CONTOURNEMENT DE L'API GOOGLE PLAY BILLING

Guillaume LOPES (@Guillaume_Lopes)
RandoriSec

D'après le blog [INVESP], le montant global des paiements dits « in-app » représentait environ 37 milliards de dollars (USD) en 2017 pour les applications mobiles (Android et Apple). Ce montant représente quasiment la moitié des revenus générés par les applications mobiles (48,2%), dépassant les revenus générés par les régies publicitaires (14%), ainsi que l'achat d'applications (37,8%). Il est donc important que la sécurité de ces paiements soit correctement implémentée afin d'éviter un manque à gagner pour les développeurs des applications. Dans le cadre de cet article, nous avons passé en revue 50 applications Android afin d'étudier le fonctionnement de l'API Google Play Billing et d'identifier les vulnérabilités liées à une mauvaise implémentation. Nous détaillerons en exemple des applications vulnérables.

mots-clés : *ANDROID / GOOGLE / PLAY BILLING / PLAY STORE / IN-APP*

1. L'API GOOGLE PLAY BILLING

1.1 Présentation

Le service Google Play Billing (ou anciennement Google InApp Billing) est un service permettant aux développeurs de vendre des produits au sein de leur application Android. Pour simplifier, deux grandes catégories de produits ont été définies par Google **[BILLINGDOC]** :

- les « one-time products » : ce sont des produits qui nécessitent un paiement en une seule fois. Il peut s'agir d'acheter une version premium d'une application ou d'acheter des vies supplémentaires dans un jeu ;
- les « subscriptions » : comme leur nom l'indique, il s'agit d'abonnements, qui vont nécessiter un paiement régulier de la part de l'utilisateur (mensuel, hebdomadaire, etc.). Par exemple, l'abonnement à un journal.

Il est à noter que l'ensemble des produits vendus par l'application doivent être définis dans la Google Play Console **[PLAYCONSOLE]**. Il s'agit de la même interface permettant aux développeurs de publier leur application.

Le principal intérêt d'utiliser le service Google Play Billing est que toute la partie paiement est complètement gérée par Google. Le développeur n'a pas besoin de gérer des cartes de paiement ou des données bancaires. La figure 1 résume les différentes étapes d'un paiement via l'API Google Play Billing, qui sont :

1. Au lancement de l'application, une connexion est effectuée auprès des infrastructures de Google via le Play Store afin de valider la connexion et récupérer la liste des différents produits vendus par l'application ;
2. Lorsque l'utilisateur sélectionne un produit à acheter, une requête est envoyée au Play Store afin d'initier le processus de paiement ;

3. L'application Play Store demande à l'utilisateur les informations nécessaires pour effectuer le paiement (données carte bancaire, PayPal, voucher, etc.) ;
4. Lorsque le paiement est validé par Google, le Play Store renvoie les informations sur le produit acheté par l'utilisateur à l'application ;
5. L'application valide les informations renvoyées par Google avant de délivrer le produit à l'utilisateur. Il est à noter que l'application peut utiliser un serveur externe pour valider les éléments reçus, mais cela n'est pas obligatoire.

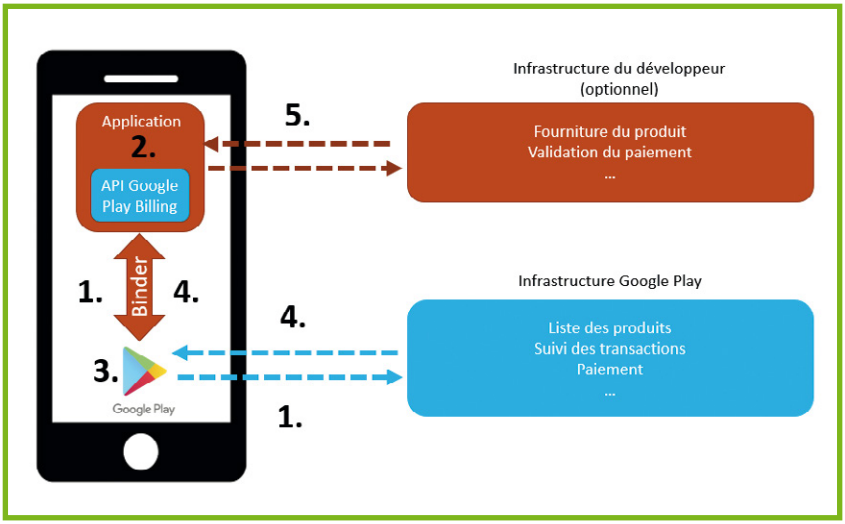

Fig. 1 : Schéma détaillant les étapes du paiement via Google Play Billing.

1.2 Validation du paiement

Pour valider le paiement d'un produit, la documentation de l'API Google Play Billing **[VERIFY]** prévoit 2 cas de figure. Soit le développeur met en place un serveur externe et effectue l'ensemble des vérifications sur celui-ci, soit la vérification est effectuée localement dans l'application, mais cette dernière ne permet pas d'assurer correctement la sécurité du paiement. Dans tous les cas, Google renvoie un objet JSON contenant, notamment, les éléments suivants :

- *orderId* : il s'agit d'un numéro unique identifiant la transaction (cet identifiant est généré par Google) ;
- *packageName* : le nom de l'application (ex : `com.limasky.doodlejumpandroid`) ;
- *productId* : l'identifiant du produit qui a été défini dans le Google Play Console (ex. : `doodlejump.candys10000`) ;
- *purchaseState* : un entier indiquant l'état du paiement (la valeur 0 indique que le produit a été acheté) ;
- *purchaseToken* : un jeton à usage unique qui permet d'identifier un achat pour un produit spécifique et un utilisateur particulier.

Cet objet JSON est signé à l'aide de la clé privée de l'application. En effet, lorsque le développeur publie son application, la Google Play Console génère une paire de clés RSA pour chaque application. La clé publique doit être présente dans l'application si le développeur souhaite valider la signature dans l'application.

Dans le cas où le développeur décide d'effectuer la validation du paiement sur un serveur externe, Google recommande d'interroger directement l'API Google Play Developer **[PLAYDEV]** pour récupérer la liste des achats enregistrés. Il pourra ainsi valider que le numéro identifiant la transaction (*orderId*) et le jeton à usage unique (*purchaseToken*) correspondent bien à ce qui est contenu dans l'objet JSON. Bien entendu, comme pour la vérification effectuée localement, le développeur pourra uniquement valider la signature de l'objet JSON.

Comme nous le verrons par la suite, de nombreuses applications préfèrent effectuer la validation du paiement au sein même de l'application.

1.3 Les bibliothèques tierces

Lors de l'étude des différentes applications Android utilisant l'API Google Play Billing, nous avons identifié que certaines bibliothèques tierces étaient utilisées pour faciliter le développement. Notamment dans le domaine du jeu vidéo, le moteur de rendu graphique Unity est très souvent utilisé. À cet effet, les bibliothèques « Google In App Billing Plugin » par Prime31 **[PRIME31]** et « Unity IAP » par Unity **[UNITY]** sont

fournies aux développeurs afin de s'interfacer avec l'API Google Play Billing. Malheureusement, ces 2 bibliothèques ne permettent pas de valider le paiement sur un serveur externe.

Voici un extrait de la documentation de Prime31 :

« Google highly recommends always validating purchases on a secure server. **The plugin will do on device validation for you but Android apps are very easily hacked so this should not be relied on.** »

Et voici un extrait de la documentation d'Unity :

« Unity IAP is designed to **support local validation within your application**. [...] **Unity does not offer support for server-side validation** ».

2. FAIBLESSES DE L'API

2.1 Historique sur BillingHack

En 2013, Dominik Schürmann a identifié deux vulnérabilités majeures dans l'API Google Play Billing **[SCHURMANN]** permettant de contourner le paiement.

D'une part, il était possible pour une application installée sur le téléphone d'usurper l'identité du Play Store. En effet, comme évoqué dans la première partie de cet article, l'application Google Play Store agit comme un proxy entre l'application Android souhaitant effectuer un achat et l'infrastructure de Google. Lorsque l'application souhaitait initier le processus de paiement, elle devait envoyer un *Intent* implicite auprès du Play Store. Pour rappel, lorsque l'*Intent* est implicite c'est Android qui se charge de déterminer le destinataire du message. Il suffit donc pour une application malveillante de définir un *Intent-filter* avec une priorité élevée pour recevoir la demande d'achat avant le Play Store. Il pourra ainsi par la suite répondre à sa place et notifier que le paiement a été effectué.

D'autre part, Dominik s'est également aperçu que la fonction de validation de la signature du paiement fournie par Google (**verifyPurchase**) était validée même si la signature était vide. Voici le code de la fonction au moment des faits :

```
public static boolean verifyPurchase(String base64PublicKey, String
signedData, String signature) {
        if (signedData == null) {
            Log.e(TAG, "data is null");
            return false;
        }

        boolean verified = false;
        if (!TextUtils.isEmpty(signature)) {
            PublicKey key = Security.generatePublicKey(base64PublicKey);
            verified = Security.verify(key, signedData, signature);
            if (!verified) {
                Log.w(TAG, "signature does not match data.");
                return false;
            }
        }
        return true;
    }
```

On constate bien que si la signature est une chaîne de caractères vide alors elle n'est pas vérifiée et la fonction retournera la valeur booléenne vraie.

Afin de démontrer l'exploitation de la vulnérabilité, Dominik a implémenté une preuve de concept nommée BillingHack **[POC]**. Pour résumer, un attaquant était en mesure de contourner le paiement en installant une application malveillante usurpant le comportement du Play Store et sous réserve que la validation était effectuée localement.

Google a rapidement corrigé ces vulnérabilités en forçant, tout d'abord, les développeurs à utiliser un *Intent* explicite pour communiquer avec le Play Store. Si tel n'est pas cas, Google refuse de publier l'application. Ensuite, la fonction **verifyPurchase** a été modifiée afin de retourner la valeur booléenne faux par défaut.

2.2 Contournement du paiement

Le lecteur attentif remarquera que malgré les corrections apportées par Google, si la validation du paiement est effectuée localement alors il est très difficile de s'assurer qu'un attaquant ne sera pas en mesure de contourner le paiement. Un attaquant modifiant l'application vulnérable pourra supprimer les vérifications et les protections mises en place.

Tout d'abord, il est possible de modifier l'*Intent* explicite, dans l'application cible, afin que notre application malveillante soit utilisée à la place du Play Store (**com.android.vending**) :

```
Intent intent = new Intent("com.android.vending.
billing.InAppBillingService.BIND");
intent.setPackage("org.billinghack");
```

Ainsi, notre application cible n'enverra ses messages que vers notre application malveillante. Ainsi, on pourra confirmer à notre application cible que l'utilisateur a effectué le paiement auprès de Google.

Ensuite, il suffira d'identifier la portion de code en charge de valider la signature et de la modifier afin que n'importe quelle signature soit acceptée. En résumé, les étapes sont les suivantes :

1. Lancer notre application malveillante (ex. : BillingHack) ;
2. Désassembler l'application cible à l'aide de l'outil **apktool** ;
3. Modifier la section de code définissant l'*Intent* explicite afin d'utiliser notre application ;
4. Modifier la portion de code validant la signature afin que n'importe quelle signature soit acceptée ;
5. Reconstruire l'application avec **apktool** et la signer avec **jarsigner** ;
6. PROFIT !

Il est à noter que pour réaliser cette attaque, il n'est pas nécessaire d'avoir un équipement rooté, mais simplement de modifier l'application ciblée et d'installer une application usurpant le Play Store.

Des applications permettant d'exploiter cette « technique » sont disponibles sur Internet. La plus connue est Lucky Patcher **[PATCHER]**. Néanmoins, toutes ces applications sont basées sur la preuve de concept de Dominik.

Enfin, en 2014, Collin Mulliner et al. ont développé un outil nommé VirtualSwindle **[SWINDLE]**, qui reproduisait cette attaque en instrumentant l'application ciblée. L'outil, qui n'a jamais été publié, nécessitait un équipement rooté. Les résultats à l'époque étaient déjà impressionnants, 51 applications étaient vulnérables sur les 85 testées.

2.3 Autres techniques de contournement

Google offre différentes possibilités aux développeurs pour tester l'API Google Play Billing dans leur application **[BILLINGTEST]**. L'une d'entre elles est d'utiliser des identifiants de produit (*productId*) réservés. Si une application Android utilise l'identifiant de produit `android.test.purchased` lors d'un achat, alors le Play Store considérera cela comme un test et ne demandera pas de paiement. Une réponse contenant les informations d'achat et une signature invalide sera renvoyée par Google. En 2017, Jérémy Matos avait présenté, lors de Bsides Lisbon **[MATOS]**, des applications Android qui ne validaient pas la signature du paiement lorsqu'un identifiant de produit réservé était utilisé. Cela permettait de contourner le paiement en modifiant uniquement l'application cible et sans utiliser d'application usurpant le Play Store.

3. APPLICATIONS VULNÉRABLES

Dans cette section, nous allons passer en revue quelques applications vulnérables afin de mettre en avant différentes implémentations du Google Play Billing. Afin de contourner le paiement, il faut comme évoqué précédemment remplacer le nom du composant du Play Store par une application que l'on contrôle (par exemple, Billing Hack). Ici, nous nous concentrons uniquement sur la validation du paiement. Note : tous les développeurs des applications présentées ont été contactés par l'auteur de l'article.

3.1 Doodle Jump : cas d'école

L'application Doodle Jump est un jeu de plateforme, qui était populaire en 2015 et qui comptabilise plus de 50 millions de téléchargements. Ce jeu permet d'acheter ou gagner des sucreries (« candies »), qui permettent par la suite d'acquérir des costumes.

Lorsque l'on analyse le code Java de l'application, on identifie dans la fonction `handleActivityResult` de la classe `IABHelper` que la signature et les données JSON sont validées par la fonction `verifyPurchase` :

```
int responseCodeFromIntent = getResponseCodeFromIntent(intent);
String stringExtra = intent.getStringExtra(RESPONSE_INAPP_PURCHASE_DATA);
String stringExtra2 = intent.getStringExtra(RESPONSE_INAPP_SIGNATURE);
    if (i2 == -1 && responseCodeFromIntent == 0) {
        if (stringExtra == null || stringExtra2 == null) {
            logError("BUG: either purchaseData or dataSignature is null.");
            logDebug("Extras: " + intent.getExtras().toString());
            iabResult = new IabResult(IABHELPER_UNKNOWN_ERROR, "IAB returned
null purchaseData or dataSignature");
            if (this.mPurchaseListener != null) {
                this.mPurchaseListener.onIabPurchaseFinished(iabResult, null);
            }
            return true;
        }
        try {
            Purchase purchase = new
Purchase(this.mPurchasingItemType,
stringExtra, stringExtra2);
            String sku = purchase.
getSku();
            if (Security.
verifyPurchase(this.
mSignatureBase64, stringExtra,
stringExtra2)) {
                logDebug("Purchase
signature successfully verified.");
```

Dans la fonction `verifyPurchase`, on s'assure que les différents éléments ne sont pas vides sinon la validation échoue. Ensuite, la fonction `verify` est appelée en utilisant la clé publique de l'application afin de vérifier que la signature correspond aux données JSON renvoyées par Google :

```
public static boolean
verifyPurchase(String str,
String str2, String str3) {
    if (!TextUtils.isEmpty(str2)
&& !TextUtils.isEmpty(str) &&
!TextUtils.isEmpty(str3)) {
        return
verify(generatePublicKey(str),
str2, str3);
    }
    Log.e(TAG, "Purchase
verification failed: missing
data.");
    return false;
}
```

La fonction `verify` effectue une vérification classique de signature. Du point de vue d'un attaquant, il est ainsi possible de modifier la fonction `verifyPurchase` afin de retourner vrai quel que soit le résultat de la fonction `verify`. Il est trivial d'effectuer cette modification en travaillant au niveau du code smali ou en utilisant Frida **[FRIDA]** par exemple. Ceci est laissé en exercice au lecteur !

3.2 Snoopy Pop : Unity à la rescousse !

L'application Snoopy Pop est un jeu similaire à Bubble Witch, la seule différence est qu'il se joue avec le personnage Snoopy ! Le but est de résoudre des puzzles en faisant des associations de boules de couleur, il est possible d'acquérir des vies ou bien des pièces (« coins ») qui permettront d'acheter des objets dans le jeu. Cette application utilise la bibliothèque Unity, qui fournit également une interface pour l'API Google Play Billing. Comme évoqué précédemment, Unity ne permet pas d'effectuer la validation du paiement depuis un serveur externe. Lors de l'analyse de l'application, nous avons constaté que la validation du paiement était réalisée dans les fichiers DLL présents dans le répertoire **`/assets/bin/Data/Managed`** :

```
# ls assets/bin/Data/Managed/
Analytics.dll Assembly-CSharp-firstpass.dll
Facebook.Unity.dll  mscorlib.dll Stores.dll System.
Xml.dll UnityEngine.Analytics.dll UnityEngine.
Purchasing.dll winrt.dll Apple.dll
Common.dll Facebook.Unity.IOS.dll P31RestKit.dll
System.Core.dll System.Xml.Linq.dll UnityEngine.dll
UnityEngine.UI.dll Assembly-CSharp.dll Facebook.
Unity.Android.dll Mono.Security.dll Security.dll
System.dll Tizen.dll UnityEngine.Networking.dll
Validator.dll
```

Dans notre contexte, ces fichiers contiennent du code C#, qui est facilement décompilable en utilisant un outil comme DnSpy **[DNSPY]**. Nous avons identifié que le fichier **`Security.dll`** contenait une fonction nommée **`Validate`** qui est en charge de valider la signature du paiement. Voici un extrait de son code :

```
publicGooglePlayReceipt Validate(string receipt,
string signature){
    byte[] bytes = Encoding.UTF8.GetBytes(receipt);
    byte[] signature2 = Convert.
FromBase64String(signature);
    bool flag = !this.key.Verify(bytes, signature2);
    if (flag)
    {
        throw new InvalidSignatureException();
    }
```

La variable **`receipt`** contient les données JSON renvoyées par Google et la variable **`signature`** contient la signature de l'objet JSON. Si la vérification échoue, on constate que la fonction retourne une exception nommée **`InvalidSignatureException`**. Afin de contourner le paiement, il suffit à un attaquant de supprimer l'exception retournée par la fonction et de reconstruire l'application avec cette nouvelle DLL.

3.3 Fruit Ninja : JNI

L'application Fruit Ninja est un jeu populaire dont le but est de découper des fruits comme un ninja ! À la date de rédaction de l'article, le jeu comptait plus de 100 millions de téléchargements. Comme précédemment, l'application permet d'acheter des crédits (pommes ou étoiles), qui permettent à leur tour d'acheter des objets dans le jeu (épée, dojo, améliorations, etc.). Pour cette application, les développeurs ont décidé d'implémenter les fonctions sensibles en utilisant du code natif. Sous Android, il est possible d'appeler du code natif depuis le code Java, et réciproquement, en utilisant l'interface *Java Native Interface* (JNI) **[NDK]**. L'utilisation de la JNI est facilement identifiable grâce à l'utilisation du mot clé **`native`** dans le code Java. En analysant le code de l'application, on identifie une fonction **`onActivityResult`**

dans la classe **GooglePlayBillingService**, qui manipule la signature et l'objet JSON renvoyés par Google comme l'illustre l'extrait de code suivant :

```
public static void onActivityResult(int i, int i2,
Intent intent) {
    Exception e;
    String str;
    StringBuilder stringBuilder;
    Intent intent2 = intent;
    if (i == 3055) {
        int intExtra = intent2.getIntExtra(RESPONSE_
CODE, 0);
        String stringExtra = intent2.
getStringExtra(RESPONSE_INAPP_PURCHASE_DATA);
        String stringExtra2 = intent2.
getStringExtra(RESPONSE_INAPP_SIGNATURE);
        String str2 = s_skuId;
        if (intExtra == 1) {
            PurchaseResult(str2, false, true, null,
null, null);
        } else if (stringExtra == null) {
            PurchaseResult(str2, false, false, null,
null, null);
        } else {
            String str3;
            try {
                JSONObject jSONObject = new
JSONObject(stringExtra);
                String string = jSONObject.
getString("productId");
               try {
                    [TRUNCATED]
                    PurchaseResult(string, true,
false, stringExtra, stringExtra2, str2);
                }
```

Ces informations sont ensuite envoyées à une fonction nommée **PurchaseResult**, qui elle-même appelle une fonction **native** nommée **PurchaseResultNative** :

```
public static void PurchaseResult(String str, boolean z,
boolean z2, String str2, String str3, String str4)
{
    GooglePlayBillingService googlePlayBillingService =
s_instance;
    if (googlePlayBillingService.m_waitingForCallback) {
        googlePlayBillingService.m_waitingForCallback =
false;
        synchronized (NativeGameLib.GetSyncObj()) {
            PurchaseResultNative(str, z, z2, str2, str3,
str4);
        }
    }
}

private static native void PurchaseResultNative(String
str, boolean z, boolean z2, String str2, String str3,
String str4);
```

Malgré le temps passé à effectuer une ingénierie inverse sur les bibliothèques partagées de l'application, l'auteur de l'article n'a pas été en mesure d'identifier le mécanisme de validation du paiement. Néanmoins, il apparaît que cette validation n'a pas été correctement implémentée étant donné qu'il est possible d'acheter des crédits dans le jeu en renvoyant une signature invalide ! Il est donc probable que la signature ne soit pas vérifiée.

CONCLUSION

Le choix laissé par Google de permettre aux développeurs de valider le paiement localement dans l'application n'a pas permis d'assurer la sécurité des paiements dits « in-app » sous Android.

Lors de notre étude réalisée sur 50 applications, nous avons identifié que 29 applications effectuaient la vérification de la signature localement et étaient donc vulnérables. Lors de l'analyse, nous avons bien identifié différentes techniques pour tenter de masquer la logique de vérification du paiement (obfuscation du code, utilisation de code natif, etc.), mais elles se sont révélées insuffisantes. Il est à noter que tous les développeurs ont été contactés, mais à ce jour aucune application n'a été corrigée.

À l'inverse du Google Play Billing, les API de paiement fournis par Samsung et Amazon pour l'environnement Android imposent l'utilisation d'un serveur externe pour la validation et la distribution du produit. Pour renforcer la sécurité des paiements dits « in-app », il est primordial de valider le paiement sur un serveur externe. Néanmoins, il reste toujours le problème de fournir le produit localement, notamment pour les jeux qui vendent des crédits ou des vies. Même sans contourner le paiement, il restera possible pour un attaquant d'identifier un moyen d'augmenter ces informations en manipulant l'application.

REMERCIEMENTS

Un grand merci aux relecteurs : Davy Douhine, Clément Notin, Marc Lebrun et Benoit Baudry. ■

Retrouvez toutes les références de cet article sur le blog de MISC : https://www.miscmag.com.