MSTG - Teste de aplicações Android e iOS

Departamento de Informática, Universidade do Minho

Resumo Palavras-chave: MSTG · Android · iOS.

Introdução

1iOS

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, e um controlo de acesso mandatorio(mac) descreve os recursos que a aplicação pode usar.

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

1 Aspetos de Segurança do iOS

1.1 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.

1.2 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].

1.3 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]

1.4 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 á 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].

1.5 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.

2 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.

3 TESTES DE SEGURANÇA EM APLICAÇÕES

3.1 Data Storage on iOS

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

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

Secure Enclave Processor (SEP):

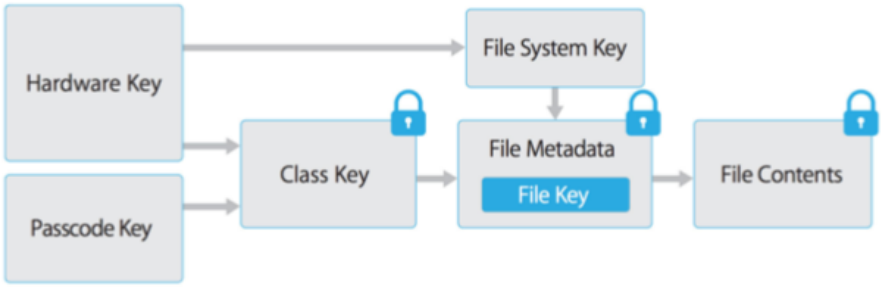

Figura 1. Secure Enclave Processor Scheme

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 á 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 á 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.

3.1.2 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.

3.1.3 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 á da figura para os UITextFields visados. Se estes não tiverem a configuração mostrada falham o teste.

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

Figura 2. Secure Text Entry Example

Para corigr 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 3. Set a secure Text Entry

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

1. Fazer reset á 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.

3.1.4 Checking for Sensitive Data Disclosed Through the User Interface (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

á 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.

3.2 iOS Cryptographic APIs

3.2.1 Verifying the Configuration of Cryptographic Standard Algorithms (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 4. Available Hash Algorithms

. *Symmetric-Key:*

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

Figura 5. Available Symmetric-key Algorithms

. *Public-Key:*

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

Figura 6. 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 7. 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.

3.2.2 Testing Key Management (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.

3.2.3 Testing Random Number Generation (MSTG- CRYPTO-6)

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

Em Swift a chamada á 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.*

3.3 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.

3.3.1 Testing Local Authentication (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 sensíveissão 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 á 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 á 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.

3.4 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.

3.4.1 App Transport Security (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 8. 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.

3.4.2 Testing Custom Certificate Stores and Certificate Pinning (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 confiaveis. 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 á 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 á 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.

3.5 iOS Platform APIs

3.5.1 Testing App Permissions (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 á 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.

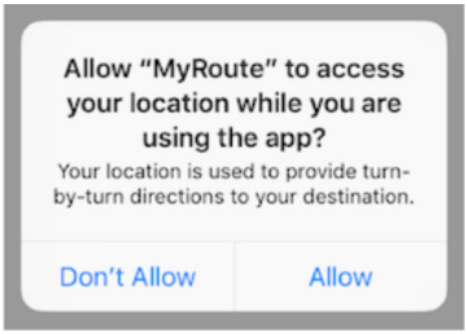

Figura 9. Example of permission request message

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 10. 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 11. 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 12. 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.

3.5.2 Testing for Sensitive Functionality Exposure Through 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 á inspeção da aplicação Telegram.

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

Figura 13. 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 14. 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 15. 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 16. 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 17. 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 á 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 18. Print proving the safe implementation of Telegram

3.6 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.

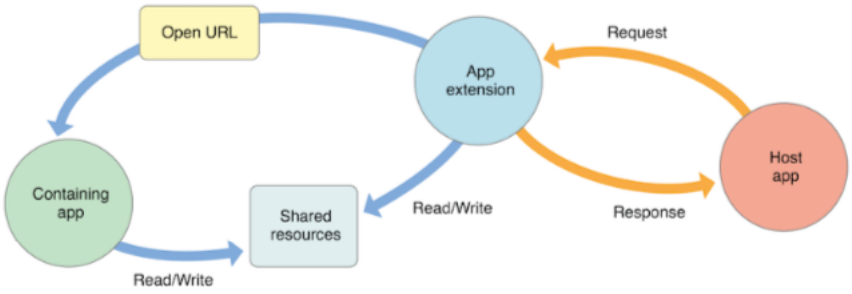

Figura 19. Aplication's shared URL implementation

3.6.1 Determining Whether Native Methods Are Exposed Through 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.

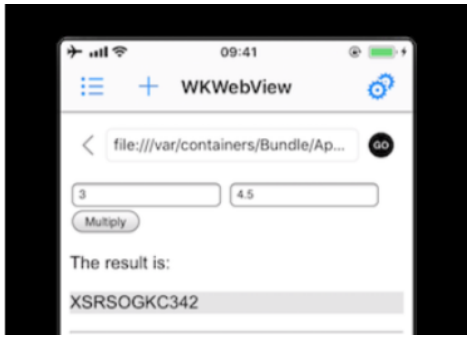

Figura 20. Leaked Secret through javaScript code injection

3.6.2 Testing enforced updating (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