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

Carter Yagemann and Wenliang Du

Syracuse University, Syracuse NY 13210, USA,
{cmyagema,wedu}@syr.edu

Abstract. Android's intent framework serves as the primary method for interprocess communication (IPC) among apps. The increased volume of intent IPC present in Android devices, coupled with intent's ability to implicitly find valid receivers for IPC, bring about new security challenges. We propose *Intentio Ex Machina* (IEM), an access control solution for Android intent security. IEM separates the logic for performing access control from the point of interception 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 modify them to a controlled extent. Since it runs as a user application, developers are able to create user firewalls that manufacturers can then integrate into their devices. In this way, IEM allows for a new genre of security application for Android systems offering a creative and interactive approach to active IPC defense.

1 Introduction

One of the constraints which has shaped the design of Android is the limited hardware resources of embedded devices. Due to small memory size, Android's creators wanted to design an architecture which encourages apps to leverage the functionalities and capabilities of apps already present on the device. This allows Android devices to conserve memory by avoiding overlapping code. From a security perspective, this increased utilization of frequently implicit interprocess communication (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, it can invoke any other app which has registered components. This is concerning for the receiving app because by registering components to handle external requests, the app is opening itself to every other app on the device. This includes apps that could be malicious or come from untrusted sources. If the receiving app's exposed components contain vulnerabilities, the attack surface for exploitation immediately becomes systemwide. Attacks related to this problem have already been observed in the wild and are categorized as component hijacking [21].

Second, since any app can register components, senders do not have full control over where their data will end up. If a malicious app registers itself to handle a wide variety of requests, apps using intents could find their data being exfiltrated into the hands of devious actors. This kind of attack has been categorized in other works as intent spoofing [10, 1].

These problems have motivated both researchers and developers to seek solutions. Our proposal for intent IPC access control is an architecture which leverages the system-centric and standardized nature of intent IPC. *Intentio Ex Machina*[1] (IEM) is motivated by the insight that while every firewall intercepts packets and takes actions based on a decision engine, these two pieces do not have to reside near each other. More specifically, IEM replaces the intent firewall's engine with an interface that can bind to a user app. This "user firewall" can then act as the system's intent firewall. By placing the decision engine in an app, rather than in the framework like the current intent firewall, developers can easily design engines to utilize all the capabilities of a user app including pushed updates that do not require rebooting or flashing, rich graphically enhanced user interaction, and simplified access to system resources such as GPS and networking.

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

2 Background

2.1 Intent

Intents are a framework level abstraction of Linux binder IPC. They provide a simplified and standardized object for communicating with other apps and system services.

The Android system server maintains a binder of handles to create a mapping between sources and destinations. Whenever a system or user app is created, it is assigned a binder handle, which allows the app to send messages to the system server. Upon startup, every app uses its binder handle to the system server to register itself as an active app on the Android device. This registration establishes a network of paths for intents to take, all centered around the system server. Apps can also register intent filters to define the kinds of external intents that their components are willing to receive.

Upon receiving an intent, the system server has to resolve which component should be the receiver. If a target is explicitly specified in the intent, the system server checks it against the receiver's intent filter and confirms that they match before delivering it. Otherwise, the system server will search its list of intent filters to find eligible receivers.

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

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

Once the receiver[3] has been resolved, the intent is then passed to the intent firewall. The firewall compares it against its policies to decide whether it should be blocked or not.

Finally, the receiver processes the intent and if a result is needed, it gives this to the system server to forward to the original sender. The overall transaction occurs asynchronously since it involves communicating across several processes.

2.2 Activity Manager Service

The system server in Android is a privileged process running in user space. Apps communicate with it via binder messages and intents. The system server itself can be further divided into a collection of services, each of which is designed to manage a particular functionality of the system.

Activity Manager Service (AMS) is responsible for handling intents. It has a collection of public methods that apps can invoke; it communicates with all the other services to make sure that intents are resolved, permissions are checked, and receivers are running and ready to receive.

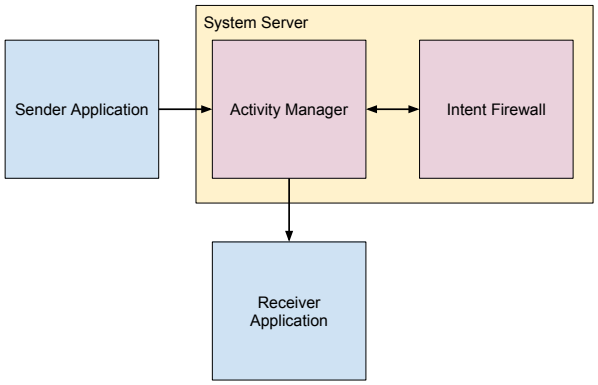

Fig. 1. Android Intent Firewall.

2.3 Intent Firewall

The intent firewall is an access control mechanism originally introduced in Android 4.3 and is present in all production devices. SEAndroid[4] configuration files support intent firewall policies, but few devices utilize it [29]. The main purpose of the intent firewall is to stall major malware outbreaks by allowing device manufacturers to push policies that will explicitly block the malware's components.

[3] There can also be multiple receivers in the case of broadcast intents.

[4] Security Enhancements for Android. Facilities device hardening via a bundle of policies.

Since such outbreaks are extremely rare, the intent firewall has almost never been used [22]. Figure 1 shows how the intent firewall fits into the framework.

AMS checks intents against the intent firewall near the end of the process. As a consequence, even though intents can be created with implicit or explicit destinations, they are all resolved to a receiver by the time they enter the firewall. Also, all three types of intents[5] are checked by the intent firewall before reaching the receiver. In other words, every intent type goes through the firewall before delivery. No intents can bypass it.

3 Design

In this section, we formulate the main architectural goal of IEM and assess the challenges in trying to achieve it. Figure 2 contains a high-level overview of the design, which will be explained in the following subsections.

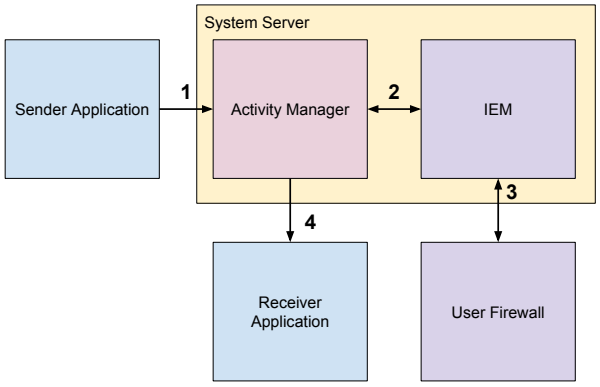

Fig. 2. Intentio Ex Machina.

3.1 Architectural Goal

All firewalls contain three critical pieces: the interceptor, the decision engine, and the policy. The interceptor captures the data packets and delivers them to the decision engine, which then decides if the packets 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 packets. On the other hand, the decision engine can be placed anywhere

[5] AMS differentiates via distinct API intents targeting activities, services, and broadcast receivers.

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 policy can only refer to attributes which the engine recognizes.

With these two observations in mind, reconsider the intent firewall in Figure 1. In this case, the interceptor and the decision engine both reside in the framework. This is the only possible placement for the interceptor, but what about the engine? Since it too resides in the framework, modifications require an OS patch, which requires the device to shutdown for several minutes. What would happen if the engine was placed inside a user app instead? 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 it can easily be changed.

With this in mind, the goal of IEM is to provide a hook in the intent flow to allow for an app to serve in place of the intent firewall. As previously stated, doing this makes changing the firewall's enforcement logic easier, which in turn allows for more flexibility. Making the firewall an app empowers the device manufacturer to deploy the solution that fits the needs of the customer. One manufacturer might deploy a user firewall which monitors location and allows the user to restrict which apps run while in the office. Another user firewall might allow a user to prevent apps from getting her location while she's driving. A parent might use 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.

Figure 2 shows IEM and how it interfaces with the rest of the framework and user apps. From here on, "IEM" specifically refers to the hook which resides in the framework while "user firewall" refers to any app which utilizes IEM. Intents first enter the system server through the public API at edge 1. This is where AMS resolves the receiver. The intent then enters IEM via edge 2. The intent is delivered to the user firewall and a decision is returned via edge 3. If the user firewall decides to allow the intent, the activity manager is alerted via edge 2 and the intent is delivered via edge 4.

In the next subsection, we identify and address the key challenges in trying to achieve this architecture.

3.2 Design Challenges

Figure 3 shows the three new pieces introduced into the framework by IEM. This subsection considers each piece in turn and addresses the challenges caused by their addition.

Intent Interceptor The first new piece an intent reaches is IEM's interceptor. The challenge here is deciding which intents are appropriate to intercept. This problem is nontrivial because we cannot assume that the user firewall will always be responsive since it is a user app. Like any other app, it can crash or freeze. This is different from the original intent firewall, which can simply intercept all intents because it is self-contained.

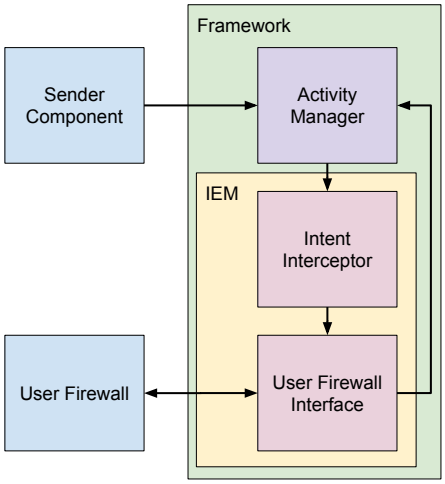

Fig. 3. The flow of intents through IEM.

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 components. The device will end up in an unrecoverable state. For this reason, intents created by the system server are exempt from being intercepted. All other intents can be safely intercepted because failing to deliver them will not impact stability.

User Firewall Interface After the interceptor, the intent next reaches the user firewall interface. In order for this interface to work, it has to be able to send intents to an app (the user firewall) and get responses containing decisions.

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 that already exists in Android: the extensible framework. This design pattern works by having a system service bind to an app service chosen by the user in the device settings. Once bound, the two services can directly exchange messages though binder IPC 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 and consequently satisfies the same security guarantees. Once the device manufacturer explicitly enables a user firewall, IEM will bind to that app's user firewall service at startup. Once bound, the intercepted intents will be sent to the user firewall via the binding. The messages include a "reply to" handle, which the user firewall can use to return a response. Note that this is a direct bidirectional binder IPC channel. The messages traversing the channel are *not* intents.

There is another challenge regarding the design of the user firewall interface. In the original architecture, intents are resolved entirely inside the framework

using additional stateful information being held in AMS. Consequently, while the intent flow as a whole is asynchronous, it is necessary that the portion that occurs inside AMS never block so resource exhaustion cannot occur. Since IEM inserts a binder IPC exchange into the middle of this resolution logic, avoiding blocks can no longer be taken for granted.

As mentioned earlier, the user firewall app could crash or freeze. If this happens, the app will cease to respond to the interface. This will cause resource exhaustion inside AMS. Even if a timeout mechanism is implemented to prevent buildup, the resulting design will 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, it no longer becomes possible for the user firewall to involve the user in critical decision making processes. This reduces the flexibility of the firewall logic, which weakens the goal our architecture strives to achieve. Alternatively if the timeout is long, the device will become unresponsive, which is unacceptable. There is no middle ground. Either choice is detrimental to the goal of designing a platform that is flexible. For this reason, there cannot be any timeouts. The interface must be completely stateless so no blocking occurs. This distinguishes IEM from many other hook designs.

Since IEM has to maintain statelessness and never be waiting on the user firewall, the messages it sends have to contain all the information necessary to reconstruct the state. To address this challenge, we introduce the concept of intent wrapper. The wrapper is a bundle containing the original intent along with everything necessary to duplicate its state. This includes sender information, which is something the original intent firewall does not consider. By wrapping these additional variables together along with the intent, not only does the user firewall now receive a complete picture of sender, receiver, and interaction being performed, but the interface no longer has to remember the intent's state because that information will be returned by the user firewall in the response. However, this state information has to be protected from being modified by the user firewall; a challenge which will be addressed next.

User Firewall The intent wrapper now reaches the user firewall. What should it 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 give 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 breaks the principal of least privilege. We introduce the intent token to address this concern.

Inspiration for the intent token comes from SYN cookies, which prevents SYN flooding by making TCP handshakes stateless [5]. Similarly, the intent token allows IEM to remain stateless while still restricting the user firewall's power. Tokens are generated inside IEM by hashing a secret created at startup with the parts of the state that should not be modified. Randomized salts are used to ensure that tokens are unique. As long as the secret is kept confidential, the user firewall will be unable to create valid tokens and consequently cannot arbitrarily modify an intent's state. Intents are also numbered sequentially and checked

against a sliding window inside IEM to prevent replay. While the window could be considered stateful, its fixed size makes it not prone to resource exhaustion and therefore acceptable for the IEM design. In this way, the user firewall is a privileged app, but only to the extent essential for performing its role as an intent firewall.

More challenging, however, is deciding which parts of the state should be included in the token to prevent modification. This answer can be determined by grouping the contents of the intent wrapper into the three mutually exclusive categories: 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. On the other hand, the sender already expects that the system will resolve the receiver. For this reason, 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.

While this subsection has provided an overview of the security mechanisms inside IEM, the following subsection explains in detail how these mechanisms mitigate threats at each surface.

3.3 Threat Model

Figure 3 shows that the boundaries between IEM, framework, and app 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 AMS since it is a part 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 component.

The first boundary crossing is from the sending app to AMS. The intents enter AMS through an API that is unmodified by IEM. Therefore, IEM assumes that this boundary is secure.

The next boundary is between IEM and AMS. Since the threat model already assumes that AMS 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 interface and the user firewall in user space. This surface can be attacked by a malicious app, but the previously mentioned security mechanisms prevent the attacker from having any success.

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 that initiates the binding and since it gets its target from the device settings, it will bind with the correct service. Since the interface is a part of the system, the user firewall can differentiate the attacker from the interface by checking the UID of the bind request. This action can also be protected using the permissions mechanism already present in Android. To attack the second action, the adversary will have to sniff and spoof binder messages. Since the binder is part of the Linux kernel, this is only doable by either gaining root privileges or by compromising the kernel. In either case, the integrity of the entire system is compromised, which is beyond the scope of IEM. Finally, an attacker targeting the third action would have to spoof a user firewall response, but as stated earlier, this is prevented by the intent token.

This completes the threat model for IEM. As a side note, we would like to mention that there are also multiple challenges pertaining to the implementation of IEM in Android, but due to length constraints we cannot go into the necessary detail to explain them in this publication.

Listing 1.1. UFW service handler template.

```
@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
}
```

4 Applications

The generic nature of IEM allows developers to create many different kinds of user firewalls to serve a variety of purposes. All the user firewalls presented in this section could be implemented by modifying the original intent firewall, but doing so would be vastly impractical. Developers would need to have Android framework expertise 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 directly into the Android framework.

The developer of a user firewall only needs to implement a service component. Listing 1.1 is a template for the handler. 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, we describe a few examples of user firewalls made possible by IEM. We have chosen just a small sample from the infinite number of possible user firewalls, which we believe are sufficient to demonstrate the flexibility of the IEM architecture.

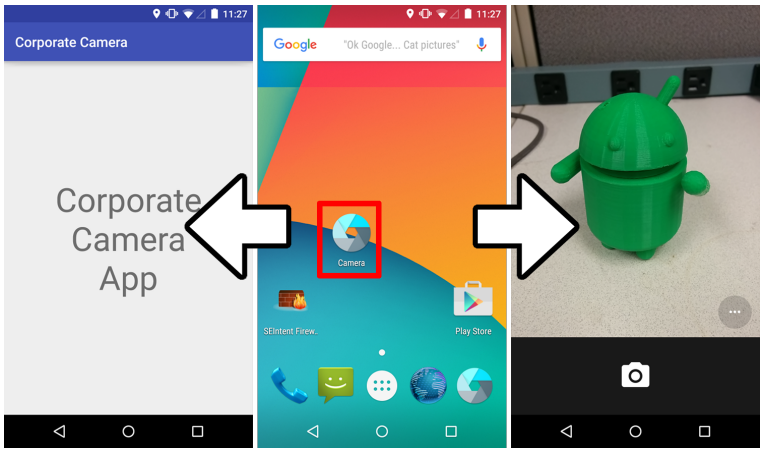

Fig. 4. Redirecting intents.

4.1 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 that 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 that could be captured when the photo is taken. 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 do not want to have to use the corporate camera app when they are not at work. A user firewall can control which camera app is launched based on GPS location using intent redirection.

Figure 4 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 are not 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 appear instead.

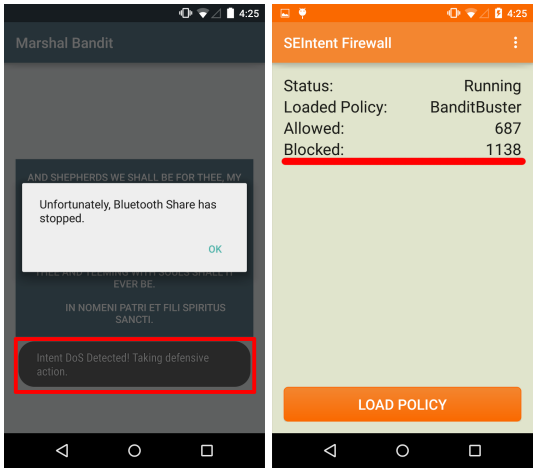

Fig. 5. Intent DoS detection.

4.2 Preventing Intent Denial of Service

When we implemented the intent token, we intentionally created a new field for it inside the intent 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. We 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 that Google has acknowledged [27].

To demonstrate the potential damage of this vulnerability, we created a malicious app called Marshal Bandit. Upon boot, Marshal Bandit queries AMS 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 the system's worker threads. The result is a denial of service, which causes the device to become unresponsive and eventually reboot. Since the user cannot access the Settings app while the attack is underway, the device is crippled. Even if the user knows how to boot the device into the recovery mode, she will not 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 Figure 5, we demonstrate a user firewall that can detect the sudden flood of intents coming from our malicious app. Upon detection, the user firewall will inform the user which app is performing the attack while stopping 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 the "stop background processes" permission and IEM. The successful thwarting of this recently discovered denial of service attack demonstrates the flexibility and capability of IEM.

Implementation logic of this complexity at the framework level would be challenging compared to the narrow scope of attacks it addresses. With IEM, a user firewall that addresses this vulnerability can be developed by a single developer in one work day.

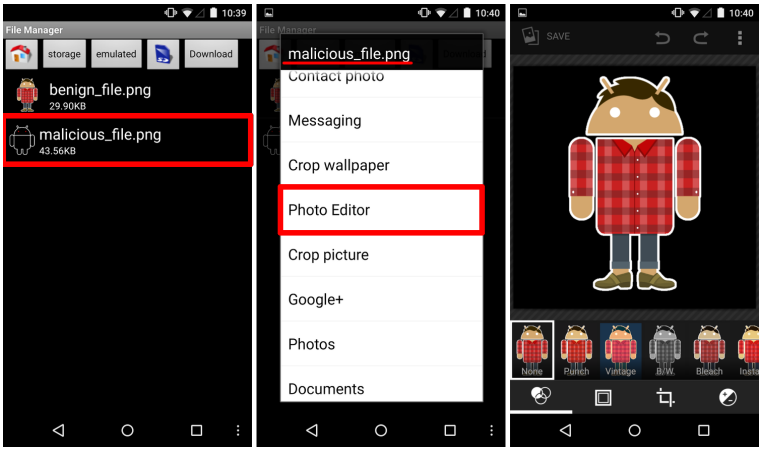

Fig. 6. Data sanitization.

4.3 Sanitizing Intent Data

Since user firewalls can modify the contents of the intent itself, they 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 that 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. Figure 6 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 not a complete solution to controlling data flow, but IEM can allow user firewalls to address some flow concerns while being easy to develop, deploy, and maintain.

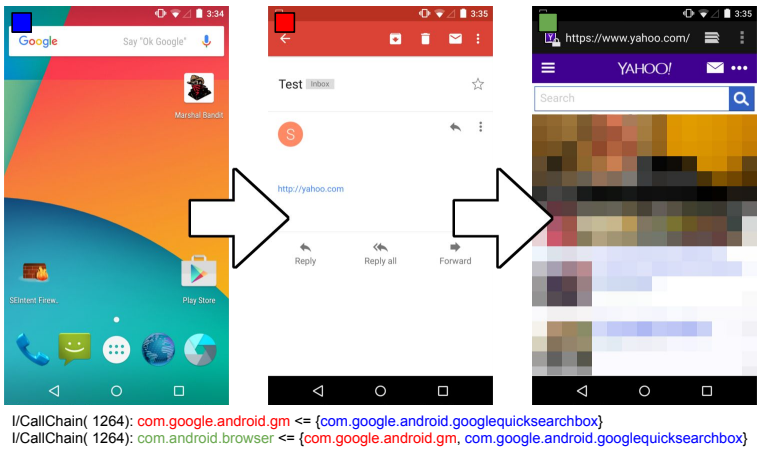

Fig. 7. Caller chain.

4.4 Determining Caller Chains

One potential shortcoming with the Android permissions architecture is 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 [14, 12, 19, 28, 6, 7]. Multiple works have identified this problem and implemented caller chains to resolve it [3]. 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. Figure 7 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 sender.

5 Evaluation

In addition to evaluating IEM in terms of what useful user firewalls it allows developers to create, we also formally evaluate it based on two additional criteria. First, how does IEM impact the stability of currently existing Android apps; both when allowing and blocking intents. Second, how does IEM impact the time it takes to route intents?

5.1 Application Stability

We 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. We explicitly included the Google apps in our evaluation because we found that they communicate with each other heavily using a wide variety of intent types.

During our tests of IEM, we did not find any cases where blocking an intent would cause an app to crash or become unresponsive. When blocking access to the Google Play Services, we did find apps that would refuse to start, prompting the user that Play Services needs to be installed. In either case, the apps behaved well even while the user firewall is blocking intents.

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

Table 1. Milliseconds to route intents, averaged over 5000 trials.

5.2 Performance

To test the intent routing performance of IEM, we created two simple apps to send and receive activity, service, and broadcast intents.

Table 1 shows the milliseconds needed to send the intent across apps. Each value shown is the average of 5000 trials. Our test device was a Nexus 5 running our modified version of Android 5.0.2 with IEM.

For our first set of trials, we configured IEM to not use a user firewall. This serves as our baseline as the logic in this configuration is identical to the original intent firewall. We 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 our applications section.

Our results in Table 1 show that the user firewall interface adds some overhead, but the difference in routing time remains well below what a human user can perceive.

6 Related works

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

6.1 Access control

In the access control category, we first find the sender permissions mechanism currently implemented in Android. This security feature allows receivers to require of the 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 limitations [12]. 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 [15, 13]. 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 possible for a malicious app to have as many, if not more, permissions than the receiving app it is trying to exploit. These problems have been the motivation for works such as XmanDroid [6], Saint [25], CRePE [11], and others [18, 16, 23, 9, 24]. Even if the developer of the receiving app wants to explicitly check who the sender of the intent is, his app can only see the last app to send the intent. ChainDroid [31] and Scippa [3] 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 [22].

Other works, such as Boxify [4], 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.

Our work is conceptually similar to Android Security Modules [17], but differentiates itself in two key aspects. First, while ASM only facilitates the monitoring of resources, our work enables modification for the purposes of redirection and data sanitization. Second, ASM uses callback timeouts; a limitation our work avoids by being stateless.

6.2 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 [2], achieves this isolation by creating virtual devices that run on top of the host device. Another solution, Airbag [30], 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 [8].

We chose an access control design for IEM because we want to leverage the unique nature of intent IPC. Specifically, we want to leverage the fact that all intents must travel through the framework 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.

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 unique properties of intent IPC, makes the study a worthy endeavor. The current Android system includes a firewall which can perform access control on intent IPC. However, it is very limited and its poor usability means it has almost never been utilized in practice. We propose IEM to separate the interceptor of the firewall from its decision engine using a novel stateless interface. This allows a normal application, called a user firewall, to become the engine for intent access control. By doing so, IEM makes it easy to develop and modify the firewall's logic, allowing for easy implementation of interesting new access control.

8 Acknowledgments

This project was supported in part by the NSF grant 1318814.

References

1. Yousra Aafer, Nan Zhang, Zhongwen Zhang, Xiao Zhang, Kai Chen, XiaoFeng Wang, Xiaoyong Zhou, Wenliang Du, and Michael Grace. Hare hunting in the wild android: A study on the threat of hanging attribute references. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 1248–1259, New York, NY, USA, 2015. ACM.
2. Jeremy Andrus, Christoffer Dall, Alexander Van't Hof, Oren Laadan, and Jason Nieh. Cells: A virtual mobile smartphone architecture. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 173–187, New York, NY, USA, 2011. ACM.
3. Michael Backes, Sven Bugiel, and Sebastian Gerling. Scippa: System-centric ipc provenance on android. In *Proceedings of the 30th Annual Computer Security Applications Conference*, ACSAC '14, pages 36–45, New York, NY, USA, 2014. ACM.
4. Michael Backes, Sven Bugiel, Christian Hammer, Oliver Schranz, and Philipp von Styp-Rekowsky. Boxify: Full-fledged app sandboxing for stock android. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 691–706, Washington, D.C., August 2015. USENIX Association.

5. D. J. Bernstein. Syn cookies. http://cr.yp.to/syncookies.html. Accessed: 2015-11-20.
6. Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, and Ahmad-Reza Sadeghi. Xmandroid: A new android evolution to mitigate privilege escalation attacks. Technical Report TR-2011-04, Technische Universität Darmstadt, April 2011.
7. Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, Ahmad-Reza Sadeghi, and Bhargava Shastry. Towards taming privilege-escalation attacks on android. In *NDSS*, 2012.
8. Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Stephan Heuser, Ahmad-Reza Sadeghi, and Bhargava Shastry. Practical and lightweight domain isolation on android. In *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '11, pages 51–62, New York, NY, USA, 2011. ACM.
9. Sven Bugiel, Stephen Heuser, and Ahmad-Reza Sadeghi. Flexible and fine-grained mandatory access control on android for diverse security and privacy policies. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, pages 131–146, Washington, D.C., 2013. USENIX.
10. Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. Analyzing inter-application communication in android. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, MobiSys '11, pages 239–252, New York, NY, USA, 2011. ACM.
11. Mauro Conti, VuThienNga Nguyen, and Bruno Crispo. Crepe: Context-related policy enforcement for android. In Mike Burmester, Gene Tsudik, Spyros Magliveras, and Ivana IliÄĞ, editors, *Information Security*, volume 6531 of *Lecture Notes in Computer Science*, pages 331–345. Springer Berlin Heidelberg, 2011.
12. Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Marcel Winandy. Privilege escalation attacks on android. In Mike Burmester, Gene Tsudik, Spyros Magliveras, and Ivana IliÄĞ, editors, *Information Security*, volume 6531 of *Lecture Notes in Computer Science*, pages 346–360. Springer Berlin Heidelberg, 2011.
13. Karim O Elish, Danfeng Daphne Yao, and Barbara G Ryder. On the need of precise inter-app icc classification for detecting android malware collusions. *Proceedings of IEEE Mobile Security Technologies (MoST), in conjunction with the IEEE Symposium on Security and Privacy*, 2015.
14. William Enck, Machigar Ongtang, and Patrick Mcdaniel. Mitigating android software misuse before it happens. 2008.
15. Adrienne Porter Felt, Steven Hanna, Erika Chin, Helen J. Wang, and Er Moshchuk. Permission re-delegation: Attacks and defenses. In *In 20th Usenix Security Symposium*, 2011.
16. Roee Hay, Omer Tripp, and Marco Pistoia. Dynamic detection of inter-application communication vulnerabilities in android. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 118–128, New York, NY, USA, 2015. ACM.
17. Stephan Heuser, Adwait Nadkarni, William Enck, and Ahmad-Reza Sadeghi. Asm: A programmable interface for extending android security. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 1005–1019, San Diego, CA, August 2014. USENIX Association.
18. David Kantola, Erika Chin, Warren He, and David Wagner. Reducing attack surfaces for intra-application communication in android. Technical Report UCB/EECS-2012-182, EECS Department, University of California, Berkeley, Jul 2012.

19. Anthony Lineberry, David Luke Richardson, and Tim Wyatt. These arenâĂŹt the permissions youâĂŹre looking for. *DefCon*, 18:2010, 2010.
20. Linux-PAM. A linux-pam page. http://www.linux-pam.org/. Accessed: 2015-12-02.
21. Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. Chex: Statically vetting android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 229–240, New York, NY, USA, 2012. ACM.
22. Adrian Ludwig. Android security state of the union. Black Hat USA 2015, 2015.
23. Amiya K. Maji, Fahad A. Arshad, Saurabh Bagchi, and Jan S. Rellermeyer. An empirical study of the robustness of inter-component communication in android. In *Proceedings of the 2012 42Nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, DSN '12, pages 1–12, Washington, DC, USA, 2012. IEEE Computer Society.
24. Adwait Nadkarni and William Enck. Preventing accidental data disclosure in modern operating systems. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, CCS '13, pages 1029–1042, New York, NY, USA, 2013. ACM.
25. M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel. Semantically rich application-centric security in android. In *Computer Security Applications Conference, 2009. ACSAC '09. Annual*, pages 340–349, Dec 2009.
26. OWASP. Web application firewall. http://tinyurl.com/3cakwty. Accessed: 2015-12-04.
27. Android Open Source Project. Android open source project - issue tracker - issue 177223: Intent/bundle security issue. https://code.google.com/p/android/issues/detail?id=177223. Accessed: 2015-11-20.
28. Roman Schlegel, Kehuan Zhang, Xiao-yong Zhou, Mehool Intwala, Apu Kapadia, and XiaoFeng Wang. Soundcomber: A stealthy and context-aware sound trojan for smartphones. In *NDSS*, volume 11, pages 17–33, 2011.
29. Stephen Smalley and Robert Craig. Security enhanced (se) android: Bringing flexible mac to android. In *NDSS*, volume 310, pages 20–38, 2013.
30. Chiachih Wu, Yajin Zhou, Kunal Patel, Zhenkai Liang, and Xuxian Jiang. Airbag: Boosting smartphone resistance to malware infection. *Proceedings of the Network and Distributed System Security Symposium*, 2014.
31. Qihui Zhou, Dan Wang, Yan Zhang, Bo Qin, Aimin Yu, and Baohua Zhao. Chaindroid: Safe and flexible access to protected android resources based on call chain. In *Trust, Security and Privacy in Computing and Communications (TrustCom), 2013 12th IEEE International Conference on*, pages 156–162, July 2013.