Practical Exploit Generation for Intent Message Vulnerabilities in Android

Daniele Gallingani
University of Illinois at Chicago
Chicago, IL
Politecnico di Milano
Milano, Italy
daniele.gallingani@mail.polimi.it

Rigel Gjomemo
University of Illinois at Chicago
Chicago, IL
rgjome1@uic.edu

V.N. Venkatakrishnan
University of Illinois at Chicago
Chicago, IL
venkat@uic.edu

Stefano Zanero
Politecnico di Milano
Milano, Italy
stefano.zanero@polimi.it

ABSTRACT

Android's Inter-Component Communication (ICC) mechanism strongly relies on Intent messages. Unfortunately, due to the lack of message origin verification in Intents, application security completely relies on the programmer's skill and attention. In this paper, we advance the state of the art by developing a method to automatically detect potential vulnerabilities and, most importantly, demonstrate whether they can be exploited or not. To this end, we adopt a formal approach to automatically produce malicious payloads that can trigger dangerous behavior in vulnerable applications. We test our methods on a representative sample of applications, and we find that 29 out of 64 tested applications are potentially vulnerable, while 26 of them are automatically proven to be exploitable.

Categories and Subject Descriptors: D.4.6 [Operating Systems]: Security and Protection

Keywords: Android, Application Security

1. Introduction

Android applications are formed by logically separated components that mainly communicate with each other through *Intents*, which carry data and request the execution of a procedure to another application. However, the Android Intent Passing mechanisms do not provide the receiving component with any information concerning the origin of an intent, thus facilitating the creation of spoofed intents with malicious input data [1, 2]. If such malicious input is not properly validated or sanitized by an application before being processed, it may subvert its state and control flow in unexpected ways. This attack vector may lead to a wide range of attacks, not only against the application itself, but also against other applications that receive and process data from the vulnerable app.

Previous research studied applications and the Android ecosystem to identify components that are exposed to receiving intents from untrusted applications and to check whether there exist dataflows from data input points to critical operations. However, this previous research is not being able to automatically verify the *practical exploitability* of the discovered dataflows [2]. This is due to two main reasons: 1) static analysis techniques approximate the behavior of programs and include additional behaviors that are not actually present and, 2) analysis techniques in state-of-the-art approaches only take into account the existence of potential suspicious paths and ignore the effect of the code along those paths, such as the use of input validation to mitigate intent spoofing vulnerabilities [1].

In this paper, we automatically develop proof-of-concept exploits against applications, to effectively prove that they are vulnerable to intent message vulnerabilities. We statically analyze the application to identify data-flows under an attacker's (indirect) control and design an analyzer that is able to follow such flows and identify Intent data that may affect either directly or indirectly the results that a component produces and sends in output. At this point, we use a constraint solver to develop concrete proof-of-concept exploits, thereby confirming the presence of the vulnerability. We test our approach on 64 popular applications from the Google Play store. Of these, 29 exhibit potential vulnerabilities, and for 26 of these, we are able to automatically generate an exploit, i.e. spoofed intents that trigger and demonstrate those vulnerabilities.

2. Problem Statement

Threat Model. In our threat model, an attacker first analyzes the manifest file to identify exposed components that can receive intent messages. An example of such a component is depicted in Listing 1, where the *onCreate* method (line 1) is called to start a component. Next, the attacker identifies statements inside those components whose execution may be subverted to the attacker' s advantage. These statements may include network operations, database operations, updates to GUI elements and so on, and their execution may be subverted by modifying their parameter values, e.g., URLs where data are sent by network operations, database queries, and the text of GUI elements (e.g., the HttpPost object creation in line 19, where the attacker may be able to modify the value of the *url*

This work was partially supported by National Science Foundation grants CNS-1065537, CNS-1069311, CNS-12416854

CODASPY 2015 Mar 02-04 2015, San Antonio, TX, USA
ACM 978-1-23456-78-9/01/23
http://dx.doi.org/10.1145/2699026.2699132.

variable to a domain under the attacker's control, or the statement on line 31, where the attacker may be able to modify the value of the variable *p*). We call such statements targeted by an attacker *sink* statements and those statements that extract intent data values *source* statements (e.g., lines 2-5).

Listing 1: Source code of a vulnerable application

```
1  void onCreate(Bundle savedInstance) {
2   Intent intent=getIntent();
3   String host = intent.getStringExtra("hostname");
4   String user = intent.getStringExtra("username");
5   String file = intent.getStringExtra("filename");
6   String url="http://www.example.com";
7   if (host.contains("example.com"))
8     url = "http://" + host + "/";
9   if (file.contains(".."))
10    file = file.replace("..", "");
11  String userId = getUserID(user);
12  if (userId != -1)
13    textView.setText(user_name);
14  String b64File = toBase64(file);
15  String httpPar = toHttpParams(b64File,user_id);
16  . . .
17  try {
18    DefaultHttpClient httpC = new DefaultHttpClient();
19    HttpPost post = new HttpPost(url+httPar);
20    . . .
21    httpC.execute(post);
22  }
23  catch(IOException e) {
24    e.printStackTrace();
25  }
26 }
27 String toBase64(String p) {
28  if(p=null || p.equals(""))
29    p = "/data/data/com.example/defaultFile.pdf";
30  else
31    p = "/data/data/com.example/public/" + p;
32  byte[] bytes = InputStream.read(p);
33  String b = Base64Encoder.toString(bytes);
34  return b;
35 }
```

As has been recognized by previous work, to detect this type of vulnerabilities, it is important to correctly identify paths that starting from the *source* statements enable an attacker to influence the variable values at the *sink* statements. However, the existence of a path does not imply that an attack is feasible. To precisely identify exploit opportunities and prevent them, the operations performed on the variable values along that path must also be considered. In fact, these operations may include sanitizations (e.g., lines 9-10), and other business logic operations (e.g., lines 11-13) that, while allowing an attacker to influence the values at the sink statements, make exploits unfeasible. In the rest of the paper, we present a method for automatically detecting such vulnerabilities and providing exploitability proofs for them.

3. Approach

3.1 Problem Formulation and Approach Overview

We formulate the problem as follows. Let the *state* of an application at a particular point in the program be defined as a set of (variable, value) pairs $V = \{(v_1, a_1), (v_2, a_2), ..., (v_n, a_n)\}$ visible at that point during execution. To successfully launch an exploit on a specific *sink*, an attacker needs to induce a state of the attacker's choosing at that sink. We denote this state by *exploit state* and represent it with a set of (variable, value) pairs $V_E = \{(v_{e1}, b_1)(v_{e2}, b_2)..., (v_{em}, b_m)\}$, where the variables v_{ei} represent the parameters of the sink statement. Furthermore, to be able to induce state V_E at the sink, the attacker can only use the input state at the *source* statements defined as a set of (variable, value) pairs $V_I = \{(v_{i1}, c_1), (v_{i2}, c_2)..., (v_{in}, c_n)\}$.

From a defensive and application vetting perspective, there can be many sink statements of interest to an attacker and they can be located anywhere inside the application. In addition, there can be many exploits for each sink statement. For example, if an attacker has control over the application state at a database query statement, there can be many exploit states V_E at his disposal, each performing a different type of SQL injection.

Therefore, we state the problem as follows. Given a point p in the program and an arbitrary exploit state V_E in p, can we automatically determine if there exists a state V_I at the *source* statements that induces V_E in p when the program executes? To answer this question, the relationship between the state at any point in the program and state V_I at the source statements must be made explicit. The discovery of such relationship and its modeling as a function F, such that we can automatically compute $V_I = F(V_E)$ is at the core of our approach. The steps of our approach are described next.

Paths computation. Since every program point p may be a sink statement, in this first step we compute all the paths between the *source* statements and every point in the program. Such analysis faces two main challenges. The first challenge arises due to the interprocedural nature of Android programs. In fact, the analysis must be able to deal with interprocedural data-flows and identify paths through deep sequences of method calls as well as recursive method calls. The second challenge is that of *path explosion*, which is a common problem in data flow analysis, and which becomes even more problematic in an interprocedural context.

To deal with these challenges, we first divide the methods into two sets: user-defined and libraries. Next, a control-flow supergraph of the user-defined methods is created by the data-flow analysis framework. In this supergraph, the call sites are joined with callee definitions and callees' exit sites are joined back with the call sites. We show a portion of the supergraph built for our example in Figure 1, where each node corresponds to a statement and is labeled with the line number from code listing 1. To further limit the size of the supergraph, we perform a preliminary *taint propagation* step. During this step, the nodes of the supergraph are partitioned into two sets, namely the set of statements that use values in V_I and can thus be influenced by an attacker, and the set of statements whose execution is independent from an attacker.

The taint propagation and the path computation steps are modeled as an IFDS data-flow analysis problem where taint and path information is collected inside facts associated with each point in the program [3, 4].

Symbolic execution. After the path computation step, symbolic execution over the paths is performed to derive the set of constraints imposed over the variable values along those paths. At the end of this step, for every program point p, a logic formula F_p is created, whose variables correspond to program variables and whose terms correspond to the program statements that modify those variables. Thus, F_p represents the relationship between the input state V_I and the application state in p. The syntax of the symbolic formulas used in our approach is described by the pseudo-grammar shown in Figure 2, where terminal symbols are represented in bold.

In particular, the terminal symbol $solv\_stat$ represent Java statements whose operations semantics can be modeled by the solver used in the exploit generation step. In addition, these are the statements that use at least one tainted variable. Conversely, $nonsolv\_stat$ represents statements that cannot be modeled by the solver. Finally, var represents tainted variables, $constant$ represents constant strings, while the $+$ terminal represents string concatenation.

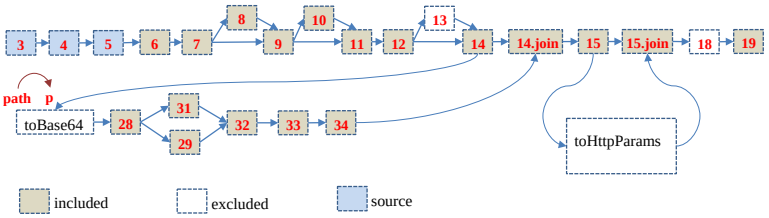

Figure 1: Supergraph built by the analysis framework.

$$
\begin{array}{lcl}
F_p & \rightarrow & F_p \vee conj \mid conj \\
conj & \rightarrow & (conj \wedge term) \mid term \\
term & \rightarrow & stat \mid \neg stat \\
stat & \rightarrow & statement \mid \boldsymbol{var} == statement \\
statement & \rightarrow & statement + single\_stat \mid single\_stat \\
single\_stat & \rightarrow & \boldsymbol{var} \mid \boldsymbol{constant} \mid library\_method \\
library\_method & \rightarrow & \boldsymbol{solv\_stat} \mid \boldsymbol{nonsolv\_stat}
\end{array}
$$

Figure 2: Grammar of Symbolic Formula

For instance, the formula F_p related to the sink statement on line 20 of Figure 1 is derived as:

$(host.contains(\text{"}example.com\text{"}) \wedge url == \text{"}http://\text{"} + host + \text{"}/\text{"}) \vee$
$(!\ host.contains(\text{"}example.com\text{"}) \wedge url == \text{"}http://www.example.com/\text{"})$

We note that each term in the symbolic formula represents a statement along the path, while each new path created by a branching statements is represented using a disjunction. Assignment operations in the code are modeled using equality constraints, in order to capture the equality conditions between two expressions.

Exploit generation. The input to this step is a program point p, the corresponding formula F_p and a set of assignments representing the exploit state V_E. The first operation of this step is the translation of the symbolic formula F_p into the solver's language. In particular, for each member of the $solv\_stat$ statements, we create a set of constraints in the language of the solver, which model the behavior of that statement. The members of the $unsolv\_stat$ statements are modeled with a particular operator in the solver's language that returns the whole domain of values for the variable.

For the implementation of this step, we chose the Kaluza solver, which provides string solving capabilities and several primitives that can model string operations [5]. Given the symbolic formula $F_p \wedge V_E$ for a particular point p, the solver provides a (possibly empty) solution containing the values of the variables in V_I, corresponding to the malicious input.

4. Results

Table 1: Experimental Results

Application	Implication
IM+	Display an arbitrary web page inside Activity && change Activity title.
Mint	Display an arbitrary web page inside an Activity
PromoQui	Load an arbitrary web page and change the common purchase process
GoSMS	Prompt to the user notification about a new message with arbitrary sender and SMS content

We evaluate our approach on 64 applications from the Google Play Store. The evaluation detected paths from sources to sinks in 29 of the 64 applications. Out of these 29 applications, the solver was able to produce exploits for 26 of them. The majority of these exploits enable an attacker to arbitrarily change different UI components inside the applications, thus potentially leading to phishing attacks. Some of our results are summarized in Table 1, which describes the security implications for each application.

Table 2 shows different measurements of our approach. In particular, for the 64 applications under evaluation, we discovered 92 paths leading to a sink out of 537, with an average of 4.2 vulnerable paths per application. For each vulnerable path the analyzer collected an average number of 17.2 statements with an average number of 5.8 statements including API calls.

	Min	Max	Avg
Per-application execution time	2.4 min	33.2 min	12.3 min
Per-application components	3	31	24.5
Per-application vulnerable paths	2	19	4.2
Per-path statements	5	81	17.2
Per-path if-statements	0	3	0.98

Table 2: Other experimental setup

Our results show that only a few applications properly implement validation of data received from Intent messages. Most applications perform only basic checks on Intent payloads parts, such as null-checks, performed only to protect against malformed Intents data. However, such checks are not strict enough to defend application resources against more sophisticated Intent spoofing attacks.

5. Conclusions

In this paper, we provide an automated analysis framework to study vulnerabilities in Android applications arising from the lack of validation checks over Intent data. Improving over the state of the art, we provide proofs of vulnerabilities by automatically generating exploits under the form of malicious data to be sent with an Intent message. We evaluate our approach and find several vulnerabilities and exploits for those vulnerabilities. Our results confirm that a large percentage of commonly-used applications do not implement appropriate security safeguards.

6. References

[1] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, "Analyzing inter-application communication in android," in *Proceedings of the 9th international conference on Mobile systems, applications, and services*. ACM, 2011, pp. 239–252.

[2] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "Chex: Statically vetting android apps for component hijacking vulnerabilities," in *ACM Conference on Computer and Communications Security (CCS)*, 2012, pp. 229–240.

[3] "Heros ifds/ide solver," http://sable.github.io/heros/, 2013.

[4] E. Bodden, "Inter-procedural data-flow analysis with ifds/ide and soot," in *ACM SIGPLAN International Workshop on State of the Art in Java Program analysis*, ser. SOAP. ACM, 2012, pp. 3–8.

[5] "Kaluza string solver," http://webblaze.cs.berkeley.edu/2010/kaluza/, 2010.