TP - La cybersécurité par la vérification environnementale : Partie 1

Louis Parent

L'objectif de ce TP est de mettre en place une couche de sécurité supplémentaire en détectant dynamiquement des vecteurs d'attaque potentiels dans l'environnement d'exécution.

Pour cela, on va se concentrer sur un sous-ensemble de vecteurs traité par le blindage dans le cas des applications *Android*.

Exercice 1 - Recherche de méthode de détection

Dans ce premier exercice, l'objectif est de construire un jeu de détecteur. Un détecteur est simplement une fonction retournant un booléen qui indique si le vecteur d'attaque est présent ou non. Nous allons nous concentrer sur la détection des vecteurs suivantes :

- Débogueur **Tu as testé le débogueur en attachant Android Studio à ton application via View → Debug → Attach debugger to Android process et en sélectionnant ton app. Ensuite, Debug.isDebuggerConnected() retourne true quand le debugger est actif.**
- Mode développeur
- Root
- Émulateur

Pour chacun de ces vecteurs d'attaque, établissez un détecteur qui peut identifier sa présence sur un appareil *Android* à l'aide d'une recherche internet. Ce détecteur peut être écrit en *Java* ou en *Kotlin* selon votre préférence (**Android Studio** est capable de faire la traduction *Java* vers *Kotlin*).

Exercice 2 - Implémentation des protections

Dans ce second exercice, l'objectif est d'implémenter ces détections au sein d'une application existante. Vous pouvez récupérer une application jouet ici afin d'avoir une application fonctionnelle comme base.

Étape 1 - Localiser le point d'injection

La première étape est de localiser l'endroit dans le code où exécuter les détections. Afin qu'un vecteur d'attaque ne puisse pas être exploité, il est nécessaire de le détecter au plus tôt. Autrement dit, il faut lancer la détection au lancement de l'application, ce qui correspond au chargement de la première activité sur *Android*.

Identifiez l'endroit concerné dans le code de l'application jouet.

Étape 2 - Définir une remédiation

Une fois ce point localisé, il est nécessaire de définir une remédiation. C'est-à-dire une action à réaliser en réponse à une détection positive. Dans le cadre de ce TP, nous allons choisir la remédiation la plus simple : tuer l'application. Cette solution permet d'empêcher immédiatement toute exploitation d'un vecteur d'attaque.

Définissez un code qui permet de fermer sa propre application.

Étape 3 - Implémentation

À cette étape, vous devriez avoir les différentes pièces pour implémenter les protections. Si tout fonctionne correctement, votre application devra se couper d'elle-même lorsqu'elle est exécutée sur l'émulateur intégré d'**Android Studio**.

Ajoutez les détections dans l'application jouet et appelez votre code de remédiation si une détection est positive.

Exercice 3 - Injection automatisée de protection

Pour ce dernier exercice, l'objectif est d'obtenir un résultat identique à celui de l'exercice précédent, mais sans intervention manuelle. Pour ce faire, vous devez produire un projet *TypeScript* qui modifiera automatiquement une application *Android* quelconque (on réutilisera ici l'application jouet de l'exercice précédent).

Étape 1 - Mise en place du projet

Afin d'accélérer la mise en place, un projet de base est disponible ici. Vous pouvez le télécharger en local ou le forker. Il contient la structure d'un projet *TypeScript* de base ainsi qu'une librairie pour parser un fichier `AndroidManifest.xml` qui sera utile pour les prochaines étapes.

Étape 2 - Refactoring de vos détecteurs

Afin de faciliter les prochaines étapes, retravaillez les détecteurs que vous avez produits en une unique classe *Java* ou *Kotlin*. Cette classe doit contenir tous les détecteurs ainsi qu'une unique méthode statique permettant de les exécuter tous et d'appliquer la remédiation si nécessaire.

Placez ensuite cette classe dans un fichier au sein du projet *TypeScript* dans un dossier `resources`, `assets` ou autre, afin qu'elle soit facilement accessible.

Étape 3 - Paramétrage de votre code

Modifiez le projet *TypeScript* afin qu'il prenne un paramètre en entrée. Ce paramètre sera le chemin vers l'application *Android* et servira de dossier racine pour les accès aux fichiers de l'application qui seront utilisés dans les étapes suivantes.

Étape 4 - Insertion de vos détecteurs

Modifiez le projet *TypeScript* afin qu'il copie votre classe créée à l'étape 2 dans le code de l'application dont le chemin a été passé en paramètre. Faites attention de conserver la nomenclature des packages *Java* en créant les dossiers correspondants (la classe `com.example.MyClass` se trouve dans le dossier `com/example/MyClass.java`), et de placer ce fichier dans l'emplacement standard pour le code dans un projet d'application *Android*.

Étape 5 - Identification de l'activité principale

Dans l'exercice précédent, vous avez manuellement identifié l'activité servant de point d'entrée dans l'application. Afin de reproduire ce comportement de façon automatisé, il est nécessaire d'analyser le fichier `AndroidManifest.xml`. Ce fichier liste entre autres toutes les activités d'une application *Android*. L'activité servant de point d'entrée est celle qui peut recevoir l'intent `android.intent.action.MAIN`.

Modifiez votre projet *TypeScript* afin de prendre en paramètre le chemin vers l'application jouet *Android* et de retrouver à partir du fichier `AndroidManifest.xml` l'activité de démarrage. Utilisez comme cible l'application jouet sans vos modifications de l'exercice précédent afin qu'il ne contienne pas le code que vous avez ajouté à la main.

Étape 6 - Localisation de la classe

Maintenant que vous connaissez le nom de l'activité utilisé au démarrage, il est nécessaire de trouver quel fichier définit cette classe afin de le modifier.

Modifiez votre projet *TypeScript* afin de parcourir les différents fichiers *Kotlin* de l'application et d'identifier lequel définit la classe de l'activité démarrage identifiée à l'étape précédente.

Étape 7 - Modification de la classe

Vous savez maintenant dans quel fichier se trouve la déclaration de la classe qui nous intéresse. Comme vu à l'exercice précédent, vous savez normalement aussi dans quelle méthode le code doit être ajouté : `onCreate(Bundle)`.

Modifiez votre projet *TypeScript* afin de changer le contenu du fichier qui nous intéresse et de rajouter un appel à la méthode statique que vous avez créé à l'étape 2.

Étape 8 - Test

Vous avez maintenant produit un programme de blindage naïf pour le code source. Vous pouvez maintenant exécuter l'application en donnant en paramètre un chemin vers l'application jouet non modifiée de l'exercice précédent. Une fois que c'est fait, ouvrez l'application dans **Android Studio**, puis lancez-la sur un émulateur. Si tout se passe bien, l'application devrait être compilée sans erreurs et avoir le même comportement que votre modification manuelle de l'exercice précédent.