Mobile Security (WS 2018/19)
Assignment 5 (Project)

Submitted By: Shubham Agarwal
Atul Anand Jha

Solution 1:- (*Provenance information for Intent messages*)

Problem Statement:-

Inter-process or Inter-application communication in Android ecosystem is a common and one of the most sought after feature that developers and users look forward for better services and user experience. For instance, an entertainment application that uses other social network application to post something on the profile of user may require to communicate with each other, thus highlighting the necessity of such channels.

Applications that share sandboxes or in more technical terminology, their UIDs as well as signed by same developer, or activities that are operated well within the boundary of the application can provide provenance information about the origin of Intent to the receiver with the help of Binder IPC (`Binder.getCallingUID(...)/Binder.getCallingPID(...)`), so that the receiver application is able to identify the intent and the originating application and thus, enforce access control on the Intent received accordingly.

However, this is not the case when an operation such as activity or broadcast is sent outside the process, i.e., inter process communication. The actual sender UID & PID of the intent or broadcast is lost within the channel and is incorrectly received by the receiver, in the sense that all the inter-app communication passes through respective Manager Services (ActivityManagerService) and the Binder may further send these intents on different threads and not on the main application thread, and the intents are usually forwarded to the receiver application with context of these Manager Services, thus with their Sender UID (1000) and not exactly of the origin app UID. This situation can further be worsened when there are multiple intermediary applications involved between source and the destination of the intent as in case of Pending Intents, for example, since the sender UIDs keep getting changed by the Binder.

Thus, the provenance information is lost within the channel and the receiver application is not able to determine whether the source application is authorized to request such operation from itself.

Design Description:-

The loss of provenance information opens android application to wide range of attack surfaces like Confused Deputy Attack, Intent Hijacking & Spoofing, etc. In Confused Deputy Attack, the malicious app may alter the Sender UID of the intent pointing to some other application and ask

for operations for which the original application is not given permission but the other application has. In this case, the receiver application would think that the intent has originated from the other application with the UID and not the actual source, thus causing Confused Deputy.

In Intent Spoofing attack, the malicious application may send an intent to other application with action string that is expected to be sent by some other application while in Intent Hijacking, the malicious app may act as man-in-the-middle and eavesdrop on the broadcast traffic or actively drop or inject data into the intent. These man-in-the-middle apps can also invoke implicit intent with malicious components such as phishing, etc.

In order to prevent such attacks, the solution could be to design or rather modify the transmission channel of the Intents or broadcasts in such a way that the provenance information is carried along from source to destination application and is not lost in between. For such implementation, the intent class could itself be modified to hold additional properties i.e. sender UID and PID, which is assigned internally at the time of intent creation and the sender UIDs are appropriately assigned at each intermediary applications according to the requirement. This design does not rely on Binder IPC to provide information about sender and is totally independent of the changes over threads. Thus, the receiver would be able to receive original creator of the intent as well as the immediate sender of the intent, and can further exercise access-controls needed for the operation to either execute or discard.

Implementation:-

As part of implementing the above-mentioned solution to ensure that the provenance information is carried all along the path that an intent traverses between an inter process communication, it is needed to modify existing framework by adding properties and access-control checks at various levels. Thus, all the files modified to serve this purpose are listed as follows:-

1. Intent.java
2. ActiveManagerService.java
3. ContextImpl.java
4. PendingIntent.java

In *Intent. Java*, two properties are added as part of Intent class – *senderUID* & *creatorUID*, which intends to carry the desired provenance information from source to destination application. Moreover, these properties are initialized in the copy constructors appropriately by assigning both as `Process.myUid(...)`, in case a fresh intent is instantiated. When a copy constructor copies the intent data to another intent object, the current sender UID is set to new *senderUID* while the original *creatorUID* from the received intent is assigned to the *creatorUID* of new intent. The latter operations are performed in functions – `writeToParcel(...), fillIn(...)` and `readFromParcel(...)`.

In *ActiveMangerService.java*, the startActivity(...) & startService(...) defined is invoked whenever a normal activity or service is started by the application. The access-control check is implemented

in these methods to check whether the *senderUID* matches with the actual UID of the user (`Binder.myUid(...)`) sending it to Activity Manager or not. In case it matches, the functionality proceeds as expected, while in the other case, it rewrites the *senderUID* as per corresponding UID in Binder IPC and passes it further.

It may even be instrumented to stop functionality and throw Security Exception if desired, but is often not a sought-after solution. This check ensures that the *senderUID* and *creatorUID* is not maliciously spoofed by the application and corresponds to actual UIDs. However, it is worth-noting that the services are always handled on the same thread by the Binder and doesn't lead to provenance information loss in normal service calls by applications and *senderUID* is preserved.

The `sendBroadCast(...)` is defined in the *ContextImpl.java* and similar access-control checks are performed in the methods when invoked by starting normal broadcasts by applications. The access-control enforced is also identical to the above-mentioned policy and thus, results in prevention of malicious alteration of senderUID by application.

The methods – `getActivity(...)`, `getBroadcast(...)`, `getService(...)`, defined in *PendingIntent.java* is invoked whenever a pending intent is supposed to be created by the application. This function internally calls `getIntentSender(…)` defined in *ActiveMangerService.java* and `send(…)` defined in *PendingRecordIntent.java*. Thus, the pending intent is created and received by the source.

The access-control check is enforced at the *PendingIntent.java* at corresponding methods itself so that correct *creatorUID* as well as *senderUID* is set initially. Further, when these intents are passed from one application to another, the intent is copied using `fillIn(…)` copy constructor, unlike default intent instantiation and the *senderUID* is appropriately assigned and also making sure that the *creatorUID* is appropriately passed to the new intent as mentioned above in normal intent scenario.

Conclusion:-

Testing the above-mentioned changes by using the TestApp module provided as well as by manual spoofing of UIDs by hardcoding it in the *Intent.java* has resulted in positive result. The *senderUID* and *creatorUID* in both the cases are assigned by source and received by the destination appropriately and as expected. Implementing this solution helps make understand that the fact that while these solutions try to eliminate widely observed attack surfaces due to loss of basic information about intent source and allowed permissions, addition of basic properties to the transaction unit – intent, here, can help prevent these losses by not hitting significant performance of the applications. However, this solution has scope to be extended further to eliminate Collusion attack surfaces by monitoring potentially malicious set of permissions requested for the application, if feasible.

Reference – System-Centric IPC Provenance on Android

Solution 2:- (*Clipboard Service as USOM*)

Problem Statement:-

There are services such as Clipboard, Location, etc. offered by Android which are common to all the applications installed in the device. However, the data created by one app using these services, sometimes, is not desired to be available to other applications in the device space but they are anyway. This may lead to potential privacy leak and thus compromise the security of the application in presence of another malicious application installed on the device. The traditional DAC permission model doesn't suffice the requirement of isolating app data object for the shared services between application.

Design Description:-

In order to enforce control over the data objects that an application utilizes while using shared services, they can be extended to User-Space Object Manager, i.e., they control the data objects created by themselves by assigning contexts which allows only those application to access who have similar subject contexts or types assigned to them, otherwise, these objects are inaccessible to them. In this project, we intend to extend ClipBoard Service as USOM by allowing corporate apps and platform apps to access primary clips set by any app domain on the device, while on the other hand, untrusted apps may be able to read primary clip set by only themselves and not by corporate apps.

The SELinux policy model based on MAC permission can be utilized extensively to extend service modules to USOM by type enforcement and hooking access-control calls to the existing functions that allow the flow of data from service space to the application. This provided an added layer of defense to unauthorized encroachment of data from user space of application to another.

Implementation:-

To implement the above-proposed solution, a new SELinux policy type is need to be defined for corporate apps along with their appropriate allow rules. Moreover, type transitions need to be defined for both untrusted apps as well as corporate apps so that correct object context is assigned each time an object, or rather a clip is created by the application. To initially assign subject contexts to all the apps based on their signature, additional keys need to be added as well as app contexts need to be modified for SELinux. All the files that are modified as part of implementing solution are as follows:-

1. mac_permissions.xml
2. keys.conf
3. seapp_contexts
4. security_classes
5. access_vectors
6. untrusted_app.te
7. system_server.te

8. app.te
9. SELinux.java
10. android_os_SELinux.cpp
11. ClipboardService.java

In *keys.conf*, the key which is used to sign corporate apps i.e. “corp.pem”, is mapped with tag – “@CORPORATE” and thus will be picked from here while checking the signatures while building the source code. The “corp.pem” file is uploaded and added appropriately in the directory where all keys are stored.

In *mac_permission.xml*, the seinfo value is assigned to the app with respect to the signature and the tag name that the app contains after build.

In *seapp_contexts*, the apps with seinfo value as “corp_app” are mapped to domain with type “corp_app” while all other apps are treated as “untrusted_app” by default.

In *security_classes*, the object class “clipservice” is defined. While in *access_vectors*, the object class – “clipservice” as well along with its operations are defined.

In *untrusted_app.te*, the type *corp_app* along with all the necessary allow rules that *untrusted_app* has is defined. Also, the type transition and allow rules for clipboard operations are defined in this file at the end of each respective sections.

In *system_server.te*, the allow rules necessary for the process to access `computeContextCreate(…)` method is defined – getattr & search.

In *app.te*, corp_app is included in the never_allow rule so that it is only allowed those actions which untrusted_app is allowed to have at initial stage.

SELinux.java holds the declaration of the computeContextCreate(…) method that assigns the context to each clipboard object/clips. This method is further defined in the native library at – *android_os_SELinux.cpp*.

The *ClipBoardService.java* file contains all the necessary hooks and access-checks that is need to be added into the existing functionality to implement Clipboard Service as USOM. While setting the clip itself – `setPrimaryClip(…)` which internally calls `setPrimaryClipInternal(…)`, the subject context and object context is determined and stored along with the clip as additional properties as per the parent app – two properties added in the *PerUserClipboard* class – *securityContext*, *objectContext*. Further, other methods such as `hasPrimaryClip(…)`, `getPrimaryClip(…)`, `hasClipBoardText(…)`, `getPrimaryClipDescription(…)`, etc. are also modified such that before doing any further operation, it checks the context of the caller with respect to the primary clip stored in the Clipboard and thus, return the clip or return empty value as per return type so that the at the receiver side, it seems as if the clipboard is empty.

Additionally, the *ListenerInfo* class and `addPrimaryClipChangedListenerListener(…)` is modified so that only those listeners which are authorized to listen the change in clip as per the

subject contexts of the listener and the subject of the primary clip is sent the onChange notifications while the others are not notified.

An additional function – `getSecurityContext(…)` is defined at the end of the file which calls the `computeContextCreate(…)` method wherever necessary as mentioned above.

Conclusion:-

With the help of provided *TestApp* module, we could test our above-mentioned changes and find that the modifications implemented have been successful with terms of the requirements that we initially decided to resolve. The Clipboard Service, acting as USOM, is successfully able to assign and enforce type contexts – *corp_app_t* or *untrusted_app_t* to the clips and able to provide or isolate accesses on the basis of subject contexts of the application along with the primary clip. This tasks help us understand that every service entity could be similarly extended to act as USOM at OS level which allows different app domains to enjoy different privileges as desired by the OS.

However, there could be significant challenge in extending USOM to some of the app domains where applications in same domain itself wants to isolate their data to some apps while provide access to others and could be taken as part of future work.

References - Android Security Framework: Enabling Generic and Extensible Access Control on Android, Flexible and Fine-Grained Mandatory Access Control on Android for Diverse Security and Privacy Policies

Note: As mentioned in the CMS, the patch has not been created, however, the VM is running as expected without any compilation error.