NIVAnalyzer: a Tool for Automatically Detecting and Verifying Next-Intent Vulnerabilities in Android Apps

Junjie Tang
Shandong University, China
Email: tajunjie@gmail.com

Xingmin Cui
The University of Hong Kong, China
Email: xmcui@cs.hku.hk

Ziming Zhao
Arizona State University, U.S.A.
Email: zmzhao@asu.edu

Shanqing Guo
Shandong University, China
Email: guoshanqing@sdu.edu.cn

Xinshun Xu
Shandong University, China
Email: xuxinshun@sdu.edu.cn

Chengyu Hu
Shandong University, China
Email: hcy@sdu.edu.cn

Tao Ban
National Institute of Information and Communications Technology, Japan
Email: bantao@nict.go.jp

Bing Mao
Nanjing University, China
Email: maobing@nju.edu.cn

***Abstract*—In the Android system design, any app can start another app's public components to facilitate code reuse by sending an asynchronous message called Intent. In addition, Android also allows an app to have private components that should only be visible to the app itself. However, malicious apps can bypass this system protection and directly invoke private components in vulnerable apps through a class of newly discovered vulnerability, which is called *next-intent vulnerability*. In this paper, we design an intent flow analysis strategy which accurately tracks the intent in smali code to statically detect next-intent vulnerabilities efficiently and effectively on a large scale. We further propose an automated approach to dynamically verify the discovered vulnerabilities by generating exploit apps. Then we implement a tool named NIVAnalyzer and evaluate it on 20,000 apps downloaded from Google Play. As the result, we successfully confirms 190 vulnerable apps, some of which even have millions of downloads. We also confirmed that an open-source project and a third-party SDK, which are still used by other apps, have next intent vulnerabilities.**

***Keywords*—Android; Intent; vulnerability; static and dynamic analysis; tool**

I. Introduction

Android apps consist of four types of components, namely Activity, Service, Broadcast Receiver and Content Provider. To make it easier for app developers to handle intra- and inter-process communication and facilitate code reuse, a component in an app can be invoked by components from the same app or other apps running on the same system using intents [1]. A component is public if it can be invoked by any app running on the same system, otherwise it is private and only the components residing in the same app or apps sharing the same user ID have the privilege to trigger it. For instance, any app can start the public `LoginActivity` of the Facebook app. But only components of the Facebook app can trigger its private `SettingsActivity` to change its configuration.

Previous work has focused on studying the security issues brought by the components that are supposed to be private but have been mistakenly set as public by app developers, assuming that the Android framework can protect private components from being invoked by other apps [8], [10]. However, a new class of vulnerability named *next-intent vulnerability (NIV)* was discovered recently, showing that even private components could be invoked by other apps [14]. More specifically, a malicious app can deliver a crafted intent to start a public component of the vulnerable app, in which a target intent is saved under a special key. Then the target intent is retrieved and passed into inter-component communication (ICC) methods such as `startActivity`. The outcome is that the attacker can invoke the designated private component of the vulnerable app.

The root cause of NIV is that developers design an unsafe way to redirect components during user interaction with the app. For example, redirecting to a specified component after the user is logged-in successfully. As a result, many apps, including open source projects and third-party SDKs, still have exploitable NIVs. However, there is no automated approach to detect and verify such vulnerabilities on a large scale. Consequently, it is not known how severe and prevailing this class of vulnerabilities is in real-world Android apps. To answer these questions, in this paper we present NIVAnalyzer, which automatically detects and verifies next-intent vulnerabilities.

NIVAnalyzer is composed of two modules: NIV discovery module and NIV exploitation module. In order to find NIV efficiently and effectively on a large scale, we design an intent flow analysis strategy which accurately tracks the intent in smali code. The NIV discovery module mainly conducts static intent flow analysis, which is designed to track the target intent instance and check whether it meets all the features of NIV. Meanwhile, it generates relevant information to guide the vulnerability exploitation. The NIV exploitation module installs the vulnerable app on the Android emulator and then automatically constructs a Robotium test project [4] which includes test cases to exploit the NIV and simulate the possible login process. With the log information collected from the Android emulator during the exploitation, we are able to see whether the vulnerability is successfully exploited.

We use NIVAnalyzer to analyze 20,000 apps and finally found 190 of them have NIV. Some of the discovered vulnerable apps are popular with over 500 million downloads. We also confirmed that an open-source project and a third-party SDK, which are still used by other apps, have next intent vulnerabilities. It's possible that all apps that include them are vulnerable.

The rest of paper is organized as follows: in Section II we will briefly introduce the background knowledge of intent and Dalvik instructions. Section III gives the detailed design of our system. Section IV presents our experimental results and gives examples of vulnerable apps with NIV. Section V discusses limitation of the current solution and future work. Section VI lists related work and Section VII concludes the paper.

II. Background

A. Intent and Android Components

An Android application consists of four types of Android components: Activity, Service, Broadcast Receiver and Content Provider. An intent is a messaging object that can be used to request an action from another component [1]. It is a means of Inter-Component Communication (ICC) in Android apps and has many use cases, such as starting an Activity or Service and delivering a broadcast.

An intent is an abstract description of the operation to be performed. An intent object carries information used by the Android system to determine which component to send to either by explicitly specifying the component name (i.e. explicit intent) or declaring general intent attributes (i.e. implicit intent). It can also carry additional information in its `extras` field for the recipient component to properly perform an operation. Android APIs, such as `startActivity`, `startService` and `sendBroadcast`, use intents as paramters to conduct inter-component communication.

An Android component can be set as private or public. A component is public if its "exported" property is set as *true* or it declares at least one intent filter in the manifest file. Otherwise, the component is private. Public components can be triggered by other apps while private components can only be started by components in the same app or apps sharing the same user ID.

B. Dalvik Instructions and Smali Code

Android apps are commonly written in Java and compiled to Java bytecode, which is then translated to Dalvik bytecode and stored in a .dex file. The .dex file along with the manifest and resource files are packaged to an .apk file and delivered to users. The Dalvik bytecode is executed on the Dalvik VM. DVM is based on a register architecture. Dalvik instructions use register arguments to indicate which data to work with. In the work described in this paper, we will disassemble the binary Dalvik bytecode to smali code using Androguard [2]. Then we will analyze the smail code to look for NIV. For instance, `move-object vx,vy` is a typical smali instruction. It consists of the opcode `move-object`, a destination register *vx* and a source register *vy*. The aim of this instruction is to move the object reference from *vy* to *vx*.

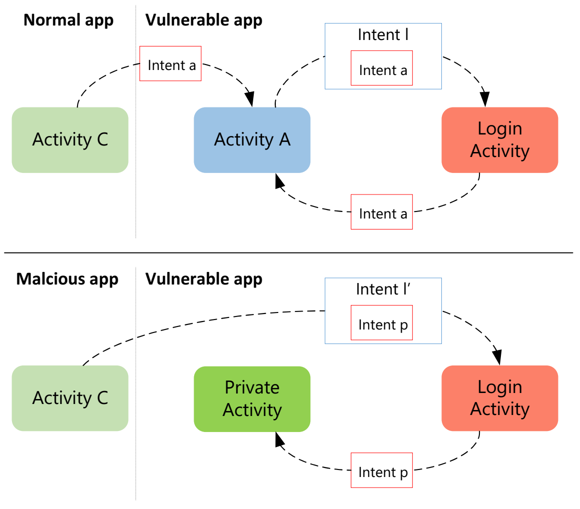

Fig. 1. Principle of NIV

C. Next-intent Vulnerability (NIV)

Wang et al. first discovered the next-intent vulnerability (NIV) in Android apps and demonstrated the vulnerability using the Dropbox and Facebook apps [14]. The attacker can invoke the private component of the vulnerable app, which is originally intended to be invoked by components within the vulnerable app. The root cause of NIV is that developers design an unsafe way to redirect components during user interaction with the app. Figure 1 illustrates an example of NIV. When a public Activity `A` is invoked by a benign component, it constructs an intent I to start Activity `Login`. Intent I is attached with another intent a under the key "next", which stores the Activity to return to after the execution of intent I. Once Activity `Login` completes its work, it would retrieve intent a under the key "next" and return to Activity `A`, which represents the redirection after a successful login.

However, the Activity `Login` is usually public for other apps to invoke. Therefore a malicious app can start it using a crafted intent I'. Intent I' is attached with the target intent p under the key "next", which will start a private Activity once executed. After Activity `Login` completes its work, the private Activity will be started and come to the foreground. Without exploiting the NIV, the malicious Activity `C` cannot invoke the private Activity.

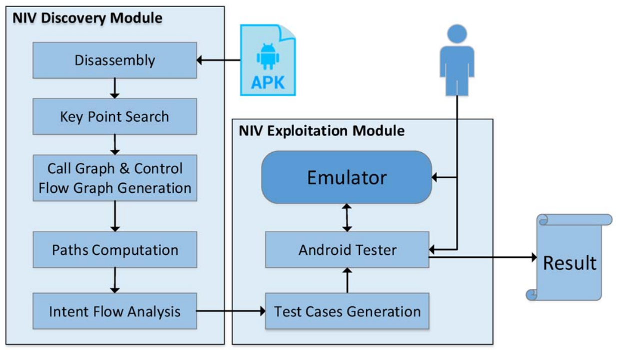

Fig. 2. Architecture of NIVAnalyzer

To better illustrate the problem and simplify further discussion, here we define some terminologies. We denote the intent extracted from another intent under a special key as the *target intent*. The *proxy intent* is the one which saves the target intent. The register that stores the target intent is called *target register* and the register that stores the proxy intent is called *proxy register*.

III. SYSTEM DESIGN

Figure 2 gives an overview of NIVAnalyzer. It is composed of two modules. The NIV discovery module takes as input the analyzed APK and checks whether it is potentially vulnerable. If yes, it activates the NIV exploitation module to generate test cases and trigger the NIV of the victim app on the emulator.

A. NIV Discovery Module

This module uses static analysis to check whether the given app contains next-intent vulnerability. It first scans the disassembled smali code to find the key points, which are used to filter the methods with potentially vulnerable code. For each method with key points, it calculates the possible execution paths of the method. We perform static intent flow analysis on each possbile execution path to determine whether NIV exists.

1) Key Points Search: We define *key points* as instructions that retrieve the target intent from another intent. To give an example, an intent invokes method `getParcelableExtra(key)` to retrieve a parcelable object with `key` as the parameter. Then the instruction with opcode `check-cast` is invoked to cast the parcelable object to an instance of `Intent`. At least one key point must exist in the code with NIV. Therefore we can quickly target the potentially vulnerable code by scanning the smali code to find all the key points. The analysis continues if key point exists in the smali code.

2) Paths Computation: Different branches may handle the target intent in different ways. Since we want to track the flow of target intent, our analysis is path-sensitive [9], [12] and takes the effect of each branch into account. For a key point, NIVAnalyzer first constructs the control flow graph (CFG) of the method that includes it. Each node in the CFG represents a basic block and directed edges represent transitions between blocks. Then we set the node with the key point as the *starting point*. The nodes that have no outgoing edges are defined as *end points*. Then we use breadth-first search to compute the possible execution paths from the starting point to all the end points. Each edge can only be traversed once in a path.

Most of the time, the number of computed paths are few. But a huge amount of paths would be generated when there are many branches in the method. To efficiently handle this case, we set the maximum threshold of the execution paths. From our statistics, we found that most methods include less than 16 branches between the key point and the sink. Therefore we set the threshold as 2^{16}. When the number of paths is bigger than the threshold, we will stop the paths calculation process and perform intent flow analysis on the already generated paths.

3) Intent Flow Analysis: The intent flow analysis takes a possible execution path as input. It first performs forward analysis to check whether this path would pass the target intent to a sink method. The *sink methods* are defined as inter-component communication (ICC) methods such as `startActivity`, `startService`, `sendBroadcast`, etc. These methods use intents as the parameter to specify the components that would be invoked subsequently. If the target intent can reach a sink method, our analysis continues to perform backward analysis to check whether the proxy intent is returned from the `getIntent` API invoked by a public component.

Forward Analysis: Smali instructions use registers to store arguments. We denote the register that stores the traced target intent as the *target register*. Our goal is to check whether the target register will be passed into a sink method. We maintain a list to track all target registers in the static analyzer. Suppose that register v is a target register, we summarize the effect of the instructions that operate on v into three cases:

- *Target retaining:* The instruction has no effect on any registers.
- *Target sanitization:* Non-target data is stored in v, meaning that v is no longer a target register and will be removed from the tracking list.
- *Target propagation:* The value in v is copied to other registers, making these registers become target registers and be added to the tracking list.

We analyze the Dalvik opcodes in [5]–[7] and divide them into three classes according to their effects. Examples of these patterns are given in table I. Method invocations belong to class I. No matter the target intent is stored in *vx* or *vy*, the instruction will not change the value of registers. The instruction of ID 2 reads an object reference instance field into *vx*. The instance is referenced by vy. If the target intent is stored in *vx*, then *vx* would be sanitized and be removed from the tracking list (class II). If the target intent is stored in *field_id*, *vx* will also become a target register and be added to the tracking list (class III). The instruction of ID 3 assigns a constant string to the target register *vx*, this would lead to the sanitization of *vx*. Here we will not explain each pattern one by one. Interested readers can refer to [5], [6] for more

TABLE I
SOME CASES OF INSTRUCTION PATTERNS

Class	Representative Patterns	Intent's Position	ID
I	invoke-virtual vx, vy, ... methodtocall	all	1
II	iget-object vx,vy, field_id	vx	2
	const-string vx, string_id	vx	3
	new-instance vx, type	vx	4
III	iget-object vx,vy, field_id	field_id	5
	move-object vx, vy	vy	6

details. When the target Intent is passed into a user-defined method, we use the call graph (CG) to find the corresponding method and continue the analysis in this method.

Several cases need to be considered carefully. For example, an app retrieves an Intent from the proxy Intent and stores it in a class field. There is a chance that the app will read the field and put it into the sink in another method. If a class field exists in the tracking list when we finish the analysis in a path, we will find all the get method of this field in the current class to track whether the filed would be passed to a sink method. When the current method returns a target register, we first find all the call sites of this method in the call graph. Then the analysis goes to the call sites in the calling methods to track the target register.

The static module of NIVAnalyzer will take corresponding actions when it encounters each class of instructions. When it finds that the register in the tracking list is passed to sink methods, it continues with the backward analysis. Otherwise, it concludes that the current path does not have NIV.

Backward Analysis: From the above analysis, we have confirmed the existence of a target intent that is retrieved from a proxy intent and finally used to invoke other components. To start our designated component, the proxy intent should be manipulable. In other words, the proxy intent should have been returned by the `getIntent` function in a public component. Therefore we do backward analysis to find out where the proxy intent comes from.

We first check whether the proxy intent is one of the current method's arguments. Registers in Dalvik bytecode have standard naming rules. For instance, suppose that a method has two arguments and uses five registers in total. Then the last two registers should store the two arguments. To be more specific, if the first register is named as v_0, arguments are saved in registers v_3 and v_4. By obtaining all the argument registers, we can easily figure out whether the proxy intent is one of the current method's arguments.

a) : If the proxy intent is not one of the current method's arguments, we begin to conduct backward analysis. Firstly we use the principles introduced in section III-A2 to calculate the possible execution paths from the starting node of the CFG of the current method to the node that includes the key point. After that we set the register that saves the proxy intent as proxy register and find out how this register is initialized on each path. Instructions of classes I and III in table I are not of our concern now since tracking only one register is enough. Suppose currently register v is the proxy register, the instruction patterns of class II can be categorized into the following cases:

- v has been assigned by another register v'. In this case we set v' as proxy register and start the tracking of v'.
- v is the parameter of the sink instruction with opcode `move-result-object`. The `move-result-object` instruction has one register to operate on and moves the return value of the previous method invocation into it. If the previous method invocation is `getIntent` and the current component is public, an vulnerability has been found and will be reported. If the previous method invocation is not `getIntent`, we find the invoking method in the call graph and continue the analysis in this method.
- v is assigned as the field value of an instance (eg. *iget-object v, v_i, field_id*). In this case we need to find where the field value is set. The analysis scans the current class to find all the initialization of this field variable (eg. *iput-object v_f, v_i, field_id*). Once it is found, we set the register that assigns the field variable as proxy register (i.e. v_f) and continue the analysis.

b) : If the proxy intent is one of the current method's arguments, we get all the callers of this method from the call graph. Then we set the register in the caller method that corresponds to the argument register in the current method as proxy register and continue with the backward analysis.

We build NIV discovery module on top of Androguard [2], an open source framework to perform various analysis of Android applications. Androguard can help us disassemble the app and translate the binary Dalvik bytecode into the more readable Smali code. It provides APIs for us to easily generate the call flow graph and the call graph of the given app.

We also adopt some strategies to gain a higher efficiency during the analysis. For example, when we find a sink in a path, we just ignore the other paths of that key point. For a key point, we maintain the method which our target intent is passed into and analyze it just once. When other paths invokes the same method, there is no need to analyze it again. What's more, we employ multi-processing so that several apps can be executed in parallel. We finish the static analysis of all the apps and store all of the relevant information to guide the subsequent dynamic analysis. Then the dynamic exploitation module is executed to exploit the potentially vulnerable apps one by one.

B. NIV Exploitation Module

The goal of this module is to verify the existence of NIV in the potentially vulnerable app and exploit the vulnerability by starting a designated private component. The NIV exploitation module installs the potentially vulnerable app on the Android emulator and then builds an attack app to exploit the NIV. With the log information collected from the Android emulator during the exploitation, we are able to tell which component is invoked and whether the vulnerability is successfully exploited.

1) Test Cases Generation: To verify the vulnerabilities discovered in the NIV discovery module, we need to generate test cases aiming at exploiting these vulnerabilities. A test case releases a crafted proxy intent, which carries a target intent, aiming to start the chosen private Activity of the vulnerable app. Basic information is required to craft the proxy intent. Besides of the vulnerable public component, the chosen private Activity and the special key to store the target intent, some well-designed key-value pairs in the proxy intent are also necessary. This is because the receiver component often verifies certain attributes from the incoming intent to check its integrity or use it to decide the control flow. Some apps even employ complex business logic and strict restriction on input values. Therefore successfully exploiting the vulnerability is complicated. In section IV we will give some practical examples to illustrate this problem.

Our approach to build meaningful proxy intents is based on the observation that if an NIV exists in one app, developers of this app will certainly use it to invoke components to fulfill the functionality of the app. More specifically, the app itself builds a proxy intent, in which a target intent is stored. So we just need to find it and extract the information we need to craft our proxy intent.

Our analysis firstly finds the special key for the target intent. After that it traces the special key forward along the already calculated execution paths of the current method. If the special key is passed into the `putExtra` method invoked by an intent as the first parameter, the calling intent is the proxy intent. Then we start searching the `putExtra` method of the proxy intent. `putExtra` has two parameters. If the first parameter is the special key to store the target intent, the second one is the current target intent. Otherwise, the parameters constitute to one key-value pair. Sometimes the second parameter (the value) is not a constant and needs to be calculated during the runtime. During test case generation, we only deal with three types of values: boolean, numeric value and string value. If the value is of type boolean, the value is set as true and false, representing the two cases respectively. For numeric and string values, we do backtrack analysis to find where they are initialized. If the value is initialized as a constant, we directly save the value. If the value is returned from a call, we only handle a few situations. For example, a `Class` variable invokes `getName` to get the name of the class and set the name to a value. We find the initialization of the `Class` variable via backward analysis, by looking for the opcode `const-class`, to retrieve the value. For other cases, we simple set the value to a preset one.

After getting the target intent, we continue to perform backward intent anlysis to find the component it wants to start. We mainly target at the methods that are used to explicitly set the class to handle the intent, eg. `init, setClass, setClassName` and `setComponent`. If it turns out the target intent aims at launching a private component, we will keep the intent value untouched and use it in the test case. Otherwise, we randomly choose a private component and set the target intent accordingly.

Finally we get a list of proxy intents for each path. Each of them corresponds to a test case.

2) Exploitation: In our manual analysis, we find that most apps launch the official login Activity before starting the vulnerable component. Once the login is done, the vulnerable code is executed. So we add a module based on Robotium to emulate human operations for automatic login. Robotium is an Android test automation framework that provides full support for native and hybrid applications and easy to write automatic black-box UI tests. The test class in the project includes all the test methods, a setUp() method that will be executed before each test method starts running and a teardown() method that automatically runs in the end to release all the resources. It guarantees that the execution of each test case is independent. Each test method will trigger a new test case and simulate the following login process. In order to test the potential vulnerable app with our test project, We have instrumented the Android system to bypass the signature verification phase. So we do not need to re-sign the potential vulnerable app.

NIVAnalyzer first runs the the potential vulnerable app. Then we manually navigate to the login Activity and pass the account information to the NIVAnalyzer console. NIVAnalyzer immediately uses UIAutomator to dump the current UI hierarchy. The exported data is stored in a XML file which we can use to find the UI widgets and simulate the login process. For instance, the type of the input field for password is an EditText and the attribute of password is set to true. Another EditText adjacent to it is the input field for username. And the hint text is usually “email”,“phone”, “username”, etc. The login button has an attribute of clickable that is set to true. And the text appeared on the button is usually “log in”, “log on”, “sign in”, and so on. There is something special that some apps have to click a login button and then the real login page is loaded. We deal with this situation by firstly finding and clicking the login button when we cannot find the input field for password. Using the above information, we can use Robotium to automatically enter the username and password into the right place in the login Activity.

In order to check whether the exploitation is successful or not, NIVAnalyzer automatically dumps the logs from the Android emulator at real time. Then it investigates the log to see whether the chosen private component is invoked, by matching the component name, start time, etc. When it finds that the chosen private component is successfully invoked, it reports that a NIV is found and starts to test the next potentially vulnerable app.

IV. EVALUATION

In this section we will give the evaluation results of NIVAnalyzer on real world apps. After that we will illustrate some practical examples to gain a clear understanding of the effects of NIVs.

A. Evaluation Result

We run our analysis on a server with 2 Intel Xeon E5-2650 CPU and 128 GB of physical memory running Ubuntu

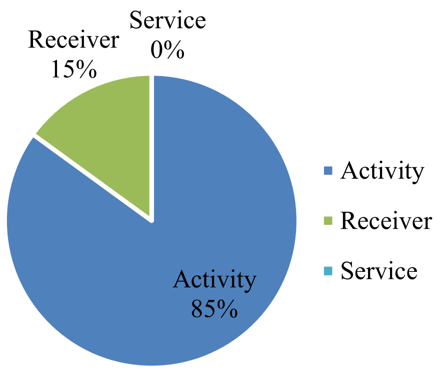

Fig. 3. Breakdown of vulnerable components.

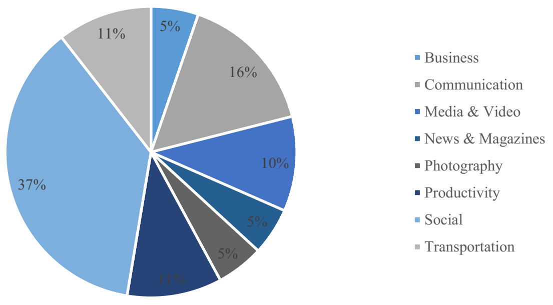

Fig. 4. Categories of vulnerable apps.

12.04. We randomly choose 20,000 apps from our database, which is crawled from Google Play Store since 2014. 503 of them failed to be disassembled, so we analyzed the remaining 19,497 apps. We use 35 processes to perform the analysis in parallel. It takes 10.8 hours to finish the static analysis.

We found 203 apps to have NIV in the NIV discovery module, and ultimately 138 apps were confirmed vulnerable by the NIV exploitation module. 65 apps can not login automatically, for a CAPTCHA is needed or complex login process or the server of these apps has been out of service. So we manually test these 65 apps and finally confirm 52 apps. We further manually analyzed 20 confirmed vulnerable apps with the highest number of downloads. Figure 3 gives the breakdown of public vulnerable components by their types. 85% of the vulnerable components are Activities, 15% are Broadcast Receivers and 0% are Services. We can see that NIV mostly exists in Activities. Among the private components that can be invoked by exploiting NIV, 91% are Activities and 9% are Services. Figure 4 shows the categories of vulnerable apps. 37% of the vulnerable apps belong to the category of social. Table II gives a fraction of these vulnerable apps. These apps are all popular on the Google Play market. Viber and Twitter have been downloaded 500 million times on Google Play.

B. Invoke Private Components Using NIV

The result of NIVAnalyzer provides the vulnerable component name and the next intent key, which makes it easy to exploit the NIV in some apps. Twitter[1] is a famous social app with more than 500 million downloads on Google play. We build an exploit intent to start the vulnerable LoginActivity, in which we construct another intent to start the private Activity under the key "android.intent.extra.INTENT". LoginActivity requires the user to login with an existing account or sign up a new account. When login is complete, it will retrieve the intent with the key "android.intent.extra.INTENT" and start it.

Obstacles exist to exploit the NIV in some apps. These apps usually extract key-value pairs in the received intent to do some validation or restriction before invoking the next intent. Hi There[2] is an SNS application with more than 5 million downloads. In the public Activity OpenTalkMain, before the app starts the intent under the key "to_intent", it retrieves the string value X stored in the received intent with key "className" and passes X to the `forName` method, causing class X to be initialized. If any error occur (eg. class X does not exist), the app will throw an exception and quit. Therefore we need to store a legitimate "className" in the crafted intent. Polaris Office + PDF[3] is a popular tool to easily view or edit documents in various formats with more than 10 million downloads. Before the vulnerable LandingActivity starts the intent under the key "runIntent", it firstly retrieves the string value stored in that intent with key "key_filename". If any error occur (eg. the key "key_filename" does not exist), the app will quit. Evernote[4] is designed for note taking, organizing, and sharing. Their official description shows that it has more than 100 million users. Its public Activity LandingActivity retrieves the target intent under the special key "EXTRA_LOGIN_PRESERVED_INTENT" when the login is complete. But before the target intent is passed to the `startActivity` method, the *action* attribute of the target intent is retrieved for evaluation. Only the action that does not equal to the already defined string values returns *true*.

These obstacles requires us to carefully design the crafted intent in order to expoit the vulnerability successfully. Knowledge of the context and operational logic of the vulnerable app, which usually demands human interaction, is a neccessity to meet this requirement. Therefore when the dynamic test fails, NIVAnalyzer just prints out the relevant information (eg. vulnerable component name, next intent key, etc) to facilitate subsequent manual analysis.

Our results show that some open-source projects and third-party SDK are also vulnerable. For example, K-9 Mail[5] is an open-source email client for Android with over 5 million downloads on Google Play. Many mail

[1]play.google.com/store/apps/details?id=com.twitter.android
[2]play.google.com/store/apps/details?id=com.psynet
[3]play.google.com/store/apps/details?id=com.infraware.office.link
[4]play.google.com/store/apps/details?id=com.evernote
[5]play.google.com/store/apps/details?id=com.fsck.k9

TABLE II
SOME VULNERABLE APPS AND COMPONENTS DISCOVERED BY NIVANALYZER

App Name	Component	Type	Next Intent Key	Downloads
MeetMe	LaunchActivity	Activity	com.myyearbook.m.extra. RESTART_INTENT	10M+
IP Webcam	Rolling	Activity	returnto	10M+
Polaris Office	FmLauncherActivity	Activity	runIntent	10M+
DB Navigator	WebAccessActivity	Activity	de.bahn.dbtickets.extra.IS_BACK_TO_ORDER	10M+
BeeTalk	BTSplashActivity	Activity	__sys__intent	10M+
MomentCam	NotificationBroadcastReceiver	Receiver	REAL_INTENT	50M+
Evernote	LandingActivity	Activity	EXTRA_LOGIN_PRESERVED_INTENT	100M+
Viber	ConversationActivity	Activity	back_intent	500M+
Twitter	LoginActivity	Activity	android.intent.extra.INTENT	500M+

apps are extended from it. One of its public Receivers BootReceiver is vulnerable. A crafted intent with action "com.fsck.k9.service.BroadcastReceiver.schedule-Intent" can start BootReceiver. BootReceiver extracts the target intent under the key "com.fsck.k9.service.BroadcastReceiver.pending Intent". After that BootReceiver constructs a new intent to start itself, in which it saves the extracted target intent under the key "com.fsck.k9.service.BroadcastReceiver.fireIntent". It uses the system alarm services, which can fire intents at the scheduled time or after the given time interval even if the current component is not active any more. Finally, BootReceiver retrieves the target intent under the key "com.fsck.k9.service.BroadcastReceiver.fireIntent" and passes it to the `startService` system API.

Similarly, a third-party SDK named BuzzBox [3] is vulnerable. The BuzzBox SDK enables developers to easily add a scheduler to their apps in order to perform background tasks or send notifications. The problem is with a public Broadcast Receiver SchedulerReceiver, which belongs to the scheduler module. The receiver retrieves the target intent under the key "message.intent" and passes it to the `startActivity` system API. We learn from the manifest file that SchedulerReceiver can be invoked by an intent with action "com.buzzbox.mob.android.scheduler.wakeup". So we can build an intent to start it and save the target intent under the key "message.intent" to invoke any private activities.

C. Attacks and Consequences

NIV can expose new attack surfaces, which cause serious consequences for the vulnerable app once being exploited. The consequence depends on the capability of the private components. Next we show some attacks to illustrate the severity of the problem.

Utanbaby[6] is an android app dedicated to young mothers. The official description shows that 20 million young mothers use it to communicate, share and learn during pregnancy and parenting in China. We confirm NIV in the public LoginActivity. We find that 6 private Activities retrieve a string value from the launch intent as a URL, which is then loaded in the WebView. By manipulating the launch intent, the attacker can open arbitrary web pages using the vulnerable app. What's worse, some Activities attach sensitive information in the request. JingpinWebActivity and ShopWebActivity attach an authentication header to the request. NativeMerchantActivity even put the userID, email, phone type, operating system type, MAC address, authentication token and other sensitive information in the request. In addition, the attacker can disturb some logic flows of the app. DjBuyDialogActivity fetches a string value "picurl" that is saved as a URL and used to load as the photo of a product. This string value can be manipulated by the attacker. The name and price are set in the same way. MallMyRebateActivity is used to show the rebate, which is also retrieved from the launch intent. At last, a lot of Activities pass "null" values using the launch intent to methods that cannot handle the "null" value. This causes the app to crash, which has the same effect as a denial-of-service attack.

[6]http://www.utanbaby.com/

What's worse, the attacker can open arbitrary web pages in almost all vulnerable apps. For example, The private PasswordResetActivity of Twitter fetches a string value "init_url" that is saved as a URL from intent and loads this url. So the attacker can launch any carefully designed phishing site in the official Twitter app using NIV to steal the user's privacy information. Similarly, the GenericWebViewActivity and the ViberOutWebViewActivity of Viber fetch a string value "extra_url" to launch and fetch a string value "extra_title" to set the title. The PromoWebActivity of Evernote retrieves a string value "URL" from the intent and load it. The BTFullScreenWebVideoActivity of BeeTalk takes a string value "video" from intent to launch.

V. DISCUSSION

Our work still has some limitations. Firstly, in the key point searching process, since the two other methods for receiving intents, namely `getParcelableArrayExtra()` and `getParcelableArrayListExtra()` are rarely used, we only considered the method `getParcelableExtra()`. Besides, we construct test cases following the way in which the app itself builds a proxy intent. In the future work, we will combine symbolic execution techniques to construct the exploit intent in a smarter way. The efficiency and accuracy of the Intent flow analysis could be further improved. And UI automation still has a bit of room for improvement by analyzing and categorizing more login pages. In addition, we look forward to finding ways to fix this vulnerability [18].

VI. RELATED WORK

With the popularity of Android platform and the widespread of Android apps, a considerable number of work has been focusing on analyzing Android apps and discovering vulnerabilities in them.

Rui Wang et al. [14] first discovered NIV and demonstrated two attacks exploiting the NIV in Dropbox and Facebook app. But they did not provide how to detect NIV in a large scale of apps. To the best of our knowledge, our work is the first one to make this effort.

Static analysis does not need to install and exercise the app, therefore it is usually more efficient than dynamic analysis. FlowDroid [11] is a context-, flow-, field-, object-sensitive and lifecycle-aware static taint analysis tool for Android applications. FlowDroid tracks all the data tainted by the source method in a path, and we only need to track the intent in our problem.

ComDroid [8] and Epicc [13] explored potential vulnerabilities brought by intent in Android apps. ComDroid focuses on one single component and gives warnings when it encounters misconfigurations such as exported components or implicit intent. ComDroid only warns about exported component, our work warns about callable private components, which is more difficult to detect. Besides, ComDroid produces too many false positives. [13] aims to connect components within one application and between different applications. Therefore it works on a large scale of applications and matches the exit/entry points of the currently analyzed application with a set of entry/exit points stored in the database. Epicc learns and records inter-component communication by matching intents with intent-filters. Our work aims at a more complex logic that uses crafted intent to launch designated private component.

Hassanshahi et al. [17] proposed an automated W2AIScanner to find and confirm web-to-app injection vulnerabilities, which can also invoke a private Activity, just like NIV. A name parameter derived from URL is passed into Android native interfaces to launch specific activities appointed by the name. So it's different from NIV explored in this paper. Felt et al. [10] built a path-finding tool to find vulnerabilities, which looks for an exploitable path in the application between a public entry point and a restricted system API call.

However static analysis becomes ineffective in the face of code obfuscation and runtime callbacks. Besides, the paths explored in static anlaysis may not necessarily execute in dynamic anlaysis. Therefore a hybrid of static and dynamic analysis is adopted to detect and further confirm the vulnerability. For example, AppIntent [15] first uses static analysis to detect sensitive data transmission paths. Then it uses dynamic symbolic execution to check whether the sensitive data transmission is user intended or not. AsDroid [16] is a work similar to AppIntent. Our work also adopts such a strategy.

VII. CONCLUSION

The next intent vulnerability in Android apps is a serious issue. By exploiting this vulnerability, attackers can directly start private components from other apps. Even though some next intent vulnerabilities have been discovered with manual analysis, it is not clear how prevalent it is in real-world apps. In this paper, we designed and implemented NIVAnalyzer to discover and confirm next intent vulnerability using static and dynamic analysis. Using our tool, we analyzed 20,000 apps downloaded from Google Play and finally found 190 apps that have NIV, some of which even have millions of downloads. In addition, we also confirmed an open-source project for Android and a third-party SDK also have NIV.

ACKNOWLEDGMENT

We would like to thank the anonymous reviewers for their comments on previous drafts of this paper. This work is partially supported by National Natural Science Foundation of China (91546203,61173068,61572295,61573212) , Program for New Century Excellent Talents in University of the Ministry of Education, the Key Science Technology Project of Shandong Province (2014GGD01063,2015GGE27033), the Independent Innovation Foundation of Shandong Province (2014CGZH1106) and the Shandong Provincial Natural Science Foundation (ZR2014FM020).

REFERENCES

[1] Intents and Intent filters, http://developer.android.com/guide/components/intents-filters.html
[2] Androguard, https://github.com/androguard/androguard
[3] buzzbox, http://hub.buzzbox.com/android-sdk/
[4] Robotium, https://github.com/RobotiumTech/robotium
[5] Dalvik opcodes, http://pallergabor.uw.hu/androidblog/dalvik_opcodes.html
[6] Opcodes — Android Developers, https://developer.android.com/reference/dalvik/bytecode/Opcodes.html
[7] A. Bartel, J. Klein, Y. Le Traon, M. Monperrus, "Dexpler: Converting Android Dalvik bytecode to Jimple for static analysis with Soot," Proc. SOAP, 2012.
[8] E. Chin, A. P. Felt, K. Greenwood, D. Wagner, "Analyzing interapplication communication in Android," Proc. MobiSys, 2011.
[9] I. Dillig, T. Dillig, A. Aiken, "Sound complete and scalable path-sensitive analysis," In PLDI, 2008.
[10] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, E. Chin, "Permission Re-Delegation: Attacks and Defenses," USENIX Security, 2011.
[11] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, P. McDaniel, "FlowDroid: Precise context flow field object-sensitive and lifecycle-aware taint analysis for Android apps", Proc. PLDI, 2014.
[12] H. Hampapuram, Y. Yang, M. Das, "Symbolic path simulation in path-sensitive dataflow analysis", PASTE'05, pp. 52-58, 2005.
[13] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, Y. le Traon, "Effective inter-component communication mapping in Android with Epicc", USENIX Security, 2013.
[14] R. Wang, L. Xing, X. Wang, S. Chen, "Unauthorized origin crossing on mobile platforms: Threats and mitigation", Proc. CCS, 2013.
[15] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, X. S. Wang, "AppIntent: Analyzing sensitive data transmission in Android for privacy leakage detection", Proc. CCS, 2013.
[16] J. Huang, X. Zhang, L. Tan, P. Wang, B. Liang, "AsDroid: Detecting stealthy behaviors in Android applications by user interface and program behavior contradiction", Proc. ICSE, 2014.
[17] B. Hassanshahi , Y. Jia , R. H. Yap , P. Saxena , Z. Liang, "Web-to-Application Injection Attacks on Android: Characterization and Detection", Springer. Computer Security C ESORICS 2015, pp. 577 C 598, Nov. 2015
[18] C. Yagemann, W. Du, "Intentio Ex Machina: Android Intent Access Control via an Extensible Application Hook", Springer. Computer Security C ESORICS 2016, pp 383-400, Sep. 2016