Uma Análise de Vulnerabilidades em Compras Dentro de Apps iOS

Izabella C. Melo[1], **Divanilson R. Campelo**[1]

[1]Centro de Informática – Universidade Federal de Pernambuco (UFPE)
Caixa Postal 7851 – 50732-970 – Recife – PE – Brasil

{imcm,dcampelo}@cin.ufpe.br

Abstract. *Monetization strategies are crucial for the sustainability and growth of mobile app-based businesses. Among the primary monetization approaches is the in-app purchase, which allows developers to offer premium features to users who want to pay for them. Consequently, in recent years, the App Store has become a goldmine for developers, being up to twice as profitable as its largest competitor. But could these revenues be even higher? In this study, we analyzed the in-app purchase implementations of 34 iOS apps available in the Top Free section of the App Store across three categories. Our analysis involved identifying potential vulnerabilities and subsequently testing them for confirmation, resulting in 3 main groups of vulnerabilities. Out of the 34 apps analyzed, 21 (or 60%) contained vulnerable implementations that allowed us to access at least one premium feature without paying.*
Keywords: in-app purchases, mobile apps, app development, iOS, mobile security

Resumo. *Estratégias de monetização são essenciais para a sustentabilidade e expansão de negócios baseados em apps móveis. Dentre as principais abordagens de monetização, destaca-se a compra dentro do apps (in-app purchase), que possibilita ao desenvolvedor oferecer funcionalidades premium para usuários dispostos a pagar. Com isso, nos últimos anos, a App Store se consolidou como uma mina de ouro para desenvolvedores, sendo até duas vezes mais rentável que sua maior concorrente. Mas esses valores poderiam ser ainda maiores? Nesse trabalho, realizamos uma análise acerca das implementações de compras dentro de 34 apps iOS disponíveis entre os Top Free da App Store em três categorias. Nossa análise abrangeu a identificação de potenciais vulnerabilidades e posterior teste para confirmação, o que resultou em 3 grupos de vulnerabilidades principais. Dos 34 apps analisados, 21 apps (ou 60%) continham implementações vulneráveis que nos permitiram ter acesso a pelo menos uma funcionalidade premium sem pagar por ela.*
Palavras-chave: compras dentro de apps, apps móveis, desenvolvimento de apps, iOS, segurança móvel

1. Introdução

Dispositivos móveis se tornaram nos últimos anos os equipamentos mais usados para comunicação e consumo de conteúdo digital no mundo. Em 2023, usuários gastaram em todo mundo cerca 171 bilhões de dólares em lojas de apps móveis [Data.ai 2024].

A maior parte desses gastos vem das duas principais lojas de apps móveis: App Store e Google Play. Entretanto, embora o Google Play seja responsável pelo maior número de downloads dado o grande número de dispositivos Android, a App Store se consolidou como a loja de apps mais rentável do mundo, com duas vezes mais ganhos do que a sua concorrente [SensorTower 2022].

Uma parte significativa dos gastos de usuário dentro de lojas de apps vem de estratégias de monetização eficazes. Entre elas, os dois principais modelos são os apps pagos, no qual usuários devem pagar para baixar o app da loja; e apps "freemium", que incluem apps gratuitos para baixar mas que oferecem recursos premium ou features adicionais por meio de compras dentro do app (in-app purchases). A implementação desse último modelo de monetização está disponível para os desenvolvedores iOS a partir do iOS 3.0 por meio de API's StoreKit disponibilizadas pela Apple [AppleDeveloper 2024f]. Em contrapartida, para cada transação realizada com sucesso por meio dessa API, a Apple cobra uma taxa de comissão de 15% a 30%, tornando a App Store parte importante dos lucros anuais da Apple [Munster 2023].

Com a grande quantidade de receita gerada pela App Store, em especial por meio de compras dentro de apps iOS, preocupações acerca de transações fraudulentas emergiram. Em maio de 2024 a Apple anunciou que detectou e bloqueou, nos últimos 4 anos, mais de 7 bilhões de dólares em transações potencialmente fraudulentas, que incluiam avaliações e contas falsas, distribuição de apps não confiáveis e uso de cartões de crédito roubados [Apple 2024]. O relatório da Apple, no entanto, não menciona fraudes em compras dentro de apps ou quaisquer tentativas de obtenção de acesso gratuito a conteúdos premium sem pagar, o que significa que o os números totais de transações fraudulentas podem ser ainda maiores. Na realidade, é possível que esses números sejam também em grande parte indetectáveis, uma vez que um meio de obter acesso inapropriado a esses conteúdos é pela exploração de implementações vulneráveis feitas pelos desenvolvedores em checagens no lado do app, que não passam pela Apple e, portanto, seriam indetectáveis por ela.

1.1. Este trabalho e Desafios

Nesse contexto, o presente trabalho tem como objetivo analisar um conjunto de apps iOS disponíveis da App Store e identificar vulnerabilidades em implementações de compras dentro de apps que permitam a um atacante acessar funcionalidades pagas de forma gratuita, o que pode resultar em potencial prejuízo financeiro para empresas e desenvolvedores. Para isso, nós analisamos um total de 34 apps nativos (feitos em Swift ou Objective-C) coletados de três categorias da App Store: Health & Fitness, Productivity e Education. Esses apps foram coletados com base em uma série de critérios, como serem gratuitos para baixar e possuirem funcionalidades que possam ser desbloqueadas por meio de compras dentro do app. Além disso, as funcionalidades não deveriam depender inerentemente do servidor, como em apps de armazenamento, ou estarem associadas a uma conta de assinatura externa, como apps de streaming como a Netflix. O objetivo desse trabalho é identificar erros de implementação do lado do cliente capazes de tornar as implementações de compra de apps vulneráveis.

Ao final da coleta de apps, iniciamos uma análise dividida em duas etapas: inicialmente realizamos uma análise estática, focada numa investigação de código e

identificação de potenciais vulnerabilidades, e, em seguida, realizamos uma análise dinâmica como forma de realizar as modificações necessárias para explorar e validar as potenciais vulnerabilidades identificadas anteriormente. Consideramos um app vulnerável se tiver sido possível obter acesso a pelo menos uma funcionalidade paga sem pagar por ela. Como resultado, nesse trabalho fomos capazes de obter acesso indevido a funcionalidades premium de 21 dos 34 apps analisados, o que representa mais de 60% do conjunto de apps, mostrando a necessidade de atenção acerca dos conhecimentos sobre desenvolvimento seguro de desenvolvedores iOS.

Durante essas etapas, enfrentamos uma série de desafios, em especial devido a restrições da Apple e do iOS. Tivemos dificuldades em expandir nosso estudo para um número maior de apps dada a impossibilidade de baixá-los automaticamente da App Store. Além disso, as análises posteriores também foram feitas de maneira manual, desde a análise do código até os testes de vulnerabilidades. No entanto, fomos capazes de vencer esses desafios e apresentar a primeira análise que mostra o estado de (in)segurança das implementações de compras dentro de apps iOS.

A Seção 2 apresenta as APIs para implementação de compras dentro de apps iOS e o passo a passo de implementação. Na Seção 3 é apresentado o potencial atacante e seu modelo de ataque. Em 4 é apresentada a metodologia desse trabalho, das ferramentas e das análises realizadas. Por fim, em 5 são mostrados os resultados, em 6 os trabalhos relacionados e em 7 a conclusão e trabalhos futuros.

2. Background Técnico

Atualmente a Apple oferece duas APIs para implementação de compras dentro de apps iOS:

- A *Original API for in-app purchase*, disponível desde o iOS 3.0, que funciona a partir de um recibo criptografado para validação de compras feitas dentro de apps [AppleDeveloper 2024d].
- A *In-App Purchase API*, disponível a partir do iOS 15.0, que usa JSON Web Signature (JWS) para assinar e validar transações com a App Store [AppleDeveloper 2024b].

Embora a In-App Purchase API, lançada durante a Apple Worldwide Developers Conference 2021 (WWDC21), ofereça uma opção mais moderna e segura para validação de transações [Apple 2021], a Original API for in-app purchase continua disponível, especialmente para apps de longa-data que suportem dispositivos com iOS anterior ao 15.0. A própria documentação da Apple estabelece opções em que o uso da Original API for in-app purchase ainda é necessário [AppleDeveloper 2024a].

Assim, neste trabalho nós focaremos exclusivamente em vulnerabilidades introduzidas durante a implementação da Original API for in-app purchase. Algumas das vulnerabilidades descritas aqui podem ser independentes do design da API e também estarem presentes na In-App Purchase API, mas nós consideramos essa API fora do escopo desse trabalho. Dessa maneira, a seção seguinte focará apenas na implementação da Original API for in-app purchase.

2.1. Original API for in-app purchase

A Fig. 1 mostra as etapas de implementação da Original API for in-app purchase em um app iOS com base na documentação oficial da Apple [AppleDeveloper 2024c]. Cada etapa é numerada de acordo com as fases descritas em seguida.

Figura 1. Fluxo para implementação da Original API for in-app purchase

2.1.1. Configuração Inicial na App Store Connect

A primeira etapa é focada inteiramente na App Store Connect, onde o desenvolvedor deve aceitar uma série de termos e contratos com a Apple. Após isso, o desenvolvedor pode iniciar as configurações criando um usuário sandbox para testar suas compras sem gastar dinheiro real.

2.1.2. Criação de Produtos

Ainda na App Store Connect, o desenvolvedor deve criar a lista de produtos que serão vendidos no seu app. Para cada produto, é necessário escolher o tipo, nome e seu Product ID, como mostrado na Fig. 2. Os tipos de produto existentes são:

- Consumível, produtos que podem ser usados apenas uma vez após a compra;

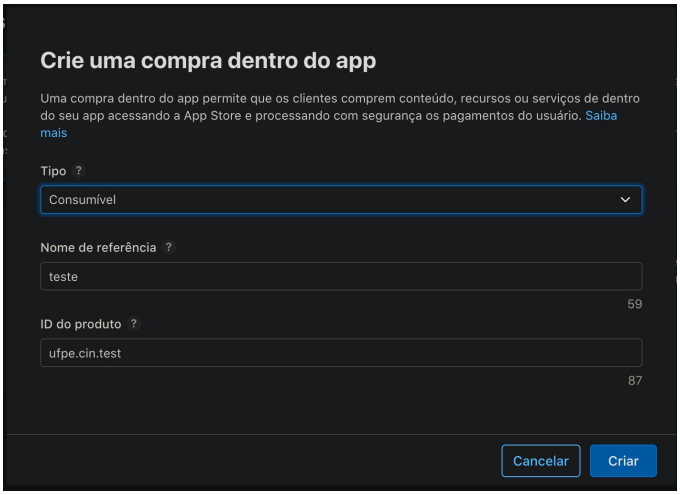

Figura 2. Criação de produtos dentro da App Store Connect

- Não-Consumíveis, produtos que podem ser usados indefinidamente;
- Assinaturas Auto-Renováveis, assinaturas que renovam automaticamente ao final do período contratado;
- Assinaturas Não-Renováveis, assinaturas que são finalizadas ao final do período contratado.

Finalmente, também é possível escolher o preço e as ofertas em diversas moedas disponíveis. Essas configurações são feitas nos servidores da Apple e podem ser alteradas remotamente sem que seja necessário lançar uma atualização do app.

2.1.3. Obtenção de Produtos e Exposição para Venda

Após a criação dos produtos, o desenvolvedor deve usar os Product IDs criados anteriormente para recuperar os produtos e suas informações no códido do app. A lista dos Product IDs é passada no objeto `SKProductsRequest` e a resposta vem através do método delegado `productsRequest(_:didReceive:)`. Os objetos retornados nesse método possuem informações como nome, descrição e preço do produto para que seja possível disponibilizá-lo para venda no app. Mais uma vez, esses dados são modificados remotamente através das configurações da App Store Connect, o que garante certa flexibilidade para os desenvolvedores.

2.1.4. Realização de uma Compra

Para que seja possível realizar uma compra, o app deve checar o retorno do método `SKPaymentQueue.canMakePayments()`. Dispositivos com controle parental ativado, por exemplo, podem não ser habilitados para realizar compras e o ideal é que o app modifique sua interface para se adequar a isso. Por outro lado, se o dispositivo estiver habilitado para compras, o desenvolvedor deve passar o Product ID do produto escolhido pelo usuário para ser processado numa `SKPaymentQueue`. A resposta dessa transação é recebida na forma de um objeto `SKPaymentTransaction` que contém

um `transactionState`. O `transactionState` é um enum capaz de assumir qualquer um dos valores abaixo:

- `purchasing` (= 0): A App Store começou a processar a transação;
- `purchased` (= 1): A App Store processou a transação com sucesso;
- `failed` (= 2): A App Store falhou ao processar a transação;
- `restored` (= 3): O conteúdo já havia sido comprado anteriormente;
- `deferred` (= 4): A transação necessita de uma ação externa.

2.1.5. Validação de uma Transação

Apesar de retornar o resultado por meio do `transactionState`, ele sozinho não deve ser suficiente para confirmar que a compra foi realizada. Se o retorno desse resultado for sucesso, o desenvolvedor deve usar o `appStoreReceiptURL` para obter o recibo criptografado gerado pela App Store. Esse recibo contém informações sobre a compra e é essencial para validar com segurança se a compra foi realizada antes da liberação da funcionalidade.

2.1.6. Persistência de uma Compra

Por último, a documentação oficial da Apple recomenda que o desenvolvedor armazene a compra para que a funcionalidade esteja disponível mesmo que o usuário não possua acesso à internet. Algumas formas descritas pela documentação oficial são: armazenar o recibo criptografado gerado pela App Store, o que está disponível apenas para alguns tipos de compras, ou salvar um booleano (true/false) no iCloud ou no UserDefaults [AppleDeveloper 2024e, AppleDeveloper 2024g].

3. Modelo de Ataque

Nesse trabalho, definimos como modelo de ataque a descrição das possíveis ameaças a um sistema, detalhando o atacante, as ferramentas usadas e o modo que ele pode usar para explorar as vulnerabilidades do app. Assim, identificamos como modelo de ataque o fluxo mostrado na Fig. 3, que consiste em um cenário onde o atacante tem acesso ao arquivo IPA (iOS App Store Package) descriptografado do app iOS. O IPA é em um arquivo comprimido que contém o binário do app e recursos adicionais como imagens, sons, etc. Ao ser enviado para a App Store, o IPA é criptografado pelo protocolo FairPlay DRM (Digital Rights Management), o que garante que ele não seja modificado em trânsito ou após baixado [Foresman 2012]. No entanto, para que o app seja executado o IPA precisa ser descriptografado, o que ocorre quando ele é carregado na memória do iPhone. Para obter o IPA descriptografado, o atacante deve recuperá-lo da memória de um dispositivo com jailbreak ou obtê-lo a partir de sites que fornecem esse tipo de serviço online [iOSGods 2024a, und3fy.dev 2024].

Com o IPA descriptografado em mãos, o atacante pode analisar o código desmontado e decompilado para identificar potenciais vulnerabilidades. Após isso, é possível usar ferramentas como Frida [Frida 2024] para realizar modificações no app em tempo de

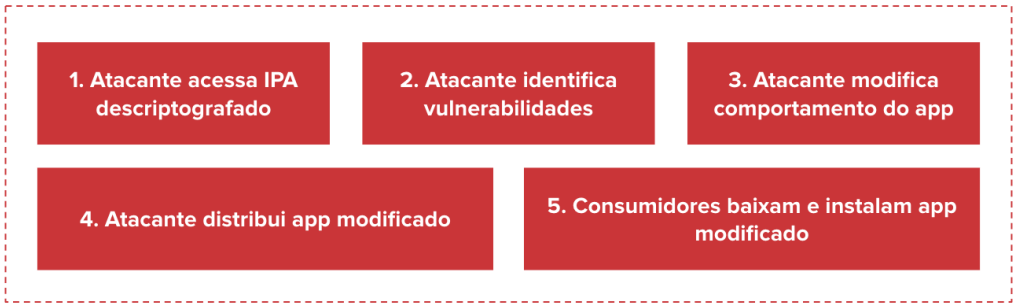

Figura 3. Modelo de ataque ilustrado

execução e testar as vulnerabilidades. Outra forma de testar essas vulnerabilidades é modificar o binário do app diretamente, uma alternativa mais complexa que envolve a necessidade de reassinar o app com um certificado válido da Apple, mas que torna o processo de distribuição do app modificado mais fácil. O app modificado pode ser posteriormente distribuído por meio de lojas de apps alternativas e os usuários podem baixá-lo sem a necessidade de um dispositivo com jailbreak [iOSGods 2024b, Rook 2019, iOSGods 2024c].

4. Metodologia

No presente trabalho nós usamos um iPhone 7 com iOS 14.6 com jailbreak para baixar, instalar e obter os IPAs descriptografados para posterior análise. A escolha de uma versão mais antiga do iOS foi intencional e nos permitiu evitar distrações com problemas relacionados a jailbreaks instáveis em versões mais recentes do sistema, o que também nos possibilitou usar o Frida iOS Dump [AloneMonkey 2024] para obter os IPAs e o Frida [Frida 2024] para testar as vulnerabilidades. No entanto, vale observar que os achados desse trabalho são focados em vulnerabilidades introduzidas nas implementações dos apps. Essas vulnerabilidades pertencem ao app e não podem ser resolvidas por meio de uma atualização de sistema operacional, o que significa que a versão do iOS não deve interferir nos cenários mostrados nesse trabalho, que devem continuar aplicáveis independente do dispositivo ou iOS usados.

Após ter acesso ao IPA descriptografado, nossa metodologia seguiu dividida em duas partes: uma análise estática e uma análise dinâmica, conforme mostrado na Fig. 4. Durante a análise estática, realizamos a desmontagem e descompilação do código do app com Hopper [Cryptic 2024] para analisar e identificar potenciais vulnerabilidades. Após isso, durante a análise dinâmica, usamos o Frida para realizar modificações necessárias para explorar essas vulnerabilidades e observar o comportamento do app.

4.1. Análise Estática

O primeiro passo para realizar a análise estática foi identificar trechos de códigos ligados a implementação de compras dentro de apps iOS. Para isso, procuramos por palavras que remetessem a essas funcionalidades, como “premium”, “vip”, “plus”, etc. Após localizar os trechos de código relacionados, analisamos cada um isoladamente e tentamos identificar potenciais falhas de segurança na implementação que pudéssemos explorar. Os códigos analisados resultaram em uma série de possíveis vulnerabilidades que puderam ser divididas em três grandes grupos:

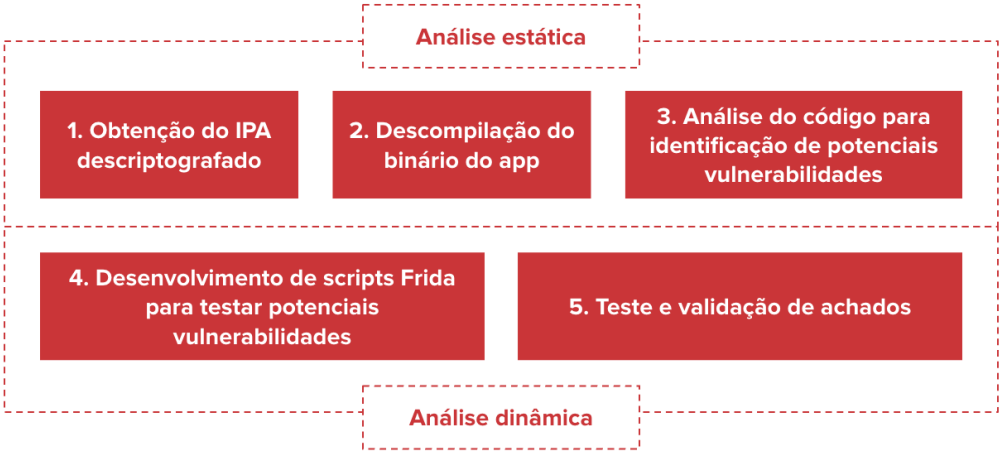

Figura 4. Metodologia ilustrada

- **Falta de Validação de Recibo**, ou seja, quando desenvolvedores não seguiram todos os passos necessários para validar a transação e não fizeram a validação do recibo criptografado disponibilizado pela Apple;
- **Código Desprotegido**, quando fomos capazes de ler e identificar facilmente trechos de códigos sensíveis e de verificação. Isso inclui a presença de código de desenvolvimento em ambiente de produção, que permitiu acessar funções que deveriam ser exclusivas de desenvolvimento e testes;
- **Armazenamento Inseguro**, que inclui o uso de métodos de armazenamento inseguro para salvar valores sensíveis. Esses tipos de armazenamento geralmente permitem a leitura e modificação de dados armazenados de maneira simples.

Abaixo, vamos descrever em mais detalhes como chegamos em cada um dos problemas acima.

4.1.1. Falta de Validação de Recibo

Das categorias acima, a Falta de Validação de Recibo é o único problema inerente ao design da Original API for in-app purchase. Como introduzido na Seção 2.1.5, o resultado da compra é recebido através da variável `transactionState`, que pode assumir uma série de valores relacionados ao estado da transação de compra dentro do app.

Após receber uma transação com valor `purchased` (= 1), a Apple recomenda verificar o recibo da App Store, pois ele é criptografado e não pode ser modificado. Assim, é evidente que a ausência de verificação do recibo representa um potencial risco de segurança. Embora não demostrado explicitamente através da análise estática do código, essa falha de implementação, inerente ao design da API, apresenta riscos significativos de segurança e pode ser facilmente testada posteriormente durante a análise dinâmica.

4.1.2. Código Desprotegido e Armazenamento Inseguro

Ao contrário da Falta de Validação de Recibo, as outras duas categorias podem ser melhor exemplificadas olhando o código decompilado de um app analisado.

A Fig. 5 mostra o pseudocódigo da função `ISVIPUser`, responsável por checar se um usuário tem acesso premium ao app. A função retorna um booleano (true/false) que pode ser facilmente modificado usando Frida. O Frida apresenta ferramentas capazes de interceptar a chamada para funções específicas e ver e modificar os argumentos passados ou o resultado obtido. A possibilidade de identificar facilmente uma função de verificação crítica como essa é o que definimos como Código Desprotegido. Embora isso possa concluir nossa análise a procura de uma vulnerabilidade que possa ser explorada, fomos mais adiante para exemplificar outra vulnerabilidade existente nesse mesmo código.

Figura 5. Pseudo-código da função `ISVIPUser` decompilada

No bloco 1 marcado na Fig. 5, a função procura a chave `Purchased` em `SAMKeychain` e checa se seu valor é `YES`. O objeto `SAMKeychain` remete ao Keychain, uma forma de armazenamento segura e criptografada. Se o valor obtido no Keychain for `YES`, o código entende que a compra foi realizada e libera os conteúdos premium.

No entanto, o bloco marcado como 2 mostra uma outra verificação feita caso o retorno do Keychain seja diferente de `YES`. Dessa vez, a verificação é feita acessando uma chave no UserDefaults. Nesse momento, há uma falha na funcionalidade de pseudo-código do Hopper Disassembler e é necessário analisar também o código assembly na Fig. 6. No bloco marcado com número 3, vemos que o app procura a chave `VPNExpired` no UserDefaults. De forma semelhante, se o UserDefaults retornar o valor 0, o app entende que deve liberar o conteúdo premium. Caso contrário, o modo premium é desativado.

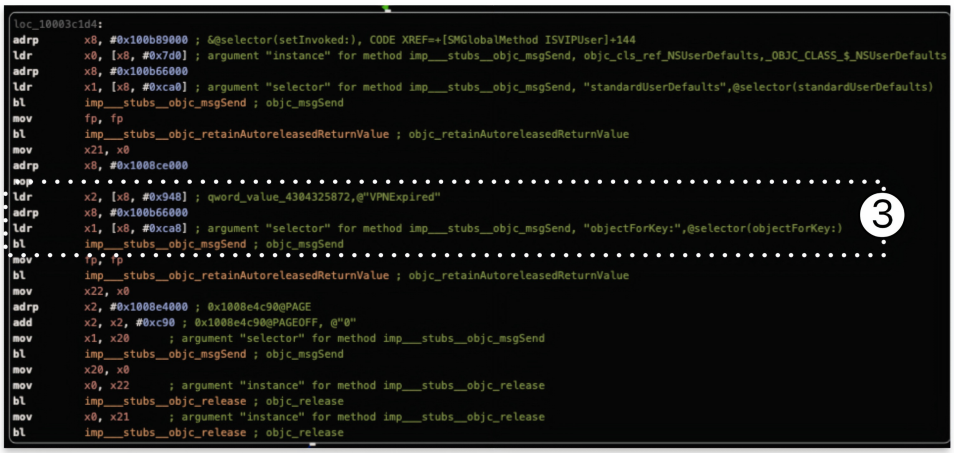

Figura 6. Código Assembly `ISVIPUser` da função decompilada

Ao contrário do Keychain usado corretamente no bloco 1 do código, o UserDefaults, apesar de ser uma das formas mais famosas e simples de salvar dados em apps iOS, não deve ser usado para salvar dados sensíveis como o estado premium do app, mesmo que a documentação da Apple sugira isso [AppleDeveloper 2024e, AppleDeveloper 2024g]. O UserDefaults armazena dados em um arquivo em texto plano no formato chave-valor e pode ser facilmente lido e modificado por um atacante. Do ponto de vista da segurança, o dado armazenado corretamente com o uso do Keychain visto no bloco 1 da Fig. 5 tornou-se irrelevante devido à verificação adicional feita de forma insegura logo depois.

4.2. Análise Dinâmica

Após identificar e categorizar as vulnerabilidades potenciais na etapa de análise estática na Seção 4.1, nós criamos scripts com Frida para modificar os códigos de acordo com o necessário e explorar cada uma dessas vulnerabilidades. A seguir, mostraremos os códigos e o funcionamento deles de acordo com o que foi analisado anteriormente.

4.2.1. Explorando a Falta de Validação de Recibo

Conforme dito anteriormente, a Falta de Validação de Recibo acontece quando o desenvolvedor não executa a verificação necessária do recibo criptografado gerado pela App Store para validar a transação e confia somente no valor retornado pela variável `transactionState` do objeto `SKPaymentTransaction`. Assim, a

exploração dessa vulnerabilidade consiste na modificação do valor retornado pela variável `transactionState` para ser sempre `purchased`, representado pelo valor 1, como visto na Fig. 7

```
1 var instance = ObjC.classes.SKPaymentTransaction["-
     transactionState"];
2 Interceptor.attach(instance.implementation, {
3     onLeave: function(oldValue) {
4         var newValue = ptr('0x1');
5         oldValue.replace(newValue);
6     }
7 });
```

Figura 7. Código Frida para modificar valor do `transactionState`

4.2.2. Explorando o Armazenamento Inseguro

Como também falado anteriormente, valores salvos no UserDefaults podem ser facilmente modificados. Uma forma simples de fazer isso é salvando um novo valor por cima do valor antigo com a mesma chave. Na Fig. 8 mostramos um código Frida que salva o valor 0 para a chave `VPNExpired`, permitindo explorar a vulnerabilidade comentada na Seção 4.1.2.

```
1 var ud = ObjC.classes.NSUserDefaults;
2 ud.standardUserDefaults().setBool_forKey_(0, 'VPNExpired');
```

Figura 8. Código Frida que salva valor 0 para `VPNExpired`

Com o valor 0 salvo com essa chave, o código recuperava o valor esperado por nós e liberava os conteúdos premium do app.

4.2.3. Explorando Código Desprotegido

O Código Desprotegido é a vulnerabilidade mais comum encontrada nos apps analisados, mas também a mais heterogênea. De maneira geral, classificamos como Código Desprotegido todos os casos em que o desenvolvedor falhou em proteger e ofuscar códigos críticos de verificação dentro do app, deixando-os acessíveis para atacantes. Isso também inclui códigos que deveriam ser usados apenas durante o desenvolvimento do app, para facilitar o teste e debug, e que não deveriam estar disponíveis no app enviado para a App Store. As formas de explorar essas vulnerabilidades podem ser bastante diferentes entre si, mas, na maior parte dos casos, nos deparamos com funções de checagem que retornavam um booleano para liberar ou não funcionalidades premium, como mostrado de maneira genérica na Fig. 9.

Do mesmo jeito que as funções são inúmeras, os códigos para explorá-las também podem se diferenciar bastante. No entanto, usando como exemplo o código alvo exibido na Fig. 9, podemos criar um código Frida para modificar o retorno da função `isPremium`

```
/* @class User */
+(bool)isPremium {
    r0 = [r0 subscription];
    if(([r0 isValid] & 0x1) != 0x0) {
        r21 = 0x1;
    } else {
        r21 = [r0 isSubscribed];
    }
    r0 = r21;
    return r0;
}
```

Figura 9. Função genérica de checagem de usuário premium

e forçá-la a retornar sempre true, o que significaria que o usuário é premium, como mostrado na Fig. 10

```
var instance = ObjC.classes.User
Interceptor.attach(
    instance.isPremium.implementation, {
        onLeave: function(oldValue) {
        var newValue = ptr(0x1);
        oldValue.replace(newValue);
    }
});
```

Figura 10. Código Frida para explorar função da Fig. 9

5. Resultados

Durante esse trabalho, nós baixamos e instalamos 34 apps coletados de Março a Setembro de 2022 dos Top Free apps de três categorias da App Store: Health & Fitness, Productivity e Education. Para cada app, nós decompilamos o binário, analisamos, identificamos potenciais vulnerabilidades e testamos formas de explorar essas vulnerabilidades. Como resultado, fomos capazes de identificar 3 principais categorias de vulnerabilidades que nos deram acesso a recursos premium sem pagar por eles em quase dois terços dos apps analisados.

Ao final desse trabalho, fomos capazes de driblar as implementações e checagens de 21 dos 34 apps analisados, o que representa mais de 60% deles. Esses resultados são comparáveis ao obtido pelo VirtualSwindle, um trabalho semelhante que focou em apps Android e foi capaz de driblar implementações de compras dentro de apps de 60% dos 85 apps examinados [C. Mulliner and Kirda 2014]. Vale observar, no entanto, que os resultados obtidos pelo VirtualSwindle representavam o estado da segurança das compras dentro de apps Android em 2014.

Além disso, fomos capazes de identificar os grupos de vulnerabilidades mais representativos dentro dos apps analisados. Em sequências, elas foram: i) Código Desprotegido (11 apps); ii) Armazenamento Inseguro (6 apps); e iii) Falta de Validação de Recibo (6 apps).

Tabela 1. Resultados compilados

Apps Analisados	Vulneráveis	Vulnerabilidade		
		Código Desprotegido	Armazenamento Inseguro	Falta de Validação de Recibo
34	21	11	6	6

Os resultados indicam que, de modo geral, os desenvolvedores de apps iOS têm pouca ou nenhuma preocupação com segurança ao criar suas aplicações. Muitas das vulnerabilidades identificadas poderiam ser facilmente mitigadas com pequenas melhorias nos processos de desenvolvimento. A adoção de práticas como o uso de métodos mais seguros para armazenamento de dados, a ofuscação de código e a implementação de soluções que sigam as recomendações da documentação oficial seriam medidas simples e eficazes para reforçar a segurança dos apps analisados.

6. Trabalhos Relacionados

A maioria dos estudos sobre segurança em dispositivos móveis foca em dispositivos Android devido às menores restrições do sistema operacional. Por exemplo, os autores de [M. Egele and Kruegel 2013] mostraram que 88% dos apps Android que utilizavam APIs criptográficas apresentavam ao menos um erro de implementação. Similarmente, os autores de [B. Reaves and Butler 2017] identificaram várias implementações inseguras em 46 apps de fintechs analisados, que comprometiam ativamente a segurança deles.

No contexto de compras dentro de apps, o VirtualSwindle [C. Mulliner and Kirda 2014] propôs um ataque automatizado para desbloquear recursos premium em apps Android em tempo de execução, substituindo o código da API do sistema pelo código dos autores. O VirtualSwindle teve uma taxa de sucesso de 60% dentre os 85 apps examinados, semelhante ao obtido no nosso trabalho, mas focando em apenas um tipo de vulnerabilidade. Outro estudo focado em apps Android foi o FreeMarket [D. Reynaud and Shin 2012], onde os autores desenvolveram uma técnica para reescrever apps sem verificações de compra no app e obtiveram sucesso em 59% dos 295 apps analisados. Essa técnica, no entanto, apresentou mais de 18% de falha e, nesses casos, o app reescrito parou de funcionar. Em outro estudo mais recente, PaymentScope [Zuo and Lin 2022] analisou jogos Android feitos com Unity. Os autores testaram estaticamente 39.121 binários compilados e detectaram que 23% deles analisados eram vulneráveis. Há ainda outros estudos que investigaram o ecossistema de pagamento móvel de terceiros, analisando SDKs chineses em milhares de apps Android com compras no app [W. Yang and Gu 2017, S. Shi and Lau 2021].

Embora tenham buscado identificar problemas nas compras dentro de apps, nenhum dos estudos anteriores buscou entender o estado atual das compras dentro de apps iOS, mesmo o iOS sendo um dos sistemas operacionais mais usados no mundo. Até onde sabemos, nosso trabalho é o primeiro a analisar o estado atual de segurança de implementações de compras dentro de apps iOS. Nossos resultados demostram uma porcentagem de apps vulneráveis semelhante aos trabalhos mais antigos em cima de apps Android, embora com foco em vulnerabilidades diferentes, dado as particularidades de cada sistema operacional.

7. Conclusão

Nesse trabalho, conduzimos o primeiro estudo sobre a segurança de implementações de compras dentro de apps iOS. Durante nosso trabalho, pudemos driblar checagens e implementações de 60% dos apps analisados. Nossos resultados mostram que, embora a segurança do iOS seja inquestionável, desenvolvedores iOS têm pouca ou nenhuma noção de boas práticas de segurança necessárias para implementação de compras dentro de apps, muitas vezes se confiando inteiramente nas restrições do sistema operacional para proteger seus apps.

7.1. Trabalhos Futuros

Embora tenhamos certeza da importância dessa análise para chamar atenção acerca da segurança de apps iOS, entendemos que ainda há bastante espaco para aprimorar essa análise. Assim, alguns dos trabalhos futuros propostos são:

- **Análise mais compreensiva** Nós fomos capazes de analisar 34 apps iOS de três categorias específicas, e, embora nossa análise seja a primeira capaz de fornecer algum entendimento acerca do estado atual da segurança de implementações de compras dentro de apps iOS, acreditamos que há espaço para expandir. Trabalhos futuros podem aumentar a quantidade de apps e de categorias analisados para fornecer uma perspectiva ainda mais realista desse cenário.
- **Criação de uma ferramenta de análise automática** Durante esse trabalho, desenvolvemos uma metodologia que envolveu análises estática e dinâmica manuais para identificar e testar vulnerabilidades em compras dentro de apps iOS. Entretanto, essa análise manual é demorada e não-escalável. Acreditamos que é possível automatizar uma ou mais etapas da exploração dessas vulnerabilidades para tornar a análise mais simples, rápida e escalável.
- **In-App Purchase API** Neste trabalho nós analisamos e identificamos vulnerabilidades com foco na Original API for in-app purchase porque entendemos que ela ainda hoje é a API usada na maioria dos apps iOS disponíveis na App Store. No entanto, trabalhos futuros devem focar também na In-App Purchase API, lançada mais recentemente pela Apple como uma forma de substituir a Original API for In-App Purchase, verificando se as vulnerabilidades identificadas aqui permanecem presentes na nova API e quaisquer novas vulnerabilidades que possam ter sido introduzidas junto a ela.

Referências

AloneMonkey (2024). frida-ios-dump. `https://github.com/AloneMonkey/frida-ios-dump` Acessado em Julho, 2024.

Apple (2021). Meet StoreKit 2. `https://developer.apple.com/videos/play/wwdc2021/10114/` Acessado em Julho de 2024.

Apple (2024). App Store stopped over $7 billion in potentially fraudulent transactions in four years. `https://www.apple.com/newsroom/2024/05/app-store-stopped-over-7-billion-usd-in-potentially-fraudulent-transactions/` Acessado em Julho, 2024.

AppleDeveloper (2024a). Choosing a StoreKit API for in-app purchases. https://developer.apple.com/documentation/storekit/in-app_purchase/choosing_a_storekit_api_for_in-app_purchases Acessado em Julho, 2024.

AppleDeveloper (2024b). In-app purchase. https://developer.apple.com/documentation/storekit/in-app_purchase Acessado em Julho, 2024.

AppleDeveloper (2024c). Offering, completing, and restoring in-app purchases. https://developer.apple.com/documentation/storekit/in-app_purchase/original_api_for_in-app_purchase/offering_completing_and_restoring_in-app_purchases Acessado em Julho, 2024.

AppleDeveloper (2024d). Original API for in-app purchase. https://developer.apple.com/documentation/storekit/in-app_purchase/original_api_for_in-app_purchase Acessado em Julho, 2024.

AppleDeveloper (2024e). Persisting a purchase. https://developer.apple.com/documentation/storekit/in-app_purchase/original_api_for_in-app_purchase/persisting_a_purchase Acessado em Julho, 2024.

AppleDeveloper (2024f). StoreKit. https://developer.apple.com/documentation/storekit Acessado em Julho, 2024.

AppleDeveloper (2024g). Unlocking purchased content. https://developer.apple.com/documentation/storekit/in-app_purchase/original_api_for_in-app_purchase/unlocking_purchased_content Acessado em Julho, 2024.

B. Reaves, J. Bowers, N. S. A. B. A. B. P. T. and Butler, K. R. B. (2017). Mo(bile) money, mo(bile) problems: Analysis of branchless banking applications. *ACM Trans. Priv. Secur.*, 20(3).

C. Mulliner, W. R. and Kirda, E. (2014). VirtualSwindle: An automated attack against in-app billing on Android. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS '14, page 459–470, New York, NY, USA. Association for Computing Machinery.

Cryptic (2024). Hopper Disassembler. https://www.hopperapp.com/ Acessado em Julho, 2024.

D. Reynaud, D. Song, T. M. E. W. and Shin, R. (2012). Freemarket: Shopping for free in Android applications. In *19th Annual Network And Distributed System Security Symposium*.

Data.ai (2024). State of mobile 2024. https://www.data.ai/en/go/state-of-mobile-2024/ Acessado em Julho, 2024.

Foresman, C. (2012). Recent iOS, Mac app crashes linked to botched FairPlay DRM. https://arstechnica.com/gadgets/2012/07/recent-ios-mac-app-crashes-linked-to-botched-fairplay-drm/ Acessado em Julho, 2024.

Frida (2024). Frida. https://frida.re/docs/ios/ Acessado em Julho, 2024.

iOSGods (2024a). Decrypted iOS IPA App Store. https://armconverter.com/decryptedappstore/us Acessado em Julho, 2024.

iOSGods (2024b). iOSGods App Store. https://app.iosgods.com/ Acessado em Julho, 2024.

iOSGods (2024c). No jailbreak section. https://iosgods.com/forum/79-no-jailbreak-section/ Acessado em Julho, 2024.

M. Egele, D. Brumley, Y. F. and Kruegel, C. (2013). An empirical study of cryptographic misuse in Android applications. CCS '13, page 73–84, New York, NY, USA. Association for Computing Machinery.

Munster, G. (2023). Apple's App Store is an important part of its profitability — and it's not changing. https://deepwatermgmt.com/apples-app-store-is-an-important-part-of-its-profitability-and-its-not-changing/ Acessado em Julho, 2024.

Rook (2019). Downloading & installing apps from the iOSGods app using Sideloadly on your PC! https://iosgods.com/topic/93697-downloading-installing-apps-from-the-iosgods-app-using-sideloadly-on-your-pc/#comment-2881754 Acessado em Julho, 2024.

S. Shi, X. W. and Lau, W. C. (2021). Breaking and fixing third-party payment service for mobile apps. In *Applied Cryptography and Network Security: 19th International Conference, ACNS 2021, Kamakura, Japan, June 21–24, 2021, Proceedings, Part II*, page 3–26, Berlin, Heidelberg. Springer-Verlag.

SensorTower (2022). Mobile market forecast 2022-2026. https://go.sensortower.com/rs/351-RWH-315/images/Sensor-Tower-2022-2026-Market-Forecast.pdf Acessado em Julho, 2024.

und3fy.dev (2024). Decrypt.day. https://decrypt.day/ Acessado em Julho, 2024.

W. Yang, Y. Zhang, J. L. H. L. Q. W. Y. Z. and Gu, D. (2017). Show me the money! Finding flawed implementations of third-party in-app payment in Android apps. In *24th Annual Network And Distributed System Security Symposium*.

Zuo, C. and Lin, Z. (2022). Playing without paying: Detecting vulnerable payment verification in native binaries of Unity mobile games. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3093–3110, Boston, MA. USENIX Association.