NinjaBox™: App Level Sandboxing API

Irene Chen <chenix@mit.edu>
Danny Chiao <dchiao@mit.edu>
Bradley Wu <bwubbles@mit.edu>
Stephanie Yu <styu@mit.edu>

Overview

NinjaBox™ is an API that allows developers to easily include the functionality of sandboxing in their Android applications. Sandboxing enables an Android user to restrict a guest user to their own unique session, as well as prevent them from leaving the developer's application without providing a password.

Problem Statement

Our primary use case is when an Android user lends their phone to another user to use a specific application. Android does not provide an easy way of switching accounts from a personal account to a 'guest' account, and although Permissions Control have been introduced in Android 4.3, the new features target guest users repeatedly using the phone, such as a child on a parent's phone. The new features in Android 4.3 do not address the issue of a one-time guest trying to use a single application with their own account, nor the issue of restricting a guest user to a single application on the phone.

Our Solution: NinjaBox™

We would like to address the issue of restricting a guest user to a single application on the phone. To provide this sandboxing functionality, we developed the NinjaBox™ library to provide developers with an easy way to enable and disable application level sandboxing (termed NinjaMode™) in their Android applications. When NinjaMode™ is enabled, the user of the phone can only exit the app by supplying a password. In addition, any state changes that occur in NinjaMode™ are removed upon exit.

For example, if user A wishes to use a notepad app that uses the NinjaBox™ library on user B's phone, user B can turn on NinjaMode™ within the app and specify a session password. At this point, user A can freely use the notepad on the phone with no knowledge of any state in the phone. User B's privacy is maintained since user A cannot leave the notepad app or see user B's previous notes and settings. When user A is done, user B can simply exit NinjaMode™ by inputting the session password and any state changes induced by user A (new notes, etc.) will be removed.

Threat Model

The NinjaBox™ API needs to ensure that attackers, in the form of either users with ill intentions or malicious/buggy applications, cannot circumvent NinjaMode™ and allow arbitrary users to exit

the sandbox without the owner's permission. Attackers can also attempt to persist user changes within the sandbox.

Design and Implementation

We designed NinjaBox™ as an API that developers can easily integrate into their application. Initially, we considered creating an application to sandbox any other application, but upon further research we realized that the capturing of intents from another application not only violates Android's security policy but was outside the scope of the project given the timeframe we had.

In order to simplify the integration of NinjaBox™ with any developer's application, we wrote a `NinjaActivity` class that extends the normal `Activity` class in Android. By then asking the developer to simply extend `NinjaActivity` instead of `Activity` in their application, we effectively take control over the entire lifecycle of the activity. The developer then has access to our API to enter and exit NinjaMode™.

What is NinjaMode™?

When an application enters NinjaMode™, user activity becomes restricted such that no actions may take the user outside the application. These restrictions occur on the various levels of application and phone usage: hardware, system dialogs, and launching other applications. In addition, files belonging to the owner of the phone cannot be accessed by the guest, and any files created by the guest are stored in separate NinjaMode™ directories.

Hardware

We disable the 'back', 'home', 'recent apps', 'search' hardware buttons on the phone.

To override the functionality of the 'back' and 'search' buttons, we overrode and modified the `OnKeyDown()` and `OnKeyLongPress()` methods, which are invoked when any phone hardware keys are pressed. Inside these methods, if we detect a key event (e.g. `KeyEvent.KEYCODE_BACK`), we return immediately instead of performing the associated action.

The 'home' button key event has been unable to be overwritten beginning with Android 4.0 (ICS). To achieve our desired functionality, we request that the Android manifest for the application declare itself as a launcher application. When a user starts NinjaMode, in addition to creating a password the user should click the home button and specify to use the current application as the default launcher. Now, any home button press will attempt to launch the current application, but since it is already launched, no action will be performed.

In addition to specifying a launcher, we also request that the main activity be specified as two alias activities.

```
<activity android:name="com.example.sample_ninjabox.LoginActivity"
android:exported="true">
<activity-alias android:name="LoginAlias"
android:targetActivity="com.example.sample_ninjabox.LoginActivity">
```

```
<activity-alias android:name="LoginAlias-Copy"
android:targetActivity="com.example.sample_ninjabox.LoginActivity">
```

We use a method called `refreshLauncherDefault()` which switches the alias and causes Android to think a new launcher has been installed and will re-prompt the user to select a default launcher. This method is essential because if not included, the current application will be stuck as the default launcher.

We call `refreshLauncherDefault()` whenever NinjaMode™ is entered or exited.

System Events

System events such as incoming phone calls, alarms, and system dialogs are intercepted.

To handle phone calls, we use a system level call to the android Telephony class and disable phone call listening. For alarms and system dialogs, we hijack `onWindowFocusChanged()` and detect whether the activity on top is outside the scope of the application (i.e. the activity was declared in the manifest). For system dialogs, we launch an intent to close all system dialogs. There is an edge case with the Recent Applications activity, though we worked around this by using the `TOGGLE_RECENTS` intent instead. Finally, we also put the application in fullscreen mode, preventing the user from accessing the notifications on the top in Android.

Launching Other Applications

All intents to start a different application are intercepted. The user will be prompted with a password dialog to exit NinjaMode™ to allow the new application to be launched.

We override a few dozen functions in the `Activity` class such as `startActivity(Intent intent)` to first launch a password prompt. This prevents the application itself from launching an external intent. Further, any other application that attempts to steal focus will instantly lose focus because we hijack `onWindowFocusChanged()` and regain focus.

Shared Preferences

`SharedPreferences` in Android by default writes to a preferences XML file. When in NinjaMode™, we want to prevent this behavior and instead write to a sandboxed file to sandbox any data saved by a guest user and prevent the original user's data from being accessed or modified. To do this, we developed a `NinjaPreferences` class which implements the `SharedPreferences` interface. This class will read and write from a sandboxed file different from that used by `SharedPreferences`, and will delete the file upon exiting NinjaMode™.

Internal and External Storage

Methods accessing the Android file system are intercepted so that application files are read from and written to a separate set of NinjaMode™ folders.

Each app is typically given application-specific directories to store files on both internal external storage and is also given both internal and external storage cache folders. On running

`initializeNinjaMode()`, we create a set of folders, saved as subfolders in the application's internal storage directory, for storing files created in NinjaMode™ and override all `Context` class methods accessing the file system to read from and write to our folders. On `stopNinjaMode()`, we clear our NinjaMode™ storage folders.

Databases

Methods accessing the database directory are intercepted to point to a special NinjaMode™ database folder.

The same approach for internal and external storage is taken here as well. We override all `Context` class methods and redirect attempts to access the true database folder to our own.

Application Programming Interface (API)

We expose three basic functions through our API: `startNinjaMode()`, `stopNinjaMode()`, and `isNinjaMode()`. These functions allow the developer to give users the chance to start and stop NinjaMode™.

`startNinjaMode()`

We envisioned a user starting NinjaMode™ via a button press. A developer could call `startNinjaMode()` in an onClick listener, and this would prompt the shift of the application into NinjaMode™. This will prompt the user to create a session password and restart the activity in a sandbox.

`stopNinjaMode()`

When a user wants to exit NinjaMode™ via either an external intent or the explicit disabling of NinjaMode™ (e.g. a button), `stopNinjaMode()` will be called. This will cause a dialog to appear and prompt the user for the session password they set during `startNinjaMode()`. If they enter the correct password, then our API will clean up and delete the sandbox. This involves deleting all sandboxed files the user might have written to, such as SharedPreferences, application-specific files in internal or external storage, and application-specific databases. The API will then restart the activity, restoring the previous user session and data.

`isNinjaMode()`

This method lets the developer know whether their application is currently in NinjaMode™ or not.

Remarks and Further Research

There are a few cases in which NinjaMode™ cannot completely prevent guest users from viewing or modifying existing user state. If applications make use of the static methods in the `Environment` class to traverse the file system, then NinjaMode™ does not intercept file I/O methods properly because it is unable to override those static methods as a library. However, it may be possible to modify the Android OS to support interception of any file I/O method.

Finally, if the guest user powers off the phone within NinjaMode™, state is preserved temporarily. We clear this state as soon as the application is launched, though we could potentially improve on this handling.