1-1-2014

Analysis and Prevention of Code-Injection Attacks on Android OS

Grant Joseph Smith
University of South Florida, gsmith@mail.usf.edu

Analysis and Prevention of Code-Injection Attacks on Android OS

by

Grant J. Smith

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science in Computer Science
Department of Computer Science and Engineering
College of Engineering
University of South Florida

Major Professor: Jay Ligatti, Ph.D.
Yao Liu, Ph.D.
Hao Zheng, Ph.D.

Date of Approval:
October 22, 2014

Keywords: Injection Attacks, BroNIEs, Formal Definitions, SQL, OS Shell

ACKNOWLEDGMENTS

I would like to thank Dr. Jay Ligatti for being an inexhaustibly helpful, constructive, and optimistic advisor. I would also like to thank Clayton Whitelaw for his help in reviewing and proofreading this thesis. Finally, I would like to thank all the rest of my friends and family, who have been very supportive of and interested in my work.

TABLE OF CONTENTS

LIST OF FIGURES ii

ABSTRACT iii

CHAPTER 1 INTRODUCTION 1
- 1.1 OS Shell Injection 2
- 1.2 SQLite Injection 3
- 1.3 Summary of Contributions 7
- 1.4 Thesis Outline 8

CHAPTER 2 RELATED WORK 10
- 2.1 General Android Security 10
- 2.2 Theory of Injection Attacks 11
- 2.3 Taint Tracking 12

CHAPTER 3 ATTACK SURFACES 14
- 3.1 SQLite 14
- 3.2 OS Shell 15
- 3.3 Possible Attacks into OS Shell 18

CHAPTER 4 A REVIEW OF BRONIES 21
- 4.1 Template String 21
- 4.2 Token Expansion 22
- 4.3 Token Classification 22
- 4.4 The NIE Property 23
- 4.5 Stopping a BroNIE 23

CHAPTER 5 PROTECTION MECHANISM IMPLEMENTATION 25
- 5.1 Taint Tracker 25
- 5.2 Weaving with AspectJ 28
- 5.3 Detecting BroNIE's 29

CHAPTER 6 CONCLUSION 32

LIST OF REFERENCES 33

LIST OF FIGURES

Figure 1.1	Java code for an Android application vulnerable to shell injection	2
Figure 1.2	Vulnerable Java code.	4
Figure 1.3	Code for an application that uses Android's SQL injection security mechanism	6
Figure 3.1	Code for our timing test application	19
Figure 3.2	A graph showing timing data gathered from the test application.	19
Figure 5.1	Code for the taint tracking module we have added to the Java libraries	27
Figure 5.2	AspectJ code to call the taint tracker from application code	28
Figure 5.3	Algorithm to detect BroNIEs using a template string and a program string	30

ABSTRACT

Injection attacks are the top two causes of software errors and vulnerabilities, according to the MITRE Common Vulnerabilities list [1]. This thesis presents a threat analysis of injection attacks on applications built for Android, a popular but not rigorously studied operating system designed for mobile devices. The following thesis is argued: Injection attacks are possible on off-the-shelf Android systems, and such attacks have the capacity to compromise the device through resource denial and leaking private data. Specifically, we demonstrate that injection attacks are possible through the OS shell and through the SQLite API. To mitigate these attacks, we augment the Android OS with a taint-tracking mechanism to monitor the flow of untrusted character strings through application execution. We use this taint information to implement a mechanism to detect and prevent these injection attacks. A good definition of an attack being critical to preventing it, our mechanism is based on Ray and Ligatti's formalized "NIE" property, which states that untrusted inputs must only insert or expand noncode tokens in output programs. If this property is violated, an injection attack has occurred. This definition's detection algorithm, in combination with our taint tracker, allow our mechanism to defend against these attacks.

CHAPTER 1

INTRODUCTION

As of January 2014, a study has shown that 58% of American adults own and use a smartphone [2]. The increasing ubiquity of this new computing platform has also opened a new battleground in computer security. Smartphones are filled with private user data such as contact databases, call histories, camera images, and the contents of any mounted SD card. If this private data is compromised by an attacker, it could have high value to advertisers, spammers, and stalkers. Phones provided by the workplace may also store confidential company information, making it even more critical to secure these devices against software vulnerabilities. A leak of sensitive company data is a constant threat and 33% of IT professionals said that the most vulnerable points for data leakage on some secure network are the USB storage devices [3]. Android phones, with their USB connectivity and SD card mounting capability, fall into this category.

Android OS is an open-source mobile device operating system [4] built on the Linux kernel. It implements a security system modeled on Linux, with several modifications. Android restricts at install time what an application may access on the system [5, 6]. Applications declare in their package manifest which permissions they require to run properly and these permissions are displayed to the user when downloading an application from the Google Play store or installing an application from any other source. The user may then choose to install the application and grant the required permissions, or abort the install. The user cannot force an application to run with fewer permissions than it has requested in off-the-shelf distributions of Android. There is no monitoring of how an application uses data or services; permissions determine only whether an application may use certain data or services.

This lack of fine-grained runtime monitoring makes Android vulnerable to injection attacks. This thesis will argue that such attacks are possible on Android OS, through both the OS shell and SQLite API, and that these attacks are able to compromise the device.

```
// Obtain a process running an OS Shell
Process proc = Runtime.getRuntime.exec(``sh -i'');
// Obtain a stream to send commands to that process
OutputStream programInput = proc.getOutputStream();
// Obtain some untrusted input
String userinput = getUntrustedInput();
// Create a command with untrusted input, and execute the command
programInput.write(``ping -c 4'' + userinput + ``\n'');
```

Figure 1.1: Java code for an Android application vulnerable to shell injection

1.1 OS Shell Injection

Consider an example application that uses user input in a simple shell program to "`ping`" a remote host specified by the user. Application code for this functionality might look like the code in Figure 1.1. This program uses the `Runtime` class and its method `.exec()` to obtain a shell running inside an instance of the `Process` class called `proc` and obtains an `OutputStream` to that `Process`. It then uses `proc` to execute a shell program that pings a remote host specified by user input with four packets. The newline character at the end of the program instructs the shell to execute the command given by the string preceding the newline.

Unfortunately, this application accepts input from an untrusted source. A malicious attacker controlling that source could perform a code injection attack on the program output by the shell. Consider the following possible input strings. The text input by the attacker is underlined for clarity.

1. `www.remotehostname.com'' && rm -rf ``data''`, to make the application output the program `ping -c 4 ``www.remotehostname.com'' && rm -rf ``data'' \n` The `&&` symbol causes the program to execute the next command, based on whether the previous command returned successfully. The first command will `ping` some supplied remote host name, as expected. The second command begins with `rm`, the command to delete directories or files from the filesystem. The first argument to `rm` is `-rf`, meaning that the program will remove subdirectories and files inside the specified directory recursively without warning the user. The second argument to `rm` is `data`, which will be used as the name of the directory

to delete. If `data` is a location containing some crucial files, this injection attack could be very damaging.

2. `www.remotehostname.com && f() { f|f& } && f \n #''`, to the make the application output the program `ping -c 4 ‘‘www.remotehostname.com'' && f() { f|f& } && f \n #'' \n''`. This program inserts a subprogram via the `&&` symbol in the same fashion as before. The second program this time begins with the creation of a function `f` that recursively executes itself indefinitely and maintains a pipe from each parent process to each child process, to prevent the parent from terminating. It then executes this function `f`. The final `#` character is used to start a line comment, and eliminate the unused closing quotes from the application code. The result of this program is a full process table, preventing the operating system from creating any new processes. To fix resource denial, the user is required to wipe volatile memory, which might be accomplished by removing the phone's battery. For mobile devices that have non-removable batteries, the user might need to wait until the device runs out of battery naturally. This attack is colloquially known as a "forkbomb".

In practice, these vulnerabilities might be used in the following way. The user decides to install some application that they trust. The user gives the application permission to access a protected resource—the SD card, say. The trusted application, unbeknownst to the user, has an injection vulnerability like the one above and takes input from an untrusted source. An attacker could then inject into this application and take advantage of its permission to access the SD card to read, write, and modify files. If the target phone was holding sensitive data, the results could be disastrous.

1.2 SQLite Injection

The Android OS ships a SQLite API with its development kit. SQL injection vulnerabilities are well studied and are the single most common vulnerability in software, according to the MITRE Common Vulnerabilities List [1]. A SQLite database in Android is not protected by a permission restriction in the application manifest but does have several countermeasures against injection. The countermeasures take the form of wrapper methods to access the database [7]. These wrapper methods all have overloads that utilize prepared statements. Prepared statements are a common

```
// Construct a SQLite query string
String query =
// Match rows where KEYVALUNAME is equal to untrusteduname and
KEYVALUNAME + '' = ''' + untrusteduname + ''' AND '' +
// Rows where KEYVALPASS is equal to untrustedpass
KEYVALPASS + '' = ''' + untrustedpass + ''' '';
// Store results from the query into a cursor
Cursor cursor = sqldatabase.query(
// Pass to the query the table to search, the columns to return
TABLEVALDEMO, new String[] { KEYVALID, KEYVALUNAME, KEYVALPASS },
// and our query string. Other optional args are null.
query, null, null, null, null, null);
// If any match was found, return true, else return false
return(cursor.moveToFirst())}
```

Figure 1.2: Vulnerable Java code.

SQL injection prevention mechanism that "prepare" a SQL query before concatenating in user input [8]. The untrusted strings are then converted to equivalent values and added to the query. We call this behavior "string binding." Closed values are noncode, so this protection mechanism fully defends against injection attacks if used properly. However, applications do not automatically implement this mechanism. The onus is on the developer to make use of these overloads. Developers who are unfamiliar with the dangers of injection attacks might not know that they need to program their applications securely.

An example of such a poorly coded application might look like Figure 1.2. The program is querying a SQLite database with columns for primarykey, username and password for the existence of a particular username and password in the database, attempting to authenticate some user. It constructs a query string that implements this behavior. Note that SQLite requires that string literals in queries be enclosed with single quote characters. The single quote characters to enclose `untrusteduname` and `untrustedpass` are included in the application code, before and after concatenating in the string variables. The query string is then used as an one of the arguments to Android's SQLite API method `.query()`. The code also passes to the `.query()` method the

name of the table to be queried and which columns of information to return in the results. Other arguments, including how to sort the results and how to group the results are left as null. The query method returns an instance of the `Cursor` class, a datatype that contains the results of the query. The code should then return true if it has found a match to the username/password combination in the database and false otherwise. The `.moveToFirst()` method of the `Cursor` class implements exactly that behavior, returning false if the `Cursor` is empty, and true otherwise. This is a standard example of how a SQLite database might be queried in an Android application. If the source of untrustedname and untrustedpass are controlled by an attacker, an injection attack could be executed on this application code.

Consider the following example. Suppose that our example database from Figure 1.2 contains only 1 entry—the entry for the administrator account. The username for the administrator account is "admin" and the password is "adminpass". According to our specifications, the code in Figure 1.2 should only return true if `untrustedname` and `untrustedpass` each match the credentials on some row in the database. A malicious attacker attempts to authenticate to the application with username `‘‘admin’’` and password `‘‘ ’ OR‘1’=‘1’’`. This results in the query `String query = KEYVALUNAME + ‘‘ = ‘’’ + admin + ‘‘ ’ AND ’’ + KEYVALPASS + ‘‘ = ‘’’ + ’OR‘1’=‘1 + ‘‘’ ’’`.

Our example database should reject this authentication attempt. The values passed into the `.authenticate()` method by the attacker do not match the values for the user account stored in the database.

This application code does not reject the authentication attempt as desired. The username is matched as normal but the password provided is not tokenized as a string literal. Instead, the first single quote in the untrusted input closes the string literal that should have contained the password to match against the password field in the database. The untrusted input then injects an extra `OR` token and an equality expression `‘1’=‘1`. Recall that SQLite also requires that literals be enclosed with single quotes. The single quote to the right hand side of the final number ‘1’ is left absent in the user provided string because the trusted query string has one already—the single quote that would have closed the password string literal. This injection results in the SQL query matching all rows where the column `KEYVALUNAME` is equal to "admin" and rows where the column `KEYVALPASS` is equal to "_" *or* where "`1 = 1`". The equation "`1 = 1`" is, of course, a tautology, resulting in

```
// Construct the query string marking user input locations with ?'s.
String query = KEYVALLOGIN + '' =? ''+ ''AND'' + KEYVALPASS + ''=?'';
// Pass to the query method the table to search,
Cursor cursor = db.query(TABLEVALDEMO,
// the columns to return, the query string
new String[] { KEYVALID, KEYVALLOGIN, KEYVALPASS }, query,
// and String array containing untrusted input to be bound as literals
new String[] { untrusteduname, untrustedpass}, null, null, null, null);
```

Figure 1.3: Code for an application that uses Android's SQL injection security mechanism

the query matching every row in the database. A row is returned, so the conditional at the end of the code returns true and the malicious user successfully authenticates to the database, despite not providing a valid username/password combination.

The injection attack described above was made possible by a poorly programmed application. The code of an equivalent application that takes advantage of Android's protection mechanisms would look like the code in Figure 1.3. This application code makes use of a prepared statement. The programmer can use a prepared statement by marking areas where untrusted input will be used in the query with '?' characters. The example code does this for both the `untrusuteduname` and `untrustedpass` strings. Then, when the SQLite API method is called with our query string, the fourth argument is used to pass in a `String` array containing the untrusted strings. It is then the task of the API method to insert the untrusted strings into the query and ensure that they are bound as literals, meaning that the untrusted strings will be tokenized as literals only. The new secure application, instead of searching for rows for which either the password value is the empty string or the equation "1 = 1" is true, will now search for rows for which the password value is `' OR '1'='1`. Since there is no row with that password in our example database, the query returns no rows, the conditional statement correctly returns false, and the injection attack is prevented. This mechanism for preventing SQL injection attacks is present in the off-the-shelf Android OS but is not automatic for all databases. The onus is on the programmer to understand why these overloads are important and to use them properly in their code to secure their application against injection attacks.

So far as we have been able to determine, there is no reason why an application developer would not want to use the provided SQLite security mechanisms, if s/he were aware of them. If there is no penalty for using the protection mechanism, why is SQL injection still a problem? The documentation for the SQLiteDatabase class describes the argument used to implement prepared statements as "*You may include ?'s in selection, which will be replaced by the values from selectionArgs, in order that they appear in the selection. The values will be bound as Strings.* [7]" This explanation makes no mention of what string binding is or why it is important to protect the application from injection attacks. The lack of detail might lead a developer unaware of the dangers of injection attacks to forego using the security mechanism, leaving their code vulnerable to injection attacks. In addition, prepared statements have been a feature of the Android OS since API level one—the initial release of Android in 2008. Despite the existence of these prepared statements, SQL injection continues to be a problem. Clearly, many developers are not understanding or simply neglecting to use these security features.

1.3 Summary of Contributions

This thesis has been written on the results of a collaborative research project. It is important to note which parts of this thesis are my work alone, which parts are the work of my peers, and which parts are a collaborative effort.

I am exclusively responsible for the taint tracking module of the protection mechanism and the conception of the original idea of the ping drain and card wipe exploits.

I worked jointly with Clayton Whitelaw on the detection portion of the protection mechanism and on the survey of existing Android applications.

I worked jointly with Hebron George, Ishan Mitra, and Clayton Whitelaw on the survey of the capabilities of the Android shell.

The original idea for this research into injection atttacks on Android was proposed by Jay Ligatti and Donald Ray. The formal definition of the NIE property on which this work is based is attributed to them as well [9].

1.4 Thesis Outline

This thesis makes two high-level contributions. First, we analyze the effectiveness of injection attacks on Android OS. Included in this analysis is a confirmation that such attacks are possible on the on both the OS shell and SQLite database, through the APIs provided by Android. We also demonstrate that these attacks have the capacity to compromise the target device and discuss which types of attacks may be executed depending on which permissions the target application has obtained. Finally, we will determine whether such vulnerabilities are present in applications currently released on Google Play and whether application developers are likely to include these vulnerabilities in their applications.

Our second contribution is the development and implementation of an extension to the Java libraries used by Android OS that attempts to prevent injection attacks. Any mechanism that attempts to precisely defend against injection attacks must precisely define an injection attack. This paper will utilize the definition developed by Ray and Ligatti [9, 10], which stipulates that for all applications that generate output programs based on untrusted input, the untrusted input must only cause the insertion or expansion of noncode tokens. Our extension to the Android OS libraries is itself in two parts. The first part consists of a proof of concept taint tracking module, which monitors the flow of untrusted input through an application's execution. The second part is a detection module, which contains an implementation of the detection algorithm described in Ray and Ligatti's work [9]. Components of this implementation include a classification of SQL and OS shell tokens into code and noncode, implementations of SQLite and OS shell lexers, and an implementation of the detection algorithm itself. Finally, we present data on the performance overhead of our implementation and its effectiveness in preventing code injection.

This thesis is organized as follows:

- Chapter 2 discusses previous work on Android security, taint tracking, and the definition of and defense against code injection attacks,
- Chapter 3 details the implementation of SQLite and the OS shell in Android. We cover how these two applications are accessed, what protections exist for them, how injection into them

occurs, and why these protections might be ignored by an application developer. We also demonstrate several example attacks on these applications.

- Chapter 4 discusses the formal definition of a BroNIE, the defintion of an injection attack used by this thesis.
- Chapter 5 covers the development of our prevention mechanism, detailing the implementation methods and demonstrating its effectiveness.
- Chapter 6 discusses our results, summarizes runtime overhead data, and concludes.

CHAPTER 2

RELATED WORK

This chapter covers other research into topics related to this thesis. Although not rigorously studied, work has been done on the security of the Android OS, with a large focus on its mechanism of granting permissions to applications at install time. Previous work in the area of formalizing a definition of injection attacks, another focus of this thesis, is cited and compiled here but discussed in full detail in Chapter 4. Taint tracking as a general topic has received a significant amount of study, due to its usefulness in defense against injection attacks. Research into taint tracking on Android has been focused more on its ability to implement runtime data flow policies, a feature not present in the off-the-shelf Android operating system. These runtime dataflow polices are useful to monitor how sensitive user data is used by an application.

2.1 General Android Security

Analysis and discussion of existing Andriod security mechanisms, including their implementation, has been discussed in [11]. Android creates a unique user id for each new installed application. This user id is granted permissions in the system on creation based on the application manifest, an XML file contained in the installation package of the application. The permissions specified at install time determine the capabilities the application has at runtime.

The install time permissions system relies on the "reputation" of an application. Decisions by the user to grant permissions to an application must be made before the application is ever used. Once an application is installed, it may use the permissions given in any way it desires. There are no protections for how the permissions are used—only whether they are usable. The lack of runtime permissions and protections could lead to user data being used by some application in a way that the user might wish to prevent. For example, an application might purport to only send

anonymized user location data to their own database but is in reality sending user-tagged data to advertisers.

Previous works on exploits and vulnerabilities on Android [12] have studied phishing attacks and privilege escalation. Privilege escalation is a technique also used in applications for "rooting" a mobile device. Rooting a mobile device is the process of circumventing system protections to obtain root user privileges on the device. Rooting is usually used to install modifications to the default operating system, or to install an entirely new operating system.

Privilege escalation can also be achieved by using multiple applications communicating, as discussed in [13]. If some application obtains the internet access permission from the system, it can then transmit the data it obtains from the internet via one of several communication channels to another application that does not have the internet permission. The first application functions as a medium, passing information to and from the internet and the second application. In this way, two applications can both use the permission to access some resource, but the user is only notified that one of the two is using the resource.

Research has been done on statically analyzing applications from the store, enabling users to determine before installing some appliction whether it is malware [14, 15]. Malicious applications are one of the primary concerns in Android security, given that application data usage is not monitored at runtime in off-the-shelf Android systems. Being able to detect malicious applications statically would allow for warnings to be posted on suspected malicious applications in the store, to warn users.

2.2 Theory of Injection Attacks

A significant body of work has been done on a precise and formal definition of injection attacks. It is critical, in order to develop a adequate prevention mechanism, to define when these attacks occur [16, 17, 18, 19, 20]. Previous works exhibit false positives and false negatives when faced with programs that inject only noncode tokens through tainted input. This thesis uses the NIE property [9, 10], which states that applications that generate output programs may not allow anything other than the insertion or expansion of noncode tokens from untrusted input in their

output programs. This definition avoids the drawbacks of previous work. A more detailed definition of a BroNIE will be discussed in Chapter four.

2.3 Taint Tracking

Taint tracking is the technique of marking data from some untrusted source with taint information and propagating that taint information though the execution of the program. Programs may then make policy enforcement decisions based on whether data is marked as untrusted. This information is especially useful for preventing injection attacks. The NIE property uses the idea of untrusted input in its definition to determine when injected symbols cause bad behavior in output programs.

Previous work on tracking tainted inputs has been done on a variety of platforms including Java applications [21, 22, 23] and Javascript [24]. Web Servers running Java frequently interface with SQL databases, making providing mechanisms designed to prevent SQL injection with accurate taint information a valuable research target.

Diglossia, a prevention mechanism for PHP requests [22], uses shadow characters to propogate taint information. Shadow characters are stored separately from ordinary characters and are not able to be used in the remainder of the program. At the point where some output program is generated, two distinct parsers are executed on the program string and the shadow of the program string. One parser accepts only ordinary program characters and the other accepts program characters and shadow characters. Injections are detected by comparing the parse tree results of the dual parsers and searching for differences.

There has also been platform-independent work on taint tracking. One paper focuses instead on the enforcement of some arbitrary policy [25], in addition to discussion of the "privilege hijack" attack. A privilege hijack attack can be said to occur when some program with legitimate privileges granted by the system is exploited, allowing the attacker privileges on the system that they would not ordinarily have through the program that they now control. Another paper examines syntax embedding [26], a technique that allows some source language to generate a program in some target language, with the source language aware of the syntax of the target language. This awareness allows the source language to take measures against injection attacks.

Taint tracking on Android OS has also been studied [27, 28, 29, 30, 31]. Challenges in taint tracking include the tradeoffs between the overhead of the system and its accuracy. Tracking of taint through all program variables carries a high performance cost. Compromises can be made to track only strings and characters, to reduce the load on the system. Much of the work into taint tracking on Android has been done for the purpose of extending Android's security mechanisms, which are restricted to permissions at install time for off-the-shelf Android. Marking and tracking the flow of private data through untrusted applications would allow Android to enforce policies about how the data and services are used. For instance, some policy enforcer might instrument method calls that perform network activity. If marked private data, such as contact databases, call histories, user images, or the contents of the SD card, are passed to the internet through one of these methods, an exception is thrown, and the transfer does not succeed. Concerns about mobile devices being increasingly used to store private data have given focus to this work.

CHAPTER 3

ATTACK SURFACES

In this chapter, we analyze the injection attacks we have discovered on Android OS. We analyze how each entry point may be exploited by an attacker and provide examples of exploits that might be used on vulnerable applications in the real world.

3.1 SQLite

A SQLite API is included in the Android SDK, for use by application developers. This API creates a local database for the application, as one of several options available to the developer for data storage. No declared manifest permission is required to use this database API. Data is inserted into and retrieved from the database by way of wrapper methods. There are wrapper methods for the insertion of rows through `.insert()`, deletion of rows through `.delete()`, updating of rows through `.update()`, and querying the database through `.query()` and `.rawQuery()`. A final wrapper method, `.execSQL()`, covers the execution of any commands not covered by the first four, such as the dropping of entire tables. All five of the SQLite methods carry overloads that make use of prepared statements [8]. The overloads take an additional array of strings as an argument. These strings will be bound as values before being passed into the SQL command, forcing them to be tokenized as string literals. Since the untrusted input can only be a string literal, an attacker cannot use input passed to a wrapper method in this fashion to inject code into a SQLite query.

However, as discussed in the introduction, using this protection mechanism is not an automatic feature of the SQLite implementation in Android. The programmer must go out of their way to use the appropriate method overload, and they must organize their query string with question marks in the appropriate places. Our mechanism for preventing injection attacks into SQLite is automatic, not requiring knowledge or intervention from the programmer, an advantage over off-the-shelf Android.

The vulnerability into SQLite, if exhibited by applications on the Google Play store, could leak sensitive data to an attacker from the database. To determine whether or not is it present in the real world, we conducted a study on the top 78 downloaded applications on the Google Play store for the vulnerabilities we have described. To perform our analysis, we first used "APKTool" [32], software that can decode Android applications from .apk format into Dalvik Virtual Machine bytecode. After obtaining this bytecode, we can use tools like grep, the Linux file search utility, to search through it and find potentially unsafe method invocations. Specifically, we searched for invocations of `.execSQL()`, `.rawQuery()`, and `.query()` methods that have passed "null" as their prepared statement argument. The lack of a safety argument suggests that if unsafe input is being used in the query, it is not being concatenated into the query properly. Our search returned 35 source files that exhibited this unsafe behavior while using the `.rawQuery()` method, and 55 source files that used `.query()` and `.execSQL()`. Our results suggest that some application developers are not aware of, or choose not to use, the protection mechanisms available to them through the SQLite API. We attempted to search through these potentially vulnerable applications and locate a concrete example of a SQL injection attack onto an application on the store. Lack of a static dataflow analysis tool and the fact that the majority of the applications were obfuscated made it necessary to inspect the code by hand. In our manual inspection, we were only able to find instances where trusted strings were passed to an unsafe query. As a result, we were unable to verify the existence of any such vulnerable application in our search.

3.2 OS Shell

To access the OS shell, an application may use the `Runtime` class as in Figure 1.1, which represents the computer in which the Java VM is running. Applications may obtain an instance of this `Runtime` class and then create processes running on the parent computer by calling the `.exec()` [33]. These processes can be any program accessible from the command line, including `ls`, `sh`, `rm`, and so on. The `.exec()` method may be called on either a string array or a simple string. In the case of the string array, the first element of the array is used as the program name to be executed and any remaining elements as the arguments to that program. In the case of the plain string, the `.exec()` method splits the input string by its whitespace and parses it into a

string array. The rules for execution then proceed the same as if the programmer had passed in a string array to begin with. For example, the string `rm -rf datadirectory` would be parsed into a command to execute the program `rm`, with `-rf` as the first argument, and `datadirectory` as the second.

A second class is also available to create operating system processes—the `Process Builder` class [34]. `Process Builder` separates the setting of the command and the starting of the process into the `.command()` and `.start()` methods respectively. Commands may be passed as single strings or a list of strings to the command method. The parsing of a plain string into a list of strings is also implemented with a whitespace split, the same algorithm used in the `.exec()` method. The programmer may then call the `.start()` method to create the process and begin execution.

The functionality of these two classes raises an important point. These classes do not themselves connect directly to the shell of the Android OS. Commands into these method calls are not parsed as shell programs. Only a simple program lookup is performed on the first string in the command array. The remainder of the command is passed directly to the program being executed as arguments. Due to this, injection attacks into the shell such as the one demonstrated in Figure 1.1 are not possible into these methods. Only if an attacker is given control over the full program string can they choose which program is being executed. If given control over only a single program argument in a string array, the attacker would only be able to modify that single argument. If the attacker's input were to be concatenated into a command string, the attacker would be able to insert additional program arguments. For example, if the command passed to `.exec()` or `.command()` was "`‘‘ping -c 4’’ + untrustedinput`", the attacker could input `-i .2 www.nameofhost.com `, resulting in the following string: `ping -c 4 -i .2 www.nameofhost.com`. Because these methods split the command string by whitespace, the attacker will have added two additional arguments to the program, `-i` and `.2`. This would decrease the interval between sending packets to one fifth of a second.

In order to perform a full code injection into the OS shell of an Android device, an application on the system must first obtain a `Process` that is running a full shell, as in example 1.1. The example application has a full shell running inside the `Process` 'proc', and our research indicates that

commands passed to this shell will be parsed and executed as shell programs, allowing injections like the one discussed in the introduction to be performed.

Having identified this OS shell vulnerability, we next examine whether this vulnerability is likely to exist in practice. First, consider whether an application developer would even need to use the Android shell for any functionality. It is possible that all of the OS programs accessible through the shell are implemented in the Android SDK. However, our research indicates that this is not the case. The Android SDK does not implement a `ping` program. An application developer could perhaps write their own `ping` program using the available socket tools native to Java but no ping program exists off the shelf in Android, except through the native ping program accessible through the shell. Application developers can also use the shell to determine if the device their application is being run on is "rooted". Application developers might also be more comfortable performing certain functionality in a familiar environment to them. Those more comfortable with shell scripting than with the Android SDK might choose to obtain a shell over working with the Android SDK.

Now that we know that an application developer has reason to access the OS shell, we must examine the benefits to the application developer of obtaining a full shell to execute these programs, instead of simply using the `.exec` or `.command()`. Recall that only the full shell is vulnerable to the attacks we have discovered. The most obvious potential benefit is processing overhead. If the developer decides to obtain a full shell and execute a series of OS programs using it, the overhead of calling the `.exec()` method is removed with the exception of the first `.exec()` call needed to obtain the full shell. The developer can then use the `Process` containing the shell to execute commands without the need for additional calls to `.exec()`.

To determine exactly how much processing time is saved, we performed runtime analysis on a test application that we created. The code that performs the testing is shown in Figure 3.1. The test application uses the `System` class, and its static method `.nanoTime()` to time how long it takes to perform an amount of shell tasks specified by the user, both through obtaining a full shell and through using calls to the `.exec()` method for each task. One call to `.nanoTime()` is performed before the tasks begin and the value is stored. When the tasks are complete, another call is made. The difference between these two values is the time, in nanoseconds, that elapsed during the test.

The data we collected is compiled in the graph in Figure 3.2. The number of milliseconds to complete the tasks is shown on the y axis and the number of tasks to be completed is shown on the x axis. It's obvious that using the interactive shell, shown with the solid line, has an execution time that grows very slowly with the number of tasks to be executed. Using individual `.exec()` method calls, shown by the dotted line, grows in execution time far more quickly in comparison. The inconsistency in the data for task numbers 18, 19, and 20 for the individual `.exec()` method calls could be the result of a process swap in the middle of the test. The process swap would stop the shell tasks from executing for some time, while the system time value would continue to increase. This time spent doing no work would lead to longer execution times than in the rest of our tests. Using individual `.exec()` method calls is only faster than using an interactive shell for the execution of a single task and it is only faster by 11 milliseconds. This graph demonstrates a clear motivation for an application developer to use a full shell to perform tasks in their application—it has the potential to save a significant amount of processing time if many OS tasks are being run over the execution of the application.

In a continuation of our work analyzing applications for SQLite vulnerabilities, we decompiled and analyzed the same set of 78 top applications on Google Play for shell vulnerabilities. Using the the same bytecode search methods with APKTool, we searched for invocations of the `.exec` or `.start` methods. In the majority of applications, we found that the `.exec` method call was used at some point in the code, indicating that application developers are aware of its existence and have found it to be useful. Several applications did obtain a full shell but passed only constant strings as programs into the shell. The potential for a vulnerability exists in the shell but we have been unable to verify an application that exhibits this vulnerability among the top 78 applications on Google Play.

3.3 Possible Attacks into OS Shell

With our analysis of the shell vulnerability complete, we now move on to a survey of what functionality is available through the shell and what behaviors exist that a malicious attacker could abuse. Recall that the Android OS uses an install-time permission system. This system is implemented by each distinct application having its own user id, with the permissions the application has

```
starttime = System.nanoTime(); // Start the clock for the interactive shell test
p = r.exec(new String[] {"sh", "-i"} ); // Start the interactive shell

DataOutputStream processInput =
  newDataOutputStream(p.getOutputStream()); // Get an a stream to put commands into the process

for(i = 0; i < count; i++) // Execute ''count'' number of tasks
{
  processInput.writeBytes("ls\n");
  processInput.flush();
}

processInput.writeBytes("exit\n"); // Wait for the process to exit
processInput.flush();
try {
  p.waitFor();
  endtime = System.nanoTime(); // Stop the clock for the interactive shell test
  p.destroy();
} catch (InterruptedException e) {
  e.printStackTrace();
}

starttime = System.nanoTime(); // Start the clock for the .exec method
for(i = 0; i < count; i++) // Execute ''count'' number of tasks
{
  p = r.exec("ls");
  try {
        p.waitFor();
        p.destroy();
  } catch (InterruptedException e) {
  e.printStackTrace();
}
endtime = System.nanoTime(); // Stop the clock for the .exec method
```

Figure 3.1: Code for our timing test application

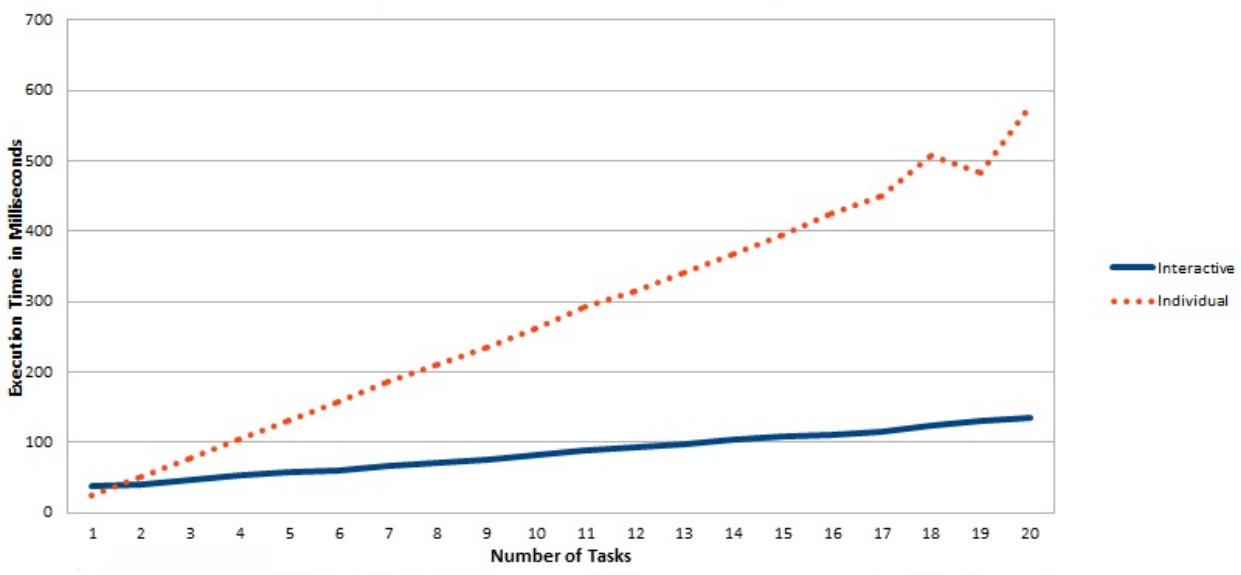

Figure 3.2: A graph showing timing data gathered from the test application.

requested attached to that id. Our research indicates that when an application obtains a full shell, that shell operates with the same permissions as that user id. The permissions of the application to be injected into affect what exploits an attacker injecting into the application has access to.

We have discovered three major exploits made possible by this vulnerability.

1. Ping Drain (Internet Permission Required) – Mobile devices can connect to the internet through a Wi-Fi connection, or though the mobile data network, where data is transmitted through cell base towers. Most cell providers have limitations and caps for "data" access to their network. An attacker can use the `ping` program, with a maximum size packet (65535 bytes) at the shortest interval (5/sec) to use 3MB/sec of data. If running constantly, this program could very quickly accrue a large amount of mobile data usage, leading to the user being assessed an excessive mobile phone bill, or simply being unable to use their mobile network data access for the remainder of the billing period.

2. Card Wipe (External Storage Permission Required) – The external storage permission gives an application read and modify permissions for the entire external storage. External storage for an Android device usually takes the form of an SD card. If an application with this permission were to be injected into, the attacker would have access to all data mounted on the SD card's filesystem. The attacker can read private data or delete some or all of the contents of the SD card.

3. Fork Bomb (No Permissions Required) – Android applications are restricted in the number of processes they may create but very loosely. A normal app may create up to 6656 processes. This number is more than enough processes to execute a resource denial attack by way of a forkbomb and lock up the target device.

We believe that these exploits into the OS shell and the lack of automatic SQL injection prevention are a significant enough concern to warrant implementing a security mechanism to mitigate these injection attacks on Android.

CHAPTER 4

A REVIEW OF BRONIES

To begin our efforts to implement a mechanism to defend against injection attacks, it is vital to decide on a formal definition of attacks. Without knowledge of what the attacks are at a low level, any protection mechanism is bound to have flaws. Our protection mechanism is based on Ray and Ligatti's definition of a BroNIE [9].

A BroNIE begins, at a high level, when some output program is generated by using tainted input. If the addition of this tainted input causes anything besides the expansion or insertion of noncode tokens in the target programming language, a BroNIE has occurred. To define the formal details of a BroNIE, Ray and Ligatti have constructed multiple subdefinitions. We review them in the following sections and demonstrate that this definition correctly identifies the injection attacks we have described earlier in this thesis.

4.1 Template String

In order to determine whether a token has been expanded or inserted by the addition of tainted input, we must first examine the original output program string, before the tainted input was added. To do this, we determine the string template. The template string is the same as the output string, but all tainted characters have been replaced with the 'ε' character, to represent the empty string. The only purpose of the 'ε' character in the template is to act as a placeholder in the program string, representing a location where an injected symbol once was. They are otherwise ignored by the tokenizer. For example, the template of the string `ping ‘‘www.hostname.com’’` is `ping ‘‘www.εεεεεεεε.com’’`.

4.2 Token Expansion

Token expansion in the BroNIE definition describes how tainted input is permitted to enlarge existing tokens. Tokens are defined in terms of what kind of token they are, the semantic value of the token, and the range of indices over which the token appears in the program string. We represent tokens in this thesis with the form $\tau_i(tokensymbols)_j$, representing a token of type τ, composed of symbols "tokensymbols' and ranging from character index i to character index j in the program string. For example, the string "2 + 10 * 3" would be tokenized as $\{\text{INT}_1(2)_1$, $\text{PLUS}_3(+)_3$, $\text{INT}_5(10)_6$, $\text{MULT}_8(*)_8$, $\text{INT}_{10}(3)_{10}\}$.

A token τ can be said to expand another token τ' if and only if:

1. The token types of the two tokens are equal.

2. The character indices of token τ are equal to, or an expansion of, the character indices of token τ'.

3. The semantic value of token τ' is a subsequence of the semantic value of token τ.

As an example, consider the string `ping ‘‘www.hostname.com’’` again. Recall that the template of this string is `ping ‘‘www.εεεεεεεε.com’’`. The token for the string literal representing the hostname in the template would be thus: $\text{LITERAL}_6(\text{www..com})_{21}$. The token for the string literal in the original program would be $\text{LITERAL}_6(\text{www.hostname.com})_{21}$. Note that the epsilon characters preserve the character indices of the template string's token, despite the tainted input being removed. By this definition of token expansion, the original program token expands the template string's token.

4.3 Token Classification

To detect BroNIEs for some output programming language, we must first partition the tokens of the target language into code and noncode tokens. Code tokens are reducible by evaluation, meaning that computation can be performed on them at runtime. A code token might be the `ping` operator. This token specifies a program to be executed, obviously necessitating some computation. Noncode tokens are the opposite; they are already values, so no computation can be performed on them.

An example noncode token might be the `String` token with semantic value "www.hostname.com". This is a `String` value and cannot be evaluated any further than it has been already.

Again considering our example of `ping ''www.hostname.com"`, shell strings can all be classified under the token WORD. WORD tokens can themselves be split into 3 different token types: code, open value, and literal. Of these three types, only the literal type is noncode. This string could thus be fully tokenized as {$\text{CODEWORD}_1(\text{ping})_4$, $\text{LITERAL}_6(\text{www.hostname.com})_{21}$}.

4.4 The NIE Property

Finally, we can formally define the NIE property. Given some program string s and the template t(s), the NIE property holds if and only if it is possible to create s from t(s) by inserting or expanding noncode tokens. If this cannot be achieved, we say that the output program exhibits a BroNIE. Continuing with our example from the rest of the chapter, we examine the string `ping www.hostname.com` and its template `ping www.`$\underline{\varepsilon\varepsilon\varepsilon\varepsilon\varepsilon\varepsilon\varepsilon\varepsilon}$`.com`. As discussed in the previous example, the LITERAL token in the output program is an expansion of the LITERAL token in the template program, and the CODEWORD token is unchanged in the template string. Thus, we can conclude that this example does not exhibit a BroNIE.

4.5 Stopping a BroNIE

Let us now examine the example injection attacks into Android that we discussed in the introduction and determine whether the definitions discussed in this chapter classify them as BroNIE's. Consider the first example shell injection program from the introduction: `ping -c 4 ''www.remotehostname.com'' && rm -rf ''data'' \n`. As before, tainted input is marked by underlining. First, we create a template of this string, as follows: `ping -c 4 ''`$\underline{\varepsilon}$`'' \n`. The original program string is tokenized as {$\text{CODEWORD}_1(\text{ping})_4$, $\text{ARGUMENT}_6(\text{-c})_7$, $\text{LITERAL}_9(4)_9$, $\text{LITERAL}_{11}(\text{“www.hostname.com”})_{26}$, $\&\&_{28}(\&\&)_{29}$, $\text{CODEWORD}_{31}(\text{rm})_{32}$, $\text{ARGUMENT}_{34}(\text{-rf})_{36}$, $\text{LITERAL}_{38}(\text{“data”})_{45}$, $\text{NEWLINE}_{46}(\backslash\text{n})_{47}$}. The template string can be tokenized as: {$\text{CODEWORD}_1(\text{ping})_4$, $\text{ARGUMENT}_6(\text{-c})_7$, $\text{LITERAL}_9(4)_9$, $\text{LITERAL}_{11}(\text{“”})_{45}$, $\text{NEWLINE}_{46}(\backslash\text{n})_{47}$}. In order to convert from the template token stream to the original token stream, we will at the very least need to add the token $\&\&_{28}(\&\&)_{29}$.

&& is a code token—it can be reduced by evaluating the expressions on either side of it. Since adding code tokens is against the NIE property, this program exhibits a BroNIE. In similar fashion, the other shell program discussed in the introduction: `ping -c 4 ‘‘www.remotehostname.com’’ && f() { f|f& } && f \n #’’ \n’’` injects an && token and the template string will be unable to add it back to restore itself to the program string in a way that satisfies the NIE property. This program too, exhibits a BroNIE.

Finally, we examine our SQLite example string from the introduction, shown here as a query string, not a snippet of Java code: `SELECT FROM ‘accounts’ WHERE KEYVALUNAME= ‘admin’ AND KEYVALPASS=‘’ OR ‘1’=‘1’`. This string's template is as follows: `SELECT FROM accounts WHERE KEYVALUNAME=‘εεεεε’ AND KEYVALPASS=‘εεεεεεεεεεεε’` The program string's tainted input areas will be tokenized as: {STRING_{43}(‘admin’)$_{49}$} and {STRING_{67}(’)$_{67}$, OR_{68}(OR)$_{69}$, STRING_{71}(‘1’)$_{73}$, EQUALS_{74}(=)$_{74}$, STRING_{75}(‘1)$_{76}$}. The same tainted areas in the template string will be tokenized as: {STRING_{43}(‘’)$_{49}$} and {STRING_{67},(‘’)$_{76}$}. Clearly, there are many code tokens present in the program string that do not appear in the template string. This program too, exhibits a BroNIE. All of our example attacks from the introduction have now been successfully detected by the NIE property and its associated definitions.

CHAPTER 5

PROTECTION MECHANISM IMPLEMENTATION

Having explored the surfaces through which injection attacks can occur on Android and defined formally how injection attacks occur, we can now implement a mechanism to defend against these attacks. There are two high-level components to this mechanism, the taint tracker and the detection algorithm. The taint tracker applies taint data to characters from untrusted sources and propagates that taint information in strings through the execution of the application. The detection algorithm will then use the taint information supplied by the taint tracker to generate the template string described in Chapter four. The detection algorithm compares the template string and the original string. If it finds that the original string cannot be reached from the template string by inserting or expanding noncode tokens, it throws an exception and the program or query is not executed [9].

5.1 Taint Tracker

In our modifications to the Java libraries used by Android OS, taint information is stored for each `String` in the system as a boolean array, with one bit for each character. A false bit appears for a specified character if and only if that character is untainted and a true if and only if the character is tainted. Taint information should be linked to each string on creation. An "untrusted" source would mark the full string created from it with taint bits.

It would be convenient to modify the `String` class directly and add our taint tracking information as a field to it. However, the Android OS performs some VM-level optimization with the `String` class. The bit-level offsets of the fields in the class are hardcoded into other sections of the OS, making adding fields to the `String` class difficult.

To circumvent this problem, we have implemented our taint information in a separate class, a new addition to the Java libraries. We call it the `TaintTrack` class. It is inserted into the `Java.lang` package, alongside the `String` class. Selections of code from the class are displayed in Figure 5.1.

The primary data structure used by the class is a `HashMap`, which stores taint information for a particular string keyed by the hash code of the string being stored. The class implements methods for adding untainted and tainted strings or character sequences to the `HashMap`. The character sequence adding method must first convert the character sequence to an instance of the `String` class. Both methods then allocate a boolean array of the size of the string and fill it with the appropriate taint data (false for untainted, true for tainted.) In Figure 5.1, only the methods for adding a tainted character sequence and an untainted string are shown. The methods for adding an untainted character sequence and a tainted string require only trivial alterations from the methods we have shown. There is a method for retrieving the taint information for a specific string, for which the code needs only access the `HashMap`. Finally, there is a method for updating taint information after two strings have been concatenated. For this method, a boolean array of the combined size of the strings is allocated, and the taint information for the two original strings is retrieved and combined together into the result array. The result array is then placed into the `HashMap`, under the hash code of the final concatenated string.

Now that we have added all of the necessary helper methods to track taints, our implementation must call them from the appropriate locations in the Java libraries and application code. To begin, we instrumented the `String` class to add all created strings as untainted by default using the `.addStringUntainted()` method from Figure 5.1. Tainted strings, in our example implementation, are obtained from a text box in the application's UI. The method for obtaining the text that a user has entered into a text box is `.getText()` from the `TextView` class. We have instrumented the `.getText()` method with the `.addCharSeqTainted()` method from Figure 5.1 to taint all strings created through it. In a full implementation of this mechanism, the developer might wish to specify other sources as tainted, and instrument those sources as well. For our example implementation, a single taint source is sufficient to demonstrate the efficacy of the system.

Next, we must call our string concatenation tracking method from the Java code. To do this, it is important to know how the Java compiler handles the "+" operator on two strings. It first instantiates an instance of the `StringBuilder` class, with the string on the left hand side as its data. Then, it calls the `.append()` method of the `StringBuilder` class, with the string on the right hand side as its argument. Finally, it builds the string, and returns the result. It would be simplest

```
//The data structure used to hold taint information
private static final HashMap<String, boolean[]> taintmap;

// Add some character sequence to the taint map as a tainted string
public static void addCharSeqTainted(CharSequence seq){
    // Convert the CharSquence to a string
    String input = seq.toString();
    // Allocate a taint array of the size of the string
    boolean[] taint = new boolean[input.length()];
    // This string is fully tainted, so the array is all tainted
    Arrays.fill(taint, true);
    // Put the array into the tracker
    taintmap.put(input, taint);}

public static void addStringUntainted(String input){
    // Allocate a taint array of the size of the string
    boolean[] taint = new boolean[input.length()];
    // This string is fully untainted, so the array is all untainted
    Arrays.fill(taint, false);
    // Put the array into the tracker
    taintmap.put(input, taint);}

// Retrieve taint data from the map for some string
public static boolean[] getDataForString(String input){
    return taintmap.get(input);}

// Propogate taint information through concatenation
public static void stringCat(String input1, String input2, String input){
    // Retrieve taint data from the two sources
    boolean[] taint1 = taintmap.get(input1);
    boolean[] taint2 = taintmap.get(input2);
    // Allocate a new taint array with the appropriate new size
    int len1 = taint1.length;
    int len2 = taint2.length;
    boolean[] taint = new boolean[len1 + len2];
    // Copy the taint information into the new array
    System.arraycopy(taint1, 0, taint, 0, len1);
    System.arraycopy(taint2, 0, taint, len1, len2);
    // Put the new taint information into the tracker
    taintmap.put(input, taint);}
```

Figure 5.1: Code for the taint tracking module we have added to the Java libraries

```
// Declare advice that runs around the target method, and returns StringBuilder
StringBuilder around(StringBuilder sb, String input):
    // This advice wraps around StringBuilder.append()
        call(StringBuilder StringBuilder.append(String))
        // Bind the calling object to variable ''sb''
        && target(sb)
        // Bind the first argument to variable ''input''
        && args(input){
                // Save the contents of the string builder before the concatenation
                String start = sb.toString();
                // Continue with execution, start this code again after done
                proceed(sb, input);
                // Save the contents of the string builder after the concatenation
                String end = sb.toString();

        // Call the taint tracker - abbreviates 42 lines in the full implementation
        callTaintTracker(start, input, end);
        // Return the stringbuilder, and continue with normal execution
                return sb;}
```

Figure 5.2: AspectJ code to call the taint tracker from application code

to instrument the `.append()` method of the `StringBuilder` class, but it also uses optimization at the VM level and we were unable to instrument it inside the source code for the Java libraries.

5.2 Weaving with AspectJ

To circumvent this problem, our implementation makes use of a tool called AspectJ, a programming language for weaving through source code. Weaving is the act of adding specific code snippets at specific points, declared by the programmer. To obtain the desired behavior, we programmed an AspectJ file to instrument the appropriate method calls. A code snippet from this file is displayed in Figure 5.2. The code begins by declaring a piece of advice, the class structure for AspectJ. It then specifies that this advice will weave "around" any calls to the `.append()` method. Recall from Figure 5.1 that our method to update taint information after concatenation requires the strings used for concatenating and the result of the concatenation. For this reason, we wrap fully around the call to `.append()` and not just before or after it. Before the `.append` call is executed, our code saves the values for the initial strings. After the call to `.append()`, it obtains the result string and calls the `stringCat()` method with all three strings.

The final component of the taint tracker is the point where an output program is being generated by the application. At the point of program generation, we again use AspectJ to call to the taint tracker to obtain the taint information for the string passed into the method that generates the

output program. The work of the taint tracker is currently complete and the rest is done by the detection algorithm.

5.3 Detecting BroNIE's

The detection algorithm is an implementation of the detection algorithm developed by Ray and Ligatti [9] and the implementation of their algorithm that we discuss in this thesis was primarily the work of Clayton Whitelaw. The algorithm begins with a program output string for some language and taint information for that string. The algorithm next generates the template string counterpart for the output string being used to generate the program. The algorithm does this by replacing all tainted characters with ε characters. The template string is then tokenized by a modified tokenizer for the target programming language. The modifications to the tokenizer allow it to ignore the ε characters in the template string. We then tokenize the original output program string in the same way, assuming that ε characters are not present in the original program output.

Now, we have 2 token streams, one from the template and one from the original. Now we must implement the BroNIE detection algorithm from [9]. The implementation of the detection algorithm is shown in Figure 5.3. The algorithm works by scanning through both token streams and comparing the tokens at each index. If the tokens are the exact same or a token has been inserted or expanded, but is noncode, the algorithm continues scanning the token streams. In any other case, a BroNIE exception is thrown. If the end of the template string is reached first, the program scans through any additional noncode tokens in the program string. Finally, it checks to see if all tokens have been checked in both token streams. If they have not, then there are either additional code tokens in the program string or there are tokens in the template string that are not in the program string. Both of these are outlawed by the definition of a BroNIE, so an exception is thrown. If the program passes all of these checks, the program is executed through the appropriate application.

The security mechanism is currently complete and should fully defend against BroNIEs on Android OS. To test the mechanism, we have attempted to execute the example injection attacks discussed in the introduction and Chapter four. All three program strings were correctly identi-

```
private static void check(String cmd) throws BronieException {
    // Retrieve taint information
    boolean[] taint = TaintTracker.getDataForString(cmd);
    // Tokenize the program string
    Tkn[] ori = tokenize(cmd);
    // Convert the program string into its template
    String template = getTemplateString(cmd);
    // Tokenize the template
    Tkn[] tem = tokenize(template);

    // Initialize indices for looping through the token streams
    int i = 0;
    int j = 0;

    while(i < ori.length && j < tem.length) {
        if(ori[i].equals(tem[i])) {
            // The tokens are the exact same, continue on
            i++;
            j++;
        } else if(ori[i].isNoncode() && tem[i].expandsTo(ori[i])){
            // The template token can expand to the original token, continue on
            i++;
            j++;
        } else if (ori[i].isNoncode()) {
            // A noncode token has been injected, continue on
            i++;
        } else {
                // Something else has happened, not allowed, this program exhibits a BroNIE
            throw new BronieException("BroNIE detected");
        }
    }

    while(i < ori.length && ori[i].isNoncode()) {
        // Additional noncode tokens were inserted, continue on
        i++;}

    if(!(i >= ori.length && j >= tem.length)) {
        System.err.println(i + "<= " + ori.length + ", " + j + "<= " + tem.length);
        // Not all tokens were scanned - this program exhibits a BroNIE
        throw new BronieException("BroNIE detected");
        }
}
```

Figure 5.3: Algorithm to detect BroNIEs using a template string and a program string

fied by the system as BroNIEs. Our proof-of-concept mechanism has been implemented and has demonstrated its effectiveness against the attacks we have described.

CHAPTER 6

CONCLUSION

This thesis set out to argue that injection attacks are both possible on Android and have the potential to compromise the target device. We have presented evidence of several injection attacks into Android that inject into both the OS shell and the SQLite API. The attacks we have demonstrated include the capability to falsely authenticate to some SQLite database, the ability to drain the mobile data allotment of the target device, the ability to modify or delete the contents of the SD card on a device, and the ability to "forkbomb" a device. There exists a mechanism to defend against SQL injection attacks on Android, but it is not automatic, relying on the programmer of the application for proper implementation. There is no mechanism to defend against OS shell attacks on Android. These attacks have the capacity to cause damage to users and we believe that they are a problem that warrants a solution.

In pursuit of that goal, we have implemented a defense mechanism to mitigate these injection attacks, based on Ray and Ligatti's BroNIE definition of injection attacks [9]. We have demonstrated that the definition of a BroNIE accurately classifies our example attacks as injection attacks. Our mechanism adds taint tracking via modification to the Java libraries used by the Andriod OS and via automated compilation of android applications with AspectJ. We have also developed an implementation of the detection algorithm discussed in Ray and Ligatti's work in AspectJ to detect injection attacks based on the taint information provided by the taint tracker.

LIST OF REFERENCES

[1] The MITRE Corporation. *CWE/SANS Top 25 Most Dangerous Software Errors*, 2011. http://cwe.mitre.org/top25/archive/2011/2011_cwe_sans_top25.pdf.

[2] Pew Internet. Mobile technology fact sheet, 2014. http://www.pewinternet.org/fact-sheets/mobile-technology-fact-sheet.

[3] CISCO. Data leakage worldwide: The high cost of insider threats, 2008. http://www.cisco.com/c/en/us/solutions/collateral/enterprise-networks/data-loss-prevention/white_paper_c11-506224.pdf.

[4] Google. Android open source project, 2014. https://source.android.com.

[5] A. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS, pages 627–638, New York, NY, USA, 2011. http://doi.acm.org/10.1145/2046707.2046779.

[6] P. Kelley, S. Consolvo, L. Cranor, J. Jung, N. Sadeh, and D. Wetherall. A conundrum of permissions: Installing applications on an android smartphone. In *Financial Cryptography and Data Security*, volume 7398 of *Lecture Notes in Computer Science*, pages 68–79. Springer Berlin Heidelberg, 2012. http://dx.doi.org/10.1007/978-3-642-34638-5_6.

[7] Google. Sqlitedatabase class documentation, 2014. http://developer.android.com/reference/android/database/sqlite/SQLiteDatabase.html.

[8] Open Web Application Security Project. *SQL Injection Prevention Cheat Sheet*, 2014. https://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet.

[9] D. Ray and J. Ligatti. Defining injection attacks. Technical report, University of South Florida, 2014. http://www.cse.usf.edu/~ligatti/papers/bronies.pdf.

[10] D. Ray and J. Ligatti. Defining code-injection attacks. In *Proceedings of the Symposium on Principles of Programming Languages*, POPL, pages 179–190, 2012.

[11] W. Enck, M. Ongtang, P.D. McDaniel, et al. Understanding android security. *IEEE Security & Privacy*, 7(1):50–57, 2009.

[12] T. Vidas, D. Votipka, and N. Christin. All your droid are belong to us: A survey of current android attacks. In *Women Of Our Time*, WOOT, pages 81–90, 2011.

[13] C. Orthacker, P. Teufl, S. Kraxberger, G. Lackner, M. Gissing, A. Marsalek, J. Leibetseder, and O. Prevenhueber. Android security permissions can we trust them? In *Security and Privacy in Mobile Information and Communication Systems*, volume 94 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 40–51. Springer Berlin Heidelberg, 2012. http://dx.doi.org/10.1007/978-3-642-30244-2_4.

[14] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In *NDSS*, 2012.

[15] F. Di Cerbo, A. Girardello, F. Michahelles, and S. Voronkova. Detection of malicious applications on android os. In *Computational Forensics*, pages 138–149. Springer Berlin Heidelberg, 2011.

[16] P. Bisht, P. Madhusudan, and V. N. Venkatakrishnan. Candid: Dynamic candidate evaluations for automatic prevention of SQL injection attacks. *ACM Trans. Inf. Syst. Secur.*, 13(2):14:1–14:39, March 2010. http://doi.acm.org/10.1145/1698750.1698754.

[17] Z. Su and G. Wassermann. The essence of command injection attacks in web applications. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL, pages 372–382, New York, NY, USA, 2006. http://doi.acm.org/10.1145/1111037.1111070.

[18] G. Wassermann and Z. Su. Sound and precise analysis of web applications for injection vulnerabilities. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, pages 32–41, 2007. http://doi.acm.org/10.1145/1250734.1250739.

[19] Oracle. How to write SQL injection proof PL/SQL, 2008. http://www.oracle.com/technetwork/database/features/plsql/overview/how-to-write-injection-proof-plsql-1-129572.pdf.

[20] G.-V. Jourdan. Securing large applications against command injections. In *Security Technology, 41st Annual IEEE International Carnahan Conference on*, pages 69–78, 2007.

[21] W.G.J. Halfond, A Orso, and P. Manolios. Wasp: Protecting web applications using positive tainting and syntax-aware evaluation. *Software Engineering, IEEE Transactions on*, 34(1):65–81, Jan 2008.

[22] S. Son, K. S. McKinley, and V. Shmatikov. Diglossia: detecting code injection attacks with precision and efficiency. In *Proceedings of the ACM SIGSAC conference on Computer & communications security*, CCS, pages 1181–1192, New York, NY, USA, 2013. http://doi.acm.org/10.1145/2508859.2516696.

[23] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically hardening web applications using precise tainting. In *Security and Privacy in the Age of Ubiquitous Computing, IFIP 20th International Conference on Information Security (SEC)*, pages 295–308, 2005.

[24] Florian N., Nenad J., Engin K., Christopher K., and Giovanni V. Cross-site scripting prevention with dynamic data tainting and static analysis. In *In Proceeding of the Network and Distributed System Security Symposium*, 2007.

[25] W. Xu, S. Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15*, USENIX-SS, Berkeley, CA, USA, 2006. http://dl.acm.org/citation.cfm?id=1267336.1267345.

[26] M. Bravenboer, E. Dolstra, and E. Visser. Preventing injection attacks with syntax embeddings. In *Proceedings of the 6th International Conference on Generative Programming and Component Engineering*, GPCE, pages 3–12, New York, NY, USA, 2007. http://doi.acm.org/10.1145/1289971.1289975.

[27] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: An information flow tracking system for real-time privacy monitoring on smartphones. *Commun. ACM*, 57(3):99–106, March 2014. http://doi.acm.org/10.1145/2494522.

[28] C. Gibler, J. Crussell, J. Erickson, and H. Chen. Androidleaks: Automatically detecting potential privacy leaks in android applications on a large scale. In *Trust and Trustworthy Computing*, volume 7344 of *Lecture Notes in Computer Science*, pages 291–307. ACM, 2012. http://dx.doi.org/10.1007/978-3-642-30921-2_17.

[29] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. These aren't the droids you're looking for: Retrofitting android to protect data from imperious applications. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS, pages 639–652, New York, NY, USA, 2011. http://doi.acm.org/10.1145/2046707.2046780.

[30] Y. Zhou, X. Zhang, X. Jiang, and V. Freeh. Taming information-stealing smartphone applications (on android). In *Trust and Trustworthy Computing*, volume 6740 of *Lecture Notes in Computer Science*, pages 93–107. ACM, 2011. http://dx.doi.org/10.1007/978-3-642-21599-5_7.

[31] G. Sarwar, O. Mehani, R. Boreli, and D. Kaafar. On the effectiveness of dynamic taint analysis for protecting against private information leaks on android-based devices. In Pierangela Samarati, editor, *SECRYPT, 10th International Conference on Security and Cryptography*, pages 461–467, Reykjvik, Iceland, July 2013.

[32] Google. Android-apktool, 2014. https://code.google.com/p/android-apktool/.

[33] Oracle. *Class Runtime Javadoc*, 1993. http://docs.oracle.com/javase/7/docs/api/java/lang/Runtime.html.

[34] Oracle. *Class Process Builder Javadoc*, 1993. http://docs.oracle.com/javase/7/docs/api/java/lang/ProcessBuilder.html.