は、これらのメソッドを検索することも可能である。例を以下に示す。

```
$ rabin2 -zz ./WheresMyBrowser | grep -i "loadHTMLString"
231 0x0002df6c 24 (4.__TEXT.__objc_methname) ascii loadHTMLString baseURL:
```

このような場合、実際にどのような種類の WebView から使用されているかを確認するために、動的解析を実行することを推奨する。ここで baseURL パラメータは "null" が設定されるため問題ありませんが、UIWebView を使用する際、適切に設定されないと問題になる可能性がある。この例については、「WebView のロード方法の確認」を参照する。

※適切な設定については「[WebView のロード方法](#)」を参照。

さらに、アプリが `loadFileURL: allowingReadAccessToURL:` というメソッドを使用しているかどうかを確認する必要がある。最初のパラメータは URL で、WebView に読み込まれる URL を含みます。2 番目のパラメータ `allowReadAccessToURL` には 1 つのファイルまたはディレクトリを含めることができます。単一のファイルを含む場合、そのファイルは WebView で利用可能である。しかし、ディレクトリを含んでいる場合、そのディレクトリ上のすべてのファイルが WebView で利用できるようになる。そのため、ディレクトリを含む場合は、その中に機密データがないことを確認する必要がある。

Swift の例「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)
```

この場合、`allowingReadAccessToURL` パラメータには、"WKWebView/scenario1.html" という 1 つのファイルが含まれており、WebView はそのファイルに対して排他的にアクセスできることを意味している。

コンパイル済みのバイナリ

```
$ rabin2 -zz ./WheresMyBrowser | grep -i "loadFileURL"
237 0x0002dff1 37 (4.__TEXT.__objc_methname) ascii
    ↪loadFileURL:allowingReadAccessToURL:
```

参考資料

- owasp-mastg Testing WebView Protocol Handlers (MSTG-PLATFORM-6) Static Analysis Testing How WebViews are Loaded

ルールブック

- WebView* でローカルコンテンツを読み込む方法（必須）
- WKWebView* でローカルコンテンツを読み込む方法（推奨）
- WKWebView.loadFileURL* のパラメータ「`allowingReadAccessTo`」に機密データを含むディレクトリを指定しない（必須）

6.6.2.2 WebView ファイルアクセス

UIWebView が使用されていることが確認できた場合は、以下のようになる。

- file:// スキームは常に有効である。
- file:// URL からのファイルアクセスは常に有効である。
- file:// URL からのユニバーサルアクセスは常に有効である。

WKWebViews について

- file:// スキームは常に有効であり、無効にすることはできない。
- デフォルトでは file:// URL からのファイルアクセスを無効にしているが、有効にすることもできる。

ファイルアクセスの設定には、以下の WebView プロパティが使用できる。

- allowFileAccessFromFileURLs (WKPreferences はデフォルトでは false): file:// スキーム URL のコンテキストで実行されている JavaScript が、他の file:// スキーム URL からのコンテンツにアクセスできるようになる。
- allowUniversalAccessFromFileURLs (WKWebViewConfiguration 、デフォルトでは false): file:// スキーム URL のコンテキストで実行する JavaScript が、任意のオリジンからのコンテンツにアクセスできるようになる。

例えば、このようにすることで、文書化されていないプロパティ allowFileAccessFromFileURLs を設定することが可能である。

Objective-C の例

```
[webView.configuration.preferences setValue:@YES forKey:@
    "allowFileAccessFromFileURLs"];
```

Swift の例

```
webView.configuration.preferences.setValue(true, forKey:
    "allowFileAccessFromFileURLs")
```

上記のプロパティが 1 つ以上有効になっている場合は、アプリが正常に動作するために本当に必要なプロパティかどうかを判断する必要がある。

参考資料

- owasp-mastg Testing WebView Protocol Handlers (MSTG-PLATFORM-6) Static Analysis Testing WebView File Access

ルールブック

- *WebView* プロパティを使用してファイルアクセスを設定する (必須)

6.6.2.3 電話番号の検出を確認

iOS の Safari では、電話番号の検出がデフォルトでオンになっている。しかし、HTML ページに電話番号と解釈できるが電話番号ではない数字が含まれている場合や、ブラウザで解析する際に DOM ドキュメントが変更されるのを防ぐために、この機能をオフにした方がよい場合がある。iOS の Safari で電話番号検出をオフにするには、format-detection meta タグ（<meta name = "format-detection" content = "telephone=no">）を使用する。この例は、Apple の開発者向けドキュメントに記載されている。電話リンクは、明示的にリンクを作成するために使用されるべきである。（例：1-408-555-5555）

参考資料

- owasp-mastg Testing WebView Protocol Handlers (MSTG-PLATFORM-6) Static Analysis Checking Telephone Number Detection

ルールブック

- 電話リンクは、明示的にリンクを作成するために使用されるべき（必須）

6.6.3 動的解析

WebView を介してローカルファイルを読み込むことが可能な場合、アプリはディレクトリトラバーサル攻撃に対して脆弱になる可能性がある。この場合、サンドボックス内のすべてのファイルへのアクセスや、（デバイスがジェイルブレイクされている場合）ファイルシステムへのフルアクセスでサンドボックスを抜け出すことが可能になる。したがって、ユーザがファイル名やファイルのロード元パスを変更できるかどうかを検証する必要があり、ロードしたファイルを編集できるようにすべきではない。

攻撃をシミュレートするために、インターフェットプロキシやダイナミックインストルメンテーションを使用して、独自の JavaScript を WebView に組み込むことができます。ローカルストレージや、JavaScript のコンテキストに公開されている可能性のあるネイティブメソッドやプロパティにアクセスすることを試してみる。

実際のシナリオでは、JavaScript は永久的なバックエンドのクロスサイトスクリプティングの脆弱性または MITM 攻撃によってのみ侵入することができる。詳しくは OWASP XSS Prevention Cheat Sheet と "iOS Network Communication" の章を参照する。

この章に関係することについて、次のことを学習する。

- WebView のロード方法の確認
- WebView ファイルアクセスの決定

参考資料

- owasp-mastg Testing WebView Protocol Handlers (MSTG-PLATFORM-6) Dynamic Analysis

6.6.3.1 WebView のロード方法の確認

上記の「WebView のロード方法」で見たように、WKWebViews の「scenario2」がロードされる場合、アプリは `URLForResource:withExtension:` と `loadHTMLString:baseURL` を呼んでそれを実行する。

これを素早く調べるには、frida-trace を使って、すべての「`loadHTMLString`」と「`URLForResource:withExtension:`」メソッドをトレースする方法がある。

```
$ 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
```

この場合、`baseURL` は `nil` に設定されており、有効なオリジンは「`null`」であることを意味する。ページの JavaScript から `window.origin` を実行することで、有効なオリジンを取得できる。(このアプリには JavaScript を記述して実行できる exploitation helper があるが、MITM を実装したり、単に Frida を使用して、WKWebView の `evaluateJavaScript:completionHandler`などを介して、JavaScript を埋め込むことも可能である)

UIWebViewに関する補足として、`baseURL` も `nil` に設定されている UIWebView から有効なオリジンを取得すると、「`null`」に設定されていないことがわかり、代わりに以下のようなものが取得できる。

```
applewebdata://5361016c-f4a0-4305-816b-65411fc1d780
```

このオリジン「`applewebdata://`」は、Same-Origin Policy を実装しておらず、ローカルファイルやあらゆる Web リソースへのアクセスを許可しているため、「`file://`」オリジンと同様である。この場合、`baseURL` を "`about:blank`" とすることで、Same-Origin Policy によってクロスオリジンアクセスを防止することができる。しかし、ここでは UIWebViews の使用を完全に避け、代わりに WKWebViews を使用することを推奨する。

参考資料

- owasp-mastg Testing WebView Protocol Handlers (MSTG-PLATFORM-6) Dynamic Analysis Checking How WebViews are Loaded

6.6.3.2 WebView ファイルアクセスの決定

元のソースコードがない場合でも、アプリの WebView がファイルアクセスを許可しているかどうか、またその種類をすぐに判断することができる。そのためには、アプリ内のターゲットの WebView に移動して、そのすべてのインスタンスを調査し、それぞれについて静的解析で言及された値、つまり `allowFileAccessFromFileURLs` と `allowUniversalAccessFromFileURLs` を取得する。これは WKWebViews にのみ適用される。(UIWebViews は常にファイルアクセスを許可する)

引き続き、「Where's My Browser?」アプリと Frida REPL を使った例で、以下の内容でスクリプトを拡張する。

```
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()); // なお、javaScriptEnabled は現在非推奨。
    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!');
  }
});
```

今すぐ実行すれば、必要な情報を全て取得することができる。

```
$ 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 // なお、javaScriptEnabled は現在非推奨。
allowFileAccessFromFileURLs: 0
hasOnlySecureContent: false
allowUniversalAccessFromFileURLs: 0
```

`allowFileAccessFromFileURLs` と `allowUniversalAccessFromFileURLs` はどちらも「0」が設定されており、これは無効になっていることを意味する。このアプリでは、WebView の設定に移動して `allowFileAccessFromFileURLs` を有効にすることができます。その後スクリプトを再実行すると、今度は「1」が設定されていることが確認できる。

```
$ frida -U -f com.authenticationfailure.WheresMyBrowser -l webviews_inspector.js
...
```

(次のページに続く)

(前のページからの続き)

allowFileAccessFromFileURLs: 1

参考資料

- owasp-mastg Testing WebView Protocol Handlers (MSTG-PLATFORM-6) Dynamic Analysis Determining WebView File Access

6.6.4 ルールブック

- WebView でローカルコンテンツを読み込む方法 (必須)
- WKWebView でローカルコンテンツを読み込む方法 (推奨)
- WKWebView.loadFileURL のパラメータ 「allowingReadAccessTo」 に機密データを含むディレクトリを指定しない (必須)
- WebView プロパティを使用してファイルアクセスを設定する (必須)
- 電話リンクは、明示的にリンクを作成するために使用されるべき (必須)

6.6.4.1 WebView でローカルコンテンツを読み込む方法 (必須)

WebView はエンドポイントからリモートコンテンツを読み込むことができるが、アプリデータディレクトリからローカルコンテンツを読み込むこともできる。ローカルコンテンツが読み込まれる場合、ユーザはファイル名やファイルの読み込みに使用されるパスに影響を与えることができないようにし、また読み込まれたファイルを編集することができないようにする必要がある。

多層防御の手段として、次のベストプラクティスを使用する。

- ローカルおよびリモートのウェブページと、読み込みを許可する URL スキームを定義したリストを作成する。
- ローカルの HTML/JavaScript ファイルのチェックサムを作成し、アプリの起動中にチェックする。JavaScript ファイルを "Minification (programming)" (最小化) して、読みにくくする。

以下のサンプルコードは、WebView でのローカルコンテンツのロード方法。

```
import UIKit
import WebKit

class webViewSample {
    var webView: WKWebView!
    // webView の生成
    // ...
    func loadLocalHTML() {
        guard let path: String = Bundle.main.path(forResource: "index", ofType:
        "html") else { return }
    }
}
```

(次のページに続く)

(前のページからの続き)

```

    let localHTMLUrl = URL(fileURLWithPath: path, isDirectory: false)
    webView.loadFileURL(localHTMLUrl, allowingReadAccessTo: localHTMLUrl)
}
}

```

これに違反する場合、以下の可能性がある。

- 悪意のあるローカルコンテンツが読み込まれる可能性がある。

6.6.4.2 WKWebView でローカルコンテンツを読み込む方法（推奨）

WKWebViews を使用する場合、Apple はローカルの HTML ファイルをロードするために `loadHTMLString:baseURL:` または `loadData:MIMEType:textEncodingName:baseURL:` を、ウェブコンテンツのために `loadRequest:` を使うことを推奨している。通常、ローカルファイルの読み込みは、特に `pathForResource ofType:`、`URLForResource:withExtension:`、`init(contentsOf:encoding:)` といったメソッドと組み合わせて行われることが多い。

記載されているメソッドのソースコードを検索し、そのパラメータを調査する。

Objective-C の例

```

#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];

    // 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;

    // UIView の view に subview 追加
    [self addSubview:self.webView];
}

```

(次のページに続く)

(前のページからの続き)

```

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

```

Swift の例「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 {}

```

WebView を介してローカルファイルを読み込むことが可能な場合、アプリはディレクトリトラバーサル攻撃に対して脆弱になる可能性がある。この場合、サンドボックス内のすべてのファイルへのアクセスや、(デバイスがジェイルブレイクされている場合) ファイルシステムへのフルアクセスでサンドボックスを抜け出すことが可能になる。したがって、ローカルファイルを読み込む場合は、ユーザがファイル名やファイルのロード元パスを変更できるかどうかを検証する必要があり、ロードしたファイルを編集できるようにすべきではない。

これに注意しない場合、以下の可能性がある。

- 悪意のあるローカルコンテンツが読み込まれる可能性がある。

6.6.4.3 WKWebView.loadFileURL のパラメータ「allowingReadAccessTo」に機密データを含むディレクトリを指定しない (必須)

allowReadAccessToURL には 1 つのファイルまたはディレクトリを含めることができる。単一のファイルを含む場合、そのファイルは WebView で利用可能である。しかし、ディレクトリを含んでいる場合、そのディレクトリ上のすべてのファイルが WebView で利用できるようになる。そのため、ディレクトリを含む場合は、その中に機密データがないことを確認する必要がある。

以下のサンプルコードは、WKWebView でのローカル HTML のロード方法である。

```

import UIKit
import WebKit

class ViewController: UIViewController {

    func load(_ bundleFileName: String) {
        let wkWebView = WKWebView()
        var url = FileManager.default.urls(for: .libraryDirectory, in: .
→ userDomainMask)[0]

```

(次のページに続く)

(前のページからの続き)

```

        url = url.appendingPathComponent("WKWebView/scenario1.html")
        wkWebView.loadFileURL(url, allowingReadAccessTo: url)
    }
}

```

ディレクトリを含む場合、仮に personal ディレクトリ内に scenario1 ファイルが無い、personal の中に入っているファイル一覧が表示される可能性がある。

```

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)
    }
}

```

これに違反する場合、以下の可能性がある。

- ディレクトリ上のすべてのファイルが WebView で利用できるようになるため、機密データが漏洩する可能性がある。

6.6.4.4 WebView プロパティを使用してファイルアクセスを設定する（必須）

ファイルアクセスの設定には、以下の WebView プロパティが使用できる。

- allowFileAccessFromFileURLs (WKPreferences はデフォルトでは false): file:// スキーム URL のコンテキストで実行されている JavaScript が、他の file:// スキーム URL からのコンテンツにアクセスできるようになる。
- allowUniversalAccessFromFileURLs (WKWebViewConfiguration 、デフォルトでは false): file:// スキーム URL のコンテキストで実行する JavaScript が、任意のオリジンからのコンテンツにアクセスできるようになる。

例えば、このようにすることで、文書化されていないプロパティ allowFileAccessFromFileURLs を設定することが可能である。

Objective-C の例

```

#import <Foundation/Foundation.h>
#import <WebKit/WebKit.h>
#import <UIKit/UIKit.h>

@interface MyWKWebViewAllowFileAccessFromFileURLs : UIView <WKNavigationDelegate, ...

```

(次のページに続く)

(前のページからの続き)

```

→WKUIDelegate> {}

@property (strong, nonatomic) WKWebView *webView;

- (void)settingWebView;
- (void)setConfiguration;
@end

@implementation MyWKWebViewAllowFileAccessFromFileURLs {}

-(void)settingWebView {

    WKWebViewConfiguration *configuration = [[WKWebViewConfiguration alloc] init];

    // webview 生成 プロパティに設定
    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];

    [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 の例

```

import Foundation
import UIKit

```

(次のページに続く)

(前のページからの続き)

```

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")
    }

}

// MARK: - WKWebView ui delegate
extension MyWKWebViewAllowFileAccessFromFileURLs: WKUIDelegate {

}

// MARK: - WKWebView WKNavigation delegate
extension MyWKWebViewAllowFileAccessFromFileURLs: WKNavigationDelegate {

}

```

上記のプロパティが 1 つ以上有効になっている場合は、アプリが正常に動作するために本当に必要なプロパティかどうかを判断する必要がある。

これに違反する場合、以下の可能性がある。

- file スキームを利用した場合、該当アプリがアクセス可能なすべてのファイルにアクセスすることが可能になる。

- 意図しない file スキームによるリクエストを受け入れてしまう。

6.6.4.5 電話リンクは、明示的にリンクを作成するために使用されるべき（必須）

iOS の Safari では、電話番号の検出がデフォルトでオンになっている。しかし、HTML ページに電話番号と解釈できるが電話番号ではない数字が含まれている場合や、ブラウザで解析する際に DOM ドキュメントが変更されるのを防ぐために、この機能をオフにした方がよい場合がある。iOS の Safari で電話番号検出をオフにするには、format-detection meta タグ（`<meta name = "format-detection" content = "telephone=no">`）を使用する。この例は、Apple の開発者向けドキュメントに記載されている。電話リンクは、明示的にリンクを作成するため使用されるべきである。（例：`1-408-555-5555`）

参考資料

- owasp-mastg Testing WebView Protocol Handlers (MSTG-PLATFORM-6) Static Analysis Checking Telephone Number Detection

これに違反する場合、以下の可能性がある。

- 電話番号ではない数字が HTML 上で電話番号として解釈される。
- ブラウザで解析する場合に DOM ドキュメントが変更される。

6.7 MSTG-PLATFORM-7

アプリのネイティブメソッドが WebView に公開されている場合、WebView はアプリパッケージ内に含まれる JavaScript のみをレンダリングしている。

6.7.1 WebView での JavaScript とネイティブとの Bridge

iOS 7 以降、Apple は WebView 内の JavaScript ランタイムと Swift または Objective-C のネイティブオブジェクトとの間の通信を可能にする API を導入した。これらの API を不用意に使用すると、悪意のあるスクリプトを WebView に注入する攻撃者に重要な機能がさらされる可能性がある（例：クロスサイトスクリプティング攻撃の成功による）。

6.7.2 静的解析

UIWebView と WKWebView はどちらも、WebView とネイティブアプリの間の通信手段を提供する。WebView の JavaScript エンジンに公開された重要なデータやネイティブの機能は、WebView で実行されている不正な JavaScript からもアクセスできるようになる。

参考資料

- owasp-mastg Determining Whether Native Methods Are Exposed Through WebViews (MSTG-PLATFORM-7) Static Analysis

6.7.2.1 UIWebView の JavaScript とネイティブブリッジのテスト

ネイティブコードと JavaScript が通信する方法には、基本的に 2 つの方法がある。

- JSContext : Objective-C や Swift のブロックが JSContext 内の識別子に割り当てられると、JavaScriptCore は自動的にそのブロックを JavaScript の関数で囲む。
- JSExport プロトコル: JSExport-inherited プロトコルで宣言されたプロパティ、インスタンスマソッド、クラスマソッドは JavaScript オブジェクトにマッピングされ、すべての JavaScript コードで利用可能である。JavaScript 環境にあるオブジェクトの修正は、ネイティブ環境に反映される。

ただし、JSExport プロトコルで定義されたクラスメンバのみが JavaScript コードからアクセス可能となることに注意してください。

WebView に関連付けられた JSContext にネイティブオブジェクトをマッピングするコードに注目し、それがどのような機能を公開しているかを分析する。たとえば、機密データは WebView にアクセスし公開すべきではありません。

Objective-C では、UIWebView に関連付けられた JSContext は以下のように取得される。

```
[webView valueForKeyPath:@"documentView.webView.mainFrame.javaScriptContext"]
```

参考資料

- owasp-mastg Determining Whether Native Methods Are Exposed Through WebViews (MSTG-PLATFORM-7) Static Analysis Testing UIWebView JavaScript to Native Bridges

ルールブック

- ネイティブコードと JavaScript が通信する方法 (必須)

6.7.2.2 WKWebView の JavaScript からネイティブブリッジのテスト

WKWebView の JavaScript コードは、ネイティブアプリにメッセージを送り返すことができるが、UIWebView とは対照的に、WKWebView の JSContext を直接参照することは不可能。その代わり、通信はメッセージシステムと postMessage 関数を使用して実装されており、JavaScript オブジェクトをネイティブの Objective-C または Swift オブジェクトに自動的にシリализする。メッセージハンドラは add(_scriptMessageHandler:name:) というメソッドで設定される。

JavaScript からネイティブへのブリッジが存在するかどうかは、WKScriptMessageHandler を検索して、公開されているすべてのメソッドをチェックすることで確認する。そして、そのメソッドがどのように呼び出されるかを確認する。

"Where's My Browser?" の次の例は、これを示している。

まず、JavaScript ブリッジがどのように有効化されているかを確認する。

```
func enableJavaScriptBridge(_ enabled: Bool) {
    options_dict["javaScriptBridge"]?.value = enabled
    let userContentController = wkWebViewConfiguration.userContentController
```

(次のページに続く)

(前のページからの続き)

```

userContentController.removeScriptMessageHandler(forName: "javaScriptBridge")

if enabled {
    let javaScriptBridgeMessageHandler = JavaScriptBridgeMessageHandler()
    userContentController.add(javaScriptBridgeMessageHandler, name:
    ↪"javaScriptBridge")
}
}

```

名前 "name" (上記の例では "javaScriptBridge") を持つスクリプトメッセージハンドラを追加すると、JavaScript 関数 window.webkit.messageHandlers.myJavaScriptMessageHandler.postMessage が、ユーザコンテンツコントローラを使用するすべてのウェビューやのすべてのフレームで定義されるようになる。そして、この関数は HTML ファイルから次のように使用することができる。

```

function invokeNativeOperation() {
    value1 = document.getElementById("value1").value
    value2 = document.getElementById("value2").value
    window.webkit.messageHandlers.javaScriptBridge.postMessage(["multiplyNumbers",
    ↪value1, value2]);
}

```

呼び出された関数は、JavaScriptBridgeMessageHandler.swift に存在する。

```

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)

```

ここで問題なのは、JavaScriptBridgeMessageHandler がその関数を含むだけでなく、機密性の高い関数を公開している点である。

```

case "getSecret":
    result = "XSRSOGKC342"

```

参考資料

- owasp-mastg Determining Whether Native Methods Are Exposed Through WebViews (MSTG-PLATFORM-7) Static Analysis Testing UIWebView JavaScript to Native Bridges

ルールブック

- *WKWebView の JavaScript コードが、ネイティブアプリにメッセージを送り返す方法（必須）*

6.7.3 動的解析

この時点では、iOS アプリ内の潜在的に興味深い WebView をすべて特定し、潜在的な攻撃対象領域の概観を得ました（静的分析、前のセクションで見た動的分析技術、またはそれらの組み合わせによって）。これには HTML と JavaScript ファイル、UIWebView の JSContext と JSExport、WKWebView の WKScriptMessageHandler の使用、そして WebView で公開されている関数が含まれている。

さらに動的解析を行うことで、これらの関数を悪用し、それらが公開している可能性のある機密データを取得することができる。静的解析で見たように、前の例ではリバースエンジニアリングによって秘密の値を取得するのは簡単でした（秘密の値はソースコード内の平文で見つかりました）しかし、公開された関数が安全なストレージから秘密を取得すると想像してください。この場合、動的解析と悪用だけが役に立つ。

関数を悪用する手順は、JavaScript のペイロードを生成し、アプリがリクエストしているファイルに注入することから始まる。このインジェクションは、例えば、さまざまな手法で実現できる。

- コンテンツの一部がインターネットから HTTP で安全に読み込まれない場合（ミックスコンテンツ）、MITM 攻撃の実装を試みることができる。
- Frida のようなフレームワークと、iOS の WebView で利用できる対応する JavaScript 評価関数（UIWebView では `stringByEvaluatingJavaScriptFromString`、WKWebView では `evaluateJavaScript:completionHandler:`）を使って、常に動的インストルメンテーションと JavaScript ペイロードをインジェクションすることが可能である。

前の例の "Where's My Browser?" アプリから秘密を取得するために、これらのテクニックの 1 つを使用して、WebView の "result" フィールドに書き込むことで秘密を明らかにする次のペイロードを注入することができる。

```
function javascriptBridgeCallBack(name, value) {
    document.getElementById("result").innerHTML=value;
};

window.webkit.messageHandlers.javaScriptBridge.postMessage(["getSecret"]);
```

もちろん、同社が提供する Exploitation Helper を利用することも可能である。



WebView に公開されている脆弱な iOS アプリと関数の別の例は、[#thiel2] page 156 で参照してください。

参考資料

- owasp-mastg Determining Whether Native Methods Are Exposed Through WebViews (MSTG-PLATFORM-7) Dynamic Analysis

6.7.4 ルールブック

- ネイティブコードと JavaScript が通信する方法（必須）
- WKWebView の JavaScript コードが、ネイティブアプリにメッセージを送り返す方法（必須）

6.7.4.1 ネイティブコードと JavaScript が通信する方法（必須）

ネイティブコードと JavaScript が通信する方法には、基本的に 2 つの方法がある。

- JSContext : Objective-C や Swift のブロックが JSContext 内の識別子に割り当てられると、JavaScriptCore は自動的にそのブロックを JavaScript の関数で囲む。
- JSExport プロトコル: JSExport-inherited プロトコルで宣言されたプロパティ、インスタンスメソッド、クラスメソッドは JavaScript オブジェクトにマッピングされ、すべての JavaScript コードで利用可能である。JavaScript 環境にあるオブジェクトの修正は、ネイティブ環境に反映される。

ただし、JSExport プロトコルで定義されたクラスメンバのみが JavaScript コードからアクセス可能となることに注意してください。

WebView に関連付けられた JSContext にネイティブオブジェクトをマッピングするコードに注目し、それがどのような機能を公開しているかを分析する。たとえば、機密データは WebView にアクセスし公開すべきで

はありません。

Objective-C では、UIWebView に関連付けられた JSContext は以下のように取得される。

```
#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];

}

@end
```

JSExport プロトコル

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

(次のページに続く)

(前のページからの続き)

```
// 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 MyJSEExample : NSObject
- (void)execute;
@end

@implementation MyJSEExample
- (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
```

参考資料

- owasp-mastg Determining Whether Native Methods Are Exposed Through WebViews (MSTG-PLATFORM-7) Static Analysis Testing UIWebView JavaScript to Native Bridges
- JSExport

これに違反する場合、以下の可能性がある。

- 悪意のあるスクリプトを WebView に注入する攻撃者に重要な機能がさらされる可能性がある。

6.7.4.2 WKWebView の JavaScript コードが、ネイティブアプリにメッセージを送り返す方法（必須）

WKWebView の JavaScript コードは、ネイティブアプリにメッセージを送り返すことができるが、UIWebView とは対照的に、WKWebView の JSContext を直接参照することは不可能。その代わり、通信はメッセージングシステムと postMessage 関数を使用して実装されており、JavaScript オブジェクトをネイティブの Objective-C または Swift オブジェクトに自動的にシリアル化する。メッセージハンドラは add(_scriptMessageHandler:name:) というメソッドで設定される。

JavaScript からネイティブへのブリッジが存在するかどうかは、WKScriptMessageHandler を検索して、公開されているすべてのメソッドをチェックすることで確認する。そして、そのメソッドがどのように呼び出されるかを確認する。

"Where's My Browser?"の次の例は、これを示している。

まず、JavaScript ブリッジがどのように有効化されているかを確認する。

```
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")
    }
}
```

公開すべきでないメソッドをメッセージハンドラに指定していないかを確認する。

これに違反する場合、以下の可能性がある。

- 悪意のあるスクリプトを WebView に注入する攻撃者に重要な機能がさらされる可能性がある。

6.8 MSTG-PLATFORM-8

オブジェクトのデシリアライゼーションは、もしあれば、安全なシリアル化 API を使用して実装されている。

6.8.1 iOS でのオブジェクトのシリアル化

iOS ではオブジェクトを永続化する方法がいくつかある。

6.8.1.1 オブジェクトのエンコード

iOS には、Objective-C または NSObjects のオブジェクトエンコードとデコードのための 2 つのプロトコルが用意されている。NSCoding と NSSecureCoding である。クラスがプロトコルのいずれかに準拠している場合、データはバイトバッファのラッパー NSData にシリализされる。Swift のデータは、NSData または変更可能な NSMutableData と同じであることに注意する。

NSCoding プロトコルは、そのインスタンス変数をエンコード/デコードするために実装する必要がある 2 つのメソッドを宣言している。NSCoding を使用するクラスは、NSObject を実装するか、@objc クラスとしてアンテーションする必要がある。NSCoding プロトコルは、以下のように encode と init を実装する必要がある。

```
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 = ""

    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.
}
```

NSCoding の問題点は、クラスタイプを評価する前に、すでにオブジェクトが構築され挿入されていることが多い。このため、攻撃者はあらゆる種類のデータを簡単に注入することができる。そこで、NSSecureCoding プロトコルが導入された。NSSecureCoding に準拠する場合、以下を含める必要がある。

```
static var supportsSecureCoding: Bool {
    return true
}
```

init(coder:) がクラスの一部であるとき。次に、オブジェクトをデコードするときに、例えば、以下のようなチェックを行う。

```
let obj = decoder.decodeObject(of:MyClass.self, forKey: "myKey")
```

NSSecureCoding への準拠は、インスタンス化されるオブジェクトが本当に期待されたものであることを保証する。しかし、データに対する追加の完全性チェックは行われず、データは暗号化されない。したがって、秘密データは追加の暗号化を必要とし、完全性が保護されなければならないデータは、追加の HMAC を取得する必要がある。

NSData (Objective-C) またはキーワード let (Swift) が使用される場合、データはメモリ内で不变であり、簡単に削除できないことに注意する。

参考資料

- owasp-mastg Testing Object Persistence (MSTG-PLATFORM-8) Overview Object Encoding

ルールブック

- オブジェクトエンコードとデコードのために *NSSecureCoding* を使用する (必須)

6.8.1.2 NSKeyedArchiver によるオブジェクトアーカイブ

NSKeyedArchiver は NSCoder のコンクリートサブクラスで、オブジェクトをエンコードしてファイルに保存する方法を提供する。NSKeyedUnarchiver はデータをデコードして元のデータを再作成する。NSCoding のセクションの例で、今度はアーカイブとアンアーカイブをみてみる。

```
// archiving:
NSKeyedArchiver.archiveRootObject(customPoint, toFile: "/path/to/archive")

// unarchiving:
guard let customPoint = NSKeyedUnarchiver.unarchiveObjectWithFile("/path/to/archive"
    ↪") as?
    CustomPoint else { return nil }
```

キー付きアーカイブをデコードする場合、値が名前で要求されるため、値が順番にデコードされなかったり、まったくデコードされなかったりすることがある。したがって、キー付きアーカイブは、前方および後方互換性のためのより良いサポートを提供する。これは、ディスク上のアーカイブは、その与えられたデータのためのキーが後の段階で提供されない限り、プログラムによって検出されない追加のデータを実際に含むことができることを意味する。

データはファイル内で暗号化されないので、機密データの場合はファイルを保護するために追加の保護が必要であることに注意する。詳しくは「Data Storage on iOS」の章を確認すること。

参考資料

- owasp-mastg Testing Object Persistence (MSTG-PLATFORM-8) Overview Object Archiving with NSKeyedArchiver

ルールブック

- オブジェクトの永続化を使用してデバイスに機密情報を保存する場合、データを暗号化する (必須)

6.8.1.3 Codable

Swift 4 では、 Codable 型のエイリアスが登場した: これは Decodable と Encodable のプロトコルの組み合わせである。文字列、Int、Double、Date、Data と URL は、本質的に Codable である:つまり、追加の作業なしに、簡単にエンコードとデコードが可能である。次のような例を見てみる。

```
struct CustomPointStruct:Codable {
    var x: Double
    var name: String
}
```

例の CustomPointStruct の継承リストに Codable を追加することで、 init(from:) と encode(to:) のメソッドが自動的にサポートされるようになった。 Codable の動作の詳細は、 [Apple Developer Documentation](#) を参照。 Codable は、様々な表現に簡単にエンコード/デコードすることができる。 NSCoding/NSSecureCoding を使用した NSData、JSON、プロパティリスト、XML などである。 詳細は、以下のセクションを参照。

参考資料

- [owasp-mastg Testing Object Persistence \(MSTG-PLATFORM-8\) Overview Codable](#)

ルールブック

- *Swift 4 では、 Decodable と Encodable のプロトコルの組み合わせで利用する（必須）*

6.8.1.4 JSON と Codable

iOS で JSON をエンコード/デコードするには、さまざまなサードパーティライブラリを使用する方法がある。

- Mantle
- JSONModel ライブラリ
- SwiftyJSON ライブラリ
- ObjectMapper ライブラリ
- JSONKit
- JSONModel
- YYModel
- SBJson 5
- Unbox
- Gloss
- Mapper
- JASON
- Arrow

ライブラリは、Swift や Objective-C の特定のバージョンへの対応、(im) mutable な結果を返すかどうか、速度、メモリ消費量、実際のライブラリサイズなどに違いがある。ここでも、不变性の場合の注意点として、機密情報をメモリから簡単に削除することはできない。

次に、Apple は、 Codable と JSONEncoder、JSONDecoder を組み合わせることで、JSON エンコード/デコードを直接サポートするようにした。

```
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 自体は、(NoSQL) データベースやファイルなど、どこにでも保存することができる。ただ、秘密を含む JSON が適切に保護されていることを確認する必要がある（例：暗号化/HMAC 化）。詳しくは「Data Storage on iOS」の章を確認すること。

参考資料

- owasp-mastg Testing Object Persistence (MSTG-PLATFORM-8) Overview JSON and Codable

ルールブック

- JSON エンコード/デコードは Codable/JSONEncoder/JSONDecoder を組み合わせて行う（必須）
- オブジェクトの永続化を使用してデバイスに機密情報を保存する場合、データを暗号化する（必須）
- オブジェクトに格納された実際の情報を処理する前に、常に HMAC /署名を検証する（必須）

6.8.1.5 プロパティリストと Codable

オブジェクトをプロパティリスト（前のセクションで plist とも呼ばれた）に永続化することができる。以下に、その使い方の例を 2 つ紹介する。

```
// archiving:  
let data = NSKeyedArchiver.archivedDataWithRootObject(customPoint) // なお、  
// archivedDataWithRootObject は現在非推奨。  
NSUserDefaults.standardUserDefaults().setObject(data, forKey: "customPoint")  
  
// unarchiving:
```

(次のページに続く)

(前のページからの続き)

```
if let data = UserDefaults.standardUserDefaults().objectForKey("customPoint") as?
    → NSData {
    let customPoint = NSKeyedUnarchiver.unarchiveObjectWithData(data) // なお、
    →unarchiveObjectWithData は現在非推奨。
}
```

この最初の例では、主要なプロパティリストである UserDefaults が使用されている。 Codable バージョンでも同じことができる。

```
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)
}
```

plist ファイルは、秘密の情報を保存するためのものではないことに注意する。アプリのユーザ設定を保持するために設計されている。

参考資料

- owasp-mastg Testing Object Persistence (MSTG-PLATFORM-8) Overview Property Lists and Codable

ルールブック

- オブジェクトをプロパティリストに永続化する方法 (必須)

6.8.1.6 XML

XML エンコードを行う方法は複数ある。JSON のベースと同様に、様々なサードパーティライブラリがある。

- Fuzi
- Ono
- AEXML
- RaptureXML
- SwiftyXMLParser

- SWXMLHash

速度、メモリ使用量、オブジェクトの永続性、そしてより重要な点として、XML 外部エンティティの扱い方が異なる。例として、Apple iOS の Office ビューアにおける XXE を確認すること。したがって、可能であれば、外部エンティティのパースを無効にすることが重要。詳しくは OWASP XXE 対策チートシートを確認すること。ライブラリの次に、Apple の XMLParser クラスを活用することができる。

サードパーティのライブラリを使わず、Apple の XMLParser を使う場合は、必ず shouldResolveExternalEntities が false を返すようにすること。

参考資料

- owasp-mastg Testing Object Persistence (MSTG-PLATFORM-8) Overview XML

ルールブック

- 外部エンティティのパースを無効にする (推奨)

6.8.1.7 オブジェクトリレーションナルマッピング (CoreData と Realm)

iOS には様々な ORM 的なソリューションがある。まず最初に紹介するのは Realm で、独自のストレージエンジンを搭載している。Realm のドキュメントで説明されているように、Realm はデータを暗号化する設定を持っている。これにより、安全なデータを扱うことができる。なお、デフォルトでは暗号化はオフになっている。

Apple は CoreData を提供しており、Apple Developer Documentation で詳しく説明されている。Apple の Persistent Store Types and Behaviors のドキュメントで説明されているように、様々なストレージバックエンドをサポートしている。Apple が推奨するストレージバックエンドの問題点は、どのタイプのデータストアも暗号化されておらず、完全性のチェックもされていない。そのため、機密性の高いデータの場合は、追加の対応が必要になる。代替案として、iMas というプロジェクトがあり、これはすぐに暗号化を行うことができる。

参考資料

- owasp-mastg Testing Object Persistence (MSTG-PLATFORM-8) Overview Object-Relational Mapping (CoreData and Realm)

ルールブック

- Realm はデータの暗号化を実施し活用する (推奨)

6.8.1.8 Protocol Buffers

Google が提供する Protocol Buffers は、Binary Data Format によって構造化データをシリализするための、プラットフォームや言語に依存しないメカニズムである。iOS では、Protobuf ライブラリにより利用可能。CVE-2015-5237 のように、Protocol Buffers を使った脆弱性がいくつか存在する。なお、Protocol Buffers は暗号化を内蔵していないため、機密性の保護はできない。

参考資料

- owasp-mastg Testing Object Persistence (MSTG-PLATFORM-8) Overview Protocol Buffers

6.8.2 静的解析

オブジェクトの永続化には、以下のような懸念がある。

- オブジェクトの永続化を使用してデバイスに機密情報を保存する場合、データが暗号化されていることを確認する。データベースレベルか、特に値レベルでの暗号化。
- 情報の完全性を保証する必要があるか、HMAC メカニズムを使用するか、保存された情報に署名する。オブジェクトに格納された実際の情報を処理する前に、常に HMAC /署名を検証する。
- 上記の 2 つの概念で使用される鍵は、Keychain に安全に格納され、十分に保護されていることを確認する。詳しくは「[Data Storage on iOS](#)」の章を参照。
- デシリアライズされたオブジェクト内のデータが、積極的に使用される前に慎重に検証されることを確認する（例えば、ビジネス / アプリケーションロジックの悪用が可能でないこと）。
- リスクの高いアプリケーションでは、オブジェクトのシリアル化/デシリアル化に [Runtime Reference](#) を使用する永続化メカニズムを使用しない。攻撃者はこのメカニズムを介してビジネスロジックを実行する手順を操作できるかもしれないから（詳細は「[iOS Anti-Reversing Defenses](#)」の章を参照）。
- Swift 2 以降では、Mirror はオブジェクトの一部を読み取るために使用できるが、オブジェクトに対して書き込むために使用することはできないことに注意する。

参考資料

- [owasp-mastg Testing Object Persistence \(MSTG-PLATFORM-8\) Static Analysis](#)

ルールブック

- オブジェクトの永続化を使用してデバイスに機密情報を保存する場合、データを暗号化する（必須）
- オブジェクトに格納された実際の情報を処理する前に、常に HMAC /署名を検証する（必須）
- リスクの高いアプリケーションでは、オブジェクトのシリアル化/デシリアル化に [Runtime Reference](#) を使用する永続化メカニズムを使用しない（必須）
- デシリアル化されたオブジェクト内のデータは慎重に検証を行う（必須）

6.8.3 動的解析

動的解析を行うには、いくつかの方法がある。

- 実際のパーシステンスの場合。「[Data Storage on iOS](#)」の章で説明したテクニックを使用する。
- シリアル化/デシリアル化のものについて。デバッグビルドを使用するか、Frida / objection を使用して、シリアル化/デシリアル化メソッドがどのように処理されるかを確認する（たとえば、アプリケーションがクラッシュするか、オブジェクトを強化することで追加情報が抽出されるなど）。

参考資料

- owasp-mastg Testing Object Persistence (MSTG-PLATFORM-8) Dynamic Analysis

6.8.4 ルールブック

1. オブジェクトエンコードとデコードのために *NSSecureCoding* を使用する（必須）
2. *Swift 4* では、*Decodable* と *Encodable* のプロトコルの組み合わせで利用する（必須）
3. *JSON* エンコード/デコードは *Codable/JSONEncoder/JSONDecoder* を組み合わせて行う（必須）
4. オブジェクトをプロパティリストに永続化する方法（必須）
5. 外部エンティティのペースを無効にする（推奨）
6. *Realm* はデータの暗号化を実施し活用する（推奨）
7. オブジェクトの永続化を使用してデバイスに機密情報を保存する場合、データを暗号化する（必須）
8. オブジェクトに格納された実際の情報を処理する前に、常に *HMAC* /署名を検証する（必須）
9. リスクの高いアプリケーションでは、オブジェクトのシリアル化/デシリアル化に *Runtime Reference* を使用する永続化メカニズムを使用しない（必須）
10. デシリアル化されたオブジェクト内のデータは慎重に検証を行う（必須）

6.8.4.1 オブジェクトエンコードとデコードのために *NSSecureCoding* を使用する（必須）

iOS には、Objective-C または *NSObject* のオブジェクトエンコードとデコードのための 2 つのプロトコルが用意されている。*NSCoding* と *NSSecureCoding* である。クラスがプロトコルのいずれかに準拠している場合、データはバイトバッファのラッパー *NSData* にシリアル化される。Swift のデータは、*NSData* または変更可能な *NSMutableData* と同じであることに注意する。*NSCoding* プロトコルは、そのインスタンス変数をエンコード/デコードするために実装する必要がある 2 つのメソッドを宣言している。*NSCoding* を使用するクラスは、*NSObject* を実装するか、@objc クラスとしてアノテーションする必要がある。

ただし *NSCoding* では、エンコードされたオブジェクトをデコードする際に、期待する型で保存したもののかを確認できない。そのため期待していない型でデコードする危険性がある。*NSSecureCoding* では、期待する型を指定できるので、期待する型のオブジェクトの場合のみデコードを行った結果を取得できる。

```
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")
    }
}
```

(次のページに続く)

(前のページからの続き)

```

    }

    init(x: Double, name: String) {
        self.x = x
        self.name = name
    }

    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)
    }

}

```

`init(coder:)` がクラスの一部であるとき。次に、オブジェクトをデコードするときに、例えば、以下のようなチェックを行う。

```
let obj = decoder.decodeObject(of:MyClass.self, forKey: "myKey")
```

`NSSecureCoding`への準拠は、インスタンス化されるオブジェクトが本当に期待されたものであることを保証する。しかし、データに対する追加の完全性チェックは行われず、データは暗号化されない。したがって、秘密データは追加の暗号化を必要とし、完全性が保護されなければならないデータは、追加の HMAC を取得する必要がある。

これに違反する場合、以下の可能性がある。

- ・インスタンス化されるオブジェクトが本当に期待されたものであることが保証されない。

6.8.4.2 Swift 4 では、Decodable と Encodable のプロトコルの組み合わせで利用する（必須）

Swift 4 では、`Codable` 型のエイリアスが登場した：これは `Decodable` と `Encodable` のプロトコルの組み合わせである。文字列、`Int`、`Double`、`Date`、`Data` と `URL` は、本質的に `Codable` である：つまり、追加の作業なしに、簡単にエンコードとデコードが可能である。次のような例を見てみる。

```

struct CustomPointStruct:Codable {
    var x: Double
    var name: String
}

```

例の `CustomPointStruct` の継承リストに `Codable` を追加することで、`init(from:)` と `encode(to:)` のメソッドが自動的にサポートされるようになった。`Codable` の動作の詳細は、[Apple Developer Documentation](#) を参照。`Codable` は、様々な表現に簡単にエンコード/デコードすることができる。`NSCoding/NSSecureCoding` を使用した `NSData`、`JSON`、プロパティリスト、`XML` などである。詳細は、以下のセクションを参照。

※違反した場合または注意しなかった場合の事象として記載可能なものなし。

6.8.4.3 JSON エンコード/デコードは Codable/JSONEncoder/JSONDecoder を組み合わせて行う（必須）

Apple は、Codable と JSONEncoder、JSONDecoder を組み合わせることで、JSON エンコード/デコードを直接サポートするようにした。

```
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"  
// })
```

※違反した場合または注意しなかった場合の事象として記載可能なものなし。

6.8.4.4 オブジェクトをプロパティリストに永続化する方法（必須）

オブジェクトをプロパティリスト（前のセクションで plist とも呼ばれた）に永続化することができる。以下に、その使い方の例を 2 つ紹介する。

```
// archiving:  
let data = NSKeyedArchiver.archivedDataWithRootObject(customPoint) // なお、  
↪archivedDataWithRootObject は現在非推奨。  
NSUserDefaults.standardUserDefaults().setObject(data, forKey: "customPoint")  
  
// unarchiving:  
if let data = UserDefaults.standardUserDefaults().objectForKey("customPoint") as?  
↪NSData {  
    let customPoint = NSKeyedUnarchiver.unarchiveObjectWithData(data) // なお、  
↪unarchiveObjectWithData は現在非推奨。  
}
```

この最初の例では、主要なプロパティリストである UserDefaults が使用されている。 Codable バージョンでも同じことができる。

```
struct CustomPointStruct: Codable {
    var point: Double
    var name: String
}
```

(次のページに続く)

(前のページからの続き)

```

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)
}

```

iOS 12 から archivedDataWithRootObject と unarchiveObjectWithData は廃止されたため、archivedDataWithRootObject:requiringSecureCoding:、unarchivedObject:ofClass:from を使用する必要がある。

```

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)
    }
}

// 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
}

```

plist ファイルは、秘密の情報を保存するためのものではないことに注意する。アプリのユーザ設定を保持するために設計されている。

これに違反する場合、以下の可能性がある。

- プロパティリストから機密情報が読み取られる可能性がある。

6.8.4.5 外部エンティティのパースを無効にする（推奨）

外部エンティティのパースを無効にすることが推奨される。詳しくは OWASP XXE 対策チートシートを確認すること。その他に、Apple の [XMLParser クラス](#) を活用することができる。

```
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()
        }
    }

    func parser(_ parser: XMLParser, didEndElement elementName: String, namespaceURI: String?, qualifiedName qName: String?) {

        if elementName == "title" || elementName == "link" {
            arrDetail.append(content)
        }
    }
}
```

(次のページに続く)

(前のページからの続き)

```

    else if elementName == "item" {
        arrFinal.append(arrDetail)
    }
}

func parser(_ parser: XMLParser, foundCharacters string: String) {
    content = string
}

func parserDidEndDocument(_ parser: XMLParser) {
    print(arrFinal)
}
}
}

```

これに注意しない場合、以下の可能性がある。

- XXE 攻撃に対して脆弱になる。

6.8.4.6 Realm はデータの暗号化を実施し活用する（推奨）

iOS の ORM 的なソリューションの Realm を活用する場合、暗号化を実施すべきである。Realm のドキュメントで説明されているように、Realm はデータを暗号化する設定を持っている。これにより、安全なデータを扱うことができる。なお、デフォルトでは暗号化はオフになっている。

例：Realm DB 暗号化方法

```

#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];
    }
}

```

(次のページに続く)

(前のページからの続き)

```

RLMRealm *realm = [RLMRealm realmWithConfiguration:configuration
                                             error:&nil];

    // 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:YES];

    // 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;
}

```

(次のページに続く)

(前のページからの続き)

@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!

  # Pods for OBCHelloWorld
  pod "Realm"

end
```

参考資料

- RealmDB 暗号化

これに注意しない場合、以下の可能性がある。

- 攻撃者や悪意のあるアプリケーションからデータを読み取られる可能性がある。

6.8.4.7 オブジェクトの永続化を使用してデバイスに機密情報を保存する場合、データを暗号化する（必須）

オブジェクトの永続化を使用してデバイスに機密情報を保存する場合、データが暗号化されていることを確認する。データベースレベルか、特に値レベルでの暗号化。

対称鍵を使用して文字列を暗号化する例：

```
class EncryptionSample {

    func EncryptionSample( rawString:String ) {

        // 対称鍵の生成と解放：
        let encryptionKey = SymmetricKey(size: .bits256)
        // ...

        // SHA-2 512-bit digest の計算：
        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
    }
}
```

これに違反する場合、以下の可能性がある。

- 永続化されたデータに含まれる機密情報が読み取られる可能性がある。

6.8.4.8 オブジェクトに格納された実際の情報を処理する前に、常に HMAC /署名を検証する（必須）

情報の完全性を保証する必要があるか？ HMAC メカニズムを使用するか、保存された情報に署名する。オブジェクトに格納された実際の情報を処理する前に、常に HMAC / 署名を検証する。

HMAC メカニズムを使用したハッシュ変換サンプルコードは以下となる。

```
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 ""
        }

        return hmac.toHexString()
    }
}
```

保存された情報に署名する（セキュアにデータを保存する方法）については、以下ルールブックのサンプルコードを参照。

ルールブック

- Keychain Services API* を使用してセキュアに値を保存する（必須）

これに違反する場合、以下の可能性がある。

- 永続化されたデータに含まれる機密情報が変更された場合に察知できない可能性がある。

6.8.4.9 リスクの高いアプリケーションでは、オブジェクトのシリアル化/デシリアル化に **Runtime Reference** を使用する永続化メカニズムを使用しない（必須）

シリアル化/デシリアル化自体は便利な機能ですが、利用にあたって注意が必要である。リスクの高いアプリケーションで、オブジェクトのシリアル化/デシリアル化に **Runtime Reference** を使用する永続化メカニズムを使用すると、外部から与えられるデータをデシリアル化する際に意図しないオブジェクトを操作され、不正な動作を引き起こしてしまうという脆弱性がある。

詳細は「[iOS Anti-Reversing Defenses](#)」の章を参照する。

※概念的なルールのため、サンプルコードはなし。

これに違反する場合、以下の可能性がある。

- 攻撃者はこの脆弱性を悪用し、リモートコードの実行・特権の昇格・任意のファイルアクセス・DoSなどを大きな被害を引き起こす可能性がある。

6.8.4.10 デシリアライズされたオブジェクト内のデータは慎重に検証を行う（必須）

デシリアライズすると、受け取ったバイト列を元にオブジェクトが生成される。アプリ上は変更不可の値も外部から値を自由に操作することができる。例えば User クラスに status というフィールドを持ち、管理者 or ユーザ権限の判断をしていた場合、ユーザが管理者権限としてデシリアライズすることが可能となってしまう。また、name フィールドなどを文字列で持っていた場合、そのままデータベースのクエリに使用してしまっていると SQL インジェクションを起こす文字列を挿入することが可能になる場合がある。

最大の対策方法は、外部から与えられるデータをデシリアライズしない設計とすることである。もしくは、他に影響しないデータのみをデシリアライズの対象とする。

もし、外部からのデータをデシリアライズする必要がある場合には、シリアル化されたデータをデジタル署名などを使用して完全性チェックを実装する。

下記の暗号的に安全なハッシュを使用してシリアル化した値とハッシュ値を比較などで対応する。

- iOS 暗号化アルゴリズム Apple CryptoKit の実装（推奨）

これに違反する場合、以下の可能性がある。

- SQL インジェクションを引き起こす。
- アクセス権限の不正な昇格。



コード品質とビルド設定要件

7.1 MSTG-CODE-1

アプリは有効な証明書で署名およびプロビジョニングされている。その秘密鍵は適切に保護されている。

7.1.1 コード署名

アプリにコード署名を行うことで、アプリの提供元が既知であり、最後に署名された後に変更されていないことをユーザに保証する。アプリケーションがアプリケーションサービスを統合したり、ジェイルブレイクされていないデバイスにインストールされたり、App Store に提出されたりする前に、Apple が発行した証明書で署名する必要がある。証明書の要求方法およびアプリケーションへのコード署名の方法について詳しくは、[App Distribution Guide](#) を参照する。

参考資料

- [owasp-mastg Making Sure that the App Is Properly Signed \(MSTG-CODE-1\) Overview](#)

ルールブック

- [アプリのコード署名方法 \(必須\)](#)

7.1.2 静的解析

アプリが最新のコード署名形式を使用していることを確認する必要がある。アプリケーションの .app ファイルから codesign を使用して署名証明書の情報を取得することができる。Codesign は、コード署名の作成、チェック、表示、およびシステム内の署名済みコードの動的ステータスの照会に使用される。

アプリケーションの IPA ファイルを取得したら、ZIP ファイルとして保存し直し、ZIP ファイルを解凍する。アプリケーションの .app ファイルがあるペイロード ディレクトリに移動する。

以下の codesign コマンドを実行すると、署名情報が表示される。

```
$ 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=455758418a5f6a878bb8fdb709ccfcfa52c0b5b9e
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
```

Apple のドキュメントに記載されているように、アプリを配布する方法には、App Store を使用する方法、Apple Business Manager を使用してカスタムまたは社内配布する方法など、さまざまな方法がある。社内配布の場合、配布用に署名したアプリにアドホック証明書が使用されていないことを確認する。

参考資料

- [owasp-mastg Making Sure that the App Is Properly Signed \(MSTG-CODE-1\) Static Analysis](#)

ルールブック

- アプリが最新のコード署名形式を使用していることを確認する必要がある（必須）

7.1.3 ルールブック

1. アプリのコード署名方法（必須）
2. アプリが最新のコード署名形式を使用していることを確認する必要がある（必須）

7.1.3.1 アプリのコード署名方法（必須）

以下手順に従い、アプリへのコード署名を行う。

1. Apple ID をアカウント設定に追加する

ユーザを識別し、ユーザのチームに関する情報をダウンロードするために、ユーザの Apple ID アカウントをアカウント環境設定に追加する。Xcode は、ユーザが属しているすべてのチームに関する情報をダウンロードするために、Apple ID の資格情報を使用する。

2. プロジェクト内のターゲットをチームに割り当てる

ユーザのプロジェクト内の各ターゲットをチームに割り当てる。Xcode は、関連するチームアカウントに署名の権限（証明書、識別子、およびプロビジョニングプロファイル）を格納する。ユーザが組織として登録した場合、プログラムの役割は、Xcode で実行できるタスクを決定する。ユーザが個人として登録する場合、ユーザは一人のチームのためのアカウント所有者である。あなたが Apple Developer Program のメンバーでない場合、Xcode はユーザのために個人的なチームを作成する。

3. アプリに機能を追加する

「署名と機能」で、使用したいアプリサービスを有効にする。Xcode は、ユーザのプロジェクトを構成し、それに応じて署名の権限を更新する。必要であれば、Xcode は、関連する App ID のアプリサービスを有効にし、それが管理するプロビジョニングプロファイルを再生成する。いくつかのアプリサービスを完全に有効にするには、開発者アカウントまたは App Store Connect にサインインする必要がある場合がある。

4. 端末上でアプリを実行する

接続された端末またはワイヤレス端末（iOS、tvOS）でアプリを初めて実行するとき、Xcode はあなたのために必要な開発署名の権限を作成する。Xcode は、あなたが選択したデバイスを登録し、それが管理するプロビジョニングプロファイルに追加する。macOS アプリの場合、Xcode は、Xcode を実行している Mac を登録する。

5. 署名証明書とプロビジョニングプロファイルのエクスポート

端末上でアプリを起動するために使用する署名証明書とプロビジョニングプロファイルは、ユーザの Mac に保存されている。署名証明書の秘密鍵は Keychain にのみ保存されるため、開発者アカウントの署名証明書とプロビジョニングプロファイルをエクスポートするには良いタイミングである。また、開発者アカウントをエクスポートして、署名権限を別の Mac に移動することもできる。

これに違反する場合、以下の可能性がある。

- ユーザに対してアプリへの署名後にソースが変更されていないことを保証できない。

7.1.3.2 アプリが最新のコード署名形式を使用していることを確認する必要がある（必須）

アプリが最新のコード署名形式を使用していることを確認する必要がある。アプリケーションの .app ファイルから codesign を使用して署名証明書の情報を取得することができる。Codesign は、コード署名の作成、チェック、表示、およびシステム内の署名済みコードの動的ステータスの照会に使用される。

アプリケーションの IPA ファイルを取得したら、ZIP ファイルとして保存し直し、ZIP ファイルを解凍する。アプリケーションの .app ファイルがあるペイロード ディレクトリに移動する。

以下の codesign コマンドを実行すると、署名情報が表示される。

```
$ 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
```

(次のページに続く)

(前のページからの続き)

```
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
```

これに違反する場合、以下の可能性がある。

- ストアへ登録できない可能性がある。

7.2 MSTG-CODE-2

アプリはリリースモードでビルドされている。リリースビルドに適した設定である（デバッグ不可など）。

7.2.1 ビルドのモード設定

iOS アプリケーションのデバッグは、lldb と呼ばれる強力なデバッガを組み込んだ Xcode を使って行うことができる。lldb は Xcode5 からデフォルトのデバッガとなり、gdb のような GNU ツールに代わって開発環境に完全に統合された。デバッグはアプリを開発する際には便利な機能だが、App Store やエンタープライズプログラムにアプリをリリースする前にはオフにする必要がある。

アプリをビルドまたはリリースモードで生成することは、Xcode のビルド設定に依存し、デバッグモードでアプリを生成すると、生成されたファイルに DEBUG flag が挿入される。

参考資料

- owasp-mastg Determining Whether the App is Debuggable (MSTG-CODE-2)

7.2.2 静的解析

まず、環境内の flag を確認するために、アプリを生成するモードを決定する必要がある。

- プロジェクトのビルド設定を選択する。
- 「Apple LVM - Preprocessing」と「Preprocessor Macros」で、「DEBUG」または「DEBUG_MODE」が選択されていないことを確認する。（Objective-C）

- または、「Swift Compiler - Custom flags」セクション / 「Other Swift Flags」において、「-D DEBUG」エントリが存在しないことを確認する。
- Manage Schemes を開き、リリース用のスキームで「Debug executable」オプションが選択されていないことを確認する。

参考資料

- [owasp-mastg Determining Whether the App is Debuggable \(MSTG-CODE-2\) Static Analysis](#)

ルールブック

- ビルドのモード設定確認（必須）

7.2.3 動的解析

Xcode を使用して、直接デバッガをアタッチできるかどうかを確認する。次に、アプリを Clutching した後、ジェイルブレイク デバイス上でデバッグできるかどうかを確認する。これは、Cydia の BigBoss リポジトリにある debug-server を使用する。

注：アプリケーションにアンチリバースエンジニアリングコントロールが装備されている場合、デバッガを検知して停止させることができる。

参考資料

- [owasp-mastg Determining Whether the App is Debuggable \(MSTG-CODE-2\) Dynamic Analysis](#)

7.2.4 ルールブック

1. ビルドのモード設定確認（必須）

7.2.4.1 ビルドのモード設定確認（必須）

アプリをビルドまたはリリースモードで生成することは、Xcode のビルド設定に依存する。そのため、リリース用のアプリケーションでは正常にリリースモードが設定されていることを確認する必要がある。デバッグモードでアプリをビルドした場合は DEBUG flag が挿入されるため、直接デバッガへのアタッチが可能になる。

環境内の DEBUG flag 設定の確認方法。

- プロジェクトのビルド設定を選択する。
- 「Apple LLVM - Preprocessing」と「Preprocessor Macros」で、「DEBUG」または「DEBUG_MODE」が選択されていないことを確認する。（Objective-C）
- または、「Swift Compiler - Custom flags」セクション / 「Other Swift Flags」において、「-D DEBUG」エントリが存在しないことを確認する。
- Manage Schemes を開き、リリース用のスキームで「Debug executable」オプションが選択されていないことを確認する。

※ Xcode の設定に関するルールのため、サンプルコードはなし。

これに違反する場合、以下の可能性がある。

- デバッガヘアタッチされることで、アプリ情報が漏洩する。

7.3 MSTG-CODE-3

デバッグシンボルはネイティブバイナリから削除されている。

7.3.1 デバッグシンボルの有無

グッドプラクティスとして、コンパイルされたバイナリにはできる限り詳細な情報を与えないようにすべきである。デバッグシンボルのような追加のメタデータが存在すると、コードに関する重要な情報、例えば関数が何をするかについての情報を漏洩する関数名が提供される可能性がある。このメタデータはバイナリを実行するのに必要ではないので、リリースビルトでは破棄するのが安全である。これは、適切なコンパイラ設定を使用することで可能である。テストとして、アプリと一緒に提供されるすべてのバイナリを調査し、デバッグシンボル（少なくとも、コードに関する重要な情報を明らかにするもの）が存在しないことを確認する必要がある。

iOS アプリケーションがコンパイルされると、コンパイラはアプリ内の各バイナリファイル（メインアプリの実行ファイル、フレームワーク、アプリの拡張機能）に対してデバッグシンボルのリストを生成する。これらのシンボルには、クラス名、グローバル変数、メソッド名、関数名が含まれ、それらが定義されている特定のファイルと行番号に対応している。アプリのデバッグビルトでは、デフォルトでコンパイル済みバイナリにデバッグシンボルを配置するが、アプリのリリースビルトでは、配布アプリのサイズを縮小するためにコンパニオンデバッグシンボルファイル（dSYM）に配置する。

参考資料

- [owasp-mastg Finding Debugging Symbols \(MSTG-CODE-3\) Overview](#)

ルールブック

- [リリースビルトではコード情報を出力しないようにする（必須）](#)

7.3.2 静的解析

デバッグシンボルの存在を確認するには、`binutils` の `objdump` か `llvm-objdump` を使って、すべてのアプリのバイナリを確認することができる。

次の例では、TargetApp（iOS のメインアプリの実行ファイル）に対して `objdump` を実行し、`d (debug)` flag でマークされたデバッグシンボルを含むバイナリの典型的な出力を示している。その他のシンボルフラグ文字については、`objdump` の `man` ページを参照する。

```
$ objdump --syms TargetApp

0000000100007dc8 1      d  *UND*  -[ViewController handleSubmitButton:]
000000010000809c 1      d  *UND*  -[ViewController touchesBegan:withEvent:]
0000000100008158 1      d  *UND*  -[ViewController viewDidLoad]
...
000000010000916c 1      d  *UND*  _disable_gdb
00000001000091d8 1      d  *UND*  _detect_injected_dyld
00000001000092a4 1      d  *UND*  _isDebugged
...
```

デバッグシンボルを含まないようにするには、XCode プロジェクトのビルド設定から「Strip Debug Symbols During Copy」を「YES」に設定する。デバッグシンボルを取り除くと、バイナリのサイズが小さくなるだけでなく、リバースエンジニアリングの難易度が上がる。

参考資料

- [owasp-mastg Finding Debugging Symbols \(MSTG-CODE-3\) Static Analysis](#)

7.3.3 動的解析

動的解析は、デバッグ用シンボルの探索には適用されない。

参考資料

- [owasp-mastg Finding Debugging Symbols \(MSTG-CODE-3\) Dynamic Analysis](#)

7.3.4 ルールブック

1. リリースビルトではコード情報を出力しないようにする（必須）

7.3.4.1 リリースビルトではコード情報を出力しないようにする（必須）

コンパイルされたバイナリにはできる限り詳細な情報を与えないようにすべきである。デバッグシンボルのような追加のメタデータが存在すると、コードに関する重要な情報、例えば関数が何をするかについての情報を漏洩する関数名が提供される可能性がある。このメタデータはバイナリを実行するのに必要ではないので、リリースビルトでは破棄するのが安全である。これは、適切なコンパイラ設定を使用することで可能である。

デバッグシンボルを含まないようにするには、XCode プロジェクトのビルド設定から「Strip Debug Symbols During Copy」を「YES」に設定する。デバッグシンボルを取り除くと、バイナリのサイズが小さくなるだけでなく、リバースエンジニアリングの難易度が上がる。

The screenshot shows the OWASP MASTG tool's configuration interface. The 'Deployment' section is selected. Under 'Setting', there is a dropdown for 'Targeted Device Family' set to 'MSTG-CODE-Release'. Below it, the 'Strip Debug Symbols During Copy' option is set to 'Yes'. Other tabs like 'Basic', 'Customized', and 'Combined' are visible at the top.

※概念的なルールのため、サンプルコードはなし。

これに違反する場合、以下の可能性がある。

- コード内のデバッグ情報、行番号、説明的な関数名やメソッド名などの一部のメタデータを漏洩する可能性がある。

7.4 MSTG-CODE-4

デバッグコードおよび開発者支援コード（テストコード、バックドア、隠し設定など）は削除されている。アプリは詳細なエラーやデバッグメッセージをログ出力していない。

7.4.1 ロギング

検証を迅速に行い、エラーをよりよく理解するために、開発者はしばしば、APIからの応答やアプリケーションの進行状況や状態について、冗長なロギングステートメント（`NSLog`、`println`、`print`、`dump`、`debugPrint`を使用）のようなデバッグコードを含める。さらに、開発者がアプリケーションの状態を設定したり、APIからの応答を模擬するために使用する「管理機能」のデバッグコードが存在する可能性がある。リバースエンジニアは、この情報を用いて、アプリケーションで何が起こっているかを簡単に追跡できる。したがって、デバッグ用のコードは、アプリケーションのリリースバージョンから削除する必要がある。

参考資料

- [owasp-mastg Finding Debugging Code and Verbose Error Logging \(MSTG-CODE-4\)](#)

ルールブック

- デバッグコードの処理（必須）
- デバッグコードは、アプリケーションのリリースバージョンから削除する必要がある（必須）

7.4.2 静的解析

ロギングステートメントに対して、以下のような静的解析のアプローチをとることができる。

1. アプリケーションのコードを Xcode にインポートする。
2. 以下の出力関数のコードを検索する。 NSLog, println, print, dump, debugPrint.
3. それらの 1 つを見つけたら、ログに記録するステートメントのマークアップを改善するために、開発者がロギング関数の周りにラッピング関数を使用したかどうかを判断する。もしそうなら、その関数を検索に追加する。
4. 手順 2 と 3 のすべての結果について、マクロまたはデバッグ状態に関連するガードが、リリースビルドでロギングをオフにするように設定されているかどうかを確認する。Objective-C でプリプロセッサマクロを使用できることに注意する。

```
#ifdef DEBUG
    // Debug-only code
#endif
```

Swift でこの動作を有効にするための手順が変更された。スキームで環境変数を設定するか、ターゲットのビルト設定で custom compiler flags として設定する必要がある。Xcode 8 と Swift 3 はこれらの関数をサポートしていないため、以下の関数 (Swift 2.1. release-configuration でビルトされたアプリかどうかを判断できる) は推奨されないことに注意する。

- _isDebugAssertConfiguration
- _isReleaseAssertConfiguration
- _isFastAssertConfiguration

アプリケーションの設定によっては、さらにロギング機能が追加される場合がある。例えば、CocoaLumberjack を使用する場合、静的解析は少し異なる。

「デバッグ管理」コード（組み込み）：storyboards を調査し、アプリケーションがサポートすべき機能とは異なる機能を提供するフローや view-controller があるかどうかを確認する。この機能は、デバッグビューからエラーメッセージの出力、custom stub-response configurations からアプリケーションのファイルシステムまたはリモートサーバ上のファイルに書き込まれるログまで、何でも可能である。

開発者として、アプリケーションのリリース バージョンにデバッグ用のステートメントが存在しないことを確認している限り、アプリケーションのデバッグバージョンにデバッグ用のステートメントを組み込むことは問題にならない。

7.4.2.1 Objective-C ロギングステートメント

Objective-C では、開発者はプリプロセッサーマクロを使用してデバッグコードをフィルタリングすることができる。

```
#ifdef DEBUG
    // Debug-only code
#endif
```

7.4.2.2 Swift ロギングステートメント

Swift 2 (with Xcode 7) では、ターゲットごとに custom compiler flags を設定する必要があり、compiler flags は "-D" で始まらなければならない。そのため、debug flag DMSTG-DEBUG が設定されている場合、以下のような記述が可能である。

```
#if MSTG-DEBUG
    // Debug-only code
#endif
```

Swift 3 (Xcode 8) では、Build settings/Swift compiler - Custom flags で Active Compilation Conditions を設定することができる。Swift 3 では、プリプロセッサの代わりに、定義された条件に基づいて条件付きコンパイルブロックを使用する。

参考資料

- owasp-mastg Finding Debugging Code and Verbose Error Logging (MSTG-CODE-4) Static Analysis

7.4.3 動的解析

開発者がデバッグコードを実行する際に、リリース / デバッグモードに基づく関数ではなく、ターゲットに基づく関数を使用することがあるため、動的解析はシミュレータと端末の両方で実行される必要がある。

1. シミュレータ上でアプリケーションを実行し、アプリの実行中にコンソールに出力される内容を確認する。
2. Mac に端末を接続し、Xcode 経由で端末上でアプリケーションを実行し、アプリの実行中にコンソールに出力される内容を確認する。

その他の「マネージャーベース」のデバッグコードについては、シミュレーターと端末の両方でアプリケーションをクリックして、アプリのプロファイルを事前に設定できる機能、実際のサーバを選択できる機能、API からのレスポンスを選択できる機能などがあるかどうかを確認する必要がある。

参考資料

- owasp-mastg Finding Debugging Code and Verbose Error Logging (MSTG-CODE-4) Dynamic Analysis

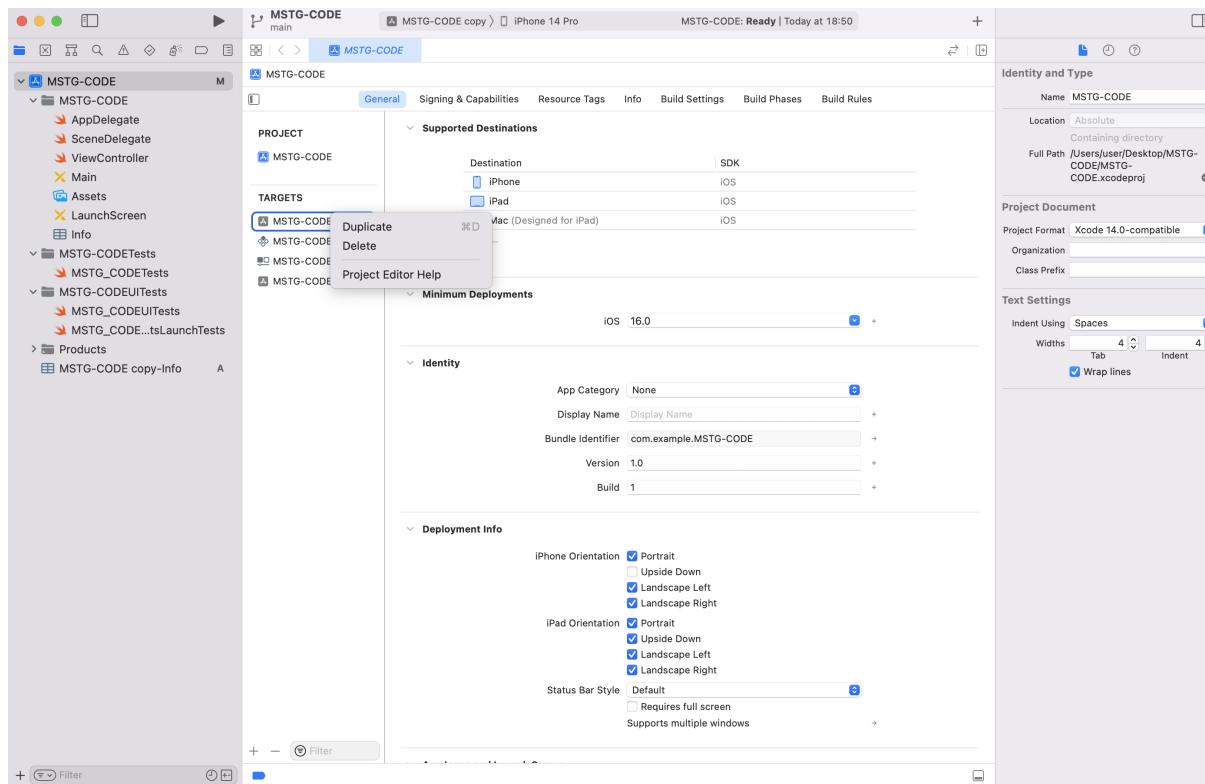
7.4.4 ルールブック

1. デバッグコードの処理（必須）
2. デバッグコードは、アプリケーションのリリースバージョンから削除する必要がある（必須）

7.4.4.1 デバッグコードの処理（必須）

スキームで環境変数を設定する方法

1. ビルドターゲットの追加既存の1つのターゲットを右クリック > duplicate を選択すると新たにコピーが作成される。

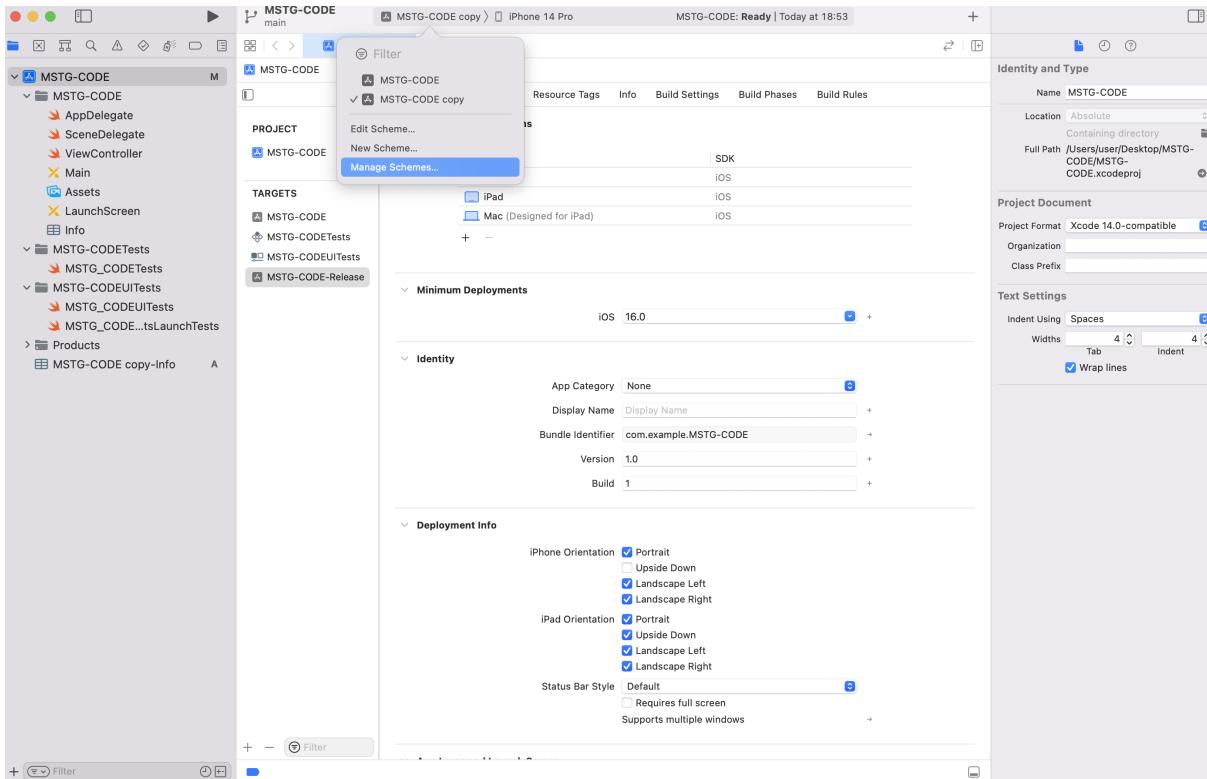


作成できたら名前を XXXXX-Release にしておく。

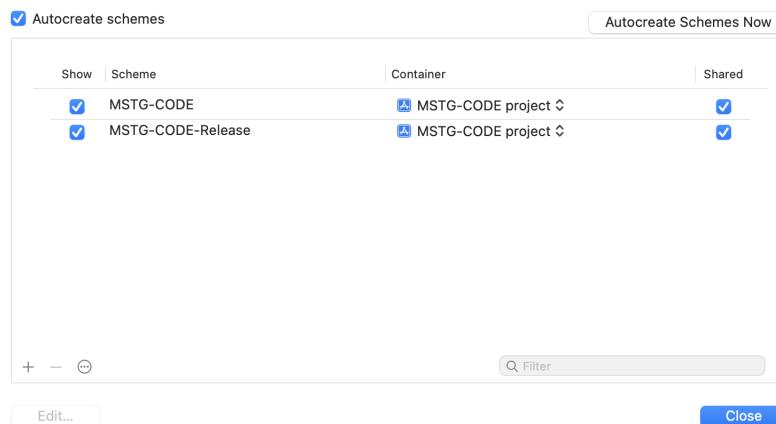


2. ビルドスキームの追加ターゲットを追加すると自動で同じ名前のスキームが自動追加される。

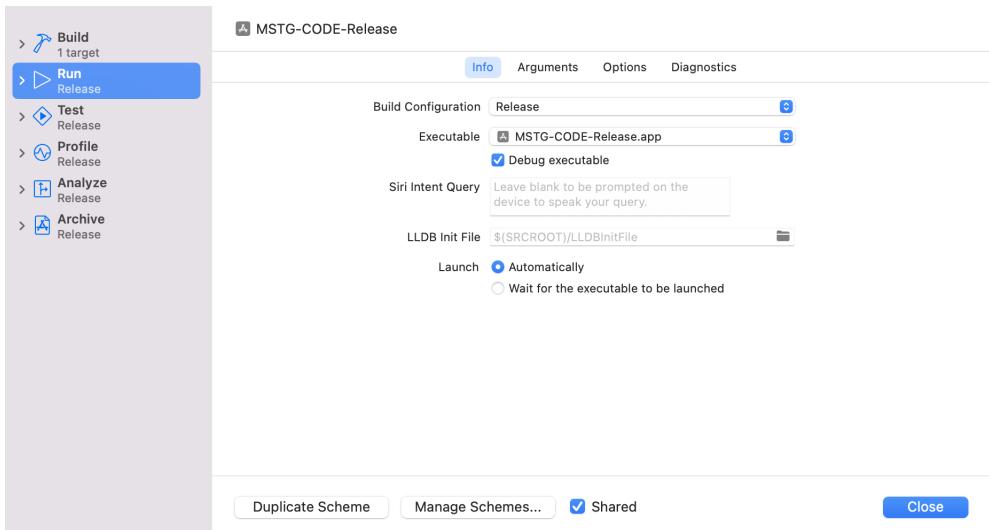
Xcode のトップバーの Run 実行停止ボタンの右にターゲット選択ボタンがある。それをクリックするとターゲットの一覧が表示される。手順 1 で名前を変更したが、スキームの方は変更されていない。



変更するには Manage Schemes... を選択しスキーム管理画面を表示させる。copy をターゲットと同じ、XXXXX-Release にリネームする。



3. Edit Scheme... をクリックする Edit Scheme... をクリックし、左タブのその他 Test、Profile、Analyze、Archive も Release に変えておく。



4. マクロの記述スキームを変えると自動でマクロの条件が true の記述のみが活性化するようになっている。

```
public class SampleClass {

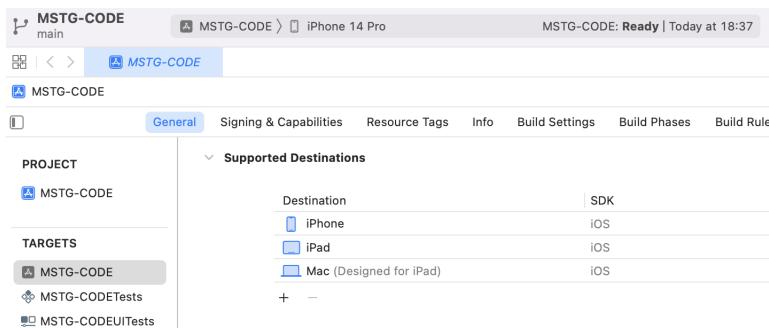
    func execute() {
        // Do any additional setup after loading the view, typically from a nib.
#if PRODUCTION
        print("PRODUCTION Runing")
#endif

#if DEBUG
        print("DEBUG Runing")
#endif

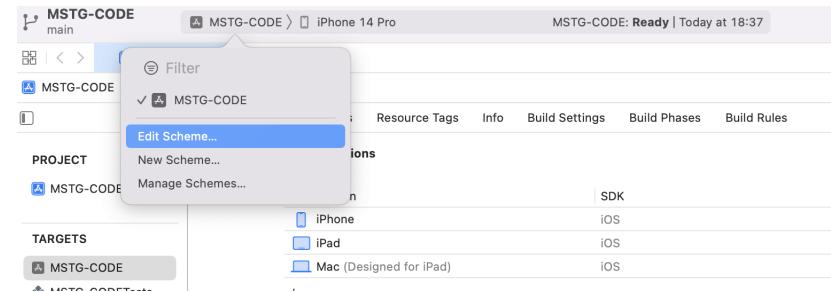
#if STAGING
        print("STAGING Runing")
#endif
    }
}
```

ターゲットのビルト設定で custom compiler flags として設定する方法

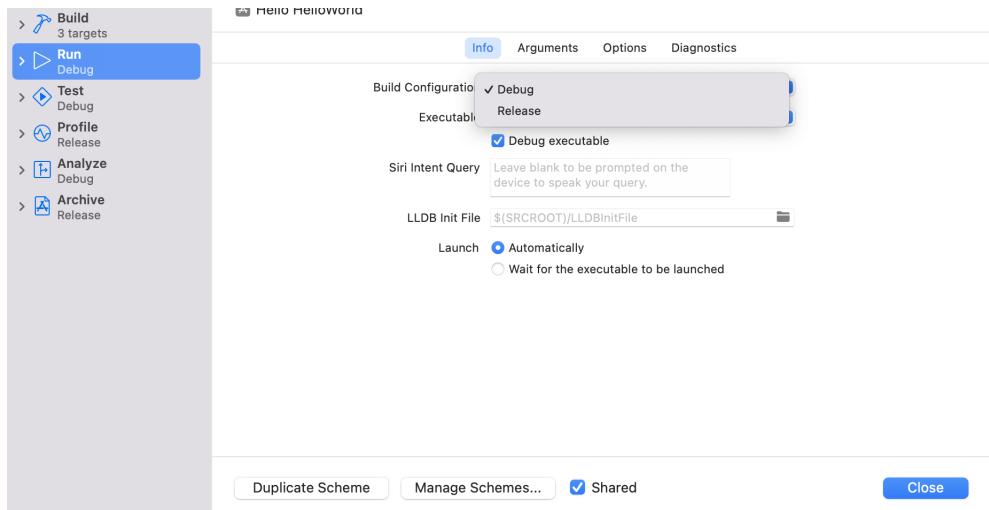
1. プロジェクト名をクリックする



2. Edit Scheme... をクリックする



3. Build Configuration で Debug と Release を切り替える



プリプロセッサの利用方法コンパイル時の flag としてデフォルトでデバッグビルドには DEBUG が付与されている。

```
public class SampleClass {

    func sample() {
#if DEBUG
        // 開発用の接続先
        let server = "api.development.com"
#else
        // 本番用の接続先
        let server = "api.com"
#endif

    }
}
```

※違反した場合または注意しなかった場合の事象として記載可能なものなし。

7.4.4.2 デバッグコードは、アプリケーションのリリースバージョンから削除する必要がある（必須）

ロギング用に追加したデバッグコードは、リリースバージョンを作成する際には削除する必要がある。

これに違反する場合、以下の可能性がある。

- デバッグコードによる情報漏洩を引き起こす可能性がある。

7.5 MSTG-CODE-5

モバイルアプリで使用されるライブラリ、フレームワークなどのすべてのサードパーティコンポーネントを把握し、既知の脆弱性を確認している。

7.5.1 サードパーティライブラリ

iOS アプリケーションでは、サードパーティライブラリを利用することが多く、開発者は問題を解決するために書くコードが少なくなるため、開発を促進することができる。しかし、サードパーティライブラリには、脆弱性、互換性のないライセンス、または悪意のあるコンテンツが含まれている可能性がある。さらに、組織や開発者にとって、ライブラリのリリースを監視し、利用可能なセキュリティパッチを適用するなど、アプリケーションの依存関係を管理することは困難である。

パッケージ管理ツールには、[Swift Package Manager](#)、[Carthage](#)、[CocoaPods](#) の 3 つがあり、幅広く利用されている。

- Swift Package Manager はオープンソースで、Swift 言語に含まれ、Xcode に統合されており（Xcode 11 以降）、Swift、Objective-C、Objective-C++、C、および C++ パッケージをサポートしている。Swift で書かれ、分散化され、プロジェクトの依存関係を文書化し管理するために Package.swift ファイルを使用する。
- Carthage はオープンソースで、Swift と Objective-C のパッケージに使用することができる。Swift で書かれ、分散化されており、Cartfile ファイルを使用してプロジェクトの依存関係をドキュメント化し管理する。
- CocoaPods はオープンソースで、Swift と Objective-C のパッケージに使用することができる。Ruby で書かれており、公開パッケージと非公開パッケージのための集中型パッケージレジストリを利用し、プロジェクトの依存関係を文書化し管理するために Podfile ファイルを使用する。

ライブラリには 2 つの種類が存在する。

- テスト用の OHHTTPStubs のように、実際のプロダクションアプリケーションに含まれない（あるいは含まれるべきではない）ライブラリ。
- Alamofire など、実際のプロダクションアプリケーションの中に詰め込まれているライブラリ。

これらのライブラリは、意図しない効果をもたらす可能性がある。

- ライブラリは脆弱性を含んでいることがあり、その場合、アプリケーションも脆弱になる。その良い例が、AFNetworking のバージョン 2.5.1 で、証明書の検証を無効にするバグが含まれている。この脆弱性

により、攻撃者は、ライブラリを使用して API に接続しているアプリケーションに対して、中間者攻撃を実行することができる。

- ライブラリのメンテナンスができなくなったり、ほとんど使われなくなったりして、脆弱性の報告や修正が行われなくなる。そのため、ライブラリを通じて、アプリケーションに脆弱なコードが含まれる可能性がある。
- ライブラリは LGPL2.1 などのライセンスを使用できる。この場合、アプリケーションの開発者は、アプリケーションを使用してそのソースの洞察を要求するユーザに、ソースコードへのアクセスを提供する必要がある。実際、アプリケーションは、ソースコードを変更して再配布できるようにする必要がある。これにより、アプリケーションの知的財産 (IP) が危険にさらされる可能性がある。

この問題は様々なレベルで発生する可能性があることに注意する必要がある。WebView で JavaScript を使用している場合、JavaScript ライブラリにもこれらの問題が発生する可能性がある。Cordova、React-native、Xamarin アプリのプラグイン/ライブラリも同様である。

参考資料

- [owasp-mastg Checking for Weaknesses in Third Party Libraries \(MSTG-CODE-5\) Overview](#)

ルールブック

- [サードパーティライブラリの使用には注意する（推奨）](#)

7.5.2 サードパーティライブラリの脆弱性の検出

アプリが使用するライブラリに脆弱性がないことを確認するためには、CocoaPods や Carthage によってインストールされた依存関係を確認するのが最適である。

参考資料

- [owasp-mastg Checking for Weaknesses in Third Party Libraries \(MSTG-CODE-5\) Detecting vulnerabilities of third party libraries](#)

ルールブック

- [ライブラリの脆弱性の確認方法（推奨）](#)

7.5.2.1 Swift Package Manager

サードパーティの依存関係を管理するために [Swift Package Manager](#) を使用している場合、サードパーティのライブラリの脆弱性を分析するために以下の手順を実行することができる。

まず、`Package.swift` があるプロジェクトのルートで、次のように入力する。

```
swift build
```

次に、実際に使用されているバージョンを `Package.resolved` ファイルで確認し、与えられたライブラリに既知の脆弱性がないかどうかを調査する。

OWASP Dependency-Check の試験的な Swift Package Manager Analyzer を利用して、すべての依存関係の Common Platform Enumeration (CPE) 命名法および対応する Common Vulnerability and Exposure (CVE) エントリを識別することが可能である。アプリケーションの Package.swift ファイルをスキャンし、以下のコマンドを使用して既知の脆弱なライブラリのレポートを生成する。

```
dependency-check --enableExperimental --out . --scan Package.swift
```

参考資料

- owasp-mastg Checking for Weaknesses in Third Party Libraries (MSTG-CODE-5) Swift Package Manager

7.5.2.2 CocoaPods

サードパーティの依存関係を管理するために CocoaPods を使用する場合、サードパーティのライブラリの脆弱性を分析するために、次の手順を実行することができる。

まず、Podfile が置かれているプロジェクトのルートで、以下のコマンドを実行する。

```
sudo gem install cocoapods
pod install
```

次に、依存関係ツリーが構築されたので、以下のコマンドを実行することで、依存関係とそのバージョンの概要を作成することができる。

```
sudo gem install cocoapods-dependencies
pod dependencies
```

上記の手順の結果を入力として、既知の脆弱性についてさまざまな脆弱性フィードを検索することができる。

備考：

1. 開発者が .podspec ファイルを使用して独自のサポートライブラリの観点からすべての依存関係をパックする場合、この .podspec ファイルは実験的な CocoaPods podspec チェッカーでチェックすることができる。
2. プロジェクトが CocoaPods と Objective-C を組み合わせて使用する場合、SourceClear を使用できる。
3. HTTPS の代わりに HTTP ベースのリンクで CocoaPods を使用すると、依存関係のダウンロード中に中間者攻撃が可能になり、攻撃者がライブラリ（の一部）を他のコンテンツに置き換えることができる可能性がある。したがって、常に HTTPS を使用すること。

OWASP Dependency-Check の試験的な CocoaPods Analyzer を利用して、すべての依存関係の Common Platform Enumeration (CPE) 命名法および対応する Common Vulnerability and Exposure (CVE) エントリを識別することができる。アプリケーションの *.podspec および / または Podfile.lock ファイルをスキャンし、以下のコマンドを使用して既知の脆弱なライブラリのレポートを生成する。

```
dependency-check --enableExperimental --out . --scan Podfile.lock
```

参考資料

- [owasp-mastg Checking for Weaknesses in Third Party Libraries \(MSTG-CODE-5\) CocoaPods](#)

7.5.2.3 Carthage

サードパーティーの依存関係に [Carthage](#) を使用している場合、サードパーティーライブリの脆弱性を分析するために、以下の手順を実行することができる。

まず、Cartfile があるプロジェクトのルートで、次のように入力する。

```
brew install carthage
carthage update --platform iOS
```

次に、実際に使用されているバージョンを Cartfile.solved で確認し、与えられたライブラリに既知の脆弱性がないかどうかを調査する。

備考：この章を書いた時点では、Carthage ベースの依存性分析の自動サポートは、著者の知る限りでは存在しない。少なくとも、この機能は OWASP DependencyCheck ツールにすでに要求されていたが、まだ実装されていない。（[GitHub issue](#) を参照）

参考資料

- [owasp-mastg Checking for Weaknesses in Third Party Libraries \(MSTG-CODE-5\) Carthage](#)

7.5.2.4 使用するライブラリの脆弱性

ライブラリに脆弱性があることが判明した場合、次の理由が適用される。

- ライブラリはアプリケーションと一緒に提供されているか。そして、そのライブラリに脆弱性のパッチが適用されたバージョンがあるかどうかを確認する。もしそうでなければ、その脆弱性が実際にアプリケーションに影響を与えるかどうかを確認する。もしそうであれば、または将来そうなる可能性があるのであれば、同様の機能を提供し、かつ脆弱性のない代替品を探す。
- そのライブラリはアプリケーションに組み込まれていないか。脆弱性が修正されたパッチが存在するかどうか確認する。もしそうでない場合、その脆弱性がビルドプロセスに影響を与えるかどうかを確認する。その脆弱性がビルドの妨げになったり、ビルドパイプラインのセキュリティを弱めたりする可能性はないか？ そして、その脆弱性が修正された代替品を探す必要がある。

フレームワークをリンクライブラリとして手動で追加した場合。

1. xcdeproj ファイルを開き、プロジェクトのプロパティを確認する。
2. **Build Phases** タブを開き、いずれかのライブラリの **Link Binary With Libraries** の項目を確認する。
[MobSF](#) を使用して同様の情報を取得する方法については、以前の章を参照する。

コピペーストしたソースの場合、ヘッダーファイル（Objective-C を使用する場合）、それ以外は Swift ファイルで、既知のライブラリの既知のメソッド名を検索する。

次に、ハイブリッドアプリケーションの場合、[RetireJS](#) で JavaScript の依存関係を確認する必要があることに注意する。同様に Xamarin の場合は、C# の依存関係を確認する必要がある。

最後に、アプリケーションが高リスクのアプリケーションである場合、最終的にライブラリを手動で審査することになる。その場合、ネイティブコードに特定の要件があり、それは、アプリケーション全体に対して MASVS が確立した要件と同様である。その次に、ソフトウェアエンジニアリングのベストプラクティスがすべて適用されているかどうかを確認する必要がある。

参考資料

- owasp-mastg Checking for Weaknesses in Third Party Libraries (MSTG-CODE-5) Carthage

ルールブック

- ハイブリッドアプリケーションの脆弱性確認の注意点（必須）

7.5.3 サードパーティライブラリのライセンス

著作権を侵害していないことを確認するためには、Swift Package Manager、CocoaPods、Carthage によってインストールされた依存関係を確認するのが最適である。

参考資料

- owasp-mastg Checking for Weaknesses in Third Party Libraries (MSTG-CODE-5) Detecting the Licenses Used by the Libraries of the Application

ルールブック

- ライブラリなどのアプリの依存関係の解析方法（必須）

7.5.3.1 Swift Package Manager

アプリケーションのソースが利用可能で、Swift Package Manager を使用している場合、プロジェクトのルートディレクトリで、Package.swift ファイルがある場所で、以下のコードを実行する。

```
swift build
```

これで、プロジェクト内の `/.build/checkouts/` フォルダに、各依存ライブラリのソースがダウンロードされる。ここで、各ライブラリのライセンスは、それぞれのフォルダにある。

参考資料

- owasp-mastg Checking for Weaknesses in Third Party Libraries (MSTG-CODE-5) Swift Package Manager

7.5.3.2 CocoaPods

アプリケーションのソースが入手でき、CocoaPods を使用する場合、以下の手順を実行して、さまざまなライセンスを取得する。まず、Podfile があるプロジェクトのルートで、次のように入力する。

```
sudo gem install cocoapods  
pod install
```

これにより、すべてのライブラリがインストールされた Pods フォルダが作成され、各フォルダ内にインストールされる。これで、各フォルダ内のライセンスファイルを調査することで、各ライブラリのライセンスを確認することができる。

参考資料

- owasp-mastg Checking for Weaknesses in Third Party Libraries (MSTG-CODE-5) CocoaPods

7.5.3.3 Carthage

アプリケーションのソースが入手でき、Carthage を使用する場合、Cartfile があるプロジェクトのルートディレクトリで、以下のコードを実行する。

```
brew install carthage  
carthage update --platform iOS
```

これで、プロジェクト内の Carthage/Checkouts フォルダに、各依存ライブラリのソースがダウンロードされる。ここには、各ライブラリのライセンスがそれぞれのフォルダに存在する。

参考資料

- owasp-mastg Checking for Weaknesses in Third Party Libraries (MSTG-CODE-5) Carthage

7.5.3.4 ライブラリライセンスに関する問題点

アプリの知的財産 (IP) をオープンソースにする必要があるライセンスがライブラリに含まれている場合、同様の機能を提供するために使用できるライブラリの代替品があるかどうかを確認する。

備考：ハイブリッドアプリの場合、使用されているビルドツールを確認する。ほとんどのビルドツールには、使用されているライセンスを検索するためのライセンス列挙プラグインがある。

参考資料

- owasp-mastg Checking for Weaknesses in Third Party Libraries (MSTG-CODE-5) Issues with library licenses

ルールブック

- ライブラリなどのアプリの依存関係の解析方法（必須）
- ライセンスの著作権が順守されているかどうかの検証（必須）

7.5.4 動的解析

この章の動的解析は、実際のライセンス確認と、ソースがない場合にどのライブラリが関係しているかの確認の 2 つのパートから構成されている。

ライセンスの著作権が順守されているかどうかを検証する必要がある。これは、アプリケーションに、サードパーティライブラリのライセンスが要求する著作権に関する記述がある、または EULA セクションを設ける必要があることを意味することが多い。

参考資料

- owasp-mastg Checking for Weaknesses in Third Party Libraries (MSTG-CODE-5) Dynamic Analysis

ルールブック

- ライセンスの著作権が順守されているかどうかの検証（必須）

7.5.4.1 アプリケーションライブラリの一覧

アプリの解析を行う際には、アプリの依存関係（通常、ライブラリやいわゆる iOS フレームワークの形になっている）も解析し、脆弱性が含まれていないことを確認する必要がある。ソースコードがない場合でも、Objection、MobSF、otool などのツールを使って、アプリの依存関係の一部を特定することができる。Objection は、最も正確な結果を提供し、使いやすいので、推奨されるツールである。iOS Bundles を扱うためのモジュールが含まれており、list_bundles と list_frameworks の 2 つのコマンドを提供する。

list_bundles コマンドは、Frameworks に関連しないアプリケーションのバンドルをすべてリストアップする。出力には、実行ファイル名、バンドル ID、ライブラリのバージョン、ライブラリへのパスが含まれる。

```
...itudehacks.DVIA.swiftv2.develop on (iPhone: 13.2.3) [usb] # ios bundles list_bundles
Executable      Bundle                               Version  Path
-----  -----
→
DVIA-v2          com.highaltitudehacks.DVIA.swiftv2.develop      2    ...-1F0C-4DB1-
→8C39-04ACBFFEE7C8/DVIA-v2.app
CoreGlyphs       com.apple.CoreGlyphs                  1    ...m/Library/
→CoreServices/CoreGlyphs.bundle
```

```

list\_frameworks コマンドは、Frameworks を表すアプリケーションのバンドルを全てリストアップする。

```
...itudehacks.DVIA.swiftv2.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

```

(次のページに続く)

(前のページからの続き)

```
↳ Library/Frameworks/IOKit.framework
...
...system/
```

## 参考資料

- owasp-mastg Checking for Weaknesses in Third Party Libraries (MSTG-CODE-5) Dynamic Analysis Listing Application Libraries

## ルールブック

- ライブラリなどのアプリの依存関係の解析方法（必須）

### 7.5.5 ルールブック

- ライブラリの脆弱性の確認方法（推奨）
- ライブラリなどのアプリの依存関係の解析方法（必須）
- ライセンスの著作権が順守されているかどうかの検証（必須）
- ハイブリッドアプリケーションの脆弱性確認の注意点（必須）
- サードパーティライブラリの使用には注意する（推奨）

#### 7.5.5.1 ライブラリの脆弱性の確認方法（推奨）

アプリが使用するライブラリに脆弱性がないことを確認するためには、CocoaPods や Carthage によってインストールされた依存関係を確認するのが最適である。

これに注意しない場合、以下の可能性がある。

- アプリが脆弱性になる可能性がある。

#### 7.5.5.2 ライブラリなどのアプリの依存関係の解析方法（必須）

著作権を侵害していないことを確認する必要がある。ソースコードがある場合は、Swift Package Manager、CocoaPods、Carthage によってインストールされた依存関係を確認するのが最適である。

### Swift Package Manager

アプリケーションのソースが利用可能で、Swift Package Manager を使用している場合、プロジェクトのルートディレクトリで、Package.swift ファイルがある場所で、以下のコードを実行する。

```
swift build
```

これで、プロジェクト内の `/.build/checkouts/` フォルダに、各依存ライブラリのソースがダウンロードされる。ここで、各ライブラリのライセンスは、それぞれのフォルダにある。

## CocoaPods

アプリケーションのソースが入手でき、CocoaPods を使用する場合、以下の手順を実行して、さまざまなライセンスを取得する。まず、Podfile があるプロジェクトのルートで、次のように入力する。

```
sudo gem install cocoapods
pod install
```

これにより、すべてのライブラリがインストールされた Pods フォルダが作成され、各フォルダ内にインストールされる。これで、各フォルダ内のライセンスファイルを調査することで、各ライブラリのライセンスを確認することができる。

## Carthage

アプリケーションのソースが入手でき、Carthage を使用する場合、Cartfile があるプロジェクトのルートディレクトリで、以下のコードを実行する。

```
brew install carthage
carthage update --platform iOS
```

これで、プロジェクト内の Carthage/Checkouts フォルダに、各依存ライブラリのソースがダウンロードされる。ここには、各ライブラリのライセンスがそれぞれのフォルダに存在する。

ソースコードがない場合でも、ツールを使って、アプリの依存関係の一部を特定することができる。

## Objection

Objection は、Frida を利用したランタイム モバイル探索ツールキットであり、ジェイルブレイクを必要とせずにモバイルアプリケーションのセキュリティ体制を評価するのに役立つように構築されており、以下のような特徴がある。

- iOS と Android の両方をサポートする。
- コンテナファイルシステムを検査して操作する。
- SSL ピン留めをバイパスする。
- Keychain をダンプする。
- ダンプやパッチ適用など、メモリ関連のタスクを実行する。
- ヒープ上のオブジェクトを探索して操作する。

## MobSF ( Mobile-Security-Framework-MobSF )

MobSF は、Android/iOS のペントest、マルウェア解析、セキュリティ評価を自動化したオールインワンのフレームワークで、静的解析と動的解析を行うことができる。MobSF は、モバイルアプリのバイナリ（APK、XAPK、IPA、APPX）と zip 形式のソースコードをサポートしている。

## otool

otool は、オブジェクトファイルやライブラリの指定された部分を表示するためのツールである。これは Mach-O ファイルとユニバーサルファイルフォーマットの両方を理解する。

これに違反する場合、以下の可能性がある。

- アプリの知的財産 (IP) をオープンソースにする必要があるライセンスがライブラリに含まれている可能性がある。

#### 7.5.5.3 ライセンスの著作権が順守されているかどうかの検証（必須）

主要なライセンスに共通する特徴として、派生ソフトウェアを頒布する際は、利用元 OSS の「著作権」「ライセンス文」「免責事項」などを表示しなければならない点が挙げられる。具体的に「何を」「どこに」「どのように」表示するかは各ライセンスによって異なる可能性があるため、詳細は個別のライセンスを丁寧に確認する必要がある。

例えば MIT License では下記の記述によって「著作権表示」および「ライセンス文」を、「ソフトウェアのすべての複製または重要な部分に記載する」よう定められている。

The above copyright notice **and** this permission notice shall be included **in all**  
**→****copies or** substantial portions of the Software.

これに違反する場合、以下の可能性がある。

- アプリの知的財産 (IP) をオープンソースにする必要があるライセンスがライブラリに含まれている可能性がある。

#### 7.5.5.4 ハイブリッドアプリケーションの脆弱性確認の注意点（必須）

ハイブリッドアプリケーションの場合、RetireJS で JavaScript の依存関係を確認する必要があることに注意する。

RetireJS は Web サイトにアクセスするだけで、アクセスした Web サイトで利用されている JavaScript ライブラリに既知の脆弱性が存在するか確認してくれる。Retire.js ができることは、「ライブラリに存在する既知の脆弱性の検知」であり、まだ公表されていない脆弱性は見つけることができない。

これに違反する場合、以下の可能性がある。

- JavaScript ライブラリに含まれる既知の脆弱性を察知できない可能性がある。

#### 7.5.5.5 サードパーティライブラリの使用には注意する（推奨）

サードパーティライブラリには以下の欠点があるため、使用する場合は吟味する必要がある。

- ライブラリに含まれる脆弱性。脆弱性が含まれるライブラリを使用すると、ライブラリを通してアプリケーションに不正なコードや脆弱性のあるコードが含まれる可能性があるため注意する。また現時点では脆弱性が発見されていない場合でも今後発見される可能性も存在する。その場合は、脆弱性に対応したバージョンに更新するか、更新バージョンがない場合は使用を控える。
- ライブラリに含まれるライセンス。ライブラリの中には、そのライブラリを使用した場合、使用したアプリのソースコードの展開を求めるライセンスが存在するため注意する。

この問題は、複数のレベルで発生する可能性があることに注意する。webview で JavaScript を使用する場合、JavaScript のライブラリにもこのような問題がある可能性がある。Cordova、React-native、Xamarin アプリのプラグイン/ライブラリも同様である。

※サードパーティライブラリの使用注意に関するルールのため、サンプルコードなし。

これに注意しない場合、以下の可能性がある。

- アプリケーションに不正なコードや脆弱性のあるコードが含まれており、悪用される可能性がある。
- サードパーティライブラリに含まれるライセンスにより、アプリのソースコードの展開を求められる可能性がある。

## 7.6 MSTG-CODE-6

アプリは可能性のある例外を catch し処理している。

### 7.6.1 例外処理

例外は、アプリケーションが異常な状態、あるいは、異常な状態に陥った後によく発生する。例外処理をテストすることは、アプリケーションが例外を処理し、ログ記録メカニズムや UI を通じて機密情報を漏洩することなく、安全な状態に移行することを確認することである。

Objective-C の例外処理は、Swift での例外処理とはまったく異なることに注意する。レガシーの Objective-C コードと Swift コードの両方で書かれているアプリケーションで 2 つのアプローチをブリッジ橋渡しだすことは、問題になることがあります。

参考資料

- owasp-mastg Testing Exception Handling (MSTG-CODE-6) Overview

ルールブック

- 例外処理のテスト目的（推奨）

### 7.6.2 Objective-C の例外処理

Objective-C には、2 種類のエラーがある。

参考資料

- owasp-mastg Testing Exception Handling (MSTG-CODE-6) Exception handling in Objective-C

### 7.6.2.1 NSException

NSException は、プログラミングや低レベルのエラー（例えば、0 による除算や境界外の配列アクセスなど）を処理するために使用される。NSException は raise で発生させるか、または @throw で throw させることができる。catch しない限り、この例外は未処理の例外ハンドラを起動し、このハンドラでステートメントをログに記録できる（ログを記録するとプログラムが停止する）。@try-@catch-block を使用している場合、@catch で例外から復帰することができる。

```
@try {
 //do work here
}

@catch (NSException *e) {
 //recover from exception
}

@finally {
 //cleanup
}
```

NSException を使用すると、メモリ管理の問題が発生することに注意する：try block から finally block にあるアロケーションをクリーンアップする必要がある。NSException オブジェクトを NSError に変換するには、@catch block で NSError をインスタンス化する必要があることに注意する。

### 7.6.2.2 NSError

NSError は、他のすべてのタイプのエラーに使用される。Cocoa フレームワーク API の中には、何か問題が発生したときのために、失敗コールバックでエラーをオブジェクトとして提供するものが存在する。それらを提供しないものは、参照によって NSError オブジェクトへのポインタを渡す。NSError オブジェクトへのポインタを取るメソッドには、成功か失敗かを示す BOOL の戻り値の型を提供することが、良い方法とされる。戻り値の型がある場合、エラーの場合は必ず nil を返すようにする必要がある。NO または nil が返された場合、エラー/失敗の理由を調査することができる。

### 7.6.3 Swift の例外処理

Swift (2 - 5) の例外処理は、かなり異なっている。try-catch block は、NSException を処理するために存在するわけではない。この block は、Error (Swift 3) または ErrorType (Swift 2) プロトコルに準拠したエラーを処理するために使用される。Objective-C と Swift のコードがアプリケーションで組み合わされている場合、困難な場合がある。したがって、両方の言語で書かれたプログラムでは、NSError は NSException よりも推奨される。さらに、エラー処理は Objective-C ではオプトインですが、Swift では throw は明示的に処理する必要がある。error-throwing を変換するには、Apple のドキュメントを参照する。エラーを throw することができるメソッドは throws キーワードを使用する。Result 型は成功または失敗を表し、Result、Swift 5 での Result の使い方、Swift における Result 型の能力、を参照する。Swift でエラーを処理する方法は 4 つ存在する。

#### 参考資料

- owasp-mastg Testing Exception Handling (MSTG-CODE-6) Exception Handling in Swift

## ルールブック

- 例外処理 (`NSError`) 使用時の注意点（必須）
- 例外処理 (`NSError`) 使用時の注意点（必須）
- Swift* の例外処理の注意点（推奨）

### 7.6.3.1 do-catch での例外処理

- 関数からその関数を呼び出すコードにエラーを伝える。この場合、do-catch はなく、実際のエラーを投げる `throw` か、`throw` したメソッドを実行する `try` があるだけである。`try` を含むメソッドには `throws` キーワードも必要である。

```
func dosomething(argumentx:TypeX) throws {
 try functionThatThrows(argumentx: argumentx)
}
```

- do-catch ステートメントでエラーを処理する。次のようなパターンが使用できる。

```
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
 }
}
```

- エラーをオプション値として処理する。

```
let x = try? functionThatThrows()
// In this case the value of x is nil in case of an error.
```

- try! 式を使用して、エラーが発生しないことを保証する。

### 7.6.3.2 Result 型での例外処理

- 一般的なエラーを Result の戻り値として処理する。

```
enum ErrorType: Error {
 case typeOne
 case typeTwo
}

func functionWithResult(param: String?) -> Result<String, ErrorType> {
 guard let value = param else {
 return .failure(.typeOne)
 }
 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)
 }
}
```

- ネットワークや JSON のデコードエラーを Result 型で処理する。

```
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
}
```

(次のページに続く)

(前のページからの続き)

```

}

func getMSTGInfo() {
 guard let url = URL(string: "https://raw.githubusercontent.com/OWASP/owasp-
→mstg/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
 }
 }
 }
}

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 静的解析

ソースコードを見て、アプリケーションが様々なタイプのエラー（IPC通信、リモートサービスの呼び出しなど）をどのように処理しているかを理解する。次の章では、この段階で各言語について確認すべきことの例を挙げる。

### 参考資料

- owasp-mastg Testing Exception Handling (MSTG-CODE-6) Static Analysis

### 7.6.4.1 Objective-C での静的解析

#### 確認点

- アプリケーションは、例外とエラーを処理するために、よく設計され、統一されたスキームを使用している。
- Cocoa フレームワークの例外は正しく処理される。
- try blocks で割り当てられたメモリは、@finally blocks で解放される。
- すべての @throw に対して、呼び出し側のメソッドまたは NSApplication/UIApplication オブジェクトのいずれかのレベルで適切な @catch があり、機密情報をクリーンアップし、場合によっては回復することができる。
- アプリケーションは、UI やログステートメントでエラーを処理する際に機密情報を公開せず、ステートメントはユーザに問題を説明するのに冗長である。
- キーイング材料や認証情報など、リスクの高いアプリケーションの機密情報は、@finally ブロックの実行中に常に削除される。
- raise はめったに使われない。（さらなる警告なしにプログラムを終了させなければならない場合に使用される）
- NSError オブジェクトは機密情報を漏洩するようなデータを含んでいない。

### 参考資料

- owasp-mastg Testing Exception Handling (MSTG-CODE-6) Static Analysis in Objective-C

#### ルールブック

- 例外/エラー処理の適切な実装と確認事項（必須）

#### 7.6.4.2 Swift での静的解析

##### 確認点

- ・アプリケーションが適切に設計され、統一されたスキームでエラーを処理する。
- ・アプリケーションは、UI やログステートメントでエラー処理中に機密情報を公開せず、ステートメントはユーザに問題を説明するのに冗長である。
- ・キーイング材料や認証情報など、リスクの高いアプリケーションの機密情報は、defer blocks の実行中に常にワイプされる。
- ・try! は、前もって適切なガード（try! で呼び出されたメソッドがエラーを投げないことをプログラムで確認すること）をした上で使用する。

##### 参考資料

- owasp-mastg Testing Exception Handling (MSTG-CODE-6) Static Analysis in Swift

##### ルールブック

- ・例外/エラー処理の適切な実装と確認事項（必須）

#### 7.6.4.3 適切なエラー処理

開発者は、いくつかの方法で適切なエラー処理を実装することができる。

- ・アプリケーションが、エラーを処理するために、よく設計され、統一されたスキームを使用していることを確認する。
- ・テストケース「デバッグコードと冗長なエラーログに関する検証」で説明されているように、すべてのログが削除されるか、保護されていることを確認する。
- ・Objective-C で書かれたリスクの高いアプリケーションの場合：簡単に取得できないはずの秘密を削除する例外ハンドラを作成する必要がある。ハンドラは NSSetUncaughtExceptionHandler で設定することができる。
- ・呼び出される throwing メソッドにエラーがないことが確実でない限り、Swift で try を使用することは控える。
- ・Swift のエラーがあまりにも多くの中間メソッドに伝搬しないことを確認する。

##### 参考資料

- owasp-mastg Testing Exception Handling (MSTG-CODE-6) Proper Error Handling

##### ルールブック

- ・例外/エラー処理の適切な実装と確認事項（必須）

## 7.6.5 動的解析

動的解析にはいくつかの手法がある。

- iOS アプリケーションの UI フィールドに想定外の値を入力する。
- カスタム URL スキーム、ペーストボード、その他のアプリ間通信コントロールに、想定外の値や例外を発生させる値を入力して検証する。
- ネットワーク通信やアプリケーションに保存されているファイルを改ざんする。
- Objective-C では、Cycript を使用してメソッドにフックし、呼び出し側が例外を throw する原因となる引数を提供することができる。

ほとんどの場合、アプリケーションはクラッシュしないはずである。その代わりに以下のようにする必要がある。

- エラーから回復するか、ユーザに続行不可能であることを知らせることができる状態になる。
- ユーザに適切な行動を取らせるためのメッセージ（機密情報を漏洩してはいけません）を提供する。
- アプリケーションのロギング機能に情報を残さないようにする。

参考資料

- [owasp-mastg Testing Exception Handling \(MSTG-CODE-6\) Dynamic Testing](#)

## 7.6.6 ルールブック

1. 例外処理のテスト目的（推奨）
2. 例外処理 (*NSError*) 使用時の注意点（必須）
3. 例外処理 (*NSError*) 使用時の注意点（必須）
4. *Swift* の例外処理の注意点（推奨）
5. 例外/エラー処理の適切な実装と確認事項（必須）

### 7.6.6.1 例外処理のテスト目的（推奨）

例外処理をテストの目的は、ログ記録メカニズムや UI を通じて機密情報を漏洩することなく、安全な状態に移行することである。

これに注意しない場合、以下の可能性がある。

- アプリケーションのクラッシュが発生する。
- 機密情報が漏洩する。

### 7.6.6.2 例外処理 (NSError) 使用時の注意点（必須）

NSError は、プログラミングや低レベルのエラー（例えば、0 による除算や 領域外の配列アクセスなど）を処理するために使用される。

try の中に記述した処理の中で例外が発生した場合に、例外が発生する直前の処理までが実行され、その後、catch の中に記述された処理が実行される。そして最後に finally の中に記述された処理が実行される。なお、この finally は、try と catch のどちらが実行されていたとしても常に最後に実行される。

一方で、例外が発生しなかった場合は、try で記述された処理が全て実行され、その後 finally の処理が実行される。catch 内の処理は実行されません。

try-catch 内でわざと例外を発生させる方法は、例外を発生させたいタイミングで下記のコードを実行する。

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

@interface MyException : NSObject {}
- (void)execute;
@end

@implementation MyException {}

- (void)execute {
 // わざと例外処理を発生させる処理
 [[NSError exceptionWithName:@"Exception" reason:@"reason" userInfo:nil] _raise];
}

@end
```

※違反した場合または注意しなかった場合の事象として記載可能なものなし。

### 7.6.6.3 例外処理 (NSError) 使用時の注意点（必須）

NSError は、他のすべてのタイプのエラーに使用される。Cocoa フレームワーク API の中には、何か問題が発生したときのために、失敗コードバックでエラーをオブジェクトとして提供するものが存在する。それらを提供しないものは、参照によって NSError オブジェクトへのポインタを渡す。NSError オブジェクトへのポインタを取るメソッドには、成功か失敗かを示す BOOL の戻り値の型を提供することが、良い方法とされる。戻り値の型がある場合、エラーの場合は必ず nil を返すようにする必要がある。NO または nil が返された場合、エラー/失敗の理由を調査することができる。

初期値を入れずに使ってしまうと、NSError を参照した時に、参照先が適当な値になっているために EXC\_BAD\_ACCESS が発生してしまう。

NSError 生成時に nil を指定して初期化することで回避できる。

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

@interface MyErrorUse : NSObject {}
```

(次のページに続く)

(前のページからの続き)

```

- (void)execute;
@end

@implementation MyErrorUse {}

- (void)execute {
 NSString *path = @"file/pass/";

 // NSError の初期化
 NSError *error = nil;
 [[NSFileManager defaultManager] removeItemAtPath: path error: &error];

 if (error != nil) {
 // エラー処理
 }
}

@end

```

引数 error は NSError 型（後述）の参照ポインタを渡している。渡すときに &error と書くことで、エラーがあった場合にメソッド内部でこの参照が更新され、NSError の実体が代入される。

つまり、ファイル削除中にエラーが起きた場合は、この error の中身を確認すればハンドルできるということであり、逆に言うとエラー処理をしないときは error: には nil を渡すこともできる。

※違反した場合または注意しなかった場合の事象として記載可能なものなし。

#### 7.6.6.4 Swift の例外処理の注意点（推奨）

Swift ( 2 - 5 ) の例外処理は、try-catch block を使用する。

この block は、Error ( Swift 3 ) または ErrorType ( Swift 2 ) プロトコルに準拠したエラーを処理するために使用され　NSException を処理するために存在するわけではない。

Objective-C と Swift のコードが組み合わされているアプリケーションのプログラムでは　NSError は NSException よりも推奨される。さらに、エラー処理は Objective-C ではオプトインですが、Swift では throw は明示的に処理する必要がある。

これに注意しない場合、以下の可能性がある。

- 例外を適切に処理できない可能性がある。

### 7.6.6.5 例外/エラー処理の適切な実装と確認事項（必須）

iOS で例外/エラー処理を実装する場合は、以下内容を確認する必要がある。

#### Objective-C での確認点

- ・ アプリケーションは、例外とエラーを処理するために、よく設計され、統一されたスキームを使用している。
- ・ Cocoa フレームワークの例外は正しく処理される。
- ・ try blocks で割り当てられたメモリは、@finally blocks で解放される。
- ・ すべての @throw に対して、呼び出し側のメソッドまたは NSApplication/UIApplication オブジェクトのいずれかのレベルで適切な @catch があり、機密情報をクリーンアップし、場合によっては回復することができる。
- ・ アプリケーションは、UI やログステートメントでエラーを処理する際に機密情報を公開せず、ステートメントはユーザに問題を説明するのに冗長である。
- ・ キーイング材料や認証情報など、リスクの高いアプリケーションの機密情報は、@finally ブロックの実行中に常に削除される。
- ・ raise はめったに使われない。（さらなる警告なしにプログラムを終了させなければならない場合に使用される）
- ・ NSError オブジェクトは機密情報を漏洩するようなデータを含んでいない。

#### Swift での確認点

- ・ アプリケーションが適切に設計され、統一されたスキームでエラーを処理する。
- ・ アプリケーションは、UI やログステートメントでエラー処理中に機密情報を公開せず、ステートメントはユーザに問題を説明するのに冗長である。
- ・ キーイング材料や認証情報など、リスクの高いアプリケーションの機密情報は、defer blocks の実行中に常にワイプされる。
- ・ try! は、前もって適切なガード（try! で呼び出されたメソッドがエラーを投げないことをプログラムで確認すること）をした上で使用する。

#### 適切なエラー処理を実装方法

- ・ アプリケーションが、エラーを処理するために、よく設計され、統一されたスキームを使用していることを確認する。
- ・ テストケース「デバッグコードと冗長なエラーログに関する検証」で説明されているように、すべてのログが削除されるか、保護されていることを確認する。
- ・ Objective-C で書かれたリスクの高いアプリケーションの場合：簡単に取得できないはずの秘密を削除する例外ハンドラを作成する必要がある。ハンドラは NSSetUncaughtExceptionHandler で設定することができる。

- 呼び出される throwing メソッドにエラーがないことが確実でない限り、Swift で try を使用することは控える。
- Swift のエラーがあまりにも多くの中間メソッドに伝搬しないことを確認する。

try-catch でエラーを処理する方法

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

@interface MyErrorUse : NSObject {}
- (void)tryCatchExample;
@end

@implementation MyErrorUse {}

- (void)tryCatchExample {
 // 実行したい処理
 @try {
 // 例外処理を発生させる
 [[NSError exceptionWithName:@"Exception" reason:@"TestCode" userInfo:nil] raise];
 } @catch (NSError *exception) {
 // 例外が起きると実行される処理
 NSLog(@"%@", exception.name);
 NSLog(@"%@", exception.reason);

 } @finally {
 // try-catch の処理が終わった時に実行したい処理
 NSLog(@"*** finally ***");
 }
}

@end
```

do-catch でエラーを処理する方法

```
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)
```

(次のページに続く)

(前のページからの続き)

```

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
 }
}

```

これに違反する場合、以下の可能性がある。

- アプリケーションのクラッシュが発生する。
- 機密情報が漏洩する。

## 7.7 MSTG-CODE-7

セキュリティコントロールのエラー処理ロジックはデフォルトでアクセスを拒否している。

※ OWASP では問題の事象 (MSTG-CODE-7) について iOS 側での記載がないため、本章の記載を省略。

## 7.8 MSTG-CODE-8

アンマネージドコードでは、メモリはセキュアに割り当て、解放、使用されている。

### 7.8.1 ネイティブコードでのメモリ破損バグ

メモリ破損バグは、ハッカーに人気のある主要なものである。この種のバグは、プログラムが意図しないメモリ位置にアクセスするようなプログラミングエラーに起因する。適切な条件下で、攻撃者はこの挙動を利用して、脆弱なプログラムの実行フローを乗っ取り、任意のコードを実行することができる。この種の脆弱性は、様々な方法で発生する。

メモリ破損を悪用する主な目的は、通常、攻撃者がシェルコードと呼ばれる組み立てられたマシン命令を配置した場所にプログラムフローをリダイレクトすることである。iOS では、データ実行防止機能（その名の通り、データセグメントとして定義されたメモリからの実行を防止する機能）がある。この機能を回避するために、攻撃者は ROP (return-oriented programming) を利用する。このプロセスでは、テキストセグメント内の小さな既存のコードチャンク（ガジェット）を連結し、これらのガジェットが攻撃者にとって有用な機能を実行したり、mprotect を呼び出して攻撃者がシェルコードを格納した場所のメモリ保護設定を変更したりする。

Android アプリは、ほとんどの場合、Java で実装されており、設計上、メモリ破損の問題から本質的に安全である。しかし、JNI ライブラリを利用したネイティブアプリは、この種のバグの影響を受けやすくなっている。同様に、iOS アプリは C/C++ の呼び出しを Obj-C や Swift でラップすることができるため、この種の攻撃を受けやすくなっている。

#### 参考資料

- [owasp-mastg Memory Corruption Bugs \(MSTG-CODE-8\)](#)

### 7.8.1.1 バッファオーバーフロー

特定の操作に対して、割り当てられたメモリの範囲を超えて書き込みを行うプログラミングエラーを指す。攻撃者は、この欠陥を利用して、関数ポインタなど、隣接するメモリにある重要な制御データを上書きすることができます。バッファオーバーフローは、以前はメモリ破壊の最も一般的なタイプの欠陥でしたが、さまざまな要因により、ここ数年はあまり見かけなくなった。特に、安全でない C ライブラリ関数を使用することのリスクについて開発者の間で認識されるようになり、さらに、バッファオーバーフローのバグを捕まえることが比較的簡単になったことが、一般的なベストプラクティスとなっている。しかし、このような不具合がないかどうかをテストする価値はまだある。

#### 参考資料

- [owasp-mastg Memory Corruption Bugs \(MSTG-CODE-8\)](#)

### 7.8.1.2 Out-of-bounds-access

ポインタの演算にバグがあると、ポインタやインデックスが、意図したメモリ構造(バッファやリストなど)の境界を超えた位置を参照することがある。アプリが境界外のアドレスに書き込もうとすると、クラッシュや意図しない動作が発生する。攻撃者が対象のオフセットを制御し、書き込まれた内容をある程度操作できれば、コード実行の悪用が可能である可能性が高い。

#### 参考資料

- [owasp-mastg Memory Corruption Bugs \(MSTG-CODE-8\)](#)

### 7.8.1.3 ダングリングポインタ

これは、あるメモリ位置への参照を持つオブジェクトが削除または解放されたにもかかわらず、オブジェクトポインタがリセットされない場合に発生する。プログラムが後でダングリングポインタを使用して、既に割り当て解除されたオブジェクトの仮想関数を呼び出すと、元の vtable ポインタを上書きして実行を乗っ取ることが可能である。また、ダングリングポインタから参照されるオブジェクト変数や他のメモリ構造の読み書きが可能である。

#### 参考資料

- [owasp-mastg Memory Corruption Bugs \(MSTG-CODE-8\)](#)

#### 7.8.1.4 Use-after-free

これは、解放された(割り当て解除された)メモリを参照するダングリングポインタの特殊なケースを指す。メモリアドレスがクリアされると、その場所を参照していたポインタはすべて無効となり、メモリマネージャはそのアドレスを使用可能なメモリのプールに戻すことになる。このメモリアドレスが再割り当てされると、元のポインタにアクセスすると、新しく割り当てられたメモリに含まれるデータが読み取られたり書き込まれたりする。これは通常、データの破損や未定義の動作につながるが、巧妙な攻撃者は、命令ポインタの制御を活用するために適切なメモリ位置を設定することができる。

参考資料

- [owasp-mastg Memory Corruption Bugs \(MSTG-CODE-8\)](#)

#### 7.8.1.5 整数オーバーフロー

演算結果がプログラマが定義した整数型の最大値を超える場合、整数型の最大値に値が「回り込む」ことになり、必然的に小さな値が格納されることになる。逆に、演算結果が整数型の最小値より小さい場合、結果が予想より大きくなる整数型アンダーフローが発生する。特定の整数オーバーフロー/アンダーフローのバグが悪用可能かどうかは、整数の使われ方によって異なる。例えば、整数型がバッファの長さを表す場合、バッファオーバーフローの脆弱性が発生する可能性がある。

参考資料

- [owasp-mastg Memory Corruption Bugs \(MSTG-CODE-8\)](#)

#### 7.8.1.6 フォーマット文字列の脆弱性

C 言語の printf 関数の format string パラメータに未チェックのユーザ入力が渡されると、攻撃者は '%c' や '%n'などのフォーマットトークンを注入してメモリにアクセスする可能性がある。フォーマット文字列のバグは、その柔軟性から悪用するのに便利である。文字列の書式設定結果を出力してしまうと、ASLR などの保護機能を回避して、任意のタイミングでメモリの読み書きができるようになる。

参考資料

- [owasp-mastg Memory Corruption Bugs \(MSTG-CODE-8\)](#)

### 7.8.2 バッファと整数のオーバーフロー

以下のコードは、バッファオーバーフローの脆弱性が発生する条件の簡単な例を示す。

```
void copyData(char *userId) {
 char smallBuffer[10]; // size of 10
 strcpy(smallBuffer, userId);
}
```

バッファオーバーフローの可能性を特定するには、安全でない文字列関数(strcpy、strcat、その他「str」接頭辞で始まる関数など)の使用や、ユーザ入力を限られたサイズのバッファにコピーするなどの潜在的に脆弱

なプログラミング構造を確認する。安全でない文字列関数の red flags と考えられるのは、以下のようなものである。

- strcat
- strcpy
- strncat
- strlcat
- strncpy
- strlcpy
- sprintf
- snprintf
- gets

また、「for」または「while」ループとして実装されたコピー操作のインスタンスを探し、長さチェックが正しく実行されていることを確認する。

以下のベストプラクティスが守られていることを確認する。

- 配列のインデックス付けやバッファ長の計算など、セキュリティ上重要な操作に整数変数を使用する場合は、符号なし整数型が使用されていることを確認し、整数の折り返しの可能性を防ぐために前提条件テストを実行する。
- アプリは、strcpy、その他「str」という接頭辞で始まるほとんどの関数、sprint、vsprintf、getsなどの安全でない文字列関数を使用していないこと。
- アプリに C++ コードが含まれる場合、ANSI C++ 文字列クラスを使用する。
- memcpy の場合、ターゲットバッファが少なくともソースと同じサイズであること、両バッファが重複していないことを確認すること。
- Objective-C で記述された iOS アプリは NSString クラスを使用する。iOS 上の C アプリでは、文字列の Core Foundation 表現である CFString を使用する必要がある。
- 信頼できないデータがフォーマット文字列に連結されることはない。

#### 参考資料

- [owasp-mastg Memory Corruption Bugs \(MSTG-CODE-8\) Buffer and Integer Overflows](#)

#### ルールブック

- バッファオーバーフローを引き起こす安全でない文字列関数を使用しない（必須）
- バッファオーバーフローのベストプラクティス（必須）

### 7.8.2.1 静的解析

低レベルコードの静的コード解析は、それだけで 1 冊の本が完成するほど複雑なトピックである。RATS のような自動化されたツールと、限られた手動での検査作業を組み合わせれば、通常、低い位置にある果実を特定するのに十分である。しかし、メモリ破損は複雑な原因によって引き起こされることがよくある。例えば、use-after-free バグは、すぐには明らかにならない複雑で直感に反するレースコンディションの結果である可能性がある。一般に、見落とされているコードの欠陥の深い部分に起因するバグは、動的解析またはプログラムを深く理解するために時間を投資するテスターによって発見される。

#### 参考資料

- [owasp-mastg Memory Corruption Bugs \(MSTG-CODE-8\) Static Analysis](#)

### 7.8.2.2 動的解析

メモリ破壊のバグは、入力ファジングによって発見するのが最適である。自動化されたブラックボックスソフトウェアテストの手法で、不正なデータをアプリケーションに継続的に送信し、脆弱性の可能性がないか調査する。このプロセスでは、アプリケーションの誤動作やクラッシュがないかどうかが監視される。クラッシュが発生した場合、(少なくともセキュリティテスト実施者にとっては) クラッシュを発生させた条件から、攻略可能なセキュリティ上の不具合が明らかになることが期待される。

ファズテストの技術やスクリプト(しばしば「ファザー」と呼ばれる)は、通常、半ば正確な方法で、構造化された入力の複数のインスタンスを生成する。基本的に、生成された値や引数は、少なくとも部分的にはターゲットアプリケーションに受け入れられるが、無効な要素も含まれており、潜在的に入力処理の欠陥や予期しないプログラムの動作を誘発する可能性がある。優れたファザーは、可能性のあるプログラム実行パスの相当量を公開する(すなわち、高いカバレッジの出力)。入力は、ゼロから生成する方法(生成ベース)と、既知の有効な入力データを変異させて生成する方法(変異ベース)の 2 種類がある。

ファジングの詳細については、[OWASP Fuzzing Guide](#) を参照する。

#### 参考資料

- [owasp-mastg Memory Corruption Bugs \(MSTG-CODE-8\) Dynamic Analysis](#)

### 7.8.3 Objective-C/Swift コードでのメモリ破損バグ

iOS アプリケーションには、メモリ破損のバグに陥る様々な原因がある。まず、一般的なメモリ破壊のバグの章で述べたような、ネイティブコードの問題点がある。次に、Objective-C と Swift の両方で、実際に問題を引き起こす可能性のあるネイティブコードをラップするために、様々な安全でない操作が存在する。最後に、Swift と Objective-C の両方の実装は、もはや使用されていないオブジェクトを保持するために、メモリリークを引き起こす可能性がある。

#### 参考資料

- [owasp-mastg Memory Corruption Bugs \(MSTG-CODE-8\)](#)

### 7.8.3.1 静的解析

ネイティブコードがある場合：一般的なメモリ破壊の章で、指定された問題がないか確認する。ネイティブコードは、コンパイル時に発見するのが少し難しい。もしソースがあれば、C ファイルは .c ソースファイルと .h ヘッダーファイル、C++ は .cpp ファイルと .h ファイルを使用していることがわかる。これは、Swift と Objective-C の .swift と .m のソースファイルとは少し異なっている。これらのファイルは、ソースの一部であったり、サードパーティのライブラリの一部であったり、フレームワークとして登録され、Carthage、Swift Package Manager、Cocoapods などの様々なツールを通してインポートされることがある。

プロジェクト内のマネージドコード（Objective-C / Swift）については、以下の項目を確認する。

- doubleFree 問題：与えられた領域に対して、free が一度ではなく二度呼ばれる場合。
- サイクルの保持：メモリ内に材料を保持するコンポーネント同士の強い参照によって、循環的な依存関係を確認する。
- UnsafePointer のインスタンスを使用すると、間違って管理される可能性があり、様々なメモリ破壊の問題を許すことになる。
- Unmanaged によってオブジェクトへの参照カウントを手動で管理しようとすると、誤ったカウンタ番号になり、リリースが遅すぎたり早すぎたりすることになる。

このテーマについては、[Realm academy](#) で素晴らしい講演が行われ、実際に何が起こっているのかを見るための素晴らしいチュートリアルが Ray Wenderlich によって提供されている。

Swift 5 では、full blocks の割り当て解除のみ可能であるため、プレイグラウンドが少し変更されたことに注意する。

#### 参考資料

- [owasp-mastg Memory Corruption Bugs \(MSTG-CODE-8\) Static Analysis](#)

#### ルールブック

- [Objective-C / Swift での確認点（推奨）](#)

### 7.8.3.2 動的解析

Xcode 8 で導入された Debug Memory グラフや、Xcode の Allocations and Leaks インストルメントなど、Xcode 内のメモリバグを特定するのに役立つ様々なツールが提供されている。

次に、アプリケーションを検証しながら、Xcode で NSAutoreleaseFreedObjectCheckEnabled 、 NSZombieEnabled 、 NSDebugEnabled を有効にすると、メモリの解放が早すぎるか遅すぎるかを確認することができる。

メモリ管理をする上で役に立つ、よく書かれた様々な解説がある。これらは、本章の参考資料を確認すること。

#### 参考資料

- [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-e84f097e>

## 7.8.4 ルールブック

1. バッファオーバーフローを引き起こす安全でない文字列関数を使用しない（必須）
2. バッファオーバーフローのベストプラクティス（必須）
3. *Objective-C / Swift* での確認点（推奨）

### 7.8.4.1 バッファオーバーフローを引き起こす安全でない文字列関数を使用しない（必須）

バッファオーバーフローを引き起こす安全でない文字列関数として以下の関数が存在し、これらの未然に使用を控える必要がある。

- strcat
- strcpy
- strncat
- strlcat
- strncpy
- strlcpy
- sprintf
- snprintf
- gets

※非推奨なルールのため、サンプルコードなし。

これに違反する場合、以下の可能性がある。

- バッファオーバーフローを引き起こす可能性がある。

### 7.8.4.2 バッファオーバーフローのベストプラクティス（必須）

バッファオーバーフローを引き起こさるために以下のことを確認する。

- 配列のインデックス付けやバッファ長の計算など、セキュリティ上重要な操作に整数変数を使用する場合は、符号なし整数型が使用されていることを確認し、整数の折り返しの可能性を防ぐために前提条件テストを実行する。

- アプリは、strcpy、その他「str」という接頭辞で始まるほとんどの関数、sprint、vsprintf、getsなどの安全でない文字列関数を使用していないこと。
- アプリに C++ コードが含まれる場合、ANSI C++ 文字列クラスを使用する。
- memcpy の場合、ターゲットバッファが少なくともソースと同じサイズであること、両バッファが重複していないことを確認すること。
- 信頼できないデータがフォーマット文字列に連結されることはない。

これに注意しない場合、以下の可能性がある。

- バッファオーバーフローを引き起こす可能性がある。

#### 7.8.4.3 Objective-C / Swift での確認点（推奨）

プロジェクト内のマネージドコード（Objective-C / Swift）については、以下の項目を確認する。

- doubleFree 問題：与えられた領域に対して、free が一度ではなく二度呼ばれる場合。
- サイクルの保持：メモリ内に材料を保持するコンポーネント同士の強い参照によって、循環的な依存関係を確認する。
- UnsafePointer のインスタンスを使用すると、間違って管理される可能性があり、様々なメモリ破壊の問題を許すことになる。
- Unmanaged によってオブジェクトへの参照カウントを手動で管理しようとすると、誤ったカウンタ番号になり、リリースが遅すぎたり早すぎたりすることになる。

このテーマについては、Realm academy で素晴らしい講演が行われ、実際に何が起こっているのかを見るための素晴らしいチュートリアルが Ray Wenderlich によって提供されている。

Swift 5 では、full blocks の割り当て解除のみ可能であるため、プレイグラウンドが少し変更されたことに注意する。

これに注意しない場合、以下の可能性がある。

- メモリ破損バグを引き起こす可能性がある。

## 7.9 MSTG-CODE-9

バイトコードの軽量化、スタック保護、PIE サポート、自動参照カウントなどツールチェーンにより提供されるフリーのセキュリティ機能が有効化されている。

### 7.9.1 バイナリ保護機能の利用

バイナリ保護機能の存在を検出するために使用されるテストは、アプリケーションの開発に使用される言語に大きく依存する。

Xcode はデフォルトですべてのバイナリ保護機能を有効にするが、古いアプリケーションの場合はこれを確認したり、compiler flags の設定ミスを確認することが適切な場合がある。

以下の機能が適用される。

- PIE ( Position Independent Executable )
  - PIE は、実行形式バイナリ（Mach-O type MH\_EXECUTE）に適用される。
  - ただし、ライブラリ（Mach-O type MH\_DYLIB）には適用されない。
- メモリ管理
  - 純粋な Objective-C、Swift、ハイブリッドバイナリのいずれも、ARC（自動参照カウント）を有効にする必要がある。
  - C/C++ ライブラリについては、開発者の責任において、適切な手動メモリ管理を行う必要がある。  
「[ネイティブコードでのメモリ破損バグ](#)」を参照する。
- スタックスマッシングプロテクション
  - 純粋な Objective-C バイナリの場合、これは常に有効であるべきである。Swift はメモリセーフに設計されているので、もしライブラリが純粋に Swift で書かれていて、stack canaries が有効にならなければ、そのリスクは最小になる。

より詳細に知る：

- OS X ABI Mach-O ファイルフォーマットリファレンス
- iOS のバイナリ保護について
- iOS および iPadOS におけるランタイムプロセスのセキュリティ
- Mach-O プログラミングトピックス - 位置依存のないコード

これらの保護機能の存在を検出するための試験は、アプリケーションを開発するために使用される言語に大きく依存する。例えば、stack canaries の存在を検出するための既存の技術は、純粋な Swift アプリでは機能しない。

#### 参考資料

- [owasp-mastg Make Sure That Free Security Features Are Activated \(MSTG-CODE-9\) Overview](#)

#### ルールブック

- [バイナリ保護機能の利用の確認（推奨）](#)

### 7.9.1.1 Xcode のプロジェクト設定

#### スタックカナリアの保護

iOS アプリケーションでスタックカナリア保護を有効にするための手順。

1. Xcode の "Targets" セクションでターゲットを選択し、"Build Settings" タブをクリックしてターゲットの設定を表示する。
2. "Other C Flags" セクションで "-fstack-protector-all" オプションが選択されていることを確認する。
3. PIE ( Position Independent Executables ) サポートが有効になっていることを確認する。

#### PIE の保護

iOS アプリケーションを PIE としてビルドする手順。

1. Xcode の "Targets" セクションでターゲットを選択し、"Build Settings" タブをクリックして、ターゲットの設定を表示する。
2. iOS Deployment Target を iOS 4.3 以降に設定する。
3. "Generate Position-Dependent Code" ( セクション "Apple Clang - Code Generation" ) がデフォルト値 ( "NO" ) に設定されていることを確認する。
4. "Generate Position-Dependent Executable" ( セクション "Linking" ) がデフォルト値 ( "NO" ) に設定されていることを確認する。

#### ARC の保護

ARC は Swift アプリでは swiftc コンパイラによって自動的に有効化される。しかし、Objective-C アプリの場合は、次の手順で有効になっていることを確認する必要がある。

1. Xcode の "Targets" セクションでターゲットを選択し、"Build Settings" タブをクリックしてターゲットの設定を表示する。
2. "Objective-C Automatic Reference Counting" がデフォルト値 ( "YES" ) に設定されていることを確認する。

技術的な Q&A QA1788 Position Independent Executable のビルドを参照する。

#### 参考資料

- [owasp-mastg Make Sure That Free Security Features Are Activated \(MSTG-CODE-9\) Xcode Project Settings](#)

## 7.9.2 静的解析

`otool` を使用すると、上記のバイナリ保護機能を確認することができる。これらの例では、すべての機能が有効になっている。

- 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 NOUNDEF 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 NOUNDEF DYLDLINK TWOLEVEL
WEAK_DEFINES BINDS_TO_WEAK PIE
```

出力は、PIE の Mach-O flag が設定されていることを示す。このチェックは、Objective-C, Swift およびハイブリッドアプリのすべてに適用されるが、メインの実行可能ファイルにのみ適用される。

- 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
```

上記の出力では、`__stack_chk_fail` の存在は、スタックカナリアが使用されていることを示す。この確認は純粋な Objective-C アプリとハイブリッドアプリに適用できるが、純粋な Swift アプリには必ずしも適用できない。(つまり Swift は設計上メモリセーフであるため、無効と表示されても問題ない)

- 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
```

(次のページに続く)

(前のページからの続き)

[SNIP]

このチェックは、自動的に有効になる純粋な Swift アプリを含む、すべての場合に適用される。

#### 参考資料

- owasp-mastg Make Sure That Free Security Features Are Activated (MSTG-CODE-9) Static Analysis

### 7.9.3 動的解析

これらのチェックは、[Objection](#) を使って動的に実行することができる。以下はその一例である。

| 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                                                              |         |           |       |      |        |            | -   |
| ...                                                                  |         |           |       |      |        |            |     |

#### 参考資料

- 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 ) とは、主記憶装置のどこかに配置されると、その絶対アドレスに関係なく正しく実行されるコードのことである。PIC は共有ライブラリによく使われ、同じライブラリコードを各プログラムのアドレス空間の中で、他の使用中のメモリ（例えば他の共有ライブラリ）と重ならない位置にロードできるようとする。

PIE ( Position Independent Executable ) は、すべて PIC で作られた実行バイナリである。PIE バイナリは、実行ファイルのベースやスタック、ヒープ、ライブラリの位置など、プロセスの重要なデータ領域のアドレス空間の位置をランダムに配置する ASLR ( アドレス空間レイアウトランダム化 ) を有効にするために使用される。

#### 参考資料

- owasp-mastg Make Sure That Free Security Features Are Activated (MSTG-CODE-9) Position Independent Code

### 7.9.3.2 メモリ管理

#### Automatic Reference Counting

ARC ( Automatic Reference Counting ) は、Objective-C と Swift 専用の Clang コンパイラのメモリ管理機能である。ARC は、クラスのインスタンスが不要になったときに、そのインスタンスが使用しているメモリを自動的に解放する。ARC は、実行時に非同期にオブジェクトを解放するバックグラウンドプロセスがない点で、トレースガベージコレクションとは異なる。

トレースガベージコレクションとは異なり、ARC は参照サイクルを自動的には処理しない。つまり、あるオブジェクトへの「強い」参照がある限り、そのオブジェクトは解放されないということである。強い相互参照は、それに応じてデッドロックやメモリリークを発生させる可能性がある。弱い参照を使ってサイクルを断ち切るのは、開発者次第である。ガベージコレクションとの違いについては、こちらで詳しく説明している。

#### Garbage Collection

Garbage Collection ( GC ) は、Java/Kotlin/Dart などの一部の言語が持つ自動メモリ管理機能である。ガベージコレクタは、プログラムによって割り当てられたが、すでに参照されていないメモリ（ガベージとも呼ばれる）を回収しようとする。Android ランタイム ( ART ) は、GC の改良版を使用している。ARC との違いについては、こちらで詳しく説明している。

**手動メモリ管理** ARC や GC が適用されない C/C++ で書かれたネイティブライブラリでは、通常、手動メモリ管理が必要である。開発者は、適切なメモリ管理を行う責任がある。手動メモリ管理は、不適切に使用された場合、プログラムにいくつかの主要なクラスのバグ、特にメモリ安全性の侵害やメモリリークを引き起こすことが知られている。

詳細は、「[ネイティブコードでのメモリ破損バグ](#)」を参照する。

#### 参考資料

- owasp-mastg Make Sure That Free Security Features Are Activated (MSTG-CODE-9) Memory management

### 7.9.3.3 スタック破壊保護

スタックカナリアは、リターンポインタの直前のスタックに隠された整数値を保存することで、スタックバッファオーバーフロー攻撃を防ぐのに役立つ。この値は、関数の return 文が実行される前に検証される。バッファオーバーフロー攻撃は、しばしばリターンポインタを上書きし、プログラムフローを乗っ取るために、メモリ領域を上書きする。スタックカナリアが有効な場合、それらも上書きされ、CPU はメモリが改ざんされたことを知ることになる。

スタックバッファオーバーフローは、バッファオーバーフロー（またはバッファオーバーラン）として知られる、より一般的なプログラミングの脆弱性の一種である。スタックには、すべてのアクティブな関数呼び出しの戻りアドレスが含まれているため、スタック上のバッファのオーバーフローは、ヒープ上のバッファのオーバーフローよりも、プログラムの実行を狂わせる可能性が高い。

#### 参考資料

- owasp-mastg Make Sure That Free Security Features Are Activated (MSTG-CODE-9) Stack Smashing Protection

## 7.9.4 ルールブック

### 1. バイナリ保護機能の利用の確認（推奨）

#### 7.9.4.1 バイナリ保護機能の利用の確認（推奨）

Xcode はデフォルトですべてのバイナリ保護機能を有効にするが、古いアプリケーションの場合はこれを確認したり、compiler flags の設定ミスを確認する必要がある。

確認を行うため以下の機能が適用される。

- **PIE ( Position Independent Executable )**

- PIE は、実行形式バイナリ（Mach-O type MH\_EXECUTE）に適用される。
- ただし、ライブラリ（Mach-O type MH\_DYLIB）には適用されない。

- **メモリ管理**

- 純粋な Objective-C、Swift、ハイブリッドバイナリのいずれも、ARC（自動参照カウント）を有効にする必要がある。
- C/C++ ライブラリについては、開発者の責任において、適切な手動メモリ管理を参照する。

- **スタック破壊保護**：純粋な Objective-C バイナリの場合、これは常に有効であるべきである。Swift はメモリセーフに設計されているので、もしライブラリが純粋に Swift で書かれていて、stack canaries が有効になっていなければ、そのリスクは最小になる。

詳細はこちら

- OS X ABI Mach-O ファイルフォーマットリファレンス
- iOS のバイナリ保護について
- iOS および iPadOS におけるランタイムプロセスのセキュリティ
- Mach-O プログラミングトピックス - 位置依存のないコード

これらの保護機能の存在を検出するための試験は、アプリケーションを開発するために使用される言語に大きく依存する。例えば、stack canaries の存在を検出するための既存の技術は、純粋な Swift アプリでは機能しない。

これに注意しない場合、以下の可能性がある。

- バイナリ保護機能が利用できない可能性がある。

8

更新履歷

2023-04-01

初版