May 2016

Intentio Ex Machina: Android Intent Access Control via an Extensible Application Hook

Carter Yagemann
Syracuse University

Abstract

Android's intent framework facilitates binder based interprocess communication (IPC) and encourages application developers to utilize IPC in their applications with a frequency unseen in traditional desktop environments. The increased volume of IPC present in Android devices, coupled with intent's ability to implicitly find valid receivers for IPC, bring about new security challenges to the computing security landscape.

This work proposes *Intentio Ex Machina*[1] (IEM), an access control solution for Android intent IPC security. IEM separates the logic for performing access control from where the intents are intercepted by placing an interface in the Android framework. This allows the access control logic to be placed inside a normal application and reached via the interface. The app, called a “user firewall”, can then receive intents as they enter the system and inspect them. Not only can the user firewall allow or block intents, but it can even—within designed limitations—modify them. Since the user firewall runs as a normal user application, developers are free to create their own user firewall applications which users can then download and enable. In this way, IEM creates a new genre of security application for Android systems allowing for creative and interactive approaches to active IPC defense.

[1] Latin for *intent of the machine*. *Ex Machina* is an acronym meaning *Extensible Mandatory Access Control Hook Integrating Normal Applications*.

Intentio Ex Machina: Android Intent Access Control via an Extensible Application Hook

by

Carter Yagemann

B.S., Syracuse University, 2015

Thesis
Submitted in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science.

Syracuse University
May 2016

Acknowledgment

I would like to thank my advisor Dr. Wenliang Du for guiding and supporting me on the two year journey that has lead to the creation of this thesis. His perspective and feedback were invaluable to this work, as were the opportunities and resources he provided me with.

I would also like to acknowledge my coworker Amit Ahlawat who aided me greatly in establishing my foundational understanding of Android security and who also assisted me in finding the direction for my research.

I am grateful to my colleagues for sharing their knowledge and insights. No research is conducted in a vacuum and my work builds on the shoulders of theirs.

Finally, I wish to acknowledge my friends and family for their encouragement and emotional support. Research is a never ending process of self doubting and without my parents, Jacqueline, Daniel, Tim, Ningen, and countless others, I would have given up long ago.

World Three forever.

Table of Contents

Table of Figures

1 Introduction

1.1 Overview

One of the constraints which has shaped the design of Android is the limited hardware resources of the devices it is intended to run on. Due to limited memory, Android's creators wanted to design an architecture which encourages apps to leverage the functionalities and capabilities of the other apps already present on the device. By doing so, Android devices can conserve memory by avoiding overlapping code between apps. This philosophy deviates from the traditional desktop environment where apps frequently try to minimize their dependency on other software by implementing all their own code.

To illustrate this difference, consider the task of taking a picture using a device's camera. In the traditional computing environment, the app would contain all the code necessary to access the camera, take the picture, and store said picture for future use. This is done to ensure that when a user installs the desktop app, it will run without the need for any additional software. On the contrary, in the Android environment, every Android device comes pre-installed with a default camera app. When a third-party app needs to take a picture using the camera, it can ask the already installed camera app, through interprocess communication (IPC), to take a picture on its behalf and return a reference for the picture to the requesting app so it can access the picture later.

It is also important to note that while this example Android app happened to utilize the capabilities of the default camera app, it did not have to know the default camera app's name or process explicitly in order to do so. Instead, Android's system server, a privileged system app running in user space, receives from the third-party app a request describing the task it is trying to perform and the system server finds on behalf of the third-party app an appropriate receiver. More of this mechanism will be explained later, but for now it is sufficient to note the implicit nature of this IPC transaction and how it strongly contrasts the explicit IPC of traditional desktop systems.

Due to this shift in app design philosophy, IPC occurs more frequently and among more apps in Android than in desktop systems like Windows or Linux. From the security perspective, this increased utilization of IPC and the frequently implicit nature of Android's IPC gives rise to interesting and unique security concerns.

First, since an app does not need to explicitly know a receiver in order to perform IPC, an app on Android is able to invoke any other app which has components registered for handling requests. This is concerning for the receiving app because by registering components for request handling, the app is opening itself to requests which could come from any other app on the device, including apps which may be malicious or come from untrusted sources. If the receiving app's exposed components happen to contain vulnerabilities, the attack surface for exploiting these vulnerabilities becomes system wide. Attacks related to this problem have already been observed in the wild and are categorized as component hijacking[1].

Second, since any app can register its components to handle any type of request, sender apps have cause for concern regarding where their data will end up. If a malicious app registers itself to handle a wide variety of requests, the sender app could find its data being exfiltrated into the hands of devious actors. This sort of attack has been categorized in other works as intent spoofing[2][3].

These problems have motivated both researchers and developers to seek solutions. My solution for intent IPC access control is an architecture which leverages the system-centric and standardized nature of intent IPC. IEM is motivated by the insight that while every firewall intercepts packets and takes actions based on some decision engine, these pieces don't have to reside near each other. Specifically, IEM replaces the intent firewall's engine with an interface which can bind to a normal app. This "user firewall" can then act as the system's intent firewall. The user is free to install a user firewall in any of the ways they would normally install an app on their device. By placing the decision engine in an app, and not in a framework component like the intent firewall, developers can easily design the engine to utilize all the capabilities of any normal app including pushed updates which don't require rebooting or flashing, rich graphically enhanced user interaction, and access to system resources such as GPS and networking. In other words, rather than trying to be the be-all end-all solution to intent security in Android, IEM is a generic platform upon which developers can easily create, deploy, and maintain user firewalls. These user firewalls can then evolve with the changing threat landscape to meet the needs of their respective users.

I have made a virtual machine image containing IEM and a proof-of-concept user firewall available for download[2].

1.2 Related Work

This subsection discusses the IPC security mechanisms already present in the Android system as well as proposed designs from related research. These security architectures can be categorized into two general categories: access control and virtualization.

1.2.1 Access Control

In the access control category, we first find the sender permissions mechanism currently implemented in official versions of Android. This security feature allows receivers to require of the request sender a particular permission. This mechanism improves security by allowing for some restriction in which senders can invoke the receiver's exposed components, but it has its glaring limitations[4]. First, the receiver can only specify a single permission which the sender must have. Since it is common for Android apps to have multiple permissions, this means that the receiver's exposed components can be invoked by apps of lesser privilege. Second, even in the case of requester apps of equal or greater privilege, privilege and trustworthiness are not strongly correlated[5][6]. Applications coming from a variety of sources can request any combination of permissions and these permissions are granted upon approval by the user during installation. This makes it plausible for a malicious app to have as many, if not more, permissions as the receiving app it is trying to exploit. These problems have been the motivation for works such as XmanDroid[7], Saint[8], CRePE[9], and others[10][11][12][13][14]. Even if the developer of the receiving app wants to explicitly check who the sender of the intent is, his app

[2]http://jupiter.syr.edu/iem.amp.html

can only see the last app to send the intent. ChainDroid[15] and Scippa[16] both demonstrate situations where this is inadequate for enforcing access control.

The other IPC access control mechanism present in Android is the intent firewall. Unfortunately, this firewall also has major shortcomings in the robustness of its rule set which is why very few production Android devices have intent firewall policies despite the firewall being present and enabled[17]. It can be configured via SEAndroid[18].

Other works, such as Boxify[19], use runtime sandboxing to force untrusted apps to send their system transactions through additional access control mechanisms. These solutions can also restrict Binder IPC, but implementing them requires expert knowledge of Linux IPC and syscalls. Since they work at the native level, the context of the transaction is obscured. IEM user firewalls use concepts the average app developer is already familiar with.

My work is conceptually similar to Android Security Modules[20], but differentiates itself in two key aspects. First, while ASM only facilitates the monitoring of resources, my work enables modification for the purposes of redirection and data sanitization. Second, ASM uses callback timeouts; a limitation which violates the design goals of this work.

1.2.2 Virtualization

On the virtualization side of Android security, solutions attempt to achieve isolation between processes by virtualizing different portions of the Android device. One solution, Cells[21], achieves this isolation by creating virtual devices which run on top of the host device. Another solution, Airbag[22], also achieves process isolation, but rather than creating full virtual devices, this solution creates virtual system servers which prevents processes from different containers from communicating. There are also other works which implement isolation, such as TrustDroid[23].

Virtualization is an attractive approach to solving Android app security because it not only controls the interactions between apps, but it goes a step further and isolates them. Not only is an app not allowed to communicate with another app, it cannot communicate with another app. This isolation, however, comes with side effects; both in implementation and in runtime reconfiguration.

First, virtualization requires the duplication of objects. Where there was originally one object, there are now multiple which increases resource consumption and creates overhead when switching between virtual objects.

Second, components can break if they are duplicated naively. Both Cells and Airbag ran into this problem when they duplicated the Android SurfaceFlinger. This Android component handles writing to the device's screen and functions under the assumption that it is the only object at its level trying to access the screen. Duplicate this component, and now multiple SurfaceFlinger are trying to write to the device's screen simultaneously. To patch this side effect, the Android system must be modified to multiplex these competing signals. This increases the complexity of implementing and maintaining the code needed for virtualization which makes real-world deployment less practical.

Lastly, virtualization solutions lack flexibility when security configurations require change. Since virtualization creates a stronger isolation between apps than access control, reconfiguring virtualization becomes a more costly task involving migrating apps from one container to another. This is in contrast to access control which simply has to load and parse the new policy.

I chose an access control design for IEM because I want to leverage the unique nature of intent IPC. Specifically, IEM leverages the fact that all intents must travel through the Android system server using a standardized message format which the system can understand. A virtualization solution would not leverage the semantic understanding the system server has of intent messages.

2 Background

2.1 Intent

Intents are a framework level abstraction of Linux Binder IPC. They serve as a simplified and standardized object for communicating with other apps and system services. Using intents also carries the added benefit that if a user apps does not explicitly know the name of a valid receiver for a particular IPC transaction, the user app can implicitly describe a receiver based on the task they're trying to perform and then the Android system server will find a suitable receiver.

The Android system server maintains a binder of handles for use in delivering intents to other apps. Whenever a system or user app is created, it is assigned a binder handle which allows it to communicate with the system server. Upon start up, every app uses its binder handle to the system server to register itself as an active app on the Android device. This registration process includes the registering app giving the system server a binder handle to itself which is then stored for future use.

Upon receiving an intent, the system server has to resolve which component should receive that intent. If a target component is explicitly specified by the sending app, then the system server just checks the intent against the specified receiver's intent filter and confirms that they match. Otherwise, the system server will search its list of intent filters to find the eligible receivers for that intent. If the intent is intended for an activity, the system server will generate a list of all the valid receivers and then the user will be prompted to pick one. If the intent is for a service, the system server will pick the first eligible receiver. If the intent is a broadcast, then all the subscribed receivers will be selected.

Once the receiver or receivers have been resolved, the intent is then passed to the intent firewall. The firewall compares the intent against its policies and decides if it should be dropped or allowed. If it is dropped, that's the end for that intent. If it is allowed, the system server then looks up the binder handle for the receiver, or starts the receiving app if it isn't currently running, and delivers the intent to the receiver's message queue.

Finally, the receiver processes the intent and if a response is necessary the system server will forward it back to the original sender. These transactions occur asynchronously since they involve communicating across processes. Therefore, the original sender receives the results via a callback invoked by the system server.

2.2 Activity Manager Service

The system server in Android is a privileged process which runs in user space and which apps communicate with via binder transactions. The system server itself can be further divided into a collection of services, each of which is designed to manage particular tasks in the Android system. Since IEM is itself an object within one of the services, some background knowledge on them is necessary.

Activity manager service (AMS) is the main service for handling intents. Every user app is created with a binder handle for reaching AMS and most communication between apps starts with the AMS. It has a collection of public methods for receiving intents and communicates with all the other system server services to make sure that the intent is resolved, permissions are checked, and the receiver is running and ready to receive the intent. Within the activity manager, there are four other components of interest to the IEM. The first is the intent firewall—explained in more detail later—which inspects every intent. Aside from that, there is the activity stack supervisor, the broadcast queue, and the active services components.

The activity stack supervisor manages the activity components of the user apps and is responsible for handling activity intents. This includes resolving the intent, checking if the intended receiver is already running, and checking with the intent firewall that the intent should be allowed.

The broadcast queue is the component within activity manager for handling broadcast intents. This includes resolving the intent, queuing up the intent for delivery to the decided receivers, and managing the queue as broadcasts are canceled or updated. As with activity stack supervisor for activity intents, this is the component which will check with the intent firewall before allowing broadcast intents.

Finally, active services is the activity manager component for handling service intents. Like the other two components, this includes resolving the intent and checking with the intent firewall that it should be allowed, but uniquely this component handles scheduling the services to be started, stopped, and restarted as necessary.

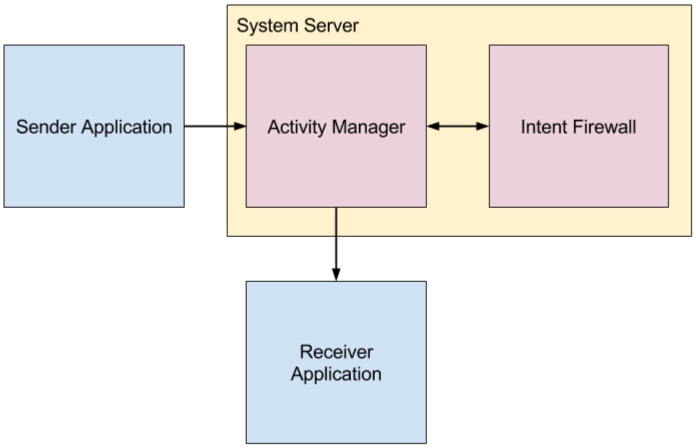


Illustration 1: Android Intent Firewall.

2.3 Intent Firewall

The intent firewall is an access control mechanism originally introduced in Android 4.3 and is present in all production Android devices; although almost none of them use it. The version of the intent firewall present in version 4.3 was more of a place holder than an actual firewall and it wasn't until version 4.4 that most of its currently existing features were implemented. As of Android 5.1 (Lollipop), almost no code has been modified from the 4.4 version. SEAndroid's configuration files support intent firewall policies[18], but few leverage this. The main purpose of the intent firewall is to buy time during a major malware outbreak by allowing device manufacturers to push policies which will explicitly block the malware's IPC[17]. Since such outbreaks are extremely rare, the intent firewall has almost never been used. Illustration 1 shows how the intent firewall fits into the framework.

Policies for the intent firewall are defined using one or more XML formatted files that are then written to a protected system directory on the Android device. Since this directory is a system directory, it is only accessible to privileged apps which prevents arbitrary apps from reconfiguring the firewall. This directory is checked at boot time by the intent firewall and then continuously monitored for changes. If a change is detected, the intent firewall will automatically try to parse any new or modified files with the XML extension for firewall rules. If a file is deleted from this directory, the intent firewall will automatically delete the rules associated with that file.

The activity manager checks intents against the intent firewall near the end of the flow. Consequently, even though intents can be implicit or explicit when the sending app creates them, all intents are explicit by the time they enter the intent firewall. Also, all three types of intents (activity, service, and broadcast) are checked by the intent firewall before reaching the receiver. In the case of pending intents, a special mechanism where an app can store an intent in the activity manager and trigger it later, the pending intent is already an explicit intent by the time it enters the firewall. Although the intent firewall checks all intents going from the sender to the receiver, any responses returned to the sender are not checked. In other words, allowing an intent to be delivered to its intended receiver implicitly allows any response to be returned to the sender. IEM also makes this implication.

3 Design

This section formulates the main architectural goal of IEM and assesses the challenges in trying to achieve it. Illustration 2 contains a high-level overview of the design which will be explained in the following subsections.

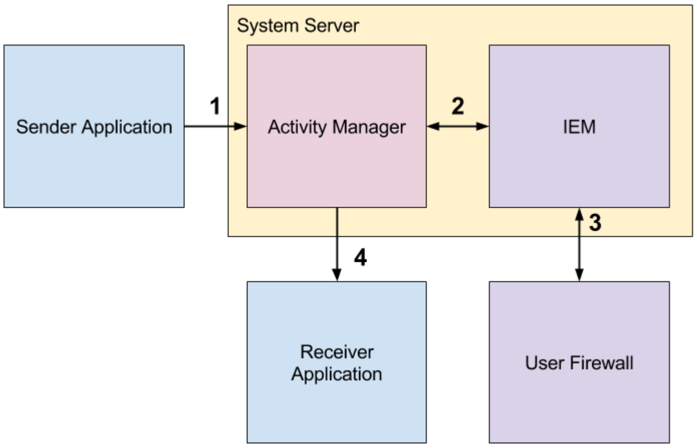


Illustration 2: Intentio Ex Machina.

3.1 Architectural Goal

All firewalls contain three critical pieces: the interceptor, the decision engine, and the policy. The interceptor captures the data packet and delivers it to the decision engine which then decides if the packet should be allowed or denied based on the configuration defined in the policy. From this model, we can make some insightful observations. First, there is little flexibility in where an interceptor can be placed since it has to be somewhere along the original path of the packet. On the other hand, the decision engine can be placed anywhere so long as the interceptor can reach it. Second, while the policy is easy to reconfigure, it is restricted by the logic of the decision engine. A particular policy can only refer to attributes which the engine defines.

With these two observations in mind, reconsider the intent firewall in Illustration 1. In this case, the interceptor and the decision engine both reside in the framework. The interceptor must be placed in the framework because it's the only point through which all intents must pass, but what about the decision engine? Since it resides in the framework, modification requires an OS patch which is quite inconvenient. Sure implementation is easiest when the engine is in the same spot as the interceptor, but what would happen if it were moved somewhere else? Specifically, what would happen if the engine was placed inside a normal app? It could then be installed, updated, and maintained with the ease of any other app. Now the policy is less restricted by the engine because said engine can easily be swapped out for new logic.

With this idea in mind, the goal of IEM is to provide a hook in the intent flow to allow for an app to serve as the intent firewall. As previously mentioned, doing this makes changing the firewall's enforcement logic easier which in turn allows for more flexibility. Making the firewall an app empowers the user to install the solution which fits their needs. One user might download a user firewall which monitors his location and restricts the apps he can use while in the office. Another user might use a user firewall which prevents apps from getting her location while she's driving. A parent might put a user firewall on her child's device to prevent him from playing video games before dinner, or maybe he can text message his friends only after he finishes his math homework. These are all apps an Android developer can program, so these are all apps which IEM can empower.

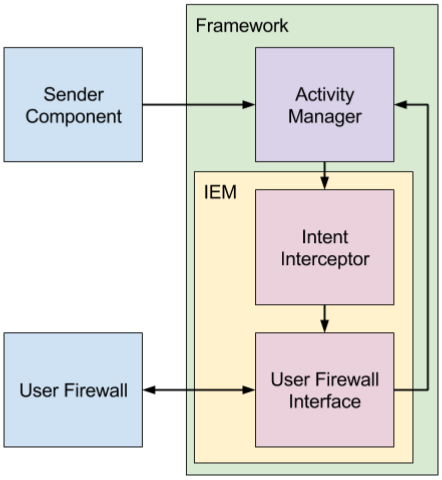


Illustration 3: The flow of intents through IEM.

Illustration 2 shows IEM and how it interfaces with the rest of the framework and other apps. From here on, “IEM” specifically refers to the hook which resides in the framework while “user firewall” refers to any app which leverages IEM. Edge 3 is the most novel part of the design because this edge does not exist in the original intent firewall. Intents first enter the system server through the public API at edge 1. This is where activity manager resolves the receiver. Following some basic security checks, the intent enters the IEM via edge 2. Illustration 3 highlights IEM's internal logic. The intent is delivered to the user firewall and a response is returned via edge 3. If the user firewall decides to allow the intent, the response will come back to IEM which will then pass the intent to the activity manager via edge 2. Finally, the intent is delivered via edge 4.

The next subsection identifies and addresses the key challenges in trying to achieve the IEM architecture.

3.2 Design Challenges

Illustration 3 shows that there are three new pieces which IEM introduces into the Android system. This subsection considers each piece in turn and addresses the challenges which arise due to their addition.

3.2.1 Intent Interceptor

The first new piece an intent reaches is IEM's interceptor. Which intents are appropriate to intercept? I began by intercepting all the intents and then quickly realized why this is a poor decision and consequently why this is not a trivial problem.

The problem is not trivial because we cannot assume that the user firewall will always be responsive since it's an app. Like any other app, it can crash or freeze. This is different from the original intent firewall which could simply intercept all intents because it didn't have to worry about the state of any other component.

Since the system server uses intents to start components, if the interceptor intercepts everything and the user firewall crashes, the system will no longer be able to start any component. The device is now stuck in an unrecoverable state. For this reason, intents created by the system are exempt from being intercepted.

Once system intents have been exempt, all that remains are app created intents. These can be safely forwarded because failing to deliver them will only impact the app and not the system.

3.2.2 User Firewall Interface

After the interceptor, the intent next reaches the user firewall interface. In order for the user firewall to be able to inspect intents, this interface has to be able to send the appropriate intents to an app and get a response on how the intent should be handled.

How can the interface forward an intent to an app and get a response? The answer to this challenge can be derived from a feature which already exists in Android: the extensible framework. This design pattern works by having a system service bind to an app service which is chosen by the user in the device settings. Once bound, the system and app can exchange messages to communicate. This allows the user to install third party apps to serve as the keyboard, the settings administrator, the “daydream” screen when the device is docked, and more.

IEM mimics this architecture. When the user enables a user firewall, IEM will bind to that app's user firewall service. Once a binding has been established, the intercepted intents will be sent to the user firewall via that binding. The messages sent through the binding include a “reply to” handle which the user firewall can then use to return a response to the interface. If the user firewall decides to allow the intent, it sends a reply message via the binder handle. Otherwise, it can just discard the message.

There is another challenge regarding the design of the user firewall interface. In the original intent firewall, intents are processed entirely inside the framework. For IEM, there is now an interface performing IPC with a user app. This change carries consequences which must be addressed.

As mentioned earlier, the user firewall app could crash or freeze. If this happens, the user firewall interface will no longer get responses from the app. With this in mind, should the interface be allowed to maintain any notion of state when communicating with the user firewall? If state is allowed, this has to be done by either having the interface's worker threads block while performing IPC with the user firewall or by storing information in a database inside the system server. Both these options carry grave consequences because they can lead to resource exhaustion. Even if a timeout mechanism is implemented for preventing the buildup of blocked threads or data entries, the design will still be weakened because now an artificial time limit has been imposed on the user firewall for making access control decisions. If this timeout is short, then it no longer becomes possible for the user firewall to involve the user in the decision making process. This reduces the flexibility of the firewall logic which weakens the goal the architecture strives to achieve. If the timeout is long, the device will become unresponsive. There is no middle ground; either choice is detrimental to the goal of designing a platform which allows for the easy creation of many different firewall decision engines. For this reason, there shouldn't be any timeouts. The interface must be completely stateless.

Since IEM has to maintain statelessness and never be waiting on the user firewall app, the message sent to the user firewall has to contain all the necessary information so that the intent's state can later be reconstructed. To address this challenge, I introduce the concept of the intent wrapper. Illustration 4 summarizes its structure. The intent wrapper is a bundle containing the original intent along with everything necessary to duplicate the sending of the intent. This includes sender information; something which the original intent firewall doesn't consider. By wrapping these additional variables together along with the intent, the user firewall receives a complete picture of who the sender is, who the receiver is, and what interaction they're performing.

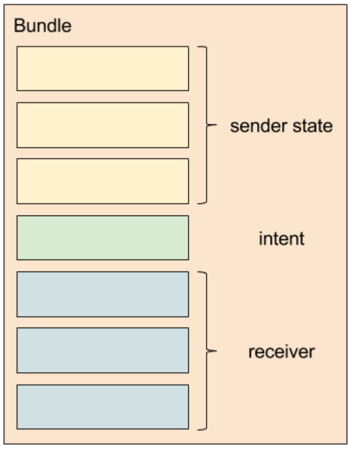


Illustration 4: Intent wrapper.

3.2.3 User Firewall

The intent now reaches the user firewall. Since this app has the intent wrapper containing its state, what should the firewall be allowed to modify? It could, for example, perform data sanitization or redirect the intent to a different receiver. However, allowing the user firewall to modify all the fields of the wrapper would be dangerous because this gives it the power to send any intent to any app on behalf of any other app. There is value in allowing for modification, but having no restrictions grants too much power. This problem is addressed by the intent token.

Inspiration for the intent token comes from the SYN cookie design which prevents SYN flooding by making the TCP handshake stateless.[24] Similarly, the intent token allows IEM to remain stateless while still restricting the user firewall app's power. Tokens are generated inside IEM by hashing a secret which it creates at startup. Randomized salts are used to ensure tokens are unique. As long as the secret is kept confidential, the user firewall will be unable to create valid tokens and consequently cannot send arbitrary intents on behalf of other apps. In this way, the user firewall is a privileged app, but only to the extent essential to it performing its role as an intent firewall.

More challenging, however, is deciding which other parts of the wrapper should be included in the hash to prevent modification. This answer can be determined by grouping the contents of the intent wrapper into the three mutually exclusive categories shown in Illustration 4: sender, intent, and receiver. The variables pertaining to the sender's identity stand out from those of the intent and receiver because the sender is the creator of the intent. If the user firewall changes who created the intent, all integrity is lost. The user firewall could create an intent and change the sender to be another app. On the other hand, changing the receiver seems reasonable because receiver resolution is the reason apps give intents to the Android system in the first place. For these reasons, the user firewall is allowed to modify the action to be performed and who will carry it out, but it cannot change who sent it. The complete intent token is shown in Illustration 5.

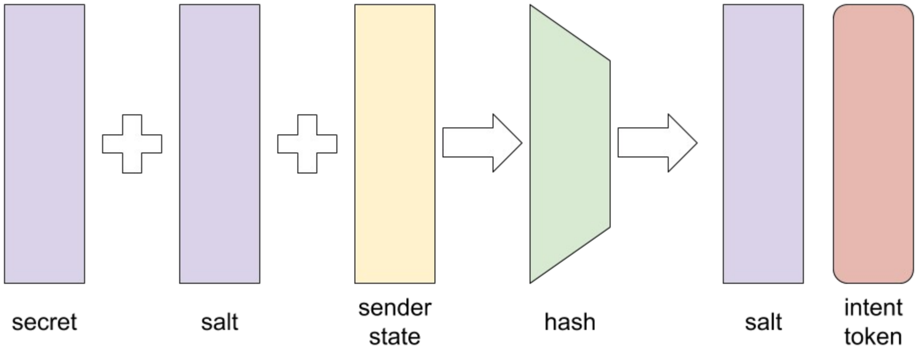


Illustration 5: Intent token.

3.3 Threat Model

Illustration 3 shows that the boundaries between the IEM, framework, and app spaces are crossed in four places. This subsection addresses the security of these crossings in the order that intents reach them.

The threat model for IEM assumes that the framework is secure and trustworthy. This includes the activity manager since it is apart of the framework. This assumption is made because IEM is designed solely to enforce intent security, so any compromise of other framework components is out-of-scope. The model also assumes that the secret created by IEM for generating tokens is kept secret. This is a safe assumption because IEM never needs to share this secret with any other party.

The first boundary crossing is from the sender in app space to the activity manager in the framework. This boundary is protected by activity manager's public API which is outside the control of IEM. Therefore, IEM assumes that this boundary is secure.

The next boundary is between IEM and the activity manager. Since the threat model already assumes that activity manager is trustworthy, the boundary is between two trusted parties. This makes the boundary secure.

After entering IEM, the intent next reaches the boundary between the user firewall interface in the IEM space and the user firewall in app space. This surface can be attacked by a malicious app so protection mechanisms are necessary.

Three actions occur over this boundary. First, the interface binds to the app. Second, the interface sends the app intents to inspect. Third, the interface receives intents from the app. Attacking the first action requires the attacker to bind to either the interface or the user firewall. The interface protects against this by disallowing apps from initiating the binding process. Instead, it is always the interface which initiates the binding and since it gets its target from the device settings, it will bind with the correct service. Since the interface is apart of the system, the user firewall can differentiate the attacker from the interface by checking the UID of the bind request. Attacking the second and third actions can be performed by either the user firewall or another app. If the attacker is another app, it will have to sniff and spoof binder messages. Since the Binder is part of the Linux kernel, this is out-of-scope for IEM. In the case where the attacker is the user firewall, spoofing cannot occur since the contents of the message are protected by the intent token. This covers all the actions which occur over this boundary.

The last boundary crossing is from the user firewall interface to the activity manager. This path is used to deliver the allowed intents to their receivers. An attacker targeting this boundary has to spoof a user firewall response, but as stated earlier, this is prevented by the intent token. Therefore, this boundary is secure.

4 Implementation

Intent Firewall & IEM	
All	intent, caller uid, caller pid, resolved type
Activity	resolved receiver
Service	resolved service, resolved application
Broadcast	receiver uid
IEM only	
Activity	caller thread, caller package name, caller userid, request code, caller binder, start flags, options bundle
Service	caller thread, caller binder, caller callbacks, caller package name, flags, caller userid
Broadcast	caller thread, intent receiver, result code, result data, required permission, appop flags, is sticky, is serialized, receiver userid

Table 1: Breakdown of parameters by intent type.

This section covers the details and challenges faced in implementing IEM in Android AOSP. The problems are presented in order of the edges they pertain to in Illustration 2. The particular versions of Android which IEM is implemented in are 5.0.2 and 5.1.1.

4.1.1 Sender API Return Values

Starting with edge 1 of Illustration 2, we must consider the return values of the methods used by apps to send intents to the activity manager. Although the actual process of delivering the intent to the intended receiver is carried out asynchronously, a return value is synchronously returned to the sender. For example, when an app requests to have a service started, activity manager will return the name of the resolved service to the sender.

This synchronously returned value poses a design challenge for IEM because once it forwards the intent to the user firewall, activity manager drops the intent. This is necessary in order to remain stateless. Since the intent was dropped, the value returned to the sending app will indicate a failure; regardless of the user firewall's response. In practice, this is not a problem for activity or broadcast intents because developers do not invoke the API directly but rather use the Context class. An exception is raised if the value returned indicates a failure, but is then immediately discarded in an empty catch block. Nothing is returned to the app code which invoked Context; neither in the successful case, nor in the unsuccessful case.

Services, however, are often invoked in order to perform tasks which are of critical importance to the app. For this reason, Context does make sure the value returned by the activity manager reaches the calling app code and some apps are designed to stop if the response doesn't match expectations. Google's music app, for example, will raise an exception and crash if it believes it failed to start its service which manages the downloading of music streams. For this reason, simply removing the return value is not an option.

Since waiting for the user firewall's response is also not an option given the design goals, IEM has to return the original value even though the user firewall might decide to block the service intent. I have evaluated this side effect considerably and deemed it to be acceptable. In the case that the user firewall decides to allow the service intent, nothing changes from the sender's perspective since the task is carried out asynchronously. If the user firewall does block the service intent, then the sender will never get a response from the service it tried to start or bind. Since this communication is inherently asynchronous, Android apps already account for the case where a response isn't received. Google's music app, for example, continues working without issue aside from the fact that it won't be able to stream music. Since the app doesn't crash or freeze, this outcome is acceptable.

4.1.2 Intent Firewall API

This brings us to edge 2. In the original intent firewall, interactions with the activity manager are performed via one of three exposed methods. They're separated based on the type of intent they process; activity, service, or broadcast. This separation is necessary since the different intents require different internal data structures. IEM also uses three public methods for this reason. However, the arguments have been modified. Since it was decided that IEM should be stateless, additional parameters which are not necessary in the original intent firewall design are now required. It is now necessary for the activity manager to give IEM these internal data structures regarding the state of the intent transaction. These new values were systematically selected based on the parameters needed by activity manager's intent sending API. Table 1 compares the original parameters with the new ones. Remember that the values pertaining to the sender's identity are tokened to prevent modification by the user firewall.

With activity and broadcast intents, all the public API in the activity manager ultimately call a single API. The different entry points are merely for convenience. This makes it easy for IEM to later invoke the correct API when it needs to restore an intent's state. This is not the case for service intents. They use four different API which each perform a unique action: start, stop, bind, or peek. If IEM doesn't invoke the correct API, the wrong action will be performed.

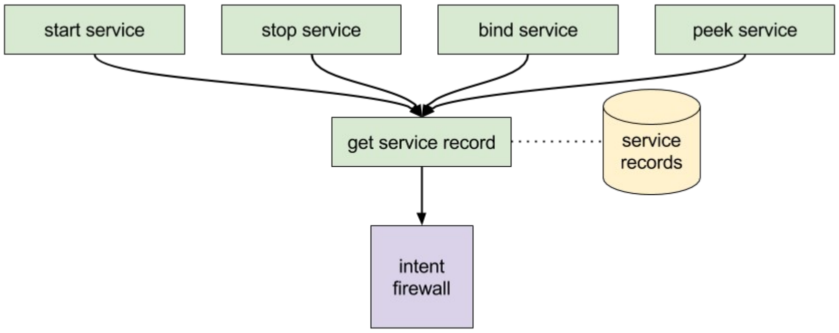


Illustration 6: Service API flow.

In the original firewall, all four service API first resolve the intent and then call the same method to retrieve the appropriate record for the receiver. It is this method which invokes the intent firewall, therefore the action being performed is not known to the firewall. The problem is illustrated in Illustration 6. This is fine for the original design, but not for IEM.

To restore this lost information, the receiver lookup method has to be modified to keep track of which action is being performed. This is a side effect of making IEM stateless. The change, however, is minimal. The total impact to the Android AOSP code is presented in the evaluation section.

4.1.3 Intent Unparceling

This brings us to edge 3. A significant implementation choice which has to be made here is where to store the intent token. Since the goal is to minimize how much of Android AOSP is impacted by the IEM implementation, I initially decided to store the token in the extras field of the intent. This ended up causing system crashes and after further study, I discovered why. The problem has to do with how the intent extras field is structured.

The extras field is a bundle containing any additional information which the receiver of an intent needs. Since intents have to be able to move through the binder, the extras bundle is comprised of only basic data types and parcelable Java objects which can be flattened into basic data types and then later reinitialized. Basic data types are always the same size, but some parcelable objects, like the Java string, are not. This means that if a process wants to read the contents of the extras, it has to somehow determine the boundary between objects in the extras bundle. This is done by placing an integer before each item in the extras which identifies the data type of the object.

The problem, however, is what happens if the parsing process reaches an item with an unknown identifying value? Since it does not know what the data type is, it cannot know how big the item is and therefore cannot parse the extras bundle beyond that point. When an app component uses a custom parcelable Java class, the developer gives this class to all the other components which may need to communicate with that component. This is fine for app developers, but not for IEM. For this reason, the token cannot be stored in the extras field of the intent.

Modifying intent is now unavoidable, so the question becomes how should it be modified? The two options are to either modify the way bundles are structured and parsed or to create a new standalone field for the intent token.

Modifying the bundle would be wonderful since it would patch a known vulnerability[25] and enable user firewalls to always be able to check the extras field. Unfortunately, bundles are used extensively throughout the Android architecture. The system impact would be difficult to measure let alone evaluate.

Instead, IEM modifies intent to have an additional field which can hold the token. This does not cause any problems for currently existing apps since they never need to read or write to this field. Due to the virtual nature of the Android runtime environment, developers don't even need to change SDKs to write apps for an Android device using IEM. The modification made to intent for IEM has no impact on current or future Android apps. It is fully compatible with the current Android ecosystem.

5 Applications

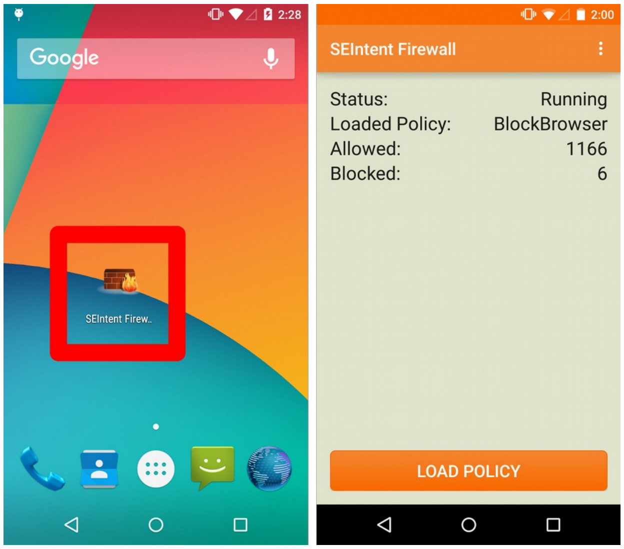
```
@Override
public void handleMessage(Message msg) {
  Bundle data = msg.getData();
  Bundle res = checkIntent(data);
  if (res != null) { //allow
    Message r = Message.obtain(null, 1);
    r.setData(res);
    try {
      msg.replyTo.send(r);
    } catch (RemoteException e) {}
  }
  //blocked intents require no action
}
```

Drawing 1: UFW service handler template.

The generic nature of IEM allows developers to create many different kinds of user firewalls which can serve a variety of purposes. All the user firewalls which are presented in this section could be implemented in the original intent firewall, but doing so would be vastly impractical. Developers would need to have Android framework specific knowledge to access resources without using an SDK and each firewall would have to be tested extensively since implementation modifies the operating system. Some of the examples are designed to address very specific needs, so it would be very challenging to anticipate and generalize these firewalls to a degree which justifies implementing them into the official Android OS source code.

The developer of a user firewall only needs to implement a service component. Drawing 1 is a template for the handler. The data bundle contains all the objects listed in Table 1. This architecture gives the user firewall developer the flexibility to design the internal logic of his app however he desires to provide whatever services and features his end-users require.

In this section, I describe a few examples of user firewalls which are made possible by IEM. I have chosen just a small sample from the infinite number of possible user firewalls which I believe to be sufficient to demonstrate the flexibility of the IEM architecture.



Status:	Running
Loaded Policy:	BlockBrowser
Allowed:	1166
Blocked:	6

Illustration 7: Blocking intents.

5.1 Simple Firewall

For the first example, I will start with a more simplistic user firewall which implements the functionality of a traditional firewall. Illustration 7, is an image of this simple user firewall. Users can select a policy and then the user firewall will enforce it.

In the case of Illustration 7, the user has selected the block browser policy. While this policy is being enforced, the user will be unable to open the browser. This means that tapping the browser's icon in the device's launcher will not start the app and tapping a URL in another app will not cause the browser's activity to start. The user firewall also contains some other example policies. For example, one policy will block the Google Play Service (GPS). While this service is blocked, apps will not be able to utilize Google Play for updating, performing in-app purchases, or any other functionality this service provides. Another policy will block Gmail's services. With this policy enabled, users can open Gmail and read their current email, but they will not be able to fetch or send new emails. This user firewall can even enforce policies which stop the user from accessing the devices settings or recent app menus, a capability which will be greatly leveraged in another example firewall. This example app also displays real-time statistics for the user like which policy is currently loaded and how many intents that policy has allowed or blocked.

Since this example is designed to adhere to the functionality of a traditional firewall, it does not leverage the intent rewriting functionality of the IEM. That will be utilized in the next example.

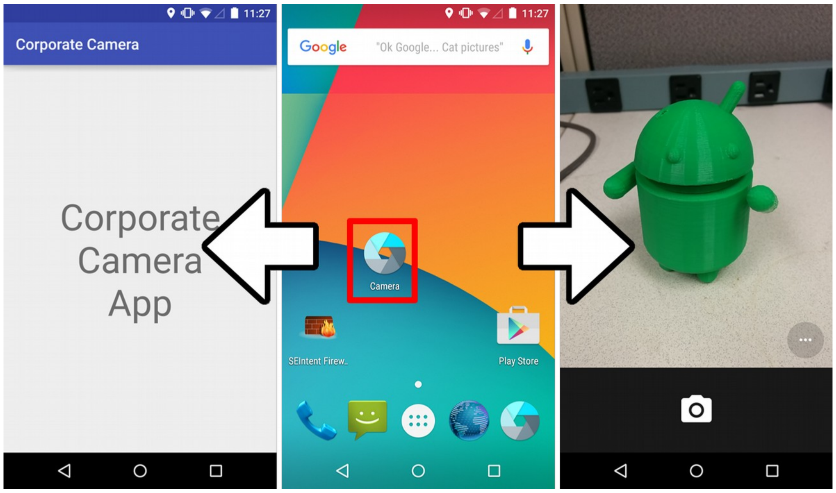


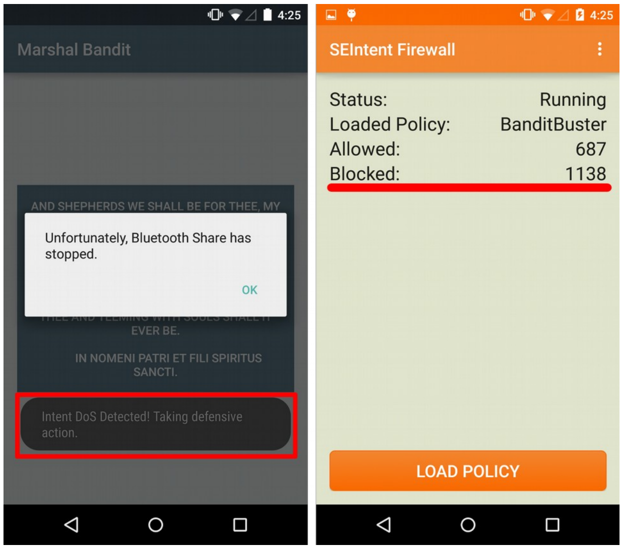
Illustration 8: Redirecting intents.

5.2 Redirecting Intents

Unlike traditional firewalls, user firewalls are not restricted to binary allow or deny access control. It is also possible for user firewalls to allow an intent, but modify some of its contents. This allows for some interesting use cases such as intent redirection.

Consider a corporation which is concerned about employees taking pictures using their phone while in the office. Banking institutions are an example since there may be sensitive information in documents and on computer monitors which could be captured when the photo is taken. Lets suppose that the corporation has created a camera app for their employees which is designed to only take “safe” photos. However, since this app is very restrictive, employees don't want to have to use the corporate camera app when they aren't at work. A user firewall can control which camera app is launched based on GPS location using intent redirection.

Illustration 8 demonstrates this case. When the user wants to launch the normal camera app, the user firewall will check the user's current location. If they aren't in the office, the intent will be allowed. If they are in the office, the intent will be redirect to the corporate camera app and it will show up instead.



AND SHEPHERDS WE SHALL BE FOR THEE, MY

Unfortunately, Bluetooth Share has stopped.

EVER BE.

IN NOMENI PATRI ET FILI SPIRITUS SANCTI.

Intent DoS Detected! Taking defensive action.

Status: Running
Loaded Policy: BanditBuster
Allowed: 687
Blocked: 1138

Illustration 9: Intent DoS detection.

5.3 Preventing Intent Denial of Service

When I implemented the intent token, I intentionally created a new field for it because attempting to read or write to any part of the extras bundle of an intent will raise an unmarshaling exception if any object in the extras is a custom class and the receiver does not have a definition for it. I discovered that most apps do not handle the unmarshaling exception and will crash. The Gmail app is one such example. This is a known vulnerability which Google has classified as being low severity[25].

To demonstrate the potential damage of this vulnerability, I created a malicious app called Marshal Bandit. Upon boot, Marshal Bandit queries the activity manager for all the running services and spams them with intents containing a custom object in the extras bundle. This causes services on the device to repeatedly crash and overwhelms activity manager's worker threads. The result is a denial of service which causes the device to become unresponsive and eventually crash. Since the user cannot access the Settings app while the attack is underway, the device is crippled. Even if the user is advance enough to know how to boot the device into safe mode, she won't know which app to uninstall. Marshal Bandit is a normal app with no granted permissions.

This type of attack cannot be stopped in current Android devices, but it can be stopped by a user firewall thanks to IEM. In Illustration 9, I demonstrate a user firewall which can detect the sudden flood of intents coming from the malicious app. Upon detection, the user firewall will inform the user which app is performing the attack while stopping the attacker's background processes, halting the spam of intents. The user can then regain control of the device and uninstall the malicious app. This user firewall is a normal app using only permissions which any app can request and IEM. The successful thwarting of this recently discovered denial of service attack demonstrates the flexibility and capability of IEM. Implementing logic of this complexity at the framework level would be too challenging compared to the narrow scope of attacks it addresses. With IEM, a user firewall app which addresses this vulnerability can be developed by a single developer in one work day.

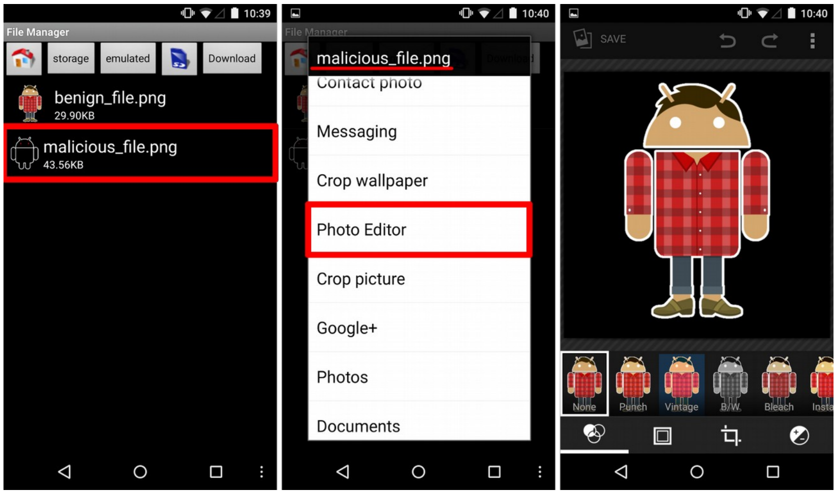


Illustration 10: Data sanitization.

5.4 Sanitizing Intent Data

Since user firewalls can modify the contents of the intent itself, a user firewall can perform data sanitization. This is applicable to trends such as “Bring Your Own Device” for the corporate environment.

If an Android app wants to share data with another app, most ways of doing so require an intent. The intent will either contain the data itself, a URI pointing to a file containing the data, or the intent will be for a service binding which will then be used to share data back and forth. In all three cases, a user firewall can either block or alter the data by dropping or modifying the initial intent. Illustration 10 demonstrates this functionality. In this example, when the user tries to open a malicious image file, the user firewall modifies the intent so a benign image is opened instead. This same technique can just as easily be applied to other types of data to either prevent data leakage or to protect apps from exploitation. This functionality is similar to the web application firewall (WAF) concept[26].

This is by no means a complete solution to controlling data flow—intents aren't involved in networking transactions—but IEM can allow user firewalls to solve some BYOD problems while being easy to develop, deploy, and maintain.

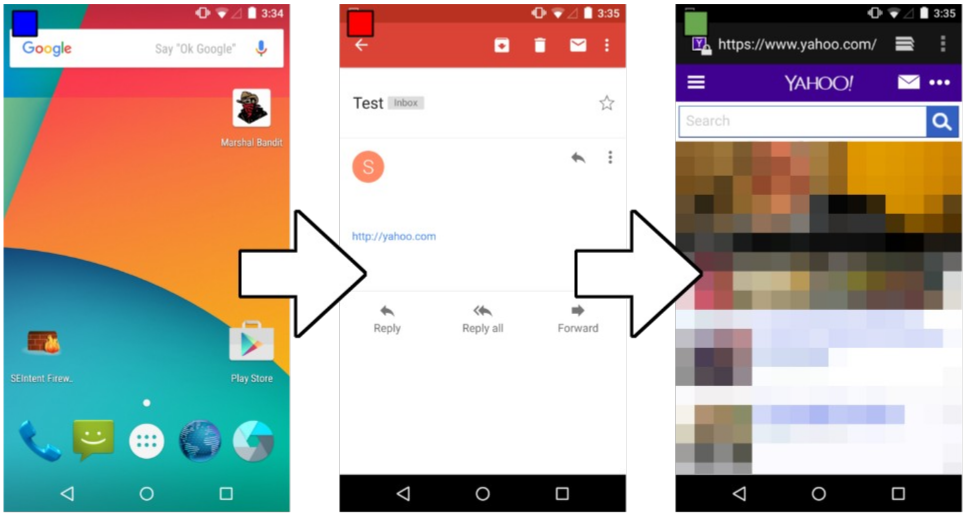
5.5 Isolating Applications

User firewalls leveraging IEM can also isolate applications from each other by inspecting the sender and receiver of every intent. This allows the user firewall app to create “secure containers”. Once the user places an app in a secure container, the user firewall can enforce a policy on the boundary between the container and the rest of the device. Maybe apps can't send intents to apps in the container. Maybe container apps can't send intents out. Maybe the contained apps can't bind to the location provider to get GPS position. There are many possible applications for this design such as parental control.

Current parental control apps struggle to keep the child confined to the safe environment because it is difficult to prevent the child from accessing the device's settings and recent apps list. Both of these actions involve sending an intent. Therefore, if the parental control apps were to become user firewalls, they could simply drop these intents and thereby prevent the child from escaping the parental control app's environment. It would also become possible for the app to enforce its parental control on the default launcher, as compared to current solutions which entail creating a new launcher on top of the default launcher.

Application isolation can be achieved knowing only the sender and receiver of the intent.

However, there are more complicated problems which user firewalls can tackle which require knowing the sender's sender. This is the problem of caller chains which is the focus of the next example application.



```
I/CallChain( 1264): com.google.android.gm <= {com.google.android.googlequicksearchbox}
I/CallChain( 1264): com.android.browser <= {com.google.android.gm,com.google.android.googlequicksearchbox}
```

Illustration 11: Caller chain.

5.6 Determining Caller Chains

One potential shortcoming with the Android permissions architecture is that it only considers the immediate sender of an intent. It does not account for the case where a chain of apps are invoked via intents. If an attacker invokes an app with slightly greater permissions and that in turn invokes another app with still greater permissions, the final receiver could be excessively more privileged than the original sender. This pattern can lead to privilege escalation[27][4][28][29][7][30]. Multiple works have identified this problem and implemented caller chains to resolve it[16]. However, all these solutions are relatively complex and require modification of the Android operating system.

Using IEM, it is easy to implement a user firewall which can track call chains. Illustration 11 demonstrates a user firewall which a single developer programmed in under an hour. When an intent enters this user firewall, it records the sender and receiver as a pair. The user firewall can then use these pairs to recursively determine all the callers associated with a particular receiving app. The user firewall can then analyze the callers to determine if the permissions of the receiver greatly exceed those of the instigator.

6 Evaluation

In addition to evaluating IEM in terms of what useful user firewalls it allows developers to create, IEM is also formally evaluated based on three additional criteria. First, how much of the framework has to be modified in order to implement IEM? Second, how does IEM impact the stability of currently existing Android apps; both when allowing and blocking intents. And third, how does IEM impact the time it takes to route intents? Security is also important, but this has already been covered in the design section.

Object	**Modified**	**Added**	**Changed**
Intent Firewall	24	606	61.6%
Active Services	14	27	1.8%
Stack Supervisor	3	2	0.2%
Broadcast Queue	4	9	1.3%
AMS	0	39	0.2%
Intent	0	13	0.6%

Table 2: Lines of code modified separated by framework object.

6.1 Code Impact

Since IEM is designed in hopes of one day being integrated into the Android Open-source Project (AOSP) source code, one of the criteria for evaluation is the amount of code which has to be modified in order to implement IEM. Minimizing the lines of code within IEM attests to a design which is efficient and maintainable while minimizing the lines of code modified outside of IEM attests to the side effect implementing this design has on the rest of the framework. The code impact is summarized in Table 2. IEM's interceptor and interface comprise one Java object and is compared against the original intent firewall.

Since IEM requires additional parameters in order to remain stateless, as summarized in Table 1, modifying framework objects other than IEM is inevitable. However, Table 2 shows that the lines of code modified outside IEM are minimal. Most of these changes are to fetch objects which the new IEM public methods require and which are already present inside the calling object. The intent class did have to be extended to include an intent token field, but this change has no impact on already existing apps.

To test the maintainability of the code, I first implemented IEM in Android 5.0.2 and then migrated it to 5.1.1. I was able to do the migration in less than a day.

6.2 Application Stability

I tested IEM using the standard Google apps which come on Nexus devices as well as the top 50 free third-party apps from the Google Play Store. I explicitly included the Google apps in the evaluation because I found that they communicate with each other heavily using a wide variety of intent types. The Google Play Service, for example, communicates with every other Google app for authentication so the user doesn't have to log into each app individually. The Google apps also leverage all the intent types with a higher frequency than other third party apps. This is once again because the Google apps are heavily interconnected while most other popular apps tend to be more standalone.

During the testing of IEM, no cases where found where blocking an intent would cause an app to crash. When blocking access to the Google Play Services, I did find apps which would refuse to start, prompting the user that Play Services needs to be installed. Blocking an app's own services can cause timeouts during operations—like fetching new emails—but once again I did not encounter any crashes or freezes. I also noticed that if a service is blocked and then later allowed, it could take up to a few minutes for the app to re-establish the service connection. Manually restarting the app expedited the reconnection time. Blocking broadcasts can disrupt some flows. For example, if broadcasts within the Play Store app are blocked, an app update will download, but not install.

Overall, I found apps to be stable, even when the user firewall decides to block most intents. I recommend that developers creating user firewalls for typical consumers place emphasis on informing the user when intents are being blocked, otherwise the lack of feedback could cause confusion.

	No User Firewall	**Allow All Intents**	**Call Chain**
Activity	346.9	348.2	352.1
Service	14.0	14.9	16.8
Broadcast	7.6	11.8	12.2

Table 3: Milliseconds to route intents from sender to receiver, averaged over 5000 trials.

6.3 Performance

To test the intent routing performance of IEM, I created two simple apps to send and receive activity, service, and broadcast intents. The sender app places the current system time inside the intent and then the receiving app retrieves this value and calculates the difference.

Table 3 shows the milliseconds needed to send the intent across apps. Each value shown is the average of 5000 trials. The test device was a Nexus 5 running the modified version of Android 5.0.2. The tested user firewalls implement no caching between intent inspections and inspect every intent using the same procedure.

For the first set of trials, I configured IEM to not use a user firewall. This serves as a baseline as the logic in this configuration is identical to the original intent firewall. I then tested the intents using two user firewall policies. The allow all policy accepts any intent and serves to measure the overhead added by the round trip between the interface and the user firewall. The call chain policy inspects, stores, and logs every intent's sender and receiver and then recursively constructs chains. This is the most performance intensive example from the applications section.

The results in Table 3 show that the user firewall interface adds some overhead, but the difference in routing time remains well below what a human user can perceive. Granted, performance is mostly dependent on the efficiency of the particular user firewall app, but the experiment shows that IEM itself is acceptably efficient and user firewalls can achieve useful functionality with reasonable performance.

7 Conclusion

Android is the most popular operating system for embedded mobile devices. It is designed to encourage apps to leverage IPC with a greater frequency than seen in operating systems which target traditional computers. This, coupled with the nature of intent communication, makes studying the security of Android IPC a valuable endeavor. The current Android system includes a firewall which can perform access control on intent IPC. However, its poor usability means it has almost never been used in practice. This work proposes IEM which separates the interceptor of the firewall from its decision engine and places it inside a normal application called a user firewall. By doing so, IEM makes it easy to develop and modify the firewall logic, allowing for easy implementation of interesting new access control.

8 References

1: Lu, Long and Li, Zhichun and Wu, Zhenyu and Lee, Wenke and Jiang, Guofei, CHEX: Statically Vetting Android Apps for Component Hijacking Vulnerabilities, 2012

2: Chin, Erika and Felt, Adrienne Porter and Greenwood, Kate and Wagner, David, Analyzing Inter-application Communication in Android, 2011

3: Chin, Erika and Felt, Adrienne Porter and Greenwood, Kate and Wagner, David, Analyzing Inter-application Communication in Android, 2011

4: Davi, Lucas and Dmitrienko, Alexandra and Sadeghi, Ahmad-Reza and Winandy, Marcel, Privilege Escalation Attacks on Android, 2011

5: Adrienne Porter Felt and Steven Hanna and Erika Chin and Helen J. Wang and Er Moshchuk, Permission re-delegation: Attacks and defenses, 2011

6: Elish, Karim O and Yao, Danfeng Daphne and Ryder, Barbara G, On the Need of Precise Inter-App ICC Classification for Detecting Android Malware Collusions, 2015

7: Sven Bugiel and Lucas Davi and Alexandra Dmitrienko and Thomas Fischer andAhmad-Reza Sadeghi, XManDroid: A New Android Evolution to Mitigate Privilege Escalation Attacks, 2011

8: Ongtang, M. and McLaughlin, S. and Enck, W. and McDaniel, P., Semantically Rich Application-Centric Security in Android, 2009

9: Conti, Mauro and Nguyen, VuThienNga and Crispo, Bruno, CRePE: Context-Related Policy Enforcement for Android, 2011

10: Kantola, David and Chin, Erika and He, Warren and Wagner, David, Reducing Attack

Surfaces for Intra-Application Communication in Android, 2012

11: Hay, Roee and Tripp, Omer and Pistoia, Marco, Dynamic Detection of Inter-application Communication Vulnerabilities in Android, 2015

12: Maji, Amiya K. and Arshad, Fahad A. and Bagchi, Saurabh and Rellermeyer, Jan S., An Empirical Study of the Robustness of Inter-component Communication in Android, 2012

13: Sven Bugiel and Stephen Heuser and Ahmad-Reza Sadeghi, Flexible and Fine-grained Mandatory Access Control on Android for Diverse Security and Privacy Policies, 2013

14: Nadkarni, Adwait and Enck, William, Preventing accidental data disclosure in modern operating systems, 2013

15: Qihui Zhou and Dan Wang and Yan Zhang and Bo Qin and Aimin Yu and Baohua Zhao, ChainDroid: Safe and Flexible Access to Protected Android Resources Based on Call Chain, 2013

16: Backes, Michael and Bugiel, Sven and Gerling, Sebastian, Scippa: System-centric IPC Provenance on Android, 2014

17: Adrian Ludwig, Android Security State of the Union, 2015

18: Smalley, Stephen and Craig, Robert, Security Enhanced (SE) Android: Bringing Flexible MAC to Android., 2013

19: Michael Backes and Sven Bugiel and Christian Hammer and Oliver Schranz and Philipp von Styp-Rekowsky, Boxify: Full-fledged App Sandboxing for Stock Android, 2015

20: Stephan Heuser and Adwait Nadkarni and William Enck and Ahmad-Reza Sadeghi, ASM: A

Programmable Interface for Extending Android Security, 2014

21: Andrus, Jeremy and Dall, Christoffer and Hof, Alexander Van't and Laadan, Oren and Nieh, Jason, Cells: A Virtual Mobile Smartphone Architecture, 2011

22: Wu, Chiachih and Zhou, Yajin and Patel, Kunal and Liang, Zhenkai and Jiang, Xuxian, Airbag: Boosting smartphone resistance to malware infection, 2014

23: Bugiel, Sven and Davi, Lucas and Dmitrienko, Alexandra and Heuser, Stephan and Sadeghi, Ahmad-Reza and Shastry, Bhargava, Practical and Lightweight Domain Isolation on Android, 2011

24: D. J. Bernstein, SYN cookies,

25: Android Open Source Project, Android Open Source Project - Issue Tracker - Issue 177223: Intent/Bundle security issue,

26: OWASP, Web Application Firewall,

27: Enck, William and Ongtang, Machigar and Mcdaniel, Patrick, Mitigating Android software misuse before it happens, 2008

28: Lineberry, Anthony and Richardson, David Luke and Wyatt, Tim, These aren’t the permissions you’re looking for, 2010

29: Schlegel, Roman and Zhang, Kehuan and Zhou, Xiao-yong and Intwala, Mehool and Kapadia, Apu and Wang, XiaoFeng, Soundcomber: A Stealthy and Context-Aware Sound Trojan for Smartphones., 2011

30: Bugiel, Sven and Davi, Lucas and Dmitrienko, Alexandra and Fischer, Thomas and Sadeghi, Ahmad-Reza and Shastry, Bhargava, Towards Taming Privilege-Escalation Attacks on Android., 2012

9 Vita

Education

Syracuse University, L. C. Smith College of Engineering and Computer Science (Syracuse, NY)
B.S. Computer Science

Experience

Cybersecurity Research (Syracuse, NY)
Undergraduate Cybersecurity Researcher at Syracuse University

- Researched intent filtering between activities in the Android framework
- Created teaching materials on the Shellshock vulnerability found in bash: https://bitbucket.org/carter-yagemann/shellshock
- Published website documenting the Android Intent Firewall including its capabilities, limitations, and possible areas for improvement: http://www.cis.syr.edu/~wedu/android/IntentFirewall/
- Implemented Android framework hook to allow for user level IPC access control between applications and system: https://bitbucket.org/intentio-ex-machina/intentio-ex-machina/

JPMorgan Chase & Co. (Syracuse, NY)
Analyst, Information Technology Security Risk Management

- Performed manual software vulnerability analysis and security penetration testing
- Coded and implemented scripts to increase efficiency of metrics reporting
- Analyzed and reverse engineered malware
- Built programs in Python, C, C# and Java to detect malware, extract critical information from infected system memory and decrypt malicious network traffic

Frontier Communications (Stanford, CT)
Web Developer

- Worked on development team for TumTiki (formerly MyFiTV)
- Coded, tested and implemented pages using the Ruby on Rails framework