The 8th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications
24-26 September 2015, Warsaw, Poland

Exploring a High-Capacity Covert Channel on the Android Operating System

Timothy Heard[1] , Daryl Johnson[2] , Bill Stackpole[3]
[1] Undergraduate, Rochester Institute of Technology, Rochester, NY 14623, Email: tjh2430@rit.edu
[2] Associate Professor, Rochester Institute of Technology, Rochester, NY 14623, Email: daryl.johnson@rit.edu
[3] Associate Professor, Rochester Institute of Technology, Rochester, NY 14623, Email: bill.stackpole@rit.edu

***Abstract* — As the use and prevalence of mobile technology increases so too does the importance of effective security for these devices. In particular, sensitive user information must be protected, which includes protecting against any covert channels that would allow such information to be compromised. This paper will examine a new covert channel capable of circumventing existing application-level protections on the Android operating system (OS). The authors will also show that this channel is capable of achieving significantly higher throughput than similar, previously discovered channels, which necessitates the implementation of additional security protections and controls at the operating system level.**

Keywords — Android, covert channel, mobile security

I. Introduction

The prevalence of mobile computing technologies, particularly devices running the Android operating system (OS), has been growing at a rapid pace in recent years, with Android devices having the largest share of the smartphone market at the time of this writing. [1] This includes an increase in both the number of people who own these devices and their range of uses, with even some e-readers and digital video recorder (DVR) devices now running Android. As a result, the amount of sensitive data which is accessible through Android devices has also increased, making the management and protection of this information all the more critical and important.

In order to accomplish the task of protecting user information Android uses a permissions-based security model which requires applications to statically declare all of the permissions they use in their corresponding AndroidManifest.xml file. Users then have to approve this set of permissions when they install any application. However, as has been discussed in previous works [2], the protections offered by this security model can be overcome if applications with different permission sets are able to communicate and collaborate with each other. This danger is particularly relevant given the existence of a number of different covert channels for Android [2]–[4].

This paper will explore a flexible, high-capacity covert channel for inter-application communication on the Android OS based on Intents. Section 2 will provide an overview of the relevant aspects of the Android OS and introduce the concept for the proposed channel, with Section 3 detailing the implementation of this new covert channel. The channel will then be evaluated in terms of throughput, reliability, and detectability in Section 4. Section 5 will illustrate the potential impact of this channel and Section 6 will provide recommendations for corresponding mitigation followed by a conclusion in Section 7.

II. Concept

Although the Android operating system is designed to isolate different applications from each other, it also provides several mechanisms for legitimate interaction and communication between applications. One of the primary mechanisms for inter-application communication is the sending and receiving of Intents -a special data type defined by the Android OS. The primary purposes of Intents are to allow applications to request that some action be performed (such as viewing a picture) or to notify interested applications of some event (such as the ringer mode being changed). Intents also allow for the transmission of additional information, most notably through the inclusion of another OS-provided data type: a Bundle, which is simply a collection of key-value mappings from Java Strings to any of a limited number of supported data types [5], [6].

Two different kinds of application components exist for sending and receiving Intents, namely Activities (essentially representations of the different application interface screens) and Services, which are used to run tasks in the background, especially those which may take a significant amount of time to complete.[1] Intents can be sent directly to a specific receiver (this is known as an explicit Intent) or they can be broadcasted across the system (an implicit Intent). Explicit Intents are generally used to send data and requests between different components of a single application while implicit Intents are most commonly used for communication between different applications. In the case of a broadcasted (implicit) Intent, the Intent will be

[1] Any length of time for which it would not be acceptable for the user interface to be unresponsive represents a significant amount of time in this context.

978-1-4673-8361-5/15/$31.00 ©2015 IEEE

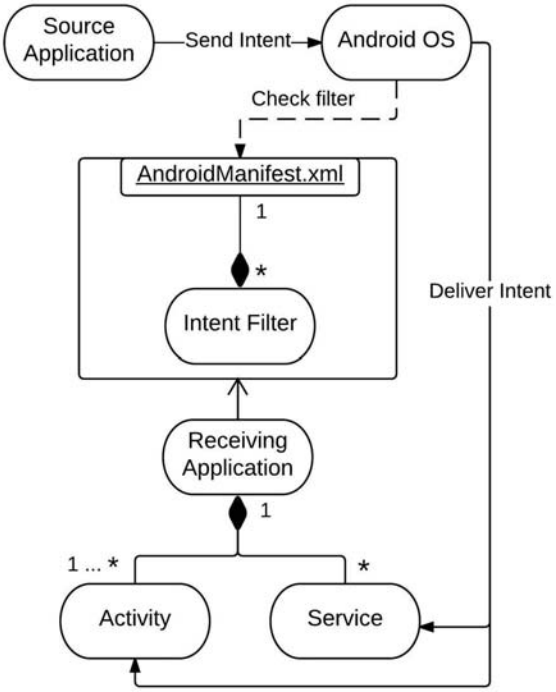

Figure 1. Sending an Intent Between Applications

delivered to an application which has a matching Intent filter [7]. If there are multiple applications with matching Intent filters, the user is asked to select one from the list of available options.

The ability to encapsulate data in an Intent using Bundles provides a flexible means of sending data between applications, as well as between components within the same application. The support for inter-application data transfer provided through Bundles is of particular interest as it also provides a flexible medium for a covert channel. This is due to the fact that Bundle keys can only map to a set number of different data types. By associating a code with each supported data type, legitimate key-value mappings can be used to encode covert bits based on what data type a given key is mapped to (the actual value stored is irrelevant in this context). Complete messages can then be encoded using a pre-defined ordering of all the keys stored in the Bundle. A receiving application which is aware of both the key ordering and the data type to code mappings can then decode such messages upon receiving the Bundle (transmitted by means of an Intent).

III. Implementation

This covert channel was implemented for version 8 of the Android API (Android 2.2 / Froyo), which allowed 22 different data types to be used for hiding numeric values from 0 to 21 for every key[2]. This API version was chosen since, at the time of this writing, it was the lowest (earliest) version of the API which has any appreciable market share according to [8] and therefore this implementation can be expected to work on nearly any Android device (given that the relevant parts of the Android OS are fully backwards compatible through all current versions). For simplicity, keys were generated in an alphabetical sequence of increasing length, with the key "a" mapping to the first value and the key "b" mapping to the second (once the single character keys were used up the keys "aa" and then "ab" were used and so on). Realistically, these keys would be meaningful strings based on the advertised purpose of the sending application; however, such keys are highly context-dependent and would be functionally equivalent to the meaningless strings used in our implementation. Once the entire message was encoded, the Bundle was placed inside an Intent which was designed to match the receiver application's Intent filter and the Intent was openly broadcasted. The receiver then extracted the Bundle from the received Intent and iterated over the set of Bundle keys in order. The type of value associated with each key was subsequently translated into a corresponding numeric code between 0 and 21 to reconstruct the original message. When implemented in such a way, this channel can easily be used to send numeric data, such as GPS coordinates.

Instead of being associated with numeric codes, each Bundle-supported data type could be mapped to a specific symbol, character, or even predetermined message to allow for the transmission of non-numeric data. Still, only being able to communicate in terms of 22 symbols may be of little use if the data being transmitted is textual (such as contact information). Of course, textual information, as well as any other form of data, could still be transmitted at the cost of the additional complexity of translating between the literal data and numeric values (e.g. using the channel to implement a bitstream). However, a simpler and more direct solution is possible due to the fact that one of the data types which a Bundle is able to contain is another Bundle, which allows numeric expansion codes to be encoded within a nested Bundle.

Using this strategy, the Bundle value type has a special meaning rather than an actual encoded value. The first key-value pairing[3] within the secondary (nested) Bundle is then interpreted using the same data-type to encoded value mapping being applied to the enclosing Bundle. This interpreted value is then used to indicate which alternate character set to use when interpreting the next key-value

[2]For version 8 of the Android API, 29 different data types are actually supported by the Bundle class, but only 22 were successfully used for this channel as some types, such as all of the specializations of the ArrayList type, were indistinguishable from each other when retrieved by the receiver.

[3]First according to the same key ordering, although a secondary ordering could be used. Additional key-value pairings could also be used, with the values retrieved from each pairing being accumulated or combined to allow for a greater number of expansion codes.

```
public static Bundle encode(int[] codes)  {
   Bundle bundle = new Bundle();
   for(int i = 0; i < codes.length; i++)  {
      switch(codes[i]) {
      case 0: bundle.putBoolean(key, true);
         break;
      case 1: bundle.putByte(key, (byte) 1);
         break;
      case 2: bundle.putInt(key, 1);
         break;
      case 3: bundle.putCharSequence(key, (CharSequence) "A");
         break;
      case 4: bundle.putDoubleArray(key, new double[0]);
         break;
      ...
      default:
         throw new CodeNotSupportedException(codes[i]);
      }
   }

   return bundle;
}
```

```
public static int[] retrieveEncodedValue(Bundle bundle) {
   List<String> keys = orderKeys(bundle.keySet());
   int[] codes = new int[keys.size()];
   int i = 0;

   for(String key: keys) {
      if(bundle.getBoolean(key))
         codes[i] = 0;
      else if(bundle.getByte(key) != 0)
         codes[i] = 1;
      else if(bundle.getInt(key) != 0)
         codes[i] = 2;
      else if(bundle.getCharSequence(key) != null)
         codes[i] = 3;
      else if(bundle.getDoubleArray(key) != null)
         codes[i] = 4;
      ...
      else
         throw new NoValueFoundException(key);

      i++;
   }

   return codes;
}
```

Figure 2. Encoding and Decoding Covert Data using a Bundle

pairing within the main Bundle.

In this way, the entire set of lowercase characters from "a" through "z" (as well as the space character to allow for more meaningful messages) was able to be successfully transmitted using this channel, with the set of these 27 characters being divided between two smaller character sets (one containing 21 characters[4] and the other containing the remaining 6) with an expansion code of 1 indicating that the next key-value pairing should be interpreted using the second character set. For our implementation, these mappings were encapsulated in concrete subclasses of an abstract dictionary class which was packaged in a library shared by both applications (the sender and the receiver), thereby allowing any other mapping to be used simply by switching to a new dictionary implementation.

It is important to note that when an Intent is delivered to an Activity, the corresponding application screen will be displayed to the user, which would establish a clear and obvious relationship between the sender and receiver applications. To avoid this, the message-bearing Intent can be broadcasted to a Service instead since when an Intent is delivered to a Service user focus is able to remain on the sender application, thereby making the transmission transparent to the user. There is an additional issue that if the sender performs data collection and transmission within an Activity, that application would only be able to perform these tasks while it has user focus [9], [10]. However, the sender application could also have a Service of its own for performing these activities, allowing collection, transmission, and receiving of data to happen continuously in the background without alerting the user.

IV. Evaluation

To evaluate this channel with regards to throughput, robustness (error rate), and detectability the channel implementation described in the preceding section was tested on a Motorola Photon Q running Android 4.0.4. A series of messages were sent between the sender and receiver applications. In our testing these were specifically textual messages, but, as explained above, these messages could contain any form of data. The messages varied with regards to the amount of data being sent, ranging from a single character up to 10,000 characters[5]. The number of Intents used to send these messages also varied, from a single Intent up to 500 Intents for a single message. For testing purposes, the receiving application sent an acknowledgement to the sending application for each Intent

[4]Only 21 characters codes are available for this implementation as the 22nd data type (Bundle) was reserved for the encoding of expansion codes.

[5]Given that the value mapping used in our implementation only directly supported 27 values, each character is only counted as being 4 bits, not 8; however, since one expansion code was used, our implementation was capable of supporting 42 different values without modification (aside from updating the mappings within the dictionary class), meaning that each character would actually equate to 5 bits of data (ignoring the remainder), making the effective throughput of the channel higher than indicated.

Table I. CHANNEL PERFORMANCE

Number of Intents Sent	Average Throughput*	Highest Recorded Throughput*	Error Rate
1	39,014.88	81,300.81	0.0
2	47,076.27	85,106.38	0.0
5	37,676.82	67,659.00	0.0
10	40,311.14	78,817.73	0.0
25	43,805.95	74,660.30	0.0
50	44,098.52	75,703.09	0.0
100	40,546.17	75,734.63	0.0
500	41,603.74	76,467.21	0.0

*Throughput is in terms of bits per second; encoding, decoding, and response times are not included in throughput calculation

it received. These responses included the text received (for error checking) and the time when the Intent was received. The difference between this time and the time each Intent was sent was accumulated and used to calculate the throughput for that transmission (without consideration for message encoding or decoding or response time from the receiver, which is not part of the actual channel). The highest achieved throughput using this calculation method was 85,106 bits per second using two Intents to send 1,000 characters (500 characters per Intent). The total transmission time, counting the encoding and decoding of all messages and response time from the receiver, was also recorded, with the highest throughput achieved when using this calculation method being 34,996 bits per second using two Intents to send 5,000 characters.

Both of these rates far exceed the bandwidth of other known, similar covert channels [2], [4], with the highest thoughput channel encountered during our research being just over 4,300 bits per second [2]. Additionally, the channel proved to be very reliable, with zero bit errors being encountered during our testing; this was unsurprising given that the application communication protocol based on the passing of Intent objects (i.e. Intent-based protocol) which was used as the carrier for this channel is a central and well tested part of the Android OS.

In addition to being well tested, the Intent-based inter-application communication protocol is also well exercised, with Intents being frequently passed through the system. As a result, this channel has the ability to blend in with legitimate Intent traffic (i.e. Intents which would be sent even if this channel were not in use on a given device). However, this requires that the communicating applications maintain an awareness of their context (what their advertised purpose is) as well as their environment (what other applications are currently running, what kinds of Intents they are sending, and how frequently these other Intents are being sent). For example, two applications with a legitimate reason to communicate using Intents containing a Bundle could select the keys used in a way that causes them to form a meaningful ordering. The types of the corresponding values could also be manipulated to communicate the desired symbols or codes by establishing a translation scheme between the different supported data types (or at least as subset of those data types). In this way, the covert information could be encoded into perfectly legitimate Intent traffic. This addresses the matter of context awareness. The other aspect of detectability for this channel, environmental awareness, requires only sending Intents (both purely legitimate Intents and those used to transmit covert messages) at a frequency and of a size which matches "normal" Intent traffic for their environment. Otherwise, analysis of Intent traffic on a given device could identify the communications between the involved applications as being suspicious. The definition of normal Intent traffic will depend on what other applications are running on the device as well as the advertised purpose of the applications using this channel.

It is important to note that this channel does not disguise the fact that there is some form of communication occurring between the involved applications due to the fact that an Intent broadcasted or sent from one application will be delivered to the other application involved in the channel by the operating system. However, the channel does allow the involved applications to effectively disguise the fact that any illegitimate communication is occurring.

V. IMPACT

As mentioned earlier in this paper, the protections offered by the Android permissions system can be violated if applications with different permission sets are able to effectively collaborate with each other. For example, if a user has installed two applications, one with permission to access the user's contact list and another which has internet access, these applications can create the behavior of an application which has been granted access to both the user's contact data and the internet even though the user never approved this combination of permissions for either application. To achieve this effect, the applications can pre-arrange a sender-receiver relationship[6] where the application with access to sensitive information (user contact data in this example) sends any or all of this information to the receiver application.

The two installed applications could also implement a master-slave relationship where one application (presumably the one with the ability to exfiltrate sensitive information) actively controls the collaboration by sending instructions to the slave application. These instructions could include requesting that any currently available information which matches some provided criteria be sent to it, or to activate a malicious functionality in the slave application which was previously dormant (to include the collection and transmission of sensitive user data). Ultimately, the exact instruction set which the master could use is highly flexible and would depend on the context and purpose of both applications. If the master application is also able to establish two-way communication with some

[6]Could also be described as a source-sink relationship. [2]

malicious actor[7] on a remote system (most likely using the internet), that actor would be able to control the behavior of both applications by using the master application as a proxy to the second application.

This relationship could be expanded even further to incorporate multiple slave applications, with each one potentially having access to a different set of protected user data such as Short Message Service (SMS) messages, location information, and the user's contact list. Each slave could then be controlled through the master application, effectively granting the external actor access to the combined permissions of every application involved in this collaboration. The implementation of this external coordination is outside the scope of the paper and is mentioned here simply to illustrate a worst-case scenario based on malicious application collaboration[8].

The communication between applications on the same Android device, however, is of direct relevance to this paper since if a covert channel were used to establish a sender-receiver or master-slave relationship as described earlier in this section, the involved applications would be able to effectively hide their malicious collaboration. A number of covert channels for the Android OS are known to exist, and any of them could potentially be used for to disguise such collaboration. However, many of these previously explored channels have much lower bandwidths, and those which are capable of higher bandwidths tend to generate large amounts of noise (excess system activity) [2], [4]. In contrast, this channel is capable of yielding high throughput with minimal system noise (i.e. number of Intents sent), and, in fact, the highest recorded throughput was achieved when using only two Intents to send a block of data.

While lower bandwidth channels would still be sufficient to leak data such as GPS coordinates, credit card information, and contact lists to a receiver application[9], the availability of a higher throughput covert channel like the one detailed in this paper could provide a benefit to applications targeting large data sets. For example, images and/or the user's messaging history could be covertly leaked using this channel[10]. Therefore the existence of this channel does pose a danger to users. This channel could also be used by malicious applications which have access to sensitive data (such as a user's contact list) but have limited time windows for transmitting that information to the receiving application (i.e. can only perform actions when the application has user focus). For such applications, being able to transmit data more quickly while still hiding that data transfer would be appealing.

[7]The term "actor" here refers to an external process or system which is actively causing some action or actions to be performed.

[8]Note that the coordination with the off-device actor could be accomplished using adaptations of known network covert channels for the Android OS [3].

[9]Here receiver refers to either the receiver application in a sender-receiver relationship or the master application in a master-slave relationship, although the slave application could also be a covert channel receiver.

[10]We say and/or because the user's messages would likely include images.

VI. Mitigation

Several options exists for preventing applications from communicating through this channel. The first is to uninstall or restrict any applications found to be using this channel, which requires the ability to accurately detect its presence. The second option is to disallow the passing of Bundles between applications. Considering that Intents, and their ability to carry Bundles, have been a core part of the Android architecture since the first released version of the Android API, this option would be very costly as it would require breaking backwards compatibility with all currently existing version of the OS. Furthermore, the removal of this ability from any or all of the currently released versions of Android would be impractical as many applications rely on it in order to function. A third option would be to implement more rigourous security controls around application communication to restrict which applications can communicate with each other and how. This could be accomplished by classifying certain application permissions as sensitive/restricted and emplacing additional barriers between applications with such permissions and those without. For example, applications with access to sensitive information could be restricted from sending Intents containing Bundles to any application which does not have explicit permission to access that data. Further protection could be provided through the implementation of active monitoring of inter-application communication since, at present, the Android OS security model relies on a static structure (permissions granted to applications upon installation and application sandboxing) to secure the system without any concept of dynamic security controls capable of reacting to suspicious application behaviour.

VII. Conclusion

The ability to violate such a central part of the Android OS's security policies as the application permissions system represents a serious issue. The fact that such an ability is available through another core part of the Android architecture, specifically the Intent application communication protocol, only serves to increase the severity of this problem. The Intent-based covert channel described in this paper provides a highly-reliable, high capacity communication channel which could allow collaborating applications to violate the existing Android security model. In order to protect users against the dangers posed by this channel, the authors recommend imposing additional levels of isolation between applications with access to sensitive information and those without and also implementing dynamic security controls based on active system monitoring to enhance the existing Android security model.

REFERENCES

[1] "Smartphone os market share 2015, 2014, 2013, and 2012." [Online]. Available: http://www.idc.com/prodserv/smartphone-os-market-share.jsp,]urldate={2015-06-08},howpublished={http://www.idc.com/prodserv/smartphone-os-market-share.jsp}

[2] H. Ritzdorf, "Analyzing covert channels on mobile devices," Ph.D. dissertation, Master Thesis ETH Zürich, 2012, 2012. [Online]. Available: http://e-collection.library.ethz.ch/view/eth:5608

[3] W. Gasior and L. Yang, "Network covert channels on the android platform," in *Proceedings of the Seventh Annual Workshop on Cyber Security and Information Intelligence Research*, ser. CSIIRW '11. New York, NY, USA: ACM, 2011, pp. 61:1–61:1. [Online]. Available: http://doi.acm.org/10.1145/2179298.2179367

[4] R. Schlegel, K. Zhang, X. Zhou, M. Intwala, A. Kapadia, and X. Wang, "Soundcomber: A stealthy and context-aware sound trojan for smartphones," in *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS)*, 2011, pp. 17–33. [Online]. Available: https://www.cs.indiana.edu/~kapadia/papers/soundcomber-ndss11.pdf

[5] "Intent | android developers," http://developer.android.com/reference/android/content/Intent.html. [Online]. Available: http://developer.android.com/reference/android/content/Intent.html

[6] "Bundle | android developers," http://developer.android.com/reference/android/os/Bundle.html. [Online]. Available: http://developer.android.com/reference/android/os/Bundle.html

[7] "Intents and intent filters | android developers," http://developer.android.com/guide/components/intents-filters.html. [Online]. Available: http://developer.android.com/guide/components/intents-filters.html

[8] "Dashboards | Android Developers." [Online]. Available: https://developer.android.com/about/dashboards/index.html

[9] "Activity | android developers," http://developer.android.com/reference/android/app/Activity.html. [Online]. Available: http://developer.android.com/reference/android/app/Activity.html

[10] "Service | android developers," http://developer.android.com/reference/android/app/Service.html. [Online]. Available: http://developer.android.com/reference/android/app/Service.html