MSTG - Teste de aplicações Android e iOS

Departamento de Informática, Universidade do Minho

Resumo Palavras-chave: MSTG · Android · iOS.

Introdução

Android

1 Arquitetura de Segurança do Android

Figura 1. Arquitetura

Android é uma plataforma Open Source baseada em Linux e desenvolvida pelo Google, que serve como um sistema operacional móvel. Como é possível observar na figura **??**, esta é composta por várias camadas diferentes. Cada camada define interfaces e serviços específicos.

Uma das particularidas das aplicações Android, é o facto destas não terem acesso direto aos recursos de hardware e cada aplicação ser executada na sua própria sandbox, o que permite assim um controlo preciso sobre recursos e aplicativos, por exemplo, quando uma aplicação falha, esta não afetará as restantes aplicações em execução no dispositivo. Outra das particularidades do Android, é que o tempo de execução deste controla o número máximo de recursos do sistema alocados nas aplicações, impedindo, deste modo, que qualquer aplicação monopolize muitos recursos.

2 Criptografia de dispositivos Android

Este sistema operativo, suporta criptografia de dispositivo desde da versão 2.3.4 (API nível 10), e que desde então tem passado por grandes mudanças. Por imposição do Google, todos os dispositivos com Android 6.0 (nível 23 da API) ou superior passaram a ter que suportar criptografia de armazenamento, o que em alguns casos veio afetar significativamente o seu desempenho. Criptografia esta que se encontra dividida em três grupos:

Full-Disk Encryption Versões 5.0 (nível de API 21) e superiores oferecem suporte à criptografia de disco completo. Esta criptografia usa uma única chave protegida pela senha do dispositivo dos utilizadores para encriptar e desencriptar a partição de dados do utilizador. Este tipo de criptografia, nos tempos atuais, é considerada obsoleta, sendo que, a criptografia baseada em arquivo deve ser, preferencialmente, usada sempre que possível, visto que, a criptografia de disco completo apresenta desvantagens, como não poder receber chamadas ou não ter alarmes operacionais após uma reinicialização se o utilizador não digitar a senha para desbloquear.

File-Based Encryption A versão 7.0 (API nível 24) já suporta criptografia baseada em arquivo. Esta permite que arquivos diferentes sejam encriptados com chaves diferentes, para que possam ser desencriptados independentemente. Fazendo com que os dispositivos que suportam este tipo de criptografia também ofereçam suporte à Inicialização Direta. Esta permite que o dispositivo tenha acesso a recursos como alarmes ou serviços de acessibilidade, mesmo que o utilizador não tenha desbloqueado o dispositivo.

Adiantum A terceira criptografia presente neste sistema operativo, é o Adiantum, em substituição ao AES, que é usado na maioria dos dispositivos Android modernos para criptografia de armazenamento. Na verdade, o AES tornou-se um algoritmo tão amplamente usado que as implementações mais recentes do processador têm um conjunto dedicado de instruções para fornecer operações de

encriptação e desencriptação aceleradas por hardware. Porém, nem todos os dispositivos são capazes de usar o AES para criptografia de armazenamento em tempo hábil.

Especialmente dispositivos de última geração com Android Go, pelo facto de usarem processadores low-end, como o ARM Cortex-A7, que não possuem AES acelerado por hardware. Daí que surgiu o Adiantum, que é uma construção cifrada projetada por Paul Crowley e Eric Biggers no Google para preencher a lacuna desse conjunto de dispositivos que não são capazes de executar o AES pelo menos a 50 MiB/s.

O Adiantum depende apenas de adições, rotações e XORs, essas operações de introdução são suportadas nativamente em todos os processadores. Portanto, os processadores low-end podem encriptar 4 vezes mais rápido e desencriptar 5 vezes mais rápido do que se estivessem usando o AES.

Adiantum é uma composição de outras cifras:

- NH: Uma função de hash
- Poly1305: um código de autenticação de mensagens (MAC)
- XChaCha12: Uma cifra de fluxo
- AES-256: Uma única invocação do AES

3 Inter-Process Communication (IPC)

Inter-Process Communication (IPC), ou Comunicação entre Processos, são os recursos que permitem que as aplicações troquem sinais e dados com segurança. Em vez de confiar nas instalações padrão do Linux IPC, o IPC do Android é baseado no Binder, uma implementação personalizada do OpenBinder. Pela segurança transmitida por este, a maioria dos serviços do sistema operativo Android e todos os serviços de comunicação entre pocessos de alto nível dependem do Binder.

4 Permissões

Como os aplicativos Android são instalados em uma sandbox e inicialmente não podem aceder a informações do utilizador e componentes do sistema (como a câmera e o microfone), o Android fornece um sistema com um conjunto predefinido de permissões para determinadas tarefas que o aplicativo pode solicitar. Por exemplo, para que a aplicação possa usar a câmera do dispositivo, solicitamos a permissão deste modo: *android.permission.CAMERA*

Anteriormente à versão 6.0 (nível 23 da API), todas as permissões solicitadas por uma aplicação eram concedidas aquando a sua instalação. Mas que a partir do nível 23 da API, o utilizador passou a ter que aprovar algumas solicitações de permissão durante a execução do aplicação.

4.1 Níveis de Proteção

As permissões do Android são classificadas com base no nível de proteção que oferecem e divididas em quatro categorias diferentes:

Normal: o nível mais baixo de proteção. Ela fornece às aplicações acesso a recursos isolados no nível do aplicativo, com risco mínimo para outras aplicações. Esta é concedida durante a instalação do aplicação e é o nível de proteção padrão. Exemplo: *android.permission.INTERNET*

Perigoso: esta permissão permite que a aplicação execute ações que possam afetar a privacidade do utilizador ou o funcionamento normal do dispositivo do utilizador. Este nível de permissão pode não ser concedido durante a instalação, pelo que, o utilizador deve decidir se a aplicação deve ter ou não essa permissão. Exemplo: *android.permission.RECORD_AUDIO*

Assinatura: esta permissão é concedida somente se a aplicação qua a solicita tiver sido assinada com o mesmo certificado que a aplicação que declarou a permissão. Se a assinatura corresponder, a permissão será concedida automaticamente. Exemplo: *android.permission.ACCESS_MOCK_LOCATION*

SystemOrSignature: esta permissão é concedida apenas a aplicações incorporadas na imagem do sistema ou assinadas com o mesmo certificado com o qual a aplicação que declarou a permissão foi assinada. Exemplo: *android.permission.ACCESS_DOWNLOAD_MAN*

4.2 Solicitação de Permissões

Como visto no início da secção Permissões, as aplicações podem ou não conceder permissões aquando a sua instalação, pelo que as que não o fazem desta forma, são feitas de um outro modo, ou seja, as aplicações podem solicitar permissões para os níveis de proteção Normal, Perigoso e Assinatura, usando para tal, uma tag deste género: *<uses-permission />* no manifesto, *AndroidManifest.xml*.

5 Processo de assinatura e publicação

Depois de uma aplicação ser desenvolvida com sucesso, o próximo passo é publicá-la e partilhá-la com outras pessoas. No entanto, as aplicaçóes não podem simplesmente ser adicionados a uma loja e partilhadas, sem antes serem assinadas. A assinatura criptográfica serve como uma marca verificável colocada pelo desenvolvedor da aplicação. Ela identifica o autor da aplicação e garante que esta não tenha sido modificada desde sua distribuição inicial.

5.1 Processo de assinatura

Durante o desenvolvimento, as aplicações são assinadas com um certificado gerado automaticamente. Este certificado é inerentemente inseguro e é apenas para depuração.

A maioria das lojas não aceita este tipo de certificado para publicação, portanto, deve ser criado um certificado com recursos mais seguros. Quando uma

aplicação é instalada no dispositivo Android, o Gerenciador de Pacotes garante que ela foi assinado com o certificado incluído no APK correspondente.

Se a chave pública do certificado corresponder à usada para assinar qualquer outro APK no dispositivo, o novo APK poderá compartilhar um UID (userID) com o APK pré-existente. Facilitando, assim, as interações entre aplicações do mesmo fornecedor. Como alternativa, é possível especificar permissões de segurança para o nível de proteção de assinatura, o que restringirá o acesso a aplicações que foram assinadas com a mesma chave.

5.2 Processo de Publicação

A distribuição de aplicações para qualquer lugar (no próprio site, qualquer loja etc.) é possível porque o ecossistema do Android está aberto. No entanto, o Google Play é a loja mais conhecida, confiável e popular, e é o próprio Google que a fornece.

A Amazon Appstore é a loja padrão confiável para dispositivos Kindle. Se os utilizadores quiserem instalar aplicativos de terceiros a partir de uma fonte não confiável, eles deverão permiti-lo explicitamente com as configurações de segurança do dispositivo.

As aplicações podem ser instaladas num dispositivo Android de várias fontes: localmente via USB, via loja de aplicativos oficial do Google (Google Play Store) ou em lojas alternativas.

Nota: Enquanto outros fornecedores podem rever e aprovar aplicações antes de serem realmente publicadas, o Google simplesmente procura assinaturas de malware conhecidas, o que faz com que o tempo entre o início do processo de publicação e a disponibilidade da aplicação ao público seja minimizado.

6 Android Application Attack Surface

O aplicativo Android pode estar vulnerável a ataques caso não:

- Valide todos os inputs através de uma comunicação IPC ou esquemas de URL
- Valide todos os inputs do utilizador nos campos de entrada.
- Valide o conteúdo carregado dentro de uma página Web.
- Comunique de forma segura com os servidores que prestam serviços ou seja suscetivel a ataques man-in-the-middle.
- Armazene com segurança todos os dados locais ou carregue dados não confiáveis do armazenamento.
- Tenha proteção contra ambientes comprometidos, reempacotamento ou outros ataques

7 TESTES DE SEGURANÇA EM APLICAÇÕES

7.1 Data Storage on Android

Os dados públicos devem estar disponíveis para todos, mas os dados confidenciais e privados devem ser protegidos ou, melhor ainda, mantidos fora do armazenamento do dispositivo.

Para além de proteger dados confidenciais, é necessário garantir que os dados lidos de qualquer fonte de armazenamento sejam validados e possivelmente limpos. Esta validação geralmente serve para garantir que os dados apresentados sejam do tipo solicitado, mas com o uso de HMAC's é possível validar a correção dos dados.

7.1.1 Testar o armazenamento local para dados confidenciais (MSTG-STORAGE-1 e MSTG-STORAGE-2)

Por norma, deve-se armazenar o mínimo possível de dados sensíveis no armazenamento local permanentemente. No entanto, algum tipo de dado do utilizador deve ser armazenado. Por exemplo, pedir ao utilizador que insira uma senha muito complexa de todas as vezes que a aplicação inicie não é uma boa idéia em termos de usabilidade. Deste modo, as aplicações devem armazenar em cache algum tipo de token de autenticação para evitar isso.

Os dados confidenciais ficam vulneráveis quando não estão adequadamente protegidos pela aplicação que os armazena persistentemente, o que pode levar, à divulgação de informações confidenciais. Em geral, um atacante pode identificar essas informações e usá-las em ataques adicionais, como engenharia social, sequestro de contas (se as informações da sessão ou um token de autenticação tiverem sido divulgados), ou ainda, a recolha de informações de aplicações que possuem uma opção de pagamento.

Os dados podem ser armazenados persistentemente de várias maneiras, entre as quais:

- Preferências Compartilhadas
- Bases de dados SQLite
- Bases de dados Firebase
- Armazenamento interno
- Armazenamento externo

Preferências Compartilhadas

A API SharedPreferences é frequentemente usada para salvar permanentemente pequenas coleções de pares de valores-chave. Os dados armazenados em um objeto SharedPreferences são gravados em um arquivo XML de texto sem formatação.

O objeto SharedPreferences pode ser declarado legível (acessível a todos as aplicações) ou privado. A API SharedPreferences geralmente pode levar à exposição de dados confidenciais.

Bases de dados SQLite

- **Bases de dados SQLite (não encriptadas)** - SQLite é um mecanismo de banco de dados SQL que armazena dados em arquivos .db. O Android SDK possui suporte interno para bancos de dados SQLite. O pacote principal usado para gerenciar os bancos de dados é android.database.sqlite.

- **Bases de dados SQLite (encriptadas)** - Se as bases de dados SQLite encriptadas forem usadas, a senha estará codificada na fonte, armazenada em preferências compartilhadas ou oculta noutro lugar no código ou no sistema de arquivos.
  As formas seguras de recuperar a chave incluem:
  - Solicitando ao utilizador que decodifique a base de dados com um PIN ou senha assim que a aplicação for aberta (senhas e PINs fracos são vulneráveis a ataques de força bruta)

  - Armazenando a chave no servidor e permitindo que ela seja acessada apenas a partir de um serviço da Web (para que a aplicação possa ser usada apenas quando o dispositivo estiver online)

Bancos de dados Firebase

O Firebase é uma plataforma de desenvolvimento que pode ser aproveitada para armazenar e sincronizar dados de uma aplicção com um banco de dados

hospedado na nuvem NoSQL. Os dados são armazenados como JSON e são sincronizados em tempo real com todos os clientes conectados e também permanecem disponíveis mesmo quando o aplicativo fica offline.

Armazenamento interno
É possível salvar arquivos no armazenamento interno do dispositivo. Os arquivos salvos no armazenamento interno são containers por padrão e não podem ser acessedidos por outras aplicações no dispositivo, e quando o utilizador desinstala a aplicação, esses arquivos são removidos.

Armazenamento externo
Outra das possibilidades, é o armazenamento externo compartilhado. Esse armazenamento pode ser removível (como um cartão SD) ou interno (não removível). Os arquivos salvos no armazenamento externo são legíveis. O utilizador pode modificá-los quando o armazenamento USB estiver ativado.

7.1.2 Determinar se dados sensíveis são enviados a terceiros (MSTG-STORAGE-4)

A possibilidade de incorporar serviços de terceiros nas aplicações, é que este tipo de serviços podem ter implementados serviços de rastreamento, monitorização do comportamento do utilizador, venda de anúncios em banner, entre outros.

Pelo que a desvantagem, neste tipo de serviços é a falta de visibilidade, ou seja, não sabermos exatamente qual o código que executam. Desta forma, deve garantir-se que apenas sejam enviadas as informações não sensíveis necessárias.

7.1.3 Procurar dados sensíveis na cache do teclado (MSTG-STORAGE-5)

A inserção de dados por parte dos utilizadores, é algo que pode ser comprometer informações confidenciais. Ou seja, quando os utilizadores inserem informação nos campos de entrada, o software sugere dados automaticamente, o que em parte pode ser muito útil para aplicações de mensagens, mas, em contrapartida, a cache do teclado pode divulgar informações confidenciais quando o utilizador seleciona um campo de entrada que aceita esse tipo de informação, pelo que se torna necessário limpar a cache do teclado de vez enquando.

7.1.4 Verificar a divulgação confidencial de dados por meio da interface do utilizador (MSTG-STORAGE-7)

São enumeras as aplicações que exigem que os utilizadores insiram vários tipos de dados para, por exemplo, registrar uma conta ou efetuar um pagamento, entre outros.

Este tipo de aplicaçóes requerem alguns cuidados, visto que ocorre a exposição de dados confidenciais, pelo que, é imperativo que estes não sejam mostrados em texto limpo na interface após inseridos. Os dados dados confidenciais devem então ser mascarados, através de asteriscos ou pontos em vez de texto não encriptado.

7.2 Android Cryptographic APIs

7.2.1 Testando a configuração de algoritmos padrão criptográficos (MSTG-CRYPTO-2, MSTG-CRYPTO-3 e MSTG-CRYPTO-4)

As APIs de criptografia do Android são baseadas em Java Cryptography Architecture (JCA). Esta separa as interfaces e a implementação, o que possibilita incluir vários provedores de segurança que podem implementar conjuntos de algoritmos criptográficos.

A maioria das interfaces e classes JCA são definidas nos pacotes java.security.* e javax.crypto.*. Para além disso, existem ainda pacotes específicos para Android, android.security.* e android.security.keystore.*.

A lista de provedores incluídos no Android varia consoante a versão do Android e as compilações específicas do Original Equipment Manufacturer (OEM). Atualmente, algumas implementações de provedor, em versões mais antigas são mais inseguras/vulneráveis. Portanto, as aplicações Android não devem apenas escolher os algoritmos corretos e fornecer uma boa configuração, em alguns casos, também devem prestar atenção à força das implementações nos provedores legados.

7.2.2 Teste do manutenção de chaves (MSTG-STORAGE-1, MSTG-CRYPTO-1 e MSTG-CRYPTO-5)

Quando o assunto são chaves criptográficas, deve ter se especial atenção, visto existirem tanto maneiras seguras como inseguras para o seu armazenamento.

A maneira mais segura de manusear conteúdos importantes, simplesmente é nunca armazená-los no dispositivo, ou seja, o que significa que o utilizador deve ser solicitado a inserir uma senha sempre que a aplicação precisar de executar uma operação criptográfica, o que do do ponto de vista da experiência do utilizador torna-se inpensável. A razão disto é que o conteúdo principal estaja disponível apenas numa matriz na memória enquanto estiver a ser usado, e quando a chave não estiver a ser necessária, a matriz pode ser zerada, o que minimiza a janela de ataque. Nenhum conteúdo importante afeta o sistema de arquivos e nenhuma senha é armazenada.

No entanto, deve-se ter em consideração que algumas cifras não limpam adequadamente as suas matrizes de bytes. Por exemplo, a codificação AES no BouncyCastle nem sempre limpa a sua última chave de trabalho. De seguida,

as chaves baseadas no BigInteger (por exemplo, chaves privadas) não podem ser removidas do heap nem zeradas. E claro, deve-se tomar especial atenção ao tentar zerar a chave.

7.2.3 Testar a geração de números aleatórios (MSTG-CRYPTO-6)

A criptografia requer geração segura de números pseudo-aleatórios (PRNG).

As classes Java padrão não fornecem aleatoriedade suficiente e, de fato, podem permitir que um atacante adivinhe o próximo valor que será gerado e use esse palpite para representar outro utilizador ou aceder a informações confidenciais.

Para tal, deve ser usado o SecureRandom, o qual deve ser instanciado por meio do construtor padrão sem argumentos. Outros construtores são para usos mais avançados e, se usados incorretamente, podem levar à diminuição da aleatoriedade e segurança. O provedor PRNG que apoia o SecureRandom usa o arquivo de dispositivo */dev/urandom* como fonte de aleatoriedade por padrão.

7.3 Local Authentication on Android

Durante a autenticação local, uma aplicação autentica um utilizador verificando con credenciais guardadas localmente no dispositivo (PIN, password, carateristicas biometricas), que são verificados por referência a dados locais. Este processo é realizado para que os utilizadores possam retomar mais convenientemente uma sessão existente com um serviço remoto ou como um meio de autenticação avançada para proteger algumas funções críticas.

7.3.1 Testar credenciais de confirmação (MSTG-AUTH-1 e MSTG-STORAGE-11)

O fluxo de confirmação da credencial está disponível desde da versão 6.0 e é usada para garantir que os utilizadores não precisem de inserir senhas específicas da aplicação junto com a proteção da tela de bloqueio.

Em vez disso, se um utilizador que fez login no dispositivo recentemente, as suas credenciais de confirmação podem ser usadas para desbloquear conteúdos criptográficos na AndroidKeystore. Ou seja, se o utilizador desbloqueou o dispositivo dentro dos limites de tempo definidos (setUserAuthenticationValidityDurationSeconds), caso contrário, o dispositivo precisará de ser desbloqueado novamente.

7.4 Android Network APIs

7.4.1 Testar a verificação de identificação de terminal (MSTG-NETWORK-3)

Uma das medidas de segurança para transporte de informação confidencial pela rede é o o uso do TLS. Porém, a comunicação encriptada entre uma aplicação móvel e a sua API de back-end não é trivial.

Por norma, as soluções mais simples, mas menos seguras, como por exemplo, aquelas que aceitam qualquer certificado, para facilitar o processo de desenvolvimento, são soluções fracas que entram na versão de produção, e que acabam por expor os utilizadores a ataques man-in-the-middle.

Daí que surgem duas medidas essencias, nomeadamente:

- Verificar se um certificado provém de uma fonte confiável, ou seja, uma CA (Autoridade de Certificação) confiável.
- Determinar se o servidor do nó da extremidade apresenta o certificado correto.

7.4.2 Testando armazenamentos de certificados personalizados e fixação de certificados (MSTG-NETWORK-4)

O processo de armazenamento de certificados e fixação de certificados é algo a que se deve prestar especial atenção, pois a aceitação de qualquer certificado assinado pode ser um processo compremetedor, ou seja, apenas deve ser feita a aceitação de certificados assinados por autoridades de certificação de confiança.

A fixação de certificado é o processo de associar o servidor back-end a um certificado X.509 ou chave pública específica, em vez de aceitar qualquer certificado assinado por uma autoridade de certificação confiável. Após armazenar o certificado ou a chave pública do servidor, a aplicação móvel conectar-se-á, posteriormente, apenas ao servidor conhecido.

Em suma, retirar a confiança de autoridades de certificação externas reduz a superfície de ataque, visto existirem enumeros casos de autoridades de certificação que foram comprometidas ou enganadas na emissão de certificados por impostores.

O certificado pode ser fixado e codificado na aplicação ou recuperado no momento em que esta se conecta ao back-end. Neste último caso, o certificado é associado ao fixado ao host, e aquando o host é visto pela primeira vez. O que torna esta alternativa menos segura, porque os atacantes que interceptem a conexão inicial podem injetar os seus próprios certificados.

7.5 Android Platform APIs

7.5.1 Testar permissões de aplicativos (MSTG-PLATFORM-1)

O Android atribui uma identidade distinta do sistema a todos as aplicações instaladas, e como cada aplicação opera numa área restrita de processo, as aplicações devem solicitar explicitamente o acesso a recursos e dados que estão fora da área restrita.

Para tal, elas solicitam este acesso declarando as permissões necessárias para usar os dados e recursos do sistema, e dependendo do quão sensíveis ou críticos

sejam os dados ou os recursos, o próprio sistema Android concederá a permissão automaticamente ou, então, solicitará que o utilizador aprove a solicitação.

Como já referido anteriormente na secção "Permissões", estas são classificadas em quatro categorias diferentes com base no nível de proteção que oferecem:

- Normal: concede às aplicações acesso a recursos isolados no nível da aplicação com risco mínimo para outras aplicações. Exemplo: android.permission.INTERNET
- Perigoso: geralmente concede à aplicação controlo sobre os dados do utilizador ou sobre o dispositivo de uma maneira que pode afetar o utilizador. Exemplo: android.permission.RECORD_AUDIO
- Assinatura: esta permissão é concedida apenas se a aplicação solicitante tiver sido assinada com o mesmo certificado usado para assinar a aplicação que declarou a permissão. Exemplo: android.permission.ACCESS_MOCK_LOCATION
- SystemOrSignature: esta permissão é concedida apenas a aplicações incorporadas na imagem do sistema ou assinadas com o mesmo certificado usado para assinar a aplicação que declarou a permissão. Exemplo: android.permission.ACCESS_DOWNLOAD_

7.5.2 Testar exposição sensível à funcionalidade através do IPC (MSTG-PLATFORM-4)

Durante a implementação de uma aplicação móvel, os desenvolvedores podem usar técnicas tradicionais para IPC (como ficheiros ou sockets partilhadas), as funcionalidades do IPC para aplicações devem ser usadas visto serem muito mais seguras para não haver comprometimento de dados.

A funcionalidade do sistema IPC oferecida pelas plataformas de aplicações móveis deve ser usada, por ser muito mais maduro que as técnicas tradicionais. O uso de mecanismos IPC sem segurança pode fazer com que a aplicação vaze ou exponha dados confidenciais.

Mecanismos IPC do Android, como Binders, Services, Bound Services, AIDL (Android Interface Definition Language), Intents, ou Content Providers, são mecanismos que podem levar à exposição de dados confidenciais.

iOS

Apps no iOS estão isoladas umas das outras através de uma sandbox(Seatbelt) cujo propósito é limitar os danos ao sistema e aos dados do utilizador caso a aplicação seja corrompida. Existe também um controlo de acesso mandatorio (MAC) que descreve os recursos que a aplicação pode usar.

Oferece poucos CIP(comunicação inter-processos) o que minimiza os vetores de ataque.

8 Aspetos de Segurança do iOS

8.1 Secure Boot

Quando Um dispositivo iOS é ligado começa por ler as instruções de uma memória read-only conhecida com Boot ROM. Esta garante que a assinatura do Low Level Bootloader está correta e o Low Level Boot Loader faz o mesmo com o iBoot bootloader, que por sua vez verifica a assinatura do iOS Kernel.

Se o Boot ROM falhar, o dispositivo entra no modo Device Firmware Upgrade (DFU)[1], se algum dos outros passos falhar o dispositivo entra em modo de recuperação.

8.2 File System Encryption

Cada dispositivo iOS tem 2 chaves AES de 256 bits (User ID e Group ID) que são guardadas no processador de aplicações durante o fabrico. Apenas os crypto-engines tem acesso ás chaves e o User ID é usado para garantir a encriptação dos ficheiros dentro do iOS.

Como os UIDs não são guardados no fabrico, nem a Apple pode restaurar as chaves de encriptação de um dispositivo[2].

8.3 Code Signing

Não é possivel correr código num iOS que não seja jailbroken[3] a não ser que a Apple o permita. Para correr uma aplicação é preciso um perfil de desenvolvedor e um certificado assinado pela Apple.

[1] DFU or Device Firmware Upgrade mode permite que um dispositivo tenha o seu software restaurado independentemente do estado em que se encontre.

[2] Isto acontece porque a chave está gravada no chip de silicone

[3] telemoveis "Jailbroken"são telemoveis modificado para fornecer acesso a todo o sistema de ficheiros[2]

8.4 App Store Data Protection

Quando se decarregam aplicações da App Store é aplicado o FairPlay Code Encryption sobre estas. O processo para a implementação desta encriptação segue os seguintes passos:

1. Quando se regista uma nova conta Apple um par de chaves publica/privada é criada e atribuida à conta sendo que a privada é armazenada no dispositivo iOS e a pública fica na App Store.
2. Quando se quer descarregar uma aplicação, a App Store encripta com a chave pública a mesma e o dispositivo quando quiser usar a aplicação pela primeira vez após ser ligado desencripta-a com a chave privada[3].

8.5 General Exploit Mitigations

Sempre que um aplicação é executada a alocação da memória para a aplicação é aleatório prevenindo ataques de injeção de código.

As bibliotecas a usar pelas aplicaçãos, como são comuns, são alocadas aleatóriamente quando o dispositivo é ligado e não sempre que uma aplicação que as requer é iniciada.

Apesar de todos os mecanismos de segurança implementados ainda podem ser encontrados problemas em:

I. Proteção de dados na memória.
II. Keychain.
III. Touch ID.
IV. dados na rede.

9 iOS Application Attack surface

Uma aplicação iOS pode estar vulneráve a ataques caso não:

. Valide todos os inputs através de uma comunicação IPC ou esquema URL.
. Valide todos os inputs que o utilizador insira nos campos que preenche.
. Valide o conteudo carregado dento de uma página web.
. Comunique de forma segura com os servidores que prestam serviços ou seja suscetivel a ataques man-in-the-middle.
. Guarde de forma segura os dados locais ou carregue dados de terceiros suspeitos.
. Tenha proteção contra ambientes comprometidos, reempacotamento ou outros ataques[3].

[3] Cada dispositivo tem um crypto-engine dedicado que fornece a geração de uma chave AES de 256 bit e um hash SHA-1.

10 TESTES DE SEGURANÇA EM APLICAÇÕES

10.1 Data Storage on iOS

A proteção de dados sensíveis como tokens de autenticação e dados privados é uma chave para a segurança no telemovel.

10.1.1 Testar o Armazenamento Local (MSTG-STORAGE-1 and MSTG-STORAGE-2)

Secure Enclave Processor (SEP):

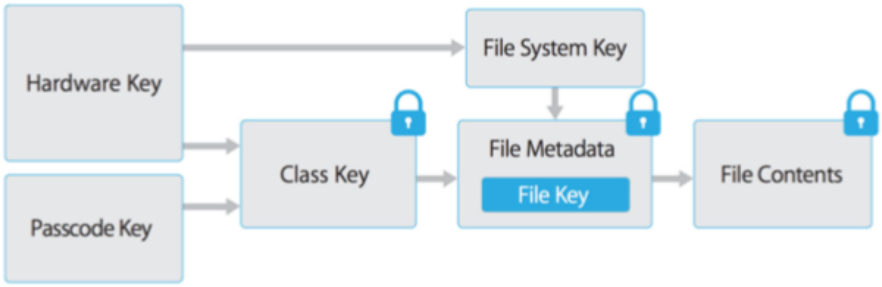

Figura 2. Secure Enclave Processor Scheme

A arquitetura é baseada numa hierarquia de chaves. O User ID e uma passcode key que é derivada da aplicação do algoritmo PBKDF2 sobre a password de desbloqueio do dispositivo. Classkey está associada ao estado do dispositivo (bloqueado ou desbloqueado). Cada ficheiro guardado no sistema de ficheiros está encriptado com uma chave por ficheiro.

Ficheiros têm 4 tipos de classes de proteção:

. *Proteção completa:* A chave de classe é apagada da memória após o dispositivo ser bloqueado.Tornando os dados inacessiveis quando o dispositivo está bloquado.

. *Proteção até ser aberta:* Igual à anterior mas a chave não é apagada quando o dispositivo é bloqueado permitindo acesso ainda bloqueado. Útil para emails.

. *Proteçã até primeira autenticação do utilizador:* O ficheiro pode ser acedido mal o utilizador desbloqueie o dispositivo após liga-lo.

. *Sem proteção:* A chave de classe é guardado na *Effaceable Storage*[0]

Para realizarmos o teste ao armazenamento local, de forma a verificarmos se conseguimos aceder ao conteúdo dos dados basta seguirmos estes passos.

1. Acionar a funcionalidade da aplicação que guarda dados sensíveis, por exemplo fazer login na aplicação.
2. navegar até à diretoria com os ficheiros sensíveis.

   `/var/mobile/Containers/Data/Application/$APP_ID/`

3. executar grep com os dados guardados por exemplo: *grep -iRn <USERID>*.

   Se os dados forem legiveis o teste falhou.

10.1.2 Determinar se dados sensíveis são enviados para terçeiros (MSTG-STORAGE-4)

Regras:

. Não deve ser enviada mais informação do que a estritamente necessária.
. Todos os dados enviados para terceiros deve ser anonimizado para impedir exposição de PII (Personal Identifiable Information).

Para realizar este teste basta usar um proxy para podermos intercetar o tráfico entre a aplicação e os terçeiros. Posteriormente podemos analisar o tráfico e se estiver a ser enviada informação sensivel pessoal, o teste falha.

10.1.3 Procurar dados sensíveis na cache do teclado (MSTG-STORAGE-5)

O protocolo UITextInputTraits é usado para a cache do teclado e possui 2 propriedades:

. *var autocorrectionType UITextAutocorrectionType:*
Determina se a correção automática está ativada durante a escrita, o valor por defeito desta propriedade é UITextAutocorrectionTypeDefault, que permite autocorreção.

. *var secureTextEntry:*
Variável responsável por definir se se mantêm as palavras escritas em cache e se se escondem as palavras escritas nos campos UITextField. Por defito é NO.

[0] A *Effaceable Storage* é uma flash memory para pequenos dados que nunca é apagada.

O objetivo deste teste é procurar no código fonte da aplicação uma implementação semelhante à da figura para os UITextFields visados. Se estes não tiverem a configuração mostrada falham o teste.

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

Figura 3. Secure Text Entry Example

Para corrigir este problema podemos programar uma solução para o UITextField alvo apicando o código: *textObject.autocorrectionType = UITextAutocorrectionTypeNo* aos *UITextFields*, *UITextViews* e *UISearchBars* que desejarmos. Para objetos que devem ser mascarados devemos definir o atributo *textObject.secureTextEntry* como *YES*. Na imagem abaixo vemos um exemplo.

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

Figura 4. Set a secure Text Entry

Para testar se a cache do teclado possui dados sensiveis que não deveriam ser guardados, basta:

1. Fazer reset à cache do teclado.
2. Entrar na aplicação e inserir, por exemplo, os dados de autenticação.
3. Sair da aplicação.
4. Reentrar na aplicação e começar a reinserir os dados.
5. Se for permitido completar automáticamente o preenchimento dos dados, o teste falha.

10.1.4 Procurar exposição de dados sensíveis na interface do utilizador (MSTG-STORAGE-7)

A inserção de informação sensível é necessária quando queremos por exemplo levantar dinheiro, fazer um pagamento entre outros. Ao utilizar aplicações que requerem esse tipo de dados sensíveis é imperativo que estes não sejam mostrados em texto limpo na interface após inseridos.

Um campo que mascare o que nele é escrito pode ser verificado de duas formas. A primeira pode ser verificada navegando pelo código da aplicação até

à configuração da caixa que recebe os dadso sensíveis e verificando se a opção *Secure Text Entry* está ativa, se estiver ativa o texto inserido é substituido por pontos. A segunda é em interação com a aplicação a testar inserir os dados sensiveis e verificar se os dados aparecem em texto limpo ou são mascarados.

10.2 iOS Cryptographic APIs

10.2.1 Verificação da configuração dos algoritmos criptográficos standard (MSTG-CRYPTO-2 and MSTG- CRYPTO-3)

O CriptoKit da Apple tem os seguintes algoritmos:

. *Hashes:*

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

Figura 5. Available Hash Algorithms

. *Symmetric-Key:*

```
- Message Authentication Codes (HMAC)
- Authenticated Encryption
    - AES-GCM
    - ChaCha20-Poly1305
```

Figura 6. Available Symmetric-key Algorithms

. *Public-Key:*

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

Figura 7. Available Public-key Algorithms

Se a aplicação usar implementações fornecidas no kit, a melhor forma de verificar o estado do algoritmo é analizando as chamadas ás funções do CommonCryptor.

```
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 */
```

Figura 8. CommonCryptor Status

Podemos depois comparar os enums para determinar o algoritmo, o padding e a chave usados. Após determinarmos os 3 podemos verificar se estes são seguros ou se já estão ultrapassados.

10.2.2 Testar Gestão de Chaves (MSTG-CRYPTO-1 and MSTG-CRYPTO-5)

Quando guardamos uma chave é recomendado o uso do keyChain desde que não se use o atributo kSecAttrAccessibleAlways.

O mecanismo de keyChain permite 2 formas de guardar chaves:

1. Usar uma chave de segurança guardada no secure-enclave para encriptar a chave.
2. Guardar a chave no secure-enclave[4].

[4] O Secure-enclave é um gestor de chaves baseado em hardware e isolado do processador para maior segurança[7].

O teste tem de ser monitorizado verificando o acesso aos ficheiros do sistema durante operações criptográficas da aplicação.

O teste falha se:

- Chaves usadas para proteger dados sensíveis são sincronizadas entre dispositivos.
- Chaves não são guardadas com proteção.
- Chaves não são derivadas de funcionalidades estáveis do dispositivo.
- Chaves não são escondidas ao usar linguagens de baixo nivel.
- Chaves são importadas de locais inseguros.

10.2.3 Testar Geração Aleatória de Números (MSTG- CRYPTO-6)

A Apple fornece uma API para gerar números aleatório criptográficamente seguros.

Em Swift a chamada à API é feita com o seguinte código:

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

A versão em Objective-C é:

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

Um exemplo de utilização é:

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

Para testar se a implementação usada é segura basta verificar que a geração de números aleatórios, se não for igual ás acima descritas, sejam wraplicaçãoers implementados sobre a API acima.

*Nota: Podemos usar o plugin **Burp's sequencer** para avaliar a qualidade da geração dos números.*

10.3 Local Authentication on iOS

Durante a autenticação local, uma aplicação autentica um utilizador verificando con credenciais guardadas localmente no dispositivo (PIN, password, carateristicas biometricas).

Atacantes podem facilmente passar a autenticação local caso nada seja retornado do processo de autenticação.

10.3.1 Testar Autenticação Local (MSTG-AUTH-8 and MSTG-STORAGE-11)

É importante lembrar que a framework de autenticação local é um processo baseado em eventos e portanto não deve ser o único método de autenticação.

Apesar de ser uma autenticação eficiente ao nivel do utilizador, esta é facilmente ultrapassável usando instrumentos de patching. Assim é melhor usar o método de serviço da keyChain, ou seja, verificar que processos sensíveis são protegidos com métodos de serviço da keyChain. ex: transações financeiras.

Verificar as flags dos controlos de acesso da keyChain para assegurar que os dados só são acedidos pelo utilizador. Podem-se utilizar as seguintes flags:

. *kSecAccessControlBiometryCurrentSet:*

Esta flag obriga o utilizador a autenticar-se biometricamente antes de aceder à keyChain. Se for adicionado um novo dado biométrico, a entrada na keyChain vai ser invalidada e só poderá ser acedida pelos utilizadores autenticados quando os dados foram adicionados.

. *kSecAccessControlBiometryAny:*

Esta flag obriga o utilizador a autenticar-se biometricamente antes de aceder à keyChain. Caso novos dados biométricos sejam adicionados a autenticação na keyChain mantem-se. Útil para utilizadores que mudam a própria impressão digital mas bom para atacantes também visto que se conseguirem registar a impresão digital conseguem aceder aos restantes dados biométricos.

. *kSecAccessControlUserPresence:*

Permite que o utilizador se autentique por password se autenticação biométrica não funcionar. É mais fraco visto ser mai facil roubar uma password por shouldersurfing do que passar pelso dados biométricos.

Para garantir que os dados biométricos podem ser usados temos de verificar se a classe kSecAttrAcessibleWhenPasscodeSetThisDeviceOnly ou a kSecAttrAcessibleWhenPasscodeSet estão implementadas[5].

Para um dispositivo *jailbroken* existem ferramentas que permitem efetuar uma quebra na segurança da autenticação local. Neste exemplo usaremos o Swizzler2, uma ferramenta que através do uso do software Frida permite que a função de autenticação retorne True mesmo que tenha falhado.

Os passos para efetuar o teste são:

1. Ir aos settings e escolher Swizzler
2. Permitir injeção do Swizzler nas aplicações
3. Permitir registar tudo no Syslog
4. Permitir registar tudo num ficheiro

[5] A variante ThisDeviceOnly garante que não há sincronização dos dados biométricos entre dispositivos iOS.

5. Entrar nos submenus da framework iOS
6. Permitir autenticação local
7. Entrar no submenu e escolher as aplicações alvo
8. Permitir que a aplicação escolhida corra
9. Reiniciar a aplicação
10. Quando for pedida a impressão digital carregar em cancelar

Se a aplicação continuar sem requerer a impressão digital esta não passou no teste.

10.4 iOS Network APIs

Quase todas as aplicações no iOS atuam como cliente de um ou mais serviços remotos. Como não podemos confiar nas redes pública. Temos portanto de garantir que as aplicações tomam medidas para mitigar os riscos.

10.4.1 Segurança da Camada de Trasporte da Aplicação (MSTG-NETWORK-2)

A App Transport Security (ATS) é um conjunto de verificações de segurança que o sistema operativo realiza quando efetua conecções com NSURLConnection[6], NSURLSession[7] e CFURL[8] a servidores públicos. Estas verificações são efetuadas por defeito do iOS SDK 9 em diante.

Quando a ligação é feita a um IP, um dominio de nomes não qualificado ou TLD de .local o ATS não é aplicado.

Agora vamos ver uma análise das configurações da ATS.

Se o código fonte estiver disponivel devemos abrir a Info.plist na diretoria da aplicação (*Bundle Directory*) que o desenvolvedor configurou. Na imagem abaixo vemos um exemplo de uma exceção configurada para desativar as restrições globais da ATS.

```
<key>NSAppTransportSecurity</key>
<dict>
    <key>NSAllowsArbitraryLoads</key>
    <true/>
</dict>
```

Figura 9. ATS example configuration

[6] Conecções NSURLConnection permitem carregar conteúdos de um URLao fornecer um pedido para um objeto URL[4].

[7] A classe NSURLSession fornece uma API para download e para upload de dados entre pontos opostos de uma conexão indicados por URLs[5].

[8] O tipo CFURL fornece facilidade para criar, percorrer e desreferenciar strings URL[6].

A desativação das restrições da ATS depende da aplicação, por exemplo a aplicação do chrome para iOS tem estas configurações, o que é aceitavel visto que por outro lado a aplicação não se conseguiria ligar a um site HTTP que não tivesse todos os requisitos da ATS.

As recomendações para o uso da ATS são:

. A ATS deve ser configurada de acordo com as melhores práticas aconcelhadas pela Apple e só deve ser desativada excecionalmente.

. Se a aplicação se conectar a um número definido de dominios que o dono da aplicação controla, esses dominios devem suportar a ATS.

. Se houver conecções a terçeiros (que não pertençam ao dono da aplicação) devem ser avaliados quais os requisitos da ATS que não são suportados e podem ser desativados.

. Se a apicação abrir sites web de terçeiros, do iOS 10 para a frente pode-se usar o *NSAllowsArbitraryLoadsInWebContent* para desabilitar as restrições da ATS para o conteúdo das páginas web.

10.4.2 Teste e Validação de Certificados (MSTG-NETWORK-3 and MSTG-NETWORK-4)

As autoridades de cetificação são uma parte fulcral da comunicação cliente-servidor. No iOS existem uma quantidade enorme de certificados confiáveis. CAs podem ser adicionadas manualmente pelo utilizador ou por malware, para precaver esta situação basta remover a confiança nas CAs adicionadas.

Falhas podem ocorrer se a aplicação esperar outra chave ou certificado do servidor, ou pode estar a ocorrer um ataque. Nestes casos basta seguir os passos referidos no capitulo *Android Network APIs*[7].

Em baixo apresentamos 3 formas de manter os certificados confiaveis por parte da aplicação.

1. Incluir os certificados do servidor na diretoria da aplicação e verifica-lo a cada conecção. Isto requer um mecanismo de atualização do certificado para o caso do certificado mudar.

2. Limitar os emissores de certificados, ex: uma CA envia à aplicação o certificado de um servidor. O certificado será seguro e limita ataques possíveis.

3. A aplicação deve possuir a chave pública da CA confiavel pertencente ao dono da aplicação. Isto não requer atualizar a aplicação sempre que o servidor mudar de certificado. Usar uma CA própria faz com que o seu certificado seja auto assinado.

Para validarmos os certificados presentes na aplicação vamos usar o Burp para intercetar o tráfico que circula entre a aplicação a testar e o seridor a que esta se está a ligar. Posteriormente realizamos os seguintes passos pela ordem em que aparecem.

1. Temos de verificar se ferramentas como o SSL Kill Switch estão desativadas.

2. Garantir que não há certificados a serem adicionados à diretoria da aplicação.

3. Iniciar a aplicação, se conseguirmos ver o tráfico não é realizada nenhuma validação do certificado e a aplicação falhou o teste.

4. Se houver uma falha no SSL handshake (está a ser verificada a validade do certificado), instalamos o certificado do Burp.

5. Se o handshake for bem sucedido e conseguimos ver o tráfico, o certificado foi validado com o certificado na diretoria da aplicação mas não pela cadeia de confiança da mesma. A aplicação falha neste ponto se tal ocorrer.

Se não conseguirmos ver o tráfico apesar do SSL handshake funcionar então todas as medidas de segurança para o estabelecimento da conecção foram tomadas e a aplicação passou no teste.

10.5 iOS Platform APIs

10.5.1 Testar Permissões das Aplicações (MSTG-PLATFORM-1)

Ao contráro do android cada aplicação no iOS corre sobre o seu ID de utilizador, todas as aplicações de terçeiros correm sobre o non-privileged mobile user.

Algumas permissões no entanto podem ser configuradas pelo desenvolvedor da aplicação e terão efeito mal a aplicação seja instalada. Este é o problema que trataremos nesta secção, aplicaçãos que pedem permissões por razões não obvias, pr exemplo um scanner de QRcodes deve ter acesso à câmara mas dificilmente necessitará de acesso ás fotografias.

Desde o iOS 10 existem areas que precisam de inspeção ás permissões:

. *Mensagens de permissões no ficheiro Info.plist*

Estas mensagens são as que aparecem por defeito quando a aplicação pretende ter acesso a um recurso.

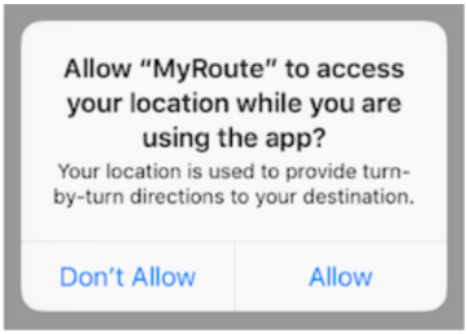

Figura 10. Example of permission request message

O objetivo de verificar o ficheiro é poder ler a justificação do uso da permissão por parte da aplicação como se pode ver na imagem abaixo.

```
<plist version="1.0">
<dict>
    <key>NSLocationWhenInUseUsageDescription</key>
    <string>Your location is used to provide turn-by-turn directions to
```

Figura 11. Example of permission request with justification

. *Ficheiro de permissões do código da aplicação*

Visualizar este ficheiro serve unicamente para verificar que a aplicação não pede permissões demasiado elevadas para o que tem de fazer permitindo assim fugas de dados. Na imagem abaixo podemos ver o exemplo da permissão pedida pela aplicação Telegram (aplicaçãolication-groups). Esta aplicação passa no teste por não pedir permissões adicionais, visto que não tem necessidade destas.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTD
<plist version="1.0">
<dict>
...
    <key>com.apple.security.application-groups</key>
    <array>
        <string>group.ph.telegra.Telegraph</string>
    </array>
</dict>
...
</plist>
```

Figura 12. Example of Telegram's permission request

. *Permissões incluidas nos ficheiros binarios compilados*

Mais uma vez vamos procurar por permissões. Podemos usar a aplicação radare2 (com -qc para correr o comando e sair sem deixar registos) para procurar em todas as strings dos ficheiros binario por *PropertyList* como podemos ver no exemplo abaixo.

```
$ r2 -qc 'izz~PropertyList' ./Telegram\ X

0x0015d2a4 ascii <?xml version="1.0" encoding="UTF-8" standalone="yes"?>\n<!DO
"-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd"
...<key>com.apple.security.application-groups</key>\n\t\t<array>
\n\t\t\t<string>group.ph.telegra.Telegraph</string>...

0x0016427d ascii H<?xml version="1.0" encoding="UTF-8"?>\n<!DOCTYPE plist PUBL
"-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd"
<dict>\n\t<key>cdhashes</key>...
```

Figura 13. Result of searching for PropertyList with radare2

. *Inspeção do código fonte*

Esta inspeção tem de ser feita para verificar se as permissões são usadas corretamente. Assim as recomendações são:

- Verificar se as strings que explicam a razão pela qual a aplicação precisa da permissão e a forma como a aplicação usa essa permissão são compatíveis.

- Verificar que as permissões concedidas não são usadas para divulgamento de dados pessoais.

10.5.2 Testar Exposição de Dados Sensiveis através de Funcionalidades IPC (MSTG-PLATFORM-4)

Durante a implementação de uma aplicação os desenvolvedores podem usar técnicas tradicionais para o IPC (como ficheiros ou sockets partilhadas), as funcionalidades do IPC para aplicações devem ser usadas visto serem muto mais seguras para não haver comprometimento de dados.

Testar um link universal requer os seguintes passos:

1. *Verificar as permissões do dominio associado*

   Links universais requerem que o desenvolvedor adicione as permissões dos dominios associados numa lista de dominios que a aplicação suporte. Podemos usar o Xcode para verificar isto. Basta ir a *Capabilities* e procurar por *Associated Domains*. Em baixo mostra-se um exemplo com recurso à inspeção da aplicação Telegram.

```
<key>com.apple.developer.associated-domains</key>
<array>
    <string>applinks:telegram.me</string>
    <string>applinks:t.me</string>
</array>
```

Figura 14. Example of domain permission request for universal link usage

2. *Obter o ficheiro de associação no site da Apple*

   Este ficheiro tem de ser acessivel por HTTPS sem redirecionamentos em *https://<domain>/.well-known/aplicaçãole-aplicação-site-assosiation.*

   Se tudo correr bem o site verificará por nós que o link é seguro e apresentará um relatório semelhante ao mostrado.

Figura 15. Safe link verifyed

3. *Verificar o método de recepção do link*

   Para receber o link e trata-lo de forma apropriada a aplicação tem de implementar a *aplicaçãolication:continueUserActivity:restorationHandler:*. Devemos portanto procurar este método para garantir que a aplicação passa o teste.

   Adicionalmente, quando a aplicação usar o método *openURL:options:completionHandler:* para abrir o link universal devemos verificar se o esquema da página URL é HTTP or HTTPS. Caso nã seja, se não for lançada uma excepção a aplicação falha o teste.

4. *Verificar o método de processamento de dados*

   Devemos verificar como são validados os dados recebidos visto que uma má verificação fornece um vetor de ataque a terçeiros. Assim todos os parâmetros fornecidos no URL não devem ser aceites sem antes serem validados. A API NSURLComponents pode ser usada para verificar e manipular os componentes do URL. Por fim deve ser verificado que as ações requeridas pelo URL não expõem dados sensíveis.

5. *Verificar se a aplicação está a usar os links universais de outra aplicação*

   Uma aplicação pode utilizar links de outra aplicação para transferir informação ou desencadiar alguma ação especifica, neste caso devemos garantir que não há vazamento de informação. Podemos para isso procurar o método *openURL:options:completionHandler:* e verificar os dados manipulados. Em baixo temos um exemplo do que foi descrito em cima. Para este exemplo usamos o rabin2 para procurar nos ficheiros binários da aplicação.

```
$ 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:
```

Figura 16. Result of searching for *openURL:options:completionHandler:* with rabin2

Note-se que a aplicação adapta o link para HTTPS antes de o usar e como só usa o URL caso este seja um link universal válido e só o abre caso haja uma aplicação capaz de o abrir.

Como esperado o método *openURL:options:completionHandler:* está implementado. Agora basta-nos garantir que dados sensíveis não são trocados entre as aplicações.

Para imprimir os dados tansmitidos vamos usar o comando:

```
$ xcrun swift-demangle S10TelegramUI15openExternalUrl7account7
context3url05for18applicationContext20navigationController12di
smissInputy0A4Core7AccountC_AA1412PresentationK0CAA0a11Applica
tionM0C7Display010NavigationO0CSgyyctF
```

Este comando apenas imprime o método *TelegramUI.openExternalUrl* sem os parâmetros que lhe são passados, para que imprima mais do que isso temos de fazer alguns ajustes.

Primeiro vamos imprimir os parâmetros passados, alterando o ficheiro stub com se vê na imagem e o respetivo resultado de voltar a correr o comando anterior.

```
// __handlers__/TelegramUI/_S10TelegramUI15openExternalUrl7_b1a3234e.js

  onEnter: function (log, args, state) {

    log("TelegramUI.openExternalUrl(account: TelegramCore.Account,
        context: TelegramUI.OpenURLContext, url: Swift.String, forceExternal:
        presentationData: TelegramUI.PresentationData,
        applicationContext: TelegramUI.TelegramApplicationContext,
        navigationController: Display.NavigationController?, dismissInput: ()
    log("\taccount: " + ObjC.Object(args[0]).toString());
    log("\tcontext: " + ObjC.Object(args[1]).toString());
    log("\turl: " + ObjC.Object(args[2]).toString());
    log("\tpresentationData: " + args[3]);
    log("\tapplicationContext: " + ObjC.Object(args[4]).toString());
    log("\tnavigationController: " + ObjC.Object(args[5]).toString());
  },
```

Figura 17. Code to print the parameters passed to *TelegramUI.openExternalUrl*

```
298382 ms  -[AppDelegate application:0x10556b3c0 continueUserActivity:0x1c4237
               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.Bo
presentationData: TelegramUI.PresentationData, applicationContext:
TelegramUI.TelegramApplicationContext, navigationController: Display.Navigatio
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
```

Figura 18. Parameters passed to *TelegramUI.openExternalUrl*

Aqui podemos observar que de facto o Telegram chama o método "aplicação-lication:continueUserActivity:restorationHandler:"como esperado e que apenas trata o URL sem o abrir, chamando o *TelegramUI.openExternalUrl* para isso.

Agora que confirmamos que o Telegram está a aplicar os métodos corretamente falta ver se temos dados sensíveis a serem trocados entre aplicações. Para isso, basta adicionar ao ficheiro stub o seguinte código: "log("userInfo:"+ ObjC.Object(args[3]).userInfo().toString())". Este código permite aceder à propriedade userInfo a partir do objeto continueUserInfo e como consequência imprime todos os dados trocados nos quais podemos verificar se existem ficheiros sensíveis. Na imagem seguinte vemos que não há dados sensíveis a serem passados.

```
298382 ms  -[AppDelegate application:0x10556b3c0 continueUserActivity:0x1c4237
               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>
```

Figura 19. Print proving the safe implementation of Telegram

10.6 App Extensions

Para podermos analisar quais os testes a fazer temos primeiro de compreender o que são as extensões e como interagem com as aplicações.

Dependendo da tarefa, a extensão da aplicação vai ter um tipo especifico (apenas 1), o chamado ponto de extensão. Alguns exemplos são:

. *Teclado personalizado:* substitui o teclado por um teclado personalizado para usar em todas as aplicaçãos.

. *Partilha:* postagem para um site ou partilha de conteudos com terceiros.

. *Hoje:* conhecidos como *widgets*, oferecem conteudo ou realizam tarefas rápidas no centro de notificações.

A forma de interação com outras aplicações também pode variar:

. *App extension:* É a extensão confinada dentro de uma aplicação. As aplicações de terceiros usam esta extensão.

. *Host aplicação:* Trata-se da aplicação de terceiros que usa a extensão de outra aplicação.

. *Containing aplicação:* Trata-se da aplicação que contem a extensão da aplicação instalada nela.

Para melhor se percecionar a interação entre aplicações apresenta-se a imagem seguinte.

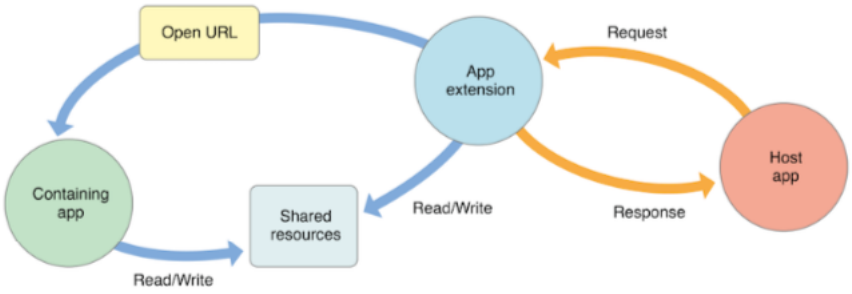

Figura 20. Aplication's shared URL implementation

10.6.1 Determinar Exposição de Métodos Nativos através de WebViews (MSTG-PLATFORM-7)

Desde o iOS a Apple introduziu APIs que permitem a comunicação entre o JavaScript na página web e o Swift ou Objective-C. Se as APIs forem usadas de forma descuidada funcionalidades podem ser expostas através de ataques de Cross-Site Scripting.

Vamos então testar a UIWebView do JavaScript. Para a testarmos temos de compreender que há 2 formas fundamentais para a comunicação do JavaScript:

. *JSContext:* Quando é atribuido um identificador a um bloco Swift ou Objective-C num contexto JSContext, o bloco é envolto em uma função JavaScript.

. *JSExport protocol:* métodos de instancia e de classes declarados no JSExport são mapeados para objetos JavaScript globais. Modificações aos objetos JavaScript são refletidas no ambiente nativo.

Devemos portanto verificar o código que resulta do mapeamento para o JSContext para ver que funcionalidades são expostas para garantir que dados sensíveis não podem ser acedidos e expostos na WebView.

Em Objective-C o JSContext associado a uma UIwebView pode-se obter com o comando:

```
[webView valueForKeyPath:@"documentView.webView.mainFrame.javaScriptContext"]
```

Para testarmos as funções devemos produzir código JavaScript para injetar num ficheiro que a aplicação utilize. Podem ser aplicadas várias técnicas para realizar isto. Vamos apresentar duas:

1. Se algum do conteúdo for carregado de forma insegura da internet por HTTP, podemos tentar implementar um ataque man-in-the-middle onde fornecemos o nosso código.

2. Podemos realizar uma injeção do código ao usar frameworks como a Frida e as funções de avaliação do javaScript correspondentes a cada WebView:
   (a) Usar "stringByEvaluatingJavaScriptFromString:"para a UIWebView

   (b) Usar "evaluateJavaScript:completionHandler:"para a WKWebView

Para efeitos de demonstração foi usada uma aplicação chamada "where's My Browser?", que fornece um serviço de calculadora web e que possui um método que, quando chamado, devolve um segredo (informação sensivel).

Para expor o segredo da aplicação "where's My Browser?"podemos usar uma das técnicas mencionadas para injetar o seguinte código:

```
function javascriptBridgeCallBack(name, value) {
          document.getElementById("result").innerHTML=value;
};

window.webkit.messageHandlers.javaScriptBridge.postMessage(["getSecret"]);
```

O resultado pode ser visto na imagem seguinte.

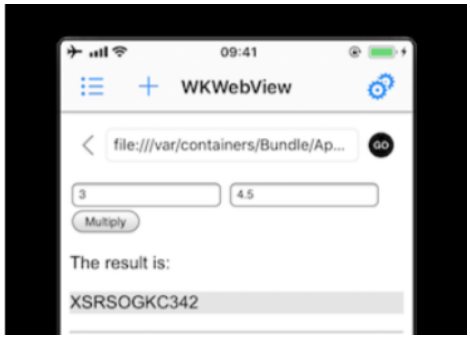

Figura 21. Leaked Secret through javaScript code injection

10.6.2 Testar Updates Forçados (MSTG-ARCH-9)

Este teste tem por base a realização de updates. Como a Apple ainda não fornece uma API para automatizar o processo de atualização de aplicações os desenvolvedores terão de o programar.

A primeira coisa a ser verificada é a presença de um mecanismo de atualização. Sem este os utilizadores não serão forçados a atualizar a aplicação.

Caso haja o dito mecanismo temos de verificar se este força a atualizar para "always latest". Temos também de verificar que de cada vez que se inicia a aplicação, esta passa pelo mecanismo de update para garantir que o mecanismo não pode ser ultrapassado.

Para testar o mecanismo devemos tentar instalar uma versão da aplicação que tenha uma vulnerabilidade. De seguida devemos iniciar a aplicação e verificar se esta corre ou se nos obriga a atualizar. Se uma janela de atualização for mostrada temos de verificar se fechando-a conseguimos continuar a correr a aplicação e, caso esta continue a correr, se o backend está preparado para não nos permitir aproveitar a vulnerabilidade bloqueando-nos o acesso enquanto a aplicação não for atualizada.

Se todas as tentativas para explorar a vulnerabilidade falharem a aplicação foi bem sucedida.

Referências

1. https://www.theiphonewiki.com/wiki/DFU_Mode
2. https://www.lifewire.com/what-is-jailbreaking-2377420
3. https://mobile-security.gitbook.io/mobile-security-testing-guide/ios-testing-guide/0x06j-testing-resiliency-against-reverse-engineering
4. https://developer.apple.com/documentation/foundation/nsurlconnection
5. https://developer.apple.com/documentation/foundation/nsurlsession
6. https://developer.apple.com/documentation/corefoundation/cfurl-rd7
7. Bernhard Mueller, Sven Schleier, Jeroen Willemsen, Carlos Holguera, The OWASP mobile team. MSTG - Mobile Security Testing Guide. The OWASP Foundation. 2 August 2019