DÉSAMORCER DES BOMBES LOGIQUES

Jordan SAMHI – jordan.samhi@uni.lu
Alexandre BARTEL – alexandre.bartel@uni.lu

Aujourd'hui, les développeurs de code malveillant sont capables de contourner les mesures de sécurité et les techniques d'analyse les plus poussées grâce à de simples mécanismes appelés « bombes logiques ». Un exemple significatif est le Google Play qui accepte toujours des applications malveillantes pouvant déjouer ses barrières de sécurité. Cette introduction aux bombes logiques permet de sensibiliser sur les différentes solutions pouvant être mises en place pour détecter ces artifices.

mots-clés : *MALWARE / ANDROID / ANTI-RÉTRO-INGÉNIERIE / ANTI-ANALYSE / ANALYSE STATIQUE / ANALYSE DYNAMIQUE / BOMBE LOGIQUE*

Nous allons, à travers cet article, nous intéresser à des mécanismes connus des spécialistes en sécurité informatique depuis bien avant les années 2000, mais qui se déclinent désormais selon de nouvelles modalités. En effet, si le concept de bombe logique en informatique peut faire référence à des malwares des années 1990, voire même à des fonctionnalités cachées de logiciels légitimes (programmés pour dysfonctionner après un temps... requérant ainsi une réparation payante !), l'émergence de nouveaux systèmes d'exploitation, notamment mobiles, lui a conféré une nouvelle actualité.

Plutôt que de généraliser les approches visant à décrire et détecter les bombes logiques, nous allons nous concentrer, dans cet article, sur la plateforme Android. En effet, même si les bombes logiques ne sont pas seulement utilisées sur un type de plateforme particulier (Windows, Linux, Android, iOS, etc.), elles restent très utilisées dans les applications Android du fait de l'architecture permettant de mettre à disposition une application publiquement. De plus, en 2020 Android représente plus de 86 % des parts de marché mobiles **[IDC]**, ce qui en fait une cible de choix pour les attaquants.

Dans un premier temps, nous allons succinctement décrire différents mécanismes anti-rétro-ingénierie/anti-analyse utilisés par les attaquants. Ensuite, nous présenterons ce qu'est une bombe logique et la forme qu'elle peut prendre, surtout dans l'univers Android. Nous expliquerons l'importance de détecter ces dispositifs et les difficultés que pose la conception d'outils de détection automatique. Enfin, nous présenterons l'implémentation d'un outil statique permettant de déceler des bombes logiques dans des applications Android.

1. MÉCANISMES ANTI-RÉTRO-INGÉNIERIE/ANTI-ANALYSE

Nous sommes arrivés dans une ère où les simples applications malveillantes sont très rapidement décelées. Que ce soit par des programmes d'analyse automatique ou des analystes qui font de la rétro-ingénierie, l'expérience accumulée permet de retrouver des modèles récurrents. C'est pourquoi l'enjeu majeur des développeurs d'applications malveillantes est aujourd'hui de trouver des astuces afin

de contourner les outils et ralentir les analystes. Pour cela, ils peuvent utiliser certaines des techniques décrites ci-après.

1.1 Obfuscation

Ce mécanisme consiste à rendre du code source totalement illisible par un être humain et/ou difficilement compréhensible, ce qui est appelé brouillage de code. Il existe plusieurs variantes que nous allons voir.

Certes, dans les applications Android, la plupart des développeurs brouillent leur code pour le protéger étant donné que la décompilation de bytecode Java est chose aisée. Pour ce faire, il existe plusieurs logiciels sur le marché permettant d'automatiser cette tâche (e.g., Proguard **[PROG]**). Cependant, dans le contexte d'applications malveillantes, cette technique permet de rendre la lecture du code épineuse afin de ne pas reconnaître certains schémas connus ou éviter la compréhension de nouveaux morceaux de code malveillants.

Le premier processus généralement utilisé est le remplacement des noms de classes, de champs et de méthodes. Cela rend la tâche difficile pour un humain, mais ne change potentiellement rien pour une analyse automatique du code. D'autres artifices comme le packing, le chiffrement de chaînes de caractères ou le brouillage du flot d'exécution peuvent être utilisés pour obfusquer du code et rendre la tâche difficile pour une analyse automatique.

On comprend bien ici l'objectif de cette technique : rendre l'analyse statique (humaine ou automatique) très difficile. Néanmoins, durant l'exécution, le code se comportera de la même manière que du code non obfusqué, ce qui permet toujours à des analyses dynamiques de déceler des comportements dangereux.

1.2 Chargement de code dynamique

La plateforme Java met à disposition une fonctionnalité intéressante pour les développeurs permettant de charger du code à la demande. Cependant, cette fonctionnalité offre aux attaquants une opportunité pour ne pas se faire détecter. Il s'agit du chargement dynamique de classes **[CLA]**.

Pour bien comprendre la dangerosité de ce mécanisme, il suffit de jeter un coup d'œil à ce morceau de code :

```
01: DexClassLoader cl = new DexCLassLoader(file, null, null,
getClass().getClassLoader());
02: Class<?> c = cl.loadClass("com.monpackage.MaClasse");
03: Object instance = c.getConstructor().newInstance();
04: Method method = c.getMethod("methodeMalveillante");
05: method.invoke(instance);
```

Pour charger une classe en mémoire, on utilise un objet de type `DexClassLoader` qui va vérifier le fichier contenant les classes et les charger si elles ne sont pas déjà en mémoire (ligne 01). Ensuite est récupérée la classe voulue (ligne 02). Il suffit désormais de créer une instance de cette classe (ligne 03), de récupérer la méthode à exécuter (ligne 04) et de l'exécuter (ligne 05). Cela fait aussi appel à un mécanisme appelé *Java Reflection* que nous verrons dans la section suivante.

Le *class loader* charge du code provenant du fichier `file` (ligne 01), ce fichier peut provenir de n'importe où, il peut même être récupéré via Internet. Cela met bien en exergue la difficulté d'analyse automatique pour un programme. Un analyste pourrait récupérer le fichier de classes et l'analyser, mais si la chaîne de caractères représentant le fichier est chiffrée, il ne pourrait plus le faire directement.

Ici, comme pour l'obfuscation, l'analyse dynamique est à préférer étant donné que l'exécution donnera des indices intéressants concernant l'utilisation de code provenant de l'extérieur et pourra aussi analyser le comportement du code chargé.

1.3 Réflexion

Le package `java.lang.reflect` **[REFL]** met à disposition des développeurs des méthodes permettant de faire de l'introspection de classe. C'est-à-dire qu'il est possible de créer des instances de classes, d'appeler des méthodes sur ces classes et d'accéder à ces champs sans connaître la classe à l'avance. Le fonctionnement ressemble au chargement dynamique de classes à la différence que cette fois-ci on considère que la classe est déjà chargée :

```
Class c = Class.forName("com.monpackage.MaClasse");
Object i = c.getConstructor().newInstance();
Method m = c.getMethod("methodeMalveillante");
m.invoke(i);
```

Cette fonctionnalité ne permet pas de lancer une analyse statique automatique, car des arcs seraient manquants dans le graphe de flot de contrôle et dans le graphe d'appels en cours d'analyse. On pourrait se dire qu'il suffit de récupérer la chaîne de caractères et d'analyser la classe/méthode correspondante. Cependant, il faut bien garder à l'esprit que le but de l'attaquant est de cacher l'information, la chaîne de caractères sera donc très probablement obfusquée. Cela bloquera aussi un analyste qui ne pourra plus décider dans quelle direction continuer son analyse.

Encore une fois, durant une analyse dynamique, une analyse comportementale de l'application reste toujours possible si le code se déclenche normalement.

1.4 Bombes logiques

Généralement, le code malveillant est caché quelque part dans une application afin d'être déclenché lorsque l'application s'exécute. Cependant, certains développeurs ont choisi d'ajouter des conditions pour lesquelles doit se déclencher ce code malveillant. Il s'agit de bombes logiques, le code se déclenchera seulement si des conditions particulières sont respectées.

Un exemple classique serait de déclencher le code malveillant après une date spécifique :

```
public class MonActivite extends Activity {
  protected void onCreate(Bundle b) {
    [...]
    Date now = new Date();
    Date dateAttaque = new Date(2020, 11, 5);
    if (now.after(dateAttaque)) {
      declencherCodeMalveillant();
    }
    [...]
  }
}
```

Cet exemple est relatif à une application qui, entrée sur le Google Play au début de l'année 2020, a potentiellement 11 mois pour se diffuser sur un maximum de smartphones pour enfin déclencher le code malveillant. Il s'agit d'une bombe logique de type temporel.

L'actualité montre que cet exemple peut aussi être utilisé par d'autres agents pathogènes, même ceux qui se baladent dans la nature ! En effet, quel virus apparaissant absolument dans tous les tabloïdes ces derniers mois ne se déclenche qu'au maximum une quinzaine de jours après infection par ce dernier ? Les plus férus de métaphores auront reconnu qu'il s'agit du SARS-CoV-2, plus communément appelé coronavirus (COVID-19) !

Plus sérieusement, on voit ici que le but est non pas de leurrer les analyses statiques, mais les analyses dynamiques. En effet, si cette application est exécutée dans un environnement de test pour analyse, le code ne se déclenchera pas si la date n'est pas passée. Ainsi, l'application apparaîtra comme bénigne.

Les techniques anti-analyse vues précédemment permettaient de contourner des analyses statiques, les bombes logiques permettent de contourner les analyses dynamiques.

Ceci est problématique étant donné que la plupart des outils les plus sophistiqués exécutent les applications pour analyser leur comportement. Combinées avec une des techniques mentionnées plus tôt dans cet article, les bombes logiques permettent l'entrée dans le Google Play d'applications malveillantes, car restées sous le radar des outils d'analyse.

On voit que l'exemple précédent ne cible personne en particulier, le but est par exemple d'infecter un maximum d'utilisateurs avec un *adware* qui va générer de l'argent au développeur. Il existe cependant un type d'attaque ciblé appelé Menace Persistante Avancée (APT) permettant à l'attaquant de ne pas se faire repérer pendant un temps indéterminé et d'avoir accès aux ressources de l'appareil. Généralement, ce type d'attaque permet de cibler des personnages politiques et/ou industriels afin de voler ou détruire des informations importantes **[APT]**. Un smartphone sous Android est une cible intéressante, car connecté au réseau téléphonique en permanence, pour ce genre d'attaque du fait du fonctionnement du framework sous-jacent, un exemple est mis en œuvre dans le code suivant :

```
public class MonReceiver extends BroadcastReceiver {
  public void onReceive(Context c, Intent i) {
    SmsMessage sms = getSmsEntrant(i);
    String body = sms.getMessageBody();
```

```
    String expediteur = sms.getOriginatingAddress();
    String cmd = null;
    if (expediteur.equals(hardCodedAddress)) {
      if (body.startsWith(hardCodedPrefix)) {
        cmd = getCmd(body);
        traiterCommande(cmd);
      }
    }
  }
}
```

Ici, le code malveillant se trouve dans la méthode **`traiterCommande()`** et permet, on l'imagine, d'effectuer des actions particulières en fonction de la commande passée au SMS entrant. À ce moment, on peut se dire : « l'utilisateur va voir ce SMS et se rendre compte de quelque chose ! ». Que nenni ! Le framework Android met à disposition la méthode **`abortBroadcast()`** qui permet de faire en sorte que les autres applications ne reçoivent pas ce SMS (à condition que la priorité de cette application soit plus élevée). Cette application pourrait donc voler subrepticement les données de l'utilisateur, agissant comme une porte dérobée. On voit que de simples morceaux de code rendent des applications très dangereuses et compliquent le travail des analystes.

Il existe bien évidemment une multitude de formes que peuvent prendre les bombes logiques. Le code malveillant ne va pas se déclencher, par exemple, à moins que :

- l'application ne tourne dans un environnement de bac à sable ;
- l'accès à Internet ne soit activé ;
- l'utilisateur ne se trouve à un endroit particulier ;
- un accéléromètre ne soit présent ;
- l'appareil n'ait une vitesse donnée ;
- un type de réseau spécifique ne soit accessible ;
- etc.

On voit qu'il peut exister un nombre assez grand de possibilités concernant la forme que peut prendre une bombe logique. L'idée principale est d'ajouter des tests avant d'exécuter du code malveillant. Cela ne change pas la problématique des détections statiques de code malveillant ni pour de la rétro-ingénierie. En revanche, cela permet de neutraliser la plupart des techniques d'analyse dynamique.

2. L'IMPORTANCE DE DÉTECTER CES MÉCANISMES

Pourquoi est-il important de s'intéresser à ces mécanismes et d'essayer de les détecter ? Pour répondre à cette question, il faut déjà bien comprendre de quoi est constituée une bombe logique, pour cela il suffit de se référer à la figure 1.

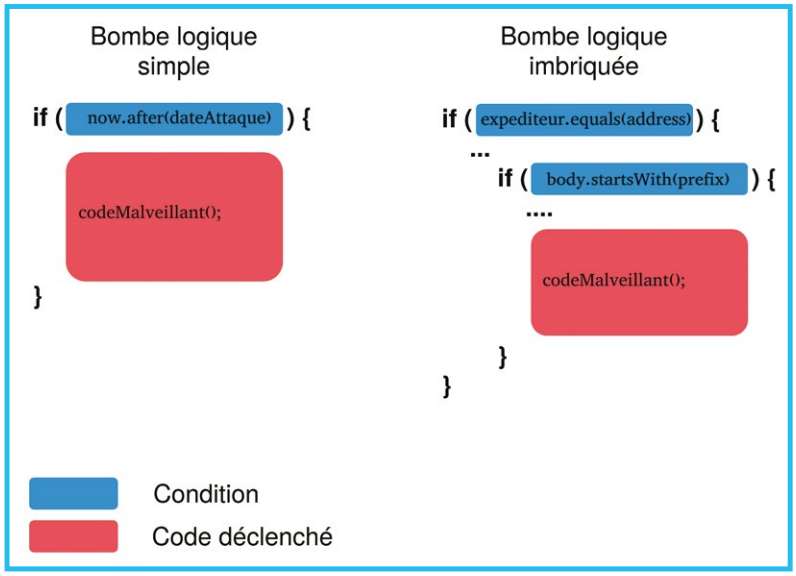

Fig. 1 : Représentation visuelle de la forme que peut prendre une bombe logique.

On voit clairement qu'une bombe logique est constituée d'une condition et du code qui va être déclenché par cette condition. Par ailleurs, la partie droite de la figure montre qu'une bombe logique peut prendre une forme imbriquée pour que la condition de déclenchement soit plus restreinte. Essayer de détecter le code déclenché réduit ce problème à la détection de code malveillant qui est indépendant des bombes logiques et donc ne justifie pas de s'intéresser à ces mécanismes.

Cependant, on peut s'arrêter aux conditions pour détecter des schémas connus. C'est pourquoi les bombes logiques sont intéressantes à étudier, elles donnent une information supplémentaire pour la détection du code malveillant.

3. DIFFICULTÉS QUE POSE LA DÉTECTION AUTOMATIQUE

Premièrement, il faut savoir que détecter n'importe quelle propriété dans un programme est impossible dans le cas général ; on dit que ce problème est indécidable, et cela a été prouvé dans les années 50 par Henry Gordon Rice **[RICE]**.

Cependant, en pratique, il est possible de mettre en place des mécanismes qui se révèlent fonctionnels dans certains cas. L'idée est de partir de schémas connus pour la forme que peut prendre la condition et ensuite d'utiliser ces heuristiques. C'est-à-dire que si on sait à l'avance que certains malwares n'exécutent leur code malveillant que si l'application reçoit un SMS particulier, on peut utiliser cette information. Mais comment ?

Le test protégeant le code malveillant prendra une forme telle que `if(smsBody.startsWith(prefix)) {...}` ou alors `if(smsBody.equals(var)) {...}`. On peut alors mettre en place une *analyse de flux de données* pour traquer les variables dans le code qui vont recevoir le corps d'un SMS (reconnaissable, car exécutant la méthode `getMessageBody()`). Ensuite, ces variables vont être propagées partout où leur valeur est transmise, c'est ce qu'on appelle une *analyse de teinte*. Ensuite, on vérifie si une des données teintées se retrouve dans une condition et est comparée avec d'autres chaînes de caractères à l'aide de méthodes telles que : `equals()`, `startsWith()`, `endsWith()`, `contains()`, etc.

Si c'est le cas, il y a des chances que l'on soit tombé sur une application malveillante. L'analyse du code couvert par cette condition peut être conduite pour en être sûr, cela réduit la surface de code à analyser, surtout si l'application est volumineuse en code.

Cette approche sous-entend partir de schémas connus, or il est aussi possible d'anticiper des schémas non connus pour le moment, mais probablement utilisés dans la nature par les développeurs de code malveillant. Cela permet, malgré l'augmentation évidente de fausses alarmes, de détecter de potentiels malwares utilisant ces méthodes.

Il faut garder à l'esprit que les applications détectées avec cette méthode ne sont pas nécessairement des malwares, car même des applications bénignes utilisent ce mécanisme, voir code ci-dessous :

```
[...]
String body = smsMessage.getMessageBody();
if(body!= null && body.startsWith("GETPOS")){
 sendPosition();
}
[...]
```

Ce code simplifié est tiré de l'application bénigne MyCarTracks. L'idée est de pouvoir recevoir la position de l'appareil en lui envoyant un simple SMS. Cette méthode est ici bénigne, mais utilisée dans un autre contexte, elle peut être considérée comme malveillante. C'est pourquoi les analyses statiques se retrouveront indéniablement avec de fausses alarmes.

C'est là toute la difficulté de la détection automatique, faire la différence entre ces deux types d'applications, ce qui n'est pas trivial. Cependant, cela permet de repérer des applications malveillantes qui utilisent les bombes logiques et ensuite d'avoir un point d'entrée sur le code malveillant pour l'analyser.

4. IMPLÉMENTATION STATIQUE

Nous avons implémenté une approche totalement statique (nommée TSOpen **[TSOP]**) permettant de prendre en entrée une application Android pour produire en sortie une liste de bombes logiques potentielles dans cette application. Une première difficulté est la modélisation de l'application sous la forme d'un graphe de flot de contrôle. En effet, les applications Android ne sont pas seulement composées d'une méthode `main` sur laquelle on lance une analyse comme d'autres programmes en C ou en Java. Elles sont construites avec des composants qui représentent les différents écrans de l'utilisateur, les services qui tournent en tâche de fond, etc. Ces composants communiquent entre eux à l'aide d'`Intents`. Il n'y a donc pas de point d'entrée unique. C'est pourquoi notre analyse repose sur un outil existant permettant de modéliser les applications en connectant les composants, il s'agit de Flowdroid **[FLOW]**.

La première phase de l'analyse permet d'annoter les variables du code avec un tag représentant leur contenu (e.g., #sms/#body, #here/#longitude, #now/#hour, etc.). Ensuite, l'idée est de regarder si une de ces variables taguées tombe dans une condition qui nous intéresse, c'est-à-dire que la condition aura une forme particulière que nous avons défini auparavant. Si c'est le cas, nous récupérons le code dominé et dépendant de la condition (potentiellement malveillant) et regardons

s'il fait appel à une méthode considérée comme sensible. Enfin, nous enregistrons la potentielle bombe logique et l'affichons à l'utilisateur.

Voici un exemple d'exécution pour une application malveillante :

```
java -jar TSOpen.jar -p platforms -f malicious.apk

Potential Logic Bombs found :
--------------------------------------------------------------------
- Statement                 : if $z4 == 0
- Class                     : com.littlephoto.server.
ShortMessageReceiver
- Method                    : onReceive
- Starting Component        : BroadcastReceiver
- Ending Component          : BroadcastReceiver
- CallStack length          : (2)
- Size of formula           : 4
- Sensitive method          : <android.content.BroadcastReceiver: void
abortBroadcast()>
- Reachable                 : Yes
- Guarded Blocks Density    : 6
- Nested                    : Yes
- Predicate                 : (#sms/#body.equals("$$@@&&@@"))
(#Suspicious)
--------------------------------------------------------------------

- Statement                 : if $z4 == 0
- Class                     : com.littlephoto.server.
ShortMessageReceiver
- Method                    : onReceive
- Starting Component        : BroadcastReceiver
- Ending Component          : BroadcastReceiver
- CallStack length          : (2)
- Size of formula           : 3
- Sensitive method          : <android.content.BroadcastReceiver: void
abortBroadcast()>
- Reachable                 : Yes
- Guarded Blocks Density    : 47
- Nested                    : No
- Predicate                 : (#sms/#body.equals("$$@@&&$$"))
(#Suspicious)
--------------------------------------------------------------------
```

Maintenant, un exemple d'exécution pour une application bénigne :

```
java -jar TSOpen.jar -p platforms -f benign.apk

Potential Logic Bombs found :
--------------------------------------------------------------------
- Statement                 : if $z0 == 0
- Class                     : hr.notfer.examtool.SMSReceiver
- Method                    : onReceive
- Starting Component        : BroadcastReceiver
- Ending Component          : BroadcastReceiver
- CallStack length          : (2)
- Size of formula           : 4
- Sensitive method          : <android.content.BroadcastReceiver: void
abortBroadcast()>
- Reachable                 : Yes
- Guarded Blocks Density    : 10
- Nested                    : No
- Predicate                 : (#sms/#body.equals("zebinjo"))
(#Suspicious)
--------------------------------------------------------------------
```

Ces deux traces démontrent la difficulté que pose la détection automatique. D'un côté, il y a de vraies bombes logiques et de l'autre le fonctionnement normal d'une application. Cependant, cela ouvre la porte à la facilité d'analyse manuelle pour une application étant donné qu'il est possible de localiser et/ou traquer le potentiel code malveillant grâce aux informations extraites.

Évidemment, cette implémentation n'est pas parfaite, car elle ne fait pas une analyse en profondeur du code déclenché pour pouvoir catégoriser le code comme malveillant avec un plus haut degré de confiance. Néanmoins, les analyses dynamiques sont plus efficaces pour prendre cette décision. Il est donc possible, après avoir détecté ces conditions, d'instrumenter l'application de base, de modifier la condition en la forçant à **vrai**, pour que le code s'exécute après avoir lancé l'application. Ainsi une approche dynamique serait en mesure d'analyser le comportement du code potentiellement malveillant.

CONCLUSION

À travers cet article, on a vu que la détection des bombes logiques n'est pas à négliger pour la compréhension des malwares sophistiqués. Même si le problème reste entier et ouvert, des solutions mettant en œuvre des techniques d'analyse statique et dynamique peuvent être mises en place pour mettre la main sur d'éventuelles applications indésirables. Ceci afin d'apporter une brique de sécurité supplémentaire aux marchés d'applications en ligne proposant des applications au grand public. ■

Retrouvez toutes les références de cet article sur : https://connect.ed-diamond.com/MISC/MISC-111