11-6-2015

Precise Detection of Injection Attacks on Concrete Systems

Clayton Whitelaw
University of South Florida, cwhitela@mail.usf.edu

Precise Detection of Injection Attacks on Concrete Systems

by

Clayton Whitelaw

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 26, 2015

Keywords: Security Mechanisms, Formal Definitions, SQL, Android, Shellshock

DEDICATION

I dedicate this thesis to the educators and enthusiastic conveyers of scientific and mathematical ideas who have sparked my interest in learning, even in the most cynical times; to the musicians whose work engages me and inspires me to push boundaries; and to my close friends and family for their lifelong support and fellowship.

ACKNOWLEDGMENTS

I have improved more in these past two years than I ever thought possible. I would not have even entered the graduate program without having met by chance my advisor, Jay Ligatti. His realistic advice, high expectations, and pedagogical clarity have been invaluable resources to my growth as a computer scientist and engineer. I would also like to thank my fellow graduate students for providing a memorable and supportive community. In particular, I would like to thank Grant Smith for working with me on the initial stages of this project.

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

Injection attacks, including SQL injection, cross-site scripting, and operating system command injection, rank the top two entries in the MITRE Common Vulnerability Enumeration (CVE) [1]. Under this attack model, an application (e.g., a web application) uses some untrusted input to produce an output program (e.g., a SQL query). Applications may be vulnerable to injection attacks because the untrusted input may alter the output program in malicious ways. Recent work has established a rigorous definition of injection attacks. Injections are benign iff they obey the NIE property, which states that injected symbols strictly insert or expand noncode tokens in the output program. Noncode symbols are strictly those that are either removed by the tokenizer (e.g., insignificant whitespace) or span closed values in the output program language, and code symbols are all other symbols. This thesis demonstrates that such attacks are possible on applications for Android—a mobile device operating system—and Bash—a common Linux shell—and shows by construction that these attacks can be detected precisely. Specifically, this thesis examines the recent Shellshock attacks on Bash and shows how it widely differs from ordinary attacks, but can still be precisely detected by instrumenting the output program's runtime. The paper closes with a discussion of the lessons learned from this study and how best to overcome the practical challenges to precisely preventing these attacks in practice.

CHAPTER 1

INTRODUCTION

Injection attacks, including SQL injection, cross-site scripting, and operating system command injection, rank the top two most commonly reported vulnerabilities in the MITRE Common Vulnerability Enumeration (CVE) [1]. Injection vulnerabilities can often occur in web applications using a relational database for persistent storage of user credentials, user information, transaction records, etc. Typically, the application's interface to that database is the Structured Query Language (SQL) [2]. Consider such an applcation that takes a user's username and password from a web form and executes the following SQL query.

```
SELECT * FROM users WHERE user='cw' AND pw='banana'
```

The injected input from the web form is underlined. The authentication is successful iff a row is returned from the database, in which case that row includes the user's real name and other personal information for use during that session. The injected symbols in this case simply caused the string literals (both originally '') to be expanded to include the data from the form. Since a string performs no operations by itself—i.e., it is dynamically passive—the first output program is benign. However, output programs contain both code (instructions) and noncode (passive parameters to those instructions); the input fed into the web form may be maliciously formed to exploit this situation.

```
SELECT * FROM users WHERE user='' OR 1=1--' AND pw=''
```

The `--` symbols start a comment in the output program, causing the rest of the symbols to be ignored. The result is that the database returns all rows in the `USERS` table where either the username is empty or 1=1. Since 1=1 is a tautology, all rows are returned and attacker subverts the authentication mechanism. This is a code-injection attack (in this case known as SQL injection [1])

because the attacker injected code to change the query. When the output language is an operating system shell or similar external program, this is known as OS Command Injection [1].

The victim of such an attack need not be a web service. Browsers today typically support Javascript, as opposed to purely rendering HTML. Without proper validation, an attacker might register an account on this website with the name `<script>alert('injected')</script>`. Consider a web-based administration console for the website where each user's information can be viewed and edited. The administrator views the details of such an attacker using a browser. Since the name field forms an HTML script element, the browser could execute the script upon rendering the page, potentially with the same permissions as an administrator. While an alert window is relatively benign, the script could include web requests to compromise the website entirely (e.g. to wipe the database or create an administrator account for the attacker) or code to attack the victim's computer directly. This form of injection attack is known as cross-site scripting (XSS) [1].

Injected symbols may be malicious without being code. Consider the following slice of a web application [3].

```
$data = '\'; securityCheck(); $data .= '&f=exit#';\n f();
```

The application intends to assign untrusted input to the variable `data`, perform a security check, concatenate another string to `$data`, and call the `f` function. The injected backslash character \ is a valid string literal. However, in some languages during lexical analysis a backslash character may be used to escape the next symbol—i.e., force the next symbol to be considered noncode. The single quote following the backlash is escaped, so some of the code following the string became part of the string literal; the resulting program can be approximately represented as follows.

```
$data = '.....'&f=exit#';\n f();.
```

That is, only noncode was injected, but the injection caused tokens to be removed. The resulting program assigns the string (with braces as delimiters for clarity) {`'; securityCheck(); $data .=` } to `$data`, binds the function reference `f` to the `exit` function, and calls `f`, thereby calling `exit` and causing a denial of service. Since only noncode was injected, but the injected symbols maliciously changed the output program, this is an instance of a *noncode-injection* attack.

The disclosure of the Shellshock vulnerability in Bash operating system shell was one of the biggest security reports of the past year. Thousands of attacks exploiting the vulnerability were reported within hours of disclosure [4, 5]. US-CERT/NIST rated the severity of all six Shellshock vulnerabilities it announced as a 10.0 out of 10 [6, 7, 8, 9, 10, 11], justified by the arbitrary code execution capability it provided to attackers. Soon after the initial disclosure, new vulnerabilities continued to be disclosed, and patches to fix them sometimes negated previous fixes or even introduced new vulnerabilities themselves [12, 13, 14, 15]. This story adds to the pile of evidence that the hack-patch cycle fails to produce secure software. For example, similar vulnerabilities could exist in Bash, either undiscovered or undisclosed; we have no way to guarantee that such attacks are prevented.

This thesis examines injection attacks in the context of the Android mobile device operating system and the Shellshock vulnerability in Bash. While Shellshock may refer to the entire family of vulnerabilities discovered after the initial disclosure, we specifically refer to the first and seemingly most prevalent vulnerability. A study of the Android operating system and the top 50 free applications on the Google Play application market reveals that injection attacks are possible on Android applications and such vulnerabilities are likely to exist in the wild. A study of Shellshock attacks shows that Shellshock differs from typical injection attacks in that it is a higher-order attack, where injected symbols start as noncode in the output program and only later become code [16]. Furthermore, the manner in which the injected symbols may become code implies that Shellshock may occur at any depth in a process tree rooted at the target application. A prototype implementation instruments Java applications to precisely detect injection attacks, including Shellshock, for SQLite (a SQL implementation available for Android applications) and a handwritten derivative of Bash.

1.1 Related Work

There have been many proposed solutions to defining (e.g., [3, 16, 17, 18, 19, 20, 21]) and detecting (e.g., [22, 23, 24, 25, 26, 27, 28, 29, 30, 31]) code-injection attacks. Halfond et al. and Nguyen-Tuong et al. consider malicious injections to be keywords and operator symbols (including quotes for string literals) [19, 20]. SQLCheck requires injected symbols to form complete derivations of nonterminals in the output program language [17]. CANDID detects attacks based by comparing

Related Work	False Positive	False Negative
Halfond & Nguyen-Tuong et al. [19, 20]	`...WHERE name='Clayton'`	`...WHERE id=exit()`
SQLCheck [17]	`...WHERE file='name.ext'`	`...WHERE id=exit()`
CANDID [18]	`...WHERE flag=TRUE`	`...WHERE id=exit()`

Figure 1.1: Examples of false positives and negatives in related work

syntactic structure of the output program and its intended representation, which is formed by replacing injected symbols with 'a' or '1' characters, depending on the injected symbol [18]. All of these approaches exhibit false positives and false negatives, as illustrated in Figure 1.1. Precisely detecting injection attacks requires carefully defining code, injection, and attacks to address these shortcomings.

Ray and Ligatti [3] define noncode as exactly the closed values in the output program language. Therefore, injected symbols should strictly either be removed by the tokenizer (e.g., insignificant whitespace) or cause noncode tokens to be inserted or expanded in the output program (the NIE property). Injections that have broken this propery (BroNIEs) are attacks, capturing exactly the set of code- and noncode-injection attacks. To my knowledge, Ray and Ligatti's work [3] is the only formal study of noncode-injection attacks. This definition, discussed further in Chapter 2, forms the theoretical foundation for this thesis.

Ray and Ligatti prove that no static or black-box mechanism can precisely prevent injection attacks; a dynamic, white-box mechanism is required [16]. In practice, programmers may elect to use conservative mechanisms to prevent injection attacks. An example of such a mechanism is a *prepared statement*, also known as a *parameterized statement* or *parameterized query* [32]. In this scheme, output programs are created and parsed in advance, using placeholders for untrusted input, and the input later fills the placeholders. Yet, as evidenced by the continued ubiquity of these attacks, such mechanisms have failed to solve the problem as a whole. Several factors hinder the scalability of prepared statements [3].

- They must be supported for all output languages. While this is standard for SQL languages [33, 34, 35, 36], we have yet to see such mechanisms for Bash scripts, for example.
- Programmers must elect to use prepared statements. Intuitively, blindly concatenating strings together to form a query is much simpler than using prepared statements. A programmer

faced with time pressures or ignorant of security issues may not elect or simply not know to use such mechanisms.

- Converting an existing application to use prepared statements would require a rewriting, and therefore re-testing of every piece of code that forms an output program. Such a decision could be very expensive.

Injection attacks would therefore ideally be detected precisely and transparently to the programmer. Since untrusted inputs could propagate continuously as data dependencies, transparently preventing such attacks would then require a taint tracking mechanism to label tainted (injected) data. Taint tracking is well-studied and generally expensive to perform precisely (e.g., [30, 21, 19, 20, 37]). Intuitively, changing an add operation on two registers to an add operation plus taint propagation would result in substantial overhead without some assumptions on the width of the data or some hardware assistance to encode a taint bit into the tracked data itself. Chin and Wagner [38] show that by confining taint tracking to strings and modifying the underlying String class, a Java application can be instrumented with little overhead.

The design and implementation of Android's security model [39] is discussed in Chapter 3.1. At a high-level, permissions requested at install time determine an application's capabilities. To enforce these permissions, Android assigns a unique Linux user id (uid) to each application and associates each application's permissions with its uid when authorization is performed at a low-level, e.g. when interacting with the file system. A malicious application may abuse its capabilities, and some research has studied the use of static analysis to detect malicious applications [40, 41]. Vidas et al. discuss privilege escalation among other attacks on Android systems [42]. Users may actually elect to use privilege escalation to circumvent security mechanisms on the existing operating system and gain root access on their devices or install modified operating systems; typically, users do not have such access. However, an attacker may use privilege escalation as well; multiple applications with disjoint permissions may collaborate using inter-process communication to act with the union of their permissions [43]. For example, an application with access to read contacts may send contacts data to an application with permissions to send SMS messages or access the internet, and that application may use that information to spam the user's contacts or upload the data to the attacker's storage. Jin et al. examine injection attacks on HTML5-enabled mobile devices, showing

that mobile devices have many unique channels in which such malicious injections may occur, such as the camera, near-field communication, and Intent objects for inter-process communication on Android [44].

1.2 Summary of Contributions and Roadmap

Chapter 2 begins with a formal review of injection attacks as defined by Ray and Ligatti [3]. With this theoretical basis established, Chapter 3 begins a study of real-world systems by examining the Android mobile operating system, it's software development kit, and a selection of popular applications, showing that Android applications are just as susceptible to injection attacks as a typical web application; it also examines injection attacks on operating system shells using Bash and its recent Shellshock vulnerabilities as a case study. Chapter 4 formalizes a custom shell language called Lash. Chapter 5 discusses a prototype implementation of Lash and mechanisms to precisely detect injection attacks with Java as the application language and both SQLite and Lash as output lanaguages. Chapter 6 discusses experiements performed to validate the implementations' effectiveness and closes with lessons learned from this project.

I began this project with Grant Smith [45]. Together, we examined the Android system, SDK, and applications. The ping drain and external storage wipe attacks discussed in Chapter 3.1, majority of the survey of related work on Android security in Section 1.1, and original taint-tracking mechanism are his contribution. The formal definition of injection attacks and language-independent detection algorithm, discussed in Chapter 2 are attributed to Ray and Ligatti [3]. The revised taint-tracking mechanism discussed in Chapter 5 and remainder of this paper are my contribution unless otherwise indicated.

CHAPTER 2

DEFINITIONS: CODE, INJECTION, AND ATTACKS

Intuitively, an injection attack occurs exactly when an injected symbol maliciously alters an output program. This chapter provides an overview of Ray and Ligatti's formal definition of injection attacks [3], which dictates the design and analysis of concrete systems as discussed in the chapters that follow.

2.1 Defining Code

Intuitively, an output program's instructions should already be provided by the application; the injected input should simply affect the parameters to those instructions. For example, when servicing a request to create an account for some new user, the injected user name should contain strictly the user name and nothing more; an attacker should not be able to inject a function call, even if it does return the user name, for it may cause malicious side-effects or never terminate.

This intuition leads us to define code by defining its complement: noncode is the set of passive terms in the output program language. In a typical programming language, this would mean characters, strings, numbers, etc. that are all fully evaluated. However, even fully-evaluated expressions may still invoke dynamic behavior; variables are bound to values, but must be substituted at runtime. Expressions are open when they contain free variables; otherwise, they are closed.

These definitions are formalized as follows [3]. Henceforth, let $|p|$ denote the length of an output program p, and $p[i]$ denote the i^{th} symbol in p.

Definition 1 (Free Variables [3])**.** *For all languages L, function $FV(p, l, h)$ over $L \times \{1..|p|\} \times \{l..|p|\}$ returns the set of free variables in the shortest term in p containing all symbols from l to h.*

Definition 2 (Values [3])**.** *For all languages L, predicate $Val(p, l, h)$ over $L \times \{1..|p|\} \times \{l..|p|\}$ is true iff the shortest term that contains the l^{th} to h^{th} symbols in L-program p is a value.*

Definition 3 (Noncode Values [3]). *For all languages L, predicate $NCV(p, l, h)$ over $L \times \{1..|p|\} \times \{l..|p|\}$ is true iff $FV(p, l, h) = \emptyset$ and $Val(p, l, h)$.*

These preliminary definitions allow us to partition code and noncode symbols in an output program. Noncode symbols are either removed by the tokenizer or within some closed value in p; code symbols are all other symbols.

Definition 4 (Tokenize [3]). *For all L-programs $p = \sigma_1\sigma_2..\sigma_n$ and position numbers $i \in \{1..|p|\}$, $Tokenize(p)$ returns the sequence of tokens within p.*

Definition 5 (Tokenizer-Removed Symbols [3]). *For all L-programs $p = \sigma_1\sigma_2..\sigma_n$ and position numbers $i \in \{1..|p|\}$, $TR(p, i)$ is true iff i is not within the bounds of any token in $Tokenize(p)$.*

Definition 6 (Noncode Symbols [3]). *For all L-programs $p = \sigma_1\sigma_2..\sigma_n$ and position numbers $i \in \{1..|p|\}$, $Noncode(p, i)$ is true iff $TR(p, i)$ or for all low and high symbol-position numbers $l \in \{1..i\}$ and $h \in \{i..|p|\}$, $NCV(p, l, h)$.*

2.2 Defining Injection

Languages are sets of strings, which are sequences of symbols over a fixed alphabet Σ. Any symbol in the output program could potentially be injected; injected symbols are distinguished by mirroring each distinct symbol $\sigma \in \Sigma$ with an injected version $\underline{\sigma}$. The union of all these symbols forms the tainted alphabet $\underline{\Sigma}$.

Definition 7 (Injected Alphabet [3]). *For all alphabets Σ, the* injected alphabet $\underline{\Sigma}$ *is:*

$$\{\sigma \mid \sigma \in \Sigma \vee (\exists \sigma' \in \Sigma : \sigma = \underline{\sigma'})\}.$$

Definition 8 (Injected Symbols [3]). *For all alphabets Σ and symbols $\sigma \in \underline{\Sigma}$, the predicate $injected(\sigma)$ is true iff $\sigma \notin \Sigma$.*

Definition 9 (Injected Output Language [3]). *For all languages L over Σ, the* injected output language $\underline{L}$ *over $\underline{\Sigma}$ is:*

$$\{\sigma_1..\sigma_n \mid \exists \sigma'_1..\sigma'_n \in L : \forall i \in \{1..n\} : (\sigma_i{=}\sigma'_i \vee \sigma_i{=}\underline{\sigma'_i})\}$$

2.3 Defining Attacks: CIAOs and BroNIEs

Using our definition of code and injection, defining CIAOs becomes trivial. A CIAO occurs when there exists a symbol in the output program that is both injected and code—i.e., the symbol is outside the normal output program alphabet Σ and cannot be part of a noncode value.

However, complications arise when certain properties of the output program language allow noncode symbols to maliciously modify the output program. The following program from Chapter 1 demonstrates that CIAOs do not capture all injection attacks.

```
$data = ‘\’; securityCheck(); $data .= ‘&f=exit#’;\n f();
```

The only injected symbol \ is within a string literal ‘\’, so it is noncode. However, a lexical feature allows the closing single quote to be escaped, rendering it as part of the following larger string literal.

```
‘\’; securityCheck(); $data .= ’
```

Here, strictly noncode injected symbols caused malicious changes in the output program; specifically, the security check was bypassed. Since strictly noncode was injected, this attack is not a CIAO; a stronger definition is required to capture it.

Typically, the first stage of a compiler will involve tokenizing the program: grouping symbols together as tokens, collectively referred to as a token stream. Each token will typically contain data members for the token type, the contained symbols, and the location of the token in the overall program. Tokens are denoted as $\tau_i(text)_j$, where τ is the token type, $text$ is the captured text (i.e., the token's semantic value), and i and j are the first and last indices where the token appears in the output program, respectively. For example, for a language supporting algebraic expressions, the program `5+10` could be tokenized to $< NUM_1(5)_1\ ,\ PLUS_2(+)_2\ ,\ NUM_3(10)_4 >$.

In the context of injections, one might assume that injected symbols expand the token stream of the output program, either by adding new tokens (e.g., 5 in 5+10) or expanding existing ones (e.g., 0 in 5+10). In a real-world scenario, a string describing a file extension (e.g., `".txt"`) might be expanded to include the file name (e.g., `"stuff.txt"`) to form a complete, relative file path in a shell script. Token expansion is defined more formally as follows.

Definition 10 (Token expansion [3]). *A token* $t = \tau_i(v)_j$ *can be* expanded *into token* $t' = \tau'_{i'}(v')_{j'}$, *denoted* $t \preceq t'$, *iff* $\tau = \tau'$, $i' \leq i \leq j \leq j'$ *and* v *is a subsequence of* v'

The inclusion of indices in token expansion ensures that tokens only expand the strings at the same point in a program. For example, a token $STRING_1(\text{“}\texttt{abc}\text{”})_5$ expands to $STRING_1(\text{“}\texttt{abcd}\text{”})_6$, but expands neither to $STRING_2(\text{“}\texttt{abcd}\text{”})_7$ nor $STRING_{11}(\text{“}\texttt{abcd}\text{”})_{16}$. This component is critical for the following method of detecting noncode-injection attacks, where the output program is compared to its template, which contains only trusted symbols.

Definition 11 (Template [3]). *The* template *of a program* p, *denoted* $\mathbb{T}_p$, *is obtained by replacing each injected symbol in* p *with an* ε.

In $\mathbb{T}_p$, the ε symbols are ignored; their only purpose is to hold the indices of injected symbols. To detect an injection attack, both the output program and its template are tokenized and the resulting token streams are compared to check for a violation fo the noncode insertion and expansion (NIE) property: injected symbols should strictly insert or expand noncode tokens in the output program.

Definition 12 (NIE Property [3]). *An L-program* p satisfies the NIE property *iff there exist:*

- $I \subseteq noncodeToks(p)$ *(i.e., a set of* p*'s inserted noncode tokens),*
- $n \in \mathbb{N}$ *(i.e., a number of* p*'s expanded noncode tokens),*
- $\{t_1..t_n\} \subseteq tokenize(\mathbb{T}_p)$ *(i.e., a set of template tokens to be expanded), and*
- $\{t'_1..t'_n\} \subseteq noncodeToks(p)$ *(i.e., a set of* p*'s expanded noncode tokens)*

such that $t_1 \preceq t'_1, \ \ldots, t_n \preceq t'_n$ *and* $tokenize(p) = ([t'_1/t_1]..[t'_n/t_n]tokenize(\mathbb{T}_p)) \cup I$.

A *BroNIE* occurs exactly when an output program has broken the NIE property.

Returning to the previous example attack, Figure 2.1 illustrates how the output program compares to its template. Code symbols are associated with unshaded boxes, noncode symbols with shaded boxes, and tokenizer-removed symbols with no boxes. The noncode token $STRING_9(\text{“}\varepsilon''\text{”})_{11}$ was expanded to $STRING_9(\text{“}\underline{\setminus}'\texttt{;securityCheck}();\texttt{\$data.} =''\text{”})_{40}$. Since $STRING$ tokens are noncode tokens, this injection is allowed. However, many prohibited changes occured to the template. The code token $ID_{14}(\texttt{securityCheck})_{26}$ and other nearby tokens were removed. The symbols

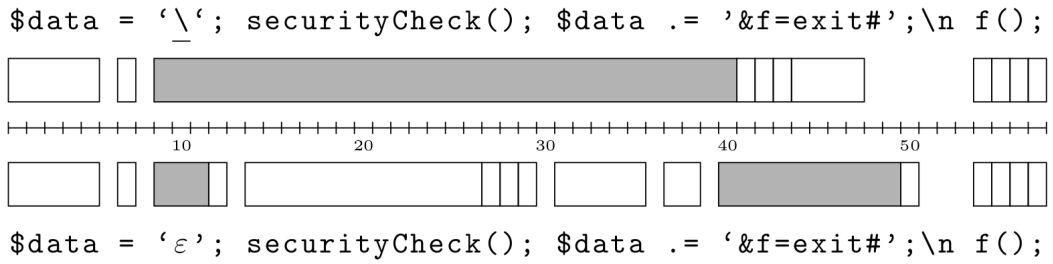


Figure 2.1: Partitioning symbols in an output program and its template. This figure is reprinted, with permission, from [46]. See Appendix A.

within `&f=exit`, while noncode in the template, became code symbols in the output program; non-code tokens were removed and code tokens were inserted. Many similar changes occured in the output program; the only type of prohibited change that did not occur was code token expansion. Any of these prohibited changes by definition exhibit a BroNIE.

Furthermore, BroNIEs are actually a superset of CIAOs; if a CIAO has occurred, then a BroNIE has occured. An injected code symbol must insert or expand a code token, which is forbidden by the NIE property. These definitions—ultimately, the NIE property—provide a formal framework to reason about injection attacks. The following chapters apply this abstract framework to concrete systems.

CHAPTER 3

CASE STUDIES

With a complete theoretical understanding of injection attacks, the following sections detail some case studies into injection attacks. These target application platforms have some unique characteristics that make them interesting to analyze versus the typical example of a web application using a database. Section 3.1 examines injection attacks on Android applications and Section 3.2 examines injection attacks on operating system shells, specifically the Shellshock attacks on Bash.[1]

3.1 Android

Portions of this section, such as the system-specific attacks on Android, were originally published in [45], but are the result of collaborative research with which I was significantly involved.

Typically, injection attacks are considered in the context of some centralized service interacting with many clients, such as web applications. Indeed, the past 15 years have seen a surge in centralized services for personal computing: cloud computing, online file hosting, media streaming, social networks, etc., and this trend seems likely to continue. Such services could easily be expected to use a database and to fall prey to SQL injections and similar attacks.

However, there has also been a trend in the development of decentralized and distributed systems stemming from the Internet, such as bitcoin cryptocurrency, the TOR relay network, and the BitTorrent file sharing protocol. In these systems, networked nodes communicate directly with each other. With injection attacks, the attack model therefore scales with the number of nodes on the network; every node is potentially exposed to and capable of performing attacks on other nodes.

Even with centralized services, client nodes may communicate with each other indirectly, using the service node as a proxy. For example, a centralized chat program propagates messages to their

[1] Portions of this chapter were previously published in [45].

intended recipients. Clients may store user information and chat histories in a SQL database, leaving them potentially vulnerable to injection attacks. This attack model is best exemplified in cross-site scripting (XSS), where a web service takes malicious data from an attacker and outputs that data to other clients, resulting in an injection attack when the page is rendered.

Today, nodes on such a network can even be mobile devices, such as cellular phones. With the increasingly information-driven economy today, mobile devices provide a vast amount of valuable information not normally exposed to the network by a traditional personal computer. This information can be attributed to the array of components integrated into modern devices, such as cameras, GPS, accelerometers, etc. Injection attacks could certainly be a method to subvert a user to obtain such information, especially when JavaScript and operating system shells are involved. In fact, such an attack has been successfully performed on Android devices before, where an implementation flaw in WebView—essentially a browser embedded into Android applications—allowed an attacker to dynamically invoke arbitrary Java code [47].

This section specifically looks into how injection attacks could (or do) happen on Android, an open-source mobile device operating system built on the Linux kernel [48]. Android protects certain device resources, such as the camera and network interface, using a static permission system [49, 50]. Applications declare in their package manifest which permissions they require to run properly. These permissions are displayed to the user when installing an application, and the user may then choose to install the application and grant the required permissions or abort the install; to our knowledge, applications cannot be installed with fewer permissions than it has requested in off-the-shelf distributions of Android, nor does the operating system monitor how an application uses its requested features.

To enforce permission at the system level, Android assigns each application a distinct Linux user id (uid), with the permissions the application has requested attached to that uid. For example, each application's uid is assigned ownership of an exclusive directory in the internal storage's file system. Also, while launching a shell is essentially escaping the limitations of Android's API, we observed that a shell launched by an application operates with the same uid as that application, granting it the same set of permissions. The permissions of a compromised application therefore determine the capabilities of an attacker performing an injection attack on that application.

```
public class PingService extends IntentService {
    public PingService() {
        super("PingService");
    }
    @Override
    protected void onHandleIntent(Intent intent) {
        Intent url = intent.getStringExtra("url")
        Runtime.getRuntime.exec("sh␣-c␣'ping␣-c3␣" + url + "'");
    }
}
```

Figure 3.1: A sample Android component using tainted input to build a command

We studied the Android operating system, its software development kit (SDK), and applications to conclude that lack of fine-grained runtime monitoring makes Android applications just as susceptible to injection attacks as any common web application. These attacks may occur through use of the operating system shell and SQLite database interface and can compromise the device in ways unique to mobile devices.

3.1.1 Potential for Attack

The Android SDK is a Java framework. Java's runtime libraries (included in the SDK) include the Runtime.exec set of methods [51] and the ProcessBuilder class [52], which allow an application to create a subprocess and execute some auxiliary program. Specifically included in the Android SDK is a SQLite database API [53], whereby an application may interact with a database using SQL queries. These capabilities reveal some interesting potential attacks.

Figure 3.1 demonstrates how a component within an Android application might fall prey to an injection attack. The component's job is to ping requested URLs. Since to our knowledge the Android SDK provides no mechanism to ping a remote host, the component resorts to the operating system shell to execute the `ping` program. Other applications send an intent—an inter-process communication construct in Android—containing the remote hostname to this service, and it constructs a request to ping the host with 3 packets (as specified by the `-c3` option). Since it does not verify the requested URL string, an attacker may inject a malicious string to change the

behavior of the shell command by closing the current command with a `;` and continuing the input string with a new command. The following is a list of some example attacks that may exploit this behavior:

1. Ping drain (Internet permission required) – Mobile devices can generally access the Internet through Wi-Fi or through the cellular data channel. Providers may limit a user's data usage per some unit time. An attacker can use the `ping` program, with a maximum size packet (65535 bytes on our device) at the shortest interval (5/sec on our device) to use 3MB/sec of data using the input string "`; ping -i 0.2 -s 65535 usf.edu`". 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. External storage wipe (write access to external storage required) – If the PingService's application has write access to external storage, an attacker may exploit this circumstance to wipe external storage without having any such permissions using a string of the form "`; rm -rf <location>/*`" to recursively delete all accessible files in the root external storage directory without requesting confirmation. The `<location>` on each individual device's file system may vary (e.g., /sdcard, /storage/sdcard), increasing the complexity required to reliably attack any Android device. If the attacker is acting through a malicious application on the target device, the attacker can reliably obtain this information using the SDK to retrieve the location of external storage. Otherwise, an attacker can attempt an attack on one location or some combination of locations. Access to external storage appears to have changed in Android KitKat, where applications receive access only to an exclusive directory in external storage, much like the segmentation of main storage [54]. However, devices with older versions of Android will not have this protection, and this storage sandboxing still allows an attacker to wipe data inside that particular directory, which is likely being used by the application since it requested permission, allowing an attacker to access, remove, or potentially modify used by the application.

3. Fork bomb (no permissions required) – Android applications are restricted only very loosely in the number of processes they may create—up to 6656 in our experiments. We were able

Amazon Shopping	Apus Launcher	Flow Free
Clean Master	CM Security	Tournament Challenge
Trivia Crack	Facebook	Facebook Messenger
My Music	Remote Control PRO	Bingo Crush
Prize Claw 2	Despicable Me	Fruit Ninja Free
CBS Sports	Coin Trip	Hulu
Temple Run 2	Instagram	GO Keyboard
GO SMS	Jelly Jump	ZigZag
Subway Surf	Candy Crush Saga	Candy Crush Soda
8 Ball Pool	Dubsmash	NCAA March Madness Live
Netflix	Pandora	Pinterest
360 Security	Kika Emoji Keyboard	Five Nights at Freddys 3 Demo
Skype	Snapchat	SoundCloud
Spotify	Clash of Clans	Flashlight
Surgery Simulator	Twitter	The Weather Channel
WhatsApp	Yahoo Mail	Crossy Road
Kik	Zedge	

Figure 3.2: Set of Android applications analyzed

to execute a resource denial attack by using an app exploiting this possibility. In this case, an attacker may execute the fork bomb using an input "`; bomb() { bomb|bomb& }; bomb`". There are 3 commands in the resulting string. The first is the original ping command, which goes unfulfilled because it never included a url to ping. The second defines a function `bomb` that recursively calls itself and pipes its output into another recursive call to itself running as a background process. The third command initiates the forkbomb, causing processes to be created exponentially until the operating system denies them, or (as was our experience) the system is rendered inoperable.

According to Jin et al., mobile devices have many unique channels in which such malicious injections may occur in addition to typical I/O over a network; these attacks may also occur through the camera, SMS, near-field communication (NFC), or even the SSID of a nearby wireless network. Further, Android-specific channels aside from Intents include shared databases like contacts and calendars, and even the file system where there exist shared space between two applications [44].

3.1.2 Application Study

Following our initial study of the platform, we examined the 50 most popular free applications (as of March 2015) on the Google Play market, as shown in Figure 3.2. These application names

were retrieved from the value of each application's `resources/string[@name="app_name"]` xpath element, or from the application's Google Play market page when the xpath's value did not exist or was ambiguous. Since we gathered the compiled packages (APKs) of these applications and they were generally not found to be open-source, inspecting these applications in detail required examining disassembled bytecode and decompiled source code. We used apktool [55] to extract and disassemble code into a human-readable form, dex2jar [56] to translate dex bytecode (a separate format for Android's Dalvik virtual machine) into Java bytecode, and JD-GUI [57] and JAD [58] to decompile the Java bytecode and browse the resulting Java code.

Of the applications we studied, 92% use the SQLite database API, all of which applications at some point call SQLite API methods not guaranteed to be safe—i.e., without support for normalizing arguments (such as a user name for a contact query). An example of a safe method is the `SQLiteDatabase.query` method, which allows arguments to be bound as closed values before being placed in the query, and examples of unsafe methods are the `SQLiteDatabase.{rawQuery,execSQL}` methods, which take a string program and execute it on the database without doing any argument binding [53]. 74% of the applications used Runtime.exec methods or ProcessBuilder classes. There are no analgous safe versions of these methods.

We were able to analyze most applications in sufficient detail to find that auxiliary programs were executed to escape the limitations of the Android SDK, allowing applications to collect information from application logs using the `logcat` program, get system properties, and use heuristics to determine whether the device is rooted. Unsafe database API methods like rawQuery were called to execute queries built from string literals hard-coded in the application, which by Java semantics precludes injection attacks via those queries.

Our understanding of these applications was limited by code obfuscation and decompiler innacuracies. The Android build system invokes the ProGuard tool by default to obfuscate the application, where bytecode-level constructs like classes and fields are renamed to make the application harder to reverse engineer, limiting the amount of information presented by the application bytecode for us to analyze [59]. For example, deciding whether a method is invoked is impossible unless the invocation is trivial, i.e., always or never occurs (c.f. Rice's theorem). The total code size we examined was also too large to manually determine when any particular method is trivially invoked. Therefore, given a case where an application includes an unsafe SQLite API method call,

we could never be certain whether the method making that unsafe call would be invoked, and it's turtles all the way down. This problem is especially of concern to our sample application set because we found that many of them use third-party libraries, e.g. for advertisement.

This analysis could be improved by more sophisticated tooling. Control-flow analysis could help to identify trivial application properties. More advanced decompilers would produce more readable code (though still within limits of ProGuard's mangling). Better tooling would also allow us to examine more cases with potential for error, such as when an application calls a safe method like `SQLiteDatabase.query`, but just inserts injected values directly into the intial program string instead of using placeholders and binding those as arguments to be normalized by the SDK.

Another limit to this study is that we examined the most popular applications. With the expectation that an application's popularity tends to correlate with application quality, and given Android's relatively open application market, we expect that such vulnerabilities could very well exist in the wild, especially with applications written by less experienced developers.

3.2 Bash and Shellshock

Bash is a Linux operating system shell. The runtime is an interpreter, reading and executing one command at a time. Generally, commands are sequences of words specifying a program to execute and its arguments, but Bash also supports structured programming, variable assignment, etc. In addition to typical programming constructs, Bash has a unique form of evaluation called word expansion. For example, the word `ba$x`, where the value of x is `sh`, expands to `bash`. Further, the word `*.c` expands to a sequence of words containing the name of each C source code file in the working directory [60].

Figure 3.3 illustrates how an injection attack may also occur on a Bash script. Consider a service that processes requests for new accounts for users wanting access to a Linux server. The server might output a script for a shell like Bash to create the account. Such a script could be maliciously formed as well.

```
adduser clayton
```
(a) Benign input

```
adduser clayton; rm -rf /
```
(b) Malicious input

Figure 3.3: Example injections into a shell script

The program `adduser` is executed with the username `clayton` to create the account on the server. Afterward, regardless of the outcome of that command, the script attempts to wipe the entire root file system. Here, `-rf` represents options to recursively apply this command to all subfolders in the root directory `/` and to force the program to delete the files without asking the user for confirmation. Notwithstanding other ad-hoc system protections against these kinds of actions, the results could be catastrophic for the server and all its users.

Bash programs can execute built-in commands or execute external programs—including subshells, generally invoked by calling "bash" in a bash script—which will be executed in a subprocess. Subshells inherit variable bindings from their parent. Variables are generally bound to terminal values such as a string literal, but these values can actually be latent function definitions for use in subshells. To create such a function, the parent sets an environment variable (e.g., via the **env** or **export** commands) to a string of the form `'() { command1; command2; ... ; commandN }'`. When a child process is invoked, the child inherits the environment from the parent, scanning for variables matching this pattern. Any such string is then evaluated as code that can be executed by referring to the associated variable as the first element of a command. Fig. 3.4 demonstrates an example. The variable `x` is bound to a string literal (bound by single quotes), but the value of the string literal represents a function. The second line creates a subshell, which scans the environment for variables starting with `() {`. Upon finding the value of `x`, Bash interprets the characters within the curly braces as a function definition, and binds the command x to the resulting code. The next line, instead of using `$x` to deference `x` as the string literal `'() { echo hi; }'`, calls `x` to execute the code bound to `x`, which is a separate program with one command (`echo hi`).

```
parent$ export x='() { echo hi; }'
parent$ bash
 child$ x
hi
```

Figure 3.4: A sample execution where a function definition is used

One of the biggest computer-security stories of the past year—Shellshock attacks—arose from injection vulnerabilities in this feature of Bash. On its 10-point scale of severity, US-CERT/NIST rated every one of the six Shellshock vulnerabilities it announced as a 10.0 [6, 7, 8, 9, 10, 11]. While

Shellshock may refer to the entire family of vulnerabilities discovered around that time, we refer to specifically the first and seemingly most prevalent vulnerability.

While the semantics for functions may seem esoteric, they appear to be intended. The vulnerability is that trailing commands following the function definition will be executed when the child shell evaluates that function. Figure 3.5 is exactly the same as the previous example, with the exception that there is code following the original function definition. Due to a bug in Bash, the trailing code was executed immediately upon launching the subshell. An attacker with control over the end of the function definition is therefore capable of arbitrary code execution, within the constraints of Bash and the target system. In this specific example, the program appears to behave the same way, but the attack code fetches some malicious script over the network and executes it without any visible output beyond the injected symbols to indicate an attack has happened.

```
parent$ export x='() { echo hi; }; curl hax.com/attack.sh | bash'
parent$ bash
 child$ x
hi
```

Figure 3.5: A subshell inadvertently executes the code following the function definition

The flaw in Bash exposing the Shellshock vulnerability was quickly remedied, but as is common in the practice of computer security, as one patch for Shellshock became available, additional vulnerabilities were discovered, sometimes in the patches themselves, and sometimes in ways that made older patches ineffective [12, 13, 14, 15]. Without foundational principles to rely on, for understanding injection attacks in Bash, the community iterated the hack-patch cycle many times.

These iterations of incrementally patching implementations in response to every newly noticed injection vulnerability don't produce trustworthy systems. For example, we have no assurance that similar, yet-unnoticed injection vulnerabilities are absent from the standard implementations of Bash shells. Achieving a higher level of trustworthiness will instead require foundational, well-tested theories for understanding injection attacks, in addition to well-understood tools—built on those foundational principles—for automatically preventing injection attacks, even the yet-unnoticed ones.

At the surface, Shellshock is yet another injection attack. However, Shellshock poses some unique challenges to building such well-understood, precise detection mechanisms because it is a

real-world example of a second-order injection attack, where tainted input is originally dynamically passive in the output program and only later becomes code [3]. Consequently, Shellshock attacks cannot be precisely detected by instrumenting the target application producing these programs. For example, an application may create a shell *parent* that launches a subshell *sub1*, which only upon some nontrivial condition defines a function using tainted input from *parent* and launches another subshell *sub2* to trigger Shellshock. It cannot be decided whether an attack has occurred until that condition is evaluated at runtime; the output program has to be instrumented as well as the target application.

The following chapters introduce the design, implementation, and evaluation of a custom shell language, a string taint tracking mechanism, and a mechanism to precisely detect arbitrary-order injection attacks on output programs SQLite and the custom language, including Shellshock. The mechanisms' system-specific implementations and constraints shed light on some of the practical challenges—beyond taint tracking overhead—to feasibly and precisely detecting injection attacks in practice.

CHAPTER 4

A CUSTOM SHELL LANGUAGE

Though Shellshock is a Bash vulnerability, Bash's implementation inherently presents a lot of challenges to designing and implementing a precise mechanism to prevent vulnerabilities like Shellshock, as detailed in Section 4.1. Traditionally, a language dictates the design of enforcement mechanisms for injection attacks on output programs in that language because the partition of code and noncode is a static, language-level concern; the implementation of interpreters, compilers, and runtimes for output languages are normally irrelevant. However, as discussed in Chapter 3.2, Shellshock is a second-order attack; the language's implementation is a great concern because the implementation must provide or be instrumented with a taint tracker and detection mechanism. Since the objective here is to evaluate Shellshock and injection attacks on operating system shells in general, not Bash specifically, I created a custom shell language as the basis for the design of injection attack detection on operating system shells.

4.1 The Case for a Custom Shell Language

Bash's implementation is both complex and nonstandard [61]. Typically, a programming language can be expressed with regular expressions for lexical analysis, a context-free grammar for syntactic analysis, and a turing machine for semantic analysis. Regular expressions and context-free grammars, while less expressive than turing machines, have the benefit of being simpler and much more amenable to analysis. Even from the beginning lexical stages, most of a Bash program is interpreted using a generated lexical analyzer, but certain parts are instead interpreted ad-hoc in a context-sensitive manner; Bash keeps a finite history of tokens to determine how to interpret the next symbols. An example is the 'time' command, which provides support for timing the execution of a command pipeline. A series of symbols `time`, when appearing for example at the beginning of a program and not as part of a larger token, will be tokenized differently—i.e., as a distinct

Token text	`time`	`echo`	`ThisIsTimed`	`|`	`time`	`echo`	`ThisIsNotTimed`
Token type	TIME	WORD	WORD	PIPE	WORD	WORD	WORD

Figure 4.1: An example of context-sensitive lexical analysis in Bash

TIME token—from the same sequence of symbols if they appeared in the same form but later in the pipeline—i.e., as just another WORD.

Figure 4.1 shows how Bash would handle tokenization for the input program `time echo ThisIsTimed | time echo ThisIsNotTimed`. In Bash, multiple commands can be chained together in a pipeline, with the output of one command sent to the input of another command, using the pipe (|) symbol. There are two commands in this program, but the beginning instance of `time` is part of neither one; it is more of a metacommand, instructing Bash to record and print the execution time of the commands represented by the tokens that follow. The second instance of `time`—even though it appears at the beginning of a command—is instead interpreted as an instruction to call the external program *time* if it exists, just as if it were any other occurence of *cat*, *grep*, etc.

Even for parsing alone, about 1000 of the approximately 6000-line Yacc grammar file is devoted to production rules; the rest are generally devoted to helper functions and macros. Not counting libraries, the source code is approximately 70,000 lines long. These complexities make for a newcomer modifying the existing code to include modules like taint propagation an intractable task.

Irrespective of code complexity, Bash is implemented in C, a language with only weak typing. Precise runtime monitoring requires control-flow integrity to prove that a security mechanism exhibits complete mediation over an aspect of a target program [62]. For example, the Bash interpreter could be instrumented with a taint tracker that propagates taint data when assigning a variable to some text in the program code (any symbols of which may be tainted). However, given C's direct memory access, another point in the program (possibly under the control of an attacker) could point directly to the address containing the taint information for a given string and clear the taint data. Control-flow integrity is easily obtained as a consequence type-safety—for example, such direct modifications to data memory would be forbidden—but in this case it would be very difficult or impossible to prove that Bash's implementation has such integrity. Given the vulner-

programs	p	::=	c^{∞}
commands	c	::=	$e^* \mid x = w \mid x \mid \text{lash}$
exprs	e	::=	$x \mid v
values	v	::=	$w \mid \varepsilon$
vars	x	::=	TV
words	w	::=	TW
functions	f	::=	$c^* c^*$

Figure 4.2: Lash first-order abstract syntax

abilities already seen in Bash, the semantics of the C and its preprocessor, and the size of Bash's code base, Bash is not likely to have such integrity. This inherent lack of guarantees precludes us from instrumenting Bash to implement any kind of precise mechanism[1].

Given these problems posed by Bash, and the goal of studying injection attacks and Shellshock directly versus Bash itself, I created a simplified shell language as the basis for studying injection vulnerabilities on operating system shells.

4.2 Lash: A Minimal Shell Language

Lash—for Lackluster Shell— is a minimal language emulating Bash-like behavior to the extent required to be vulnerable to Shellshock. Its features include:

- commands and arguments (using string literals),
- variable substitution (via word expansion),
- dynamically declared functions, and
- subshell function imports (to be vulnerable to Shellshock).

4.2.1 Syntax and Semantics

Lash's first-order abstract syntax is shown in Figure 4.2. At a glance, this syntax looks like it describes some simple shell language—it's nearly a subset of Bash. Programs are sequences of commands. Programs are potentially infinite in size to reflect that Bash is an interpreter. Since interpreters dynamically read and execute a stream of symbols, there's no guarantee that the

[1]This analysis is based on reviewing the Bash's source code at version 3.4—months before the Shellshock attacks—and may not reflect the current state of the code [61].

input stream is finite. A command can appear as a sequence of expressions, which once evaluated represent a program to execute and its arguments, if any; a variable assignment to some word; a function invocation by calling a variable bound to some command sequence; or subshell execution by calling *lash*. An expression may appear as a word or a reference to a variable bound to a word, or the empty word ε, appearing when the value of a variable is fetched but that variable is not bound to any value. The terminals indicating the concrete structure of variables and words, denoted TV and TW, respectively, are left up to the implementation. It is expected that implementation will make ε a runtime form.

The most notable difference from Bash is the handling of functions. This grammar includes an explicit nonterminal for functions, but there is no derivation from programs to functions. This detail is a consequence of the desire to make Lash minimal while still supporting Shellshock. Functions appearing in Shellshock attacks become code only later; initially they are values in the output program. Allowing statically-declared functions (as Bash does) adds complexity without enhancing our understanding of injection attacks. As a result, functions are runtime forms.

Figure 4.3 formalizes Lash's dynamic semantics, which are rather simple until functions are involved. Lash's state is a pair (F,E): a function context $F : \mathit{variables} \to \mathit{commands}^*$, and a word context $E : \mathit{variables} \to \mathit{words}$, representing traditional shell environment variable mappings.

There are two forms of evaluation in the dynamic semantics which are assigned different names: expression evaluation, occurring in the form of variable substitution; and command execution, which corresponds to statements in C-like languages. Command execution propagates variable substitution and emulates execution of the command on the system.

Figure 4.3 uses a few helper judgement forms and notation. Parentheses are used to denote tuples and to resolve potential ambiguity. When a variable describes a collection of objects, an s is appeneded to the variable name—furthermore, a variable *css* denotes a collection of collections. Since commands are sequences, $\langle\rangle$ is used to denote an empty sequence. The operator $(::) : \alpha \times \alpha^\infty \to \alpha^\infty$, borrowed from ML-style syntax, prepends an element to the beginning of a sequence. The $(@) : \alpha^* \times \alpha^\infty \to \alpha^\infty$ operator concatenates two sequences. Rules SP1 and SP2 explicitly define the judgement form $S + (x : y) = S'$, which describes a function of the form $(\alpha \times \beta) \times \{(\alpha \times \beta)\} \to \{(\alpha \times \beta)\}$, expressing that upon adding a variable-value binding to a set of bindings, the existing value for that variable, if any, is replaced. The $\oplus$ operator, defined in the rules of the judgement

$$\boxed{\text{S + (x:y) = S'}}$$

$$\frac{\exists y' : S = S' \cup (x : y')}{S + (x : y) = S' \cup (x : y)}\ \text{SP1} \qquad \frac{\forall y' : (x : y') \notin S}{S + (x : y) = S \cup (x : y)}\ \text{SP2}$$

$$\boxed{S \oplus S' = S''}$$

$$\frac{S' = (x : y) \cup S''}{S \oplus S' = (S + (x : y)) \oplus S''}\ \text{MP1} \qquad \frac{}{S \oplus \emptyset = S}\ \text{MP2}$$

$$\boxed{E \vdash e \mapsto v}$$

$$\frac{(x : w) \in E}{E \vdash \$x \mapsto w}\ \text{SE1} \qquad \frac{(x : w) \notin E}{E \vdash \$x \mapsto \varepsilon}\ \text{SE2}$$

$$\boxed{(F, E, p) \mapsto (F', E', p')}$$

$$\frac{E \vdash e_i \mapsto v_i}{(F, E, (e_1 \cdots e_i \cdots e_n) :: cs) \mapsto (F, E, (e_1 \cdots v_i \cdots e_n) :: cs)}\ \text{SC1}$$

$$\frac{}{(F, E, (v_1 \cdots v_n) :: cs) \mapsto (F, E, cs)}\ \text{SC2} \qquad \frac{}{(F, E, (x = v) :: cs) \mapsto (F, E + (x : v), cs)}\ \text{SC3}$$

$$\frac{(x : cs') \in F}{(F, E, x :: cs) \mapsto (F, E, cs'@cs)}\ \text{SC4} \qquad \frac{\forall cs' : (x : cs') \notin F}{(F, E, x :: cs) \mapsto (F, E, cs)}\ \text{SC5}$$

$$\frac{(F', css') = \{((x : cs), cs') \mid \exists w : (x : w) \in E \wedge w\ rep\ (cs, cs')\} \qquad cs' = \underset{cs' \in css'}{@} \langle\rangle}{(F, E, lash :: cs'') \mapsto (F \oplus F', E, cs'@cs'')}\ \text{SC6}$$

Figure 4.3: Lash dynamic semantics

form $S \oplus S' = S''$, simply extends the + operator to multiple bindings, replacing any bindings in S with those in S', or adding the bindings when no conflict exists.

The judgement form $E \vdash e \mapsto v$ denotes that a word context E derives that an expression steps to a value. If the word context E binds variable x to a word w, $\$x$ is substituted with w. Otherwise, $\$x$ is substituted with ε.

Rules SC1 through SC6 describe how a commands are executed. Interestingly, they appear in the judgement form $(F, E, p) \mapsto (F', E', p')$. Due to functions, commands can change the shape of the rest of the program. Hence, the behavior of commands is best specfied in the context of a program.

Rule SC1 propagates variable substitution to a command in any order—e.g., given a command `echo $x $y`, the order of subsitution of x and y is irrelevant since there is no side-effect to substitution.

Rule SC2 describes emulates a fully-evaluated being executed on the environment, which at the language level just steps to the next command; there is no feedback from the command as to its return value or whether is succeeded or failed. The first word in the sequence specifies the program to execute, and the remaining words specify arguments to the program to execute; instances of ε are ignored. A command of the form ε^* does nothing.

Rule SC3 binds a variable to a word, adding or replacing the binding in E. Rule SC4 and SC5 describe function invocation, which is performed by calling the variable to which it is bound (without using the $ operator). When calling x, if x is bound in F to a command sequence, that function body is scheduled to be executed next.

The relative complexity rule SC6, defining function importing, warrants further explanation. The command *lash* emulates launching a subshell. At this point, functions are imported, as in Bash. Lash finds functions to import with the help of the implementation, which must define a surjective function $rep : words \to functions$, which indicates that a word w represents or is tokenized and parsed as a function. With the help of *rep*, Lash searches for functions to import. Note that the F, described as a function context, actually maps variables to finite sequences of commands instead of elements of the *functions* syntactic category, which specify two finite sequences. The second sequence of commands is never stored in F because the second sequence represents the trailing code after a function definition, which is executed upon importing the function. Instead, the first sequence is bound to the variable in F and the second sequence is scheduled to be executed next, just as in Shellshock. Some of the complexity of rule SC6 arises from multiple simultaneous imports. When importing function definitions, their corresponding trailing code sequences are all concatenated together with the @ operator, starting the empty sequence $\langle\rangle$ as a seed value (order of concatenation is arbitrary), and the resulting sequence is then inserted at the beginning of the rest of the program.

The handling of variables is a bit different compared to a standard formal language like the lambda calculus. A variable may be mapped to a command and a word simultaneously, since operations depending on the two contexts are syntactically distinct; variables mapped to words

within E may be substituted using an expression of the form $\$x$, while functions bound to variables may be invoked by executing a command of the form x. Also, in both cases, operations on unmapped variables do not cause the program to get stuck. Cases of unmapped variable dependencies could just be considered ill-typed and therefore undefined for dynamic semantics, but allowing operations on unbound variables more closely resembles that of Bash.

Without functions, there would be no function context, and no commands could add other commands to execute in the program. That is, notwithstanding this behavior, Lash semantics are quite simple, which can be attributed to several key ways it differs from Bash.

1. Lash lacks conditional expressions and branching.
2. The semantics do not require the implementation to interact with the underlying system in any way.
3. Functions do not take arguments or return results.
4. One a subshell is launched, the parent is disregarded; Lash operates through the remainder of the program as if it is the subshell.

The properties specified simplify the semantics of the language. As a result of (1), programs and their executions are strictly linear. This property contrasts with assembly languages, where a static or computed jump could return a program to a previous point in execution, leaving control flow as a cyclic graph. In Lash, there is no backtracking to previously-executed commands, so the control flow graph of any given program is totally linear. Item (2) mostly simplifies implementation, since the execution of other external programs isn't a concern in this case. Items (3) and (4) preclude the language from requiring any kind of implicit or explicit tree structure in the semantics for call stacks and process trees. These aspects, coupled with finite expression evaluation (at most one substitution) and the potentially infinite program length, demonstrate Lash to be an ω-regular language.

4.2.2 Injection Attacks in Lash

Since the term *value* is already attributed to fully-evaluated expressions, when describing non-code elements they are denoted passive constructs.

Definition 13 (Lash Noncode)**.** *Noncode in Lash is defined as ε and the set of words, except when a word appears as the first word of an expression sequence.*

The constructs described in Definition 13 are expression values, which don't step. Normally, they would always be considered noncode, but one special condition is made for *words*: if appearing within an expression sequence, it must not be the first element of the sequence. Expression sequences represent a program to execute and its arguments, and the first word in that sequence is always the command to execute. That is, `adduser clayton` is a valid injection, because only an argument was injected, while `adduser clayton` is considered code, because it specifies a call to some external program. Executing an external program is analagous to calling an internal function; both specify dynamic computation. All constructs not described in Definition 13 are code.

In practice, we may also wish to prohibit options being injected into commands—e.g., `rm -r filename`. The `-r` flag makes the command recursive, applying to all child elements in the file tree. However, we chose to omit this restriction for several reasons.

- Specifying command options via dash-notation is convention rather than a language feature
- there is no clear distinction between options and their arguments. For example, "`-Ifolder`" specifies `folder` as an argument to the `-I` option in GCC, while "`-rf`" specifies two independent options for the `rm` program.
- Options don't specify dynamic computation for the shell itself.

Shellshock occurs exactly when rule SC6 is applied, and there exists a BroNIE within cs' (the trailing code immediately scheduled for execution). That is, a subshell is invoked while an environment variable is bound to some value which represents a function, and the second command sequence in the function contains a BroNIE. If the first sequence of commands contains a BroNIE then it is still a second-order injection attack, but is not as severe as Shellshock because at least the malicious function must be explicitly invoked to be executed, whereas Shellshock execution happens involuntarily upon initializing the subshell, i.e., in a way the user or system administrator likely never intended.

That said, injection attacks are very closely tied to concrete syntax, since injected symbols may affect the code-noncode partition of the resulting output program and its template. The next

chapter examines an implementation of Lash and a prototype taint tracker and injection attack detection mechanism to assess the problem further.

CHAPTER 5

DESIGN AND IMPLEMENTATION OF CONCRETE SYSTEMS

Chapter 3 examined injection attacks on Android systems and operating system shells. Chapter 4 provided a formal definition of a custom language called Lash, giving a simpler foundation on which to evaluate mechanisms to defend against injection attacks—including Shellshock—on operating system shells. This chapter builds on the previous chapters with a discussion on design, implementation, and evaluation of a concrete Lash parser and runtime, a mechanism for taint tracking, and an injection attack detection mechanism for both SQLite and Lash (including Shellshock), corresponding to the studies performed in previous chapters.

Portions of this section describe synthesis work that began while working with Smith following our evaluation of injection attacks on Android systems [45]. These components include the taint tracker and detection mechanism—at least, for SQLite. Both mechanisms have since been updated; please see their respective sections below.

Ultimately, even aside from its widespread compatibility and high-level semantics and libraries, Java proved to be the best implementation language for all components.

- Lash requires no direct system access (e.g. process forking for commands and subshells) since it only emulates shell behavior; no low-level functionality is required.
- Using Java allows us to then use AspectJ for aspect-oriented programming [63].
- Using Java also allows us to take advantage of the ANTLR parser generator [64], which has a large base of grammars [65], and for new grammars removes the need for manual abstract syntax tree creation, which would have been needed with a more primitive parser generator.
- Java also provides many safety guarantees that a language like C does not (such as by using implicit references instead of explicit pointers), allowing greater confidence in complete mediation of the implemented security mechanisms.

```
program    ::=  command (separator command)*
separator  ::=  (SEMICOLON | NEWLINE)+
function   ::=  '() {' program '}' (separator program)*
command    ::=  word word* | NAME EQUALS word
word       ::=  '$' NAME | NAME | NUMBER | GLOB
NAME       ::=  [a-zA-Z][a-zA-Z0-9_]*
NUMBER     ::=  [0-9]+
GLOB       ::=  [^ ]+ | SQUOTE .* SQUOTE
```

Figure 5.1: Lash EBNF grammar and lexicon (some lexer detail omitted for brevity)

- Since Android applications are generally programmed in Java (c.f. Section 3.1), this language choice allows both output languages to share a single taint-tracking mechanism for both output languages and a framework for injection attack detection (which requires extra implementation only for each language-specific component).

Thesis details will be elaborated when necessary in the following sections.

5.1 Output Program Languages: Lash and SQLite

Lexers and parsers for both languages were generated by ANTLR 4 [64]. SQLite had an existing grammar [65]. We were only concerned with first-order attacks on SQLite, so no runtime was required. Lash's grammar and runtime were hand-written.

Figure 5.1 shows Lash's concrete syntax in EBNF form, corresponding to the the first-order abstract syntax in Figure 4.2. Some lexer detail is omitted from the figure: single quotes are used to delimit anonymous string literals that appear in production rules (e.g., the `$` in *command* rules), while SQUOTE is used when a literal (i.e., escaped) single quote is required (e.g., in the GLOB terminal); some obviously named terminals are undefined but should still be clear; and quoted GLOBs capture only the matched text inside the quotes, rather than the quotes too.

In this grammar, programs are sequences of commands separated by some separator, which is a finite sequence of semicolons and/or newlines. Programs are actually finite in the implementation; while infinite programs were easy to integrate formally, infinite programs posed needless complexity for implementation. For example, programs could not be treated as strings; streams would be required instead.

Functions correspond exactly to the abstract syntax, allowing function definitions and trailing code to appear just as they do in Bash, beginning with "`() {`" and ending with "}", optionally followed by a separator and the Shellshock trailing command sequence.

A command may be a word sequence, where the first word is the command to execute, or a variable assignment. While the abstract syntax handled function and subshell invocations as syntactically separate commands, the implementation handles this more simply: For any fully-evaluted command sequence, if the first word is "`lash`", a subshell is lauched; if the word is bound to some function, that function is invoked; otherwise, it represents an execution of some external program identified by that word. The implementation is able to handle commands this way due to the way words are implemented, which are explained in further depth below.

A word may represent variable subsitution for some NAME by preceding the NAME with `$`. A fully-evaluated word (sometimes denoted a *word value*) is a NAME, NUMBER, or GLOB. NAME is any word beginning with an alphabetic character. NUMBER is an artifact of plans for function arguments accessed by words of the form `$0`, `$1`, etc. GLOB represents some other string without special meaning, including quoted strings. Standard lexical precedence applies: the lexer will match whatever rule allows it to match the most symbols; otherwise, it will match whatever rule is declared first (i.e., the string `x` will always be a NAME and not a GLOB). NAME terminals are used both as *variables* and as *words* in the abstract syntax; the abstract syntax identified variables and words differently, but did not require them to be lexically distinct, and this behavior mimics that of Bash. For example, the string "`directory`" can be a variable name or a literal argument to a command. The behavior, however, is equivalent. The empty word ε is implemented as an empty string, and may be introduced by the GLOB `‘‘"`. Since ε is captured in the GLOB terminal, the grammar does not need an ε production rule.

Notwithstanding some minor deviations—such as allowing subshells to take a program to execute by executing the subshell with a command string of the form `lash -c ‘<programToExecute>’`—the implementation follows the dynamic semantics a specified in Figure 4.3. Corresponding to Definition 13, noncode in this implementation is exactly the set of word values exactly when they appear as arguments to an external program invocation or to a variable assignment. The noncode partitioning in the concrete implementation is slightly less straightforward because a NAME can be a variable or word, and injection into a variable is an attack because variables are open expressions.

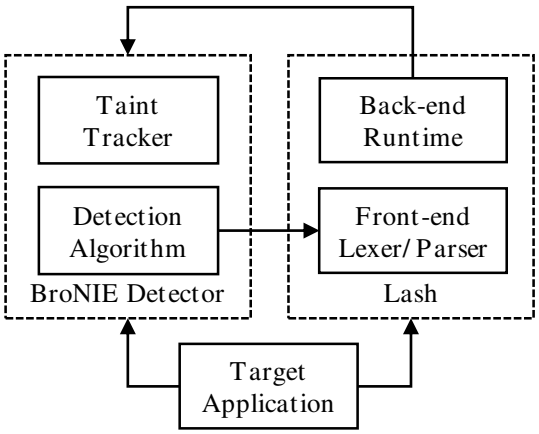


Figure 5.2: Composition of our implementation. Arrows indicate a "has-a" relation.

5.2 Security Mechanisms

Figure 5.2 illustrates the implementation arcthiecture, specifically for Lash. Solid boxes represent individual components, while dotted boxes indicate cohesion. Arrows indicate a "has-a" relation. As expected, the target application has a taint tracker and detection mechanism and access to Lash to run the output program. The detection algorithm needs to have access to the front-end of the output program language to tokenize the output program and determine which components are code. The unique part for Lash, since second-order attacks are possible, is that the runtime has a taint tracker and detection algorithm as well; the SQLite component is identical except for that detail.

For taint tracking, Chin and Wagner describe a promising solution by modifying the Java runtime's String class, which is backed by a character array, by adding a corresponding boolean array to indicate a taint bit for each character and adjusting method definitions to manage this data. When we attempted this solution on the default JVM, we appeared to encounter the same issues where making such a change to String broke expectations of predefined offsets for members of the String class, which did not yield an easily rectifiable solution; the authors even chose to pursue a different JVM [38].

In our case, we wanted a more universal solution, and taint tracking is only a means to the end of studying precise detection. Instead, we implemented the taint tracker as a separate component, and taint tracking as an aspect using AspectJ for aspect-oriented programming [63]. The

target application marks which strings are tainted and uses a special wrapper method for string concatenation—denoted *concat*. In aspect-oriented terms, the mechanism is advice inserted when *concat* returns, combining the taint information from the two strings that formed the newly created one. In practice, pointcuts can be adjusted as needed (such as when contents of a text box are extracted).

Taint information is held by mapping a String object reference to a boolean array reference, with bit i set to true in the array if and only if the i^{th} character in the string is tainted. In Java, object references support the `==` operator, which returns true iff the operands reference the same object. Objects may also override the `Object.equals(Object)` method, which returns true iff the self and the argument are considered equal. This is useful when there may exist duplicates of the same object in memory; for example, an application may have one instance of the String "Cancel" for each dialog box. A HashMap is a hash table that resolves mapping collisions by calling `equals` to determine whether two references are considered the same. Such collisions would certainly be problematic for taint tracking, where two distinct string objects are equal but only one originated from an untrusted source. Consequently, the taint tracker uses an IdentityHashMap, which resolves collisions by checking whether the object references are equal (`x == y`).

The detection mechanism is an implementation of the algorithm specified in [3]. Using a pointcut where a predefined method call is invoked to execute the output program, the detection aspect inserts advice to perform BroNIE detection.

The following chapter evaluates the effectiveness of the implementations and concludes lessons learned from the project as a whole.

CHAPTER 6

EVALUATION AND CONCLUSIONS

Related work has established a theoretical understanding of the injection attack problem domain [3] , and this paper has done so for Shellshock in particular. By formalizing a custom language, there is clear idea of the problems the implementation is designed to solve. This chapter evalutes the effectiveness of the implementations described in Chapter 5 in terms of correctness and performance, and concludes results from the project as a whole.

6.1 Experimental Methodology

The implementation was tested using a simple target application for both output languages. The target application takes a trusted prefix p, untrusted input i, and trusted suffix s of an output program; builds the output program by blindly concatnating the strings $(p + i + s)$; and outputs the resulting program for execution.

For evaluating correctness, test cases were designed to perform injections on different benign and malicious places for various synactical structures. For example, injecting a word value into the rvalue of a variable assignment should be allowed, while inserting a variable reference should not; injecting a semicolon into a quoted string is allowed since it's just a GLOB terminal, but it should not be allowed otherwise because it's a command separator.

For performance testing, the security mechanisms were tested with programs of approximate lengths 15 to 550. Performance test cases were formed by concatenating copies of correctness test programs; each program was tested with 1 to 9 copies of itself concatenated together. The test cases were all performed 20 times. The experiment then depended on whether it was testing strictly taint-tracking overhead or combined overhead. When testing strictly taint-tracking overhead, nothing additional occurs, and the entire test was carried out 1000 times. When testing program building and execution, the execution part was simulated by just making the output program language's

front-end parse the output program (which would engage the BroNIE detection pointcut), and then the entire test was carried out only 20 times. Since the first round of data was often found to be an outlier—presumably due to class loading the first time a class is encountered in the runtime—the first round was discarded.

Typically we could run the entire program instead of just parsing it to measure overhead of the operation as a whole, but Lash is pretty strictly compute-bound, and any attempts to make it otherwise would be still only simulated, compared to SQLite which already has an industrial-strength implementation. If we were to measure it this way, Lash would appear to have an extremely high overhead because the runtime would not be doing much. We also could not accurately simulate real execution time by program length, because a command `grep someWord *.txt` is small, but might be an extremely computationally intensive operation given a current working directory with extremely large text files. However, the reliable intersection between the two available to us is the parser, which we know will be invoked for each program. Therefore we include only parsing in the detection overhead measurements. This approach also allows us to view the data as an upper-bound; with any type of runtime behavior synchronized with the application, the overall runtime can be expected to decrease since the runtime of the output program execution is extended by behavior unrelated to the security mechanism.

The variable iterations per use case is due to some contraints on timing accuracy. This is perhaps the biggest drawback of having used Java. I found that Java's timing accuracy was variable: it could measure elapsed, user CPU time, and combined user and system CPU time, but the accuracy of these mechanisms was dependent on the system. On my Linux machine, the user time measurement had only 10ms accuracy, and elapsed time proved to be wildly innacurate for performance comparisons. To cope with this issue, and since user time is what we wish to ideally measure, these commands were executed in bulk to the extent that a sufficiently large enough time span would have passed—on the order of seconds or tens of seconds—to allow reliably determining overhead. Performing 1000 iterations of the already extensive parsing operations proved a prohibitively long task, so combined taint tracking and detection test iterations were greatly reduced.

#	Is BroNIE	Detected	Description
1	N	N	Assignment rvalue injected
2	N	N	Assignment rvalue partially injected
3	N	N	Assignment rvalue injected with variable reference afterward
4	Y	Y	Assignment lvalue injected
5	Y	Y	Assignment lvalue partially injected
6	Y	Y	Name of executable program injected
7	N	N	Argument to executable program injected
8	Y	Y	Variable substitution injected
9	Y	Y	Variable name injected in trusted substitution
10	Y	Y	Newline injected to separate commands
11	Y	Y	Semicolon injected to separate commands
12	N	N	Newline injected into quoted glob
13	N	N	Semicolon injected into quoted glob
14	N	N	No injections
15	Y	Y	Entire program injected
16	N	N	Function definition injected as glob, but no subshell launched
17	Y	Y	Function definition injected as glob, with subshell launched

Figure 6.1: Lash BroNIE functional test results

6.2 Results and Analysis

Figures 6.1 and 6.2 show the test results for Lash and SQLite, respectively. In all test cases, the BroNIE detection mechanism succeeded at identifying exactly when an injection attack has occurred.

Performance results are shown in Figure 6.3. Average taint-tracking overhead was 285.1% for Lash programs, and 225% for SQLite programs, with a standard deviation of 41% and 10.5%, respectively. However, we see some wild variation in the tracking and detection combined overhead, with Lash at 379.3% and Sqlite at 5.4%. This data was highly unintuitive at first and initially appeared to be a test implementation flaw. However, this data appears to be correct, though it is exaggerated from what we'd likely see in practice. SQLite's extremely low combined overhead is a consequence of the implementation and the mock output executable code (the mock SQLite library, in this case) using the same implementation language. Since the parse tree is already in memory, the mechanism simply takes the existing parse tree to evaluate the output program for BroNIEs. In practice it would probably have to be responsible for parsing the programs itself, indicating the runtime might double, leaving the overhead around 100% since the total cpu time for program parsing was found to be orders of magnitude greater than that of program building.

#	Is BroNIE	Detected	Description
1	N	N	Integer value injected
2	N	N	Integer value partially injected
3	N	N	String value partially injected
4	N	N	Null injected
5	Y	Y	CURRENT_DATE injected
6	Y	Y	Expression injected
7	Y	Y	Operator injected between trusted values
8	Y	Y	Comment injected without removing code after it
9	Y	Y	Comment injected removing code after it
10	Y	Y	Quote injected to terminate a string early, without removing code
11	Y	Y	Quote injected to terminate a string early, removing code after it
12	Y	Y	Table name injected in CREATE TABLE statement
13	Y	Y	CREATE injected in CREATE TABLE statement

Figure 6.2: SQLite BroNIE functional test results

Output Language	Metric	Min	Max	Mean	Median	Std Dev
Lash	Taint-Tracking	180.0%	320.8%	285.1%	304.0%	41.2%
Lash	Tracking + Detection	132.4%	412.0%	379.3%	412.0%	87.2%
Sqlite	Taint-Tracking	200.0%	233.3%	219.0%	225.0%	10.5%
Sqlite	Tracking + Detection	4.7%	6.4%	5.4%	5.4%	0.5%

Figure 6.3: Overhead of security mechanisms

However, we can draw two significiant conclusions from this data: there may be a workaround for the security mechanisms having to parse the output programs to determine which tokens are code by using the existing output program exectable's parse tree (and therefore this low overhead might be attainable), and the runtime of the output program's specified computation has a huge effect on what the overhead is considered to be.

Consider the effect of the target application's operation as a whole. The execution of the output script may be asynchronous, but my experience in software engineering tells me that it will not be; usually, we want to be certain when the output program has completed execution (and potentially receive feedback, e.g., whether the databse insert succeeded), and if not then we will only want to make the call asynchronous when performance issues arise. Therefore, it is safe to assume (and is indeed worst case) that the runtime of the output program affects the runtime of the entire system. This is difficult to measure with Lash since it doesn't actually interact with the environment. In any case, the output program's compiler will likely have to parse the output program once again after it has been received, making it happen twice for the same program. This runtime overhead is at the

```
public abstract class Monitor {
        ...
        abstract List<Token> getNoncodeTokens(ParseTree tree);
        abstract List<? extends Token> tokenize(String pgm);
        abstract String getTypeName(int tokenType);
        ...
}
```

Figure 6.4: Abstract Monitor methods, which could be implemented for each language

architectural level; even if we were to highly optimize the code, I can't imagine the solution would be attractive for high-throughput use cases. That said, it appears that for use cases where there is high risk and low throughput, the output program's behavior is naturally already computationally intensive, and/or there are not strict latency requirements, the benefit of transparent injection attack detection could certainly be worth paying the runtime cost.

The architectural overhead mentioned above brings up the glaring issue of software complexity. In order to minmize duplication of parsing, both the target application and the output program's compiler or interpreter would have to be engineered to take token streams and/or parse trees and/or abstract syntax trees to minimize that logic. Both applications would then have to agree on the structure of the data being passed as well. By unifying implementation to Java, I was able to create a framework for BroNIE detection and taint tracking shared between both output programs. Figure 6.4 shows a stub of that code, where other language's monitors would just inherit from Monitor and specify which tokens are noncode in a parse tree, how to tokenize a program, and mapping a token type encoding to its descriptive name.[1] In practice this luxury would not likely be present, given that applications and services are generally written in different higher-level languages today, and programs like SQLite and Bash are both implemented in C and executed as separate processes. To share data would require inter-process communication and some kind of cross-platform serialization like json, xml, or a custom binary format.

Taint-tracking has some other characteristics that also poses some high risks to implementing these mechanisms in practice. In the performance testing, tests ran up to approximately 30 string concatenations to arrive at the output program. In practice, tainted string might go through a

[1] I broke the rule of favoring composition over inheritance in this case; a monitor should simply have a language which provides this functionality.

much larger amount of a wider variety of operations—in fact, tainted data might not even be a string in the first place (e.g., it could be a number which was deserialized from some service call). The architectural complexity of identifying trusted and untrusted sources of data poses great uncertainty too. I expect taint tracking to be the biggest hinderance to the practical adoption of this technology because no software library would likely solve the issue of identifying and reliably tracking each consumer's trusted and untrusted data sources.

For more work on verifying our approach to designing and implementing enforcement mechanisms, it would be valuable to test randomly-generated programs for both correctness and performance in the hopes of generating unanticipated test cases. This could potentially yield cases of implementation errors or problems with the noncode partitioning for that target language. It also could help us deduce conclusions about performance from patterns in a large set of test data.

6.3 Closing Remarks

All overhead concerns aside, this project has successfully provided experimental verification of the ideas presented in [16] and [3], and shows that the ideas can be extended even to more procedural languages.

Chapter 4 indicates that Bash, and perhaps operating system shells as a whole, could greatly benefit from a formal analysis or even formal design from scratch in order to better understand them. Notwithstanding helper notation and judgement forms, Lash semantics are completely described with 8 rules, and that includes the extra rules for deliberately implementing Shellshock. Lash was formalized actually after it was implemented, and had the formalizing been completed first, it would have greatly changed my approach to implementing Lash. For example, I began by including a complex set of features including word quoting and expansion, function arguments, and subprocesses—the initial prototype was even in C since operating system shells are system-level tools. These features proved extraneous relative to the goal of simply implementing a shell that exhibits shellshock, and code is left over from those approaches as a result. Formally defining syntax and dynamic semantics provides literally a step-by-step guide to designed behavior. A second attempt at this implementation would be guided by the formalisms, resulting an implementation

that is simpler, easier to understand, and therefore giving a higher degree of confidence in its security.

It is quite unsurprising to find such a critical bug as Shellshock in the Bash's implementation related to a function called `stupidly_hack_special_variables` (c.f. version 4.3 in [61]), and it's not totally clear why this feature was implemented this way; for example, why not allow functions created normally [60] to just be inherited by subshells instead of being dynamically evaluted? Why set up the implementation so that code being parsed is potentially being executed simultaneously? Some features that make the language more complex are clearly beneficial, e.g. for interactive use. For example, we could make the language simpler if we forced all string literals to be quoted and variables unquoted like other programming languages, but that would then require the user to enter tedious commands in the form of `grep 'searchcontent' 'file1' 'file2' 'file3'`, when the same command without quotes is equivalent. The benefits of the functionality that caused Shellshock, however, are not so clear to me. That said, Bash is complex software that has undergone decades of design choices, and I cannot hope to understand all of them. At the same time, perhaps Shellshock is indicative that Bash has become too complex for any one person to completely understand, and that this is the optimal time to begin with something new.

LIST OF REFERENCES

[1] The MITRE Corporation. *CWE/SANS Top 25 Most Dangerous Software Errors*, 2011. Document version 1.0.2, http://cwe.mitre.org/top25/archive/2011/2011_cwe_sans_top25.pdf.

[2] Sql as understood by sqlite, 2014. http://www.sqlite.org/lang.html.

[3] Donald Ray and Jay Ligatti. Defining injection attacks. In *Proceedings of the 17th International Information Security Conference (ISC)*, pages 425–441, October 2014.

[4] Andy Greenberg. Hackers are already using the Shellshock bug to launch botnet attacks, September 2014. http://www.wired.com/2014/09/hackers-already-using-shellshock-bug-create-botnets-ddos-attacks/.

[5] Nicole Perlroth. Companies rush to fix Shellshock software bug as hackers launch thousands of attacks, September 2014. http://nyti.ms/1sz8Aaa.

[6] US-CERT/NIST. Vulnerability summary for CVE-2014-6271, September 2014. https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-6271.

[7] US-CERT/NIST. Vulnerability summary for CVE-2014-6277, September 2014. https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-6277.

[8] US-CERT/NIST. Vulnerability summary for CVE-2014-6278, September 2014. https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-6278.

[9] US-CERT/NIST. Vulnerability summary for CVE-2014-7169, September 2014. https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-7169.

[10] US-CERT/NIST. Vulnerability summary for CVE-2014-7186, September 2014. https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-7186.

[11] US-CERT/NIST. Vulnerability summary for CVE-2014-7187, September 2014. https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-7187.

[12] Sean Gallagher. New "Shellshock" patch rushed out to resolve gaps in first fix, September 2014. http://arstechnica.com/security/2014/09/new-shellshock-patch-rushed-out-to-resolve-gaps-in-first-fix/.

[13] Sean Gallagher. Still more vulnerabilities in bash? Shellshock becomes whack-a-mole, September 2014. http://arstechnica.com/security/2014/09/still-more-vulnerabilities-in-bash-shellshock-becomes-whack-a-mole/.

[14] Steve Ragan. Apple's recently released patch for Shellshock doesn't stop all attack vectors, September 2014. http://www.csoonline.com/article/2689410/vulnerabilities/apples-shellshock-patch-is-incomplete-experts-say.html/.

[15] Juha Saarinen. Further flaws render Shellshock patch ineffective, September 2014. http://www.itnews.com.au/News/396256,further-flaws-render-shellshock-patch-ineffective.aspx.

[16] Donald Ray and Jay Ligatti. Defining code-injection attacks. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*, pages 179–190, January 2012.

[17] Zhendong Su and Gary Wassermann. The essence of command injection attacks in web applications. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 372–382, 2006.

[18] Prithvi Bisht, P. Madhusudan, and V. N. Venkatakrishnan. CANDID: Dynamic candidate evaluations for automatic prevention of SQL injection attacks. *Transactions on Information and System Security (TISSEC)*, 13(2):1–39, 2010.

[19] William Halfond, Alex Orso, and Pete Manolios. Wasp: Protecting web applications using positive tainting and syntax-aware evaluation. *IEEE Trans. Softw. Eng.*, 34(1):65–81, 2008.

[20] Anh Nguyen-Tuong, Salvatore Guarnieri, Doug Greene, Jeff Shirley, and David Evans. Automatically hardening web applications using precise tainting. In *Proceedings of the International Information Security Conference (SEC)*, pages 372–382, 2005.

[21] Wei Xu, Sandeep Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: a practical approach to defeat a wide range of attacks. In *Proceedings of the 15th USENIX Security Symposium*, 2006.

[22] Martin Bravenboer, Eelco Dolstra, and Eelco Visser. Preventing injection attacks with syntax embeddings. *Science of Computer Programming (SCP)*, 75(7):473–495, 2010.

[23] Robert Hansen and Meredith Patterson. Stopping injection attacks with computational theory. In *Black Hat Briefings Conference*, 2005.

[24] Oracle. How to write injection-proof PL/SQL. An Oracle White Paper, December 2008. Page 11.

[25] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Precise alias analysis for static detection of web application vulnerabilities. In *Proceedings of the 2006 workshop on Programming languages and analysis for security*, pages 27–36. ACM, 2006.

[26] Gregory Buehrer, Bruce W. Weide, and Paolo A. G. Sivilotti. Using parse tree validation to prevent sql injection attacks. In *SEM '05: Proceedings of the 5th international workshop on software engineering and middleware*, pages 106–113, 2005.

[27] Adam Kieżun, Philip J. Guo, Karthick Jayaraman, and Michael D. Ernst. Automatic creation of SQL injection and cross-site scripting attacks. In *Proceedings of the International Conference on Software Engineering*, May 2009.

[28] Zhengqin Luo, Tamara Rezk, and Manuel Serrano. Automated code injection prevention for web applications. In *Proceedings of the Conference on Theory of Security and Applications*, 2011.

[29] Jonas Magazinius, Billy K. Rios, and Andrei Sabelfeld. Polyglots: Crossing origins by crossing formats. In *Proceedings of the Conference on Computer and Communications Security (CCS)*, pages 753–764, 2013.

[30] Tadeusz Pietraszek and Chris Vanden Berghe. Defending against injection attacks through context-sensitive string evaluation. In *Proceedings of Recent Advances in Intrusion Detection (RAID)*, 2005.

[31] Sooel Son, Kathryn S McKinley, and Vitaly Shmatikov. Diglossia: Detecting code injection attacks with precision and efficiency. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security (CCS)*, pages 1181–1192, 2013.

[32] The Open Web Application Security Project (OWASP). SQL injection prevention cheat sheet, 2012. https://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet.

[33] The PostgreSQL Global Development Group. *PostgreSQL 9.3.4 Documentation*, 2014. http://www.postgresql.org/docs/9.3/static/index.html.

[34] Microsoft. *Transact-SQL Reference (Database Engine)*, 2014. http://msdn.microsoft.com/en-us/library/bb510741(v=sql.120).aspx.

[35] Oracle. *MySQL 5.7 Reference Manual*, 2014. https://dev.mysql.com/doc/refman/5.7/en/index.html.

[36] Oracle. *Oracle Database SQL Language Reference, 12c Release 1 (12.1)*, 2013. http://docs.oracle.com/cd/E16655_01/server.121/e17209.pdf.

[37] James Clause, Wanchun Li, and Alessandro Orso. Dytan: a generic dynamic taint analysis framework. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 196–206, 2007.

[38] E. Chin and D. Wagner. Efficient character-level taint tracking for java. In *Proceedings of the ACM Workshop on Secure Web Services*, SWS, pages 3–12, New York, NY, USA, 2009. ACM. http://doi.acm.org/10.1145/1655121.1655125.

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

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] Xing Jin, Xuchao Hu, Kailiang Ying, Wenliang Du, Heng Yin, and Gautam Nagesh Peri. Code injection attacks on html5-based mobile apps: Characterization, detection and mitigation. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 66–77. ACM, 2014.

[45] Grant J. Smith. Analysis and prevention of code-injection attacks on android os. Master's thesis, University of South Florida, October 2014.

[46] Donald Ray and Jay Ligatti. Defining injection attacks. Technical Report CSE-TR-081114, University of South Florida, 2014.

[47] OSVDB. 97520: Google android multiple devices improper webview class implementation addjavascriptinterface method public arbitrary java method access, 2015. http://www.osvdb.org/97520.

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

[49] 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.

[50] 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.

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

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

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

[54] Google. External storage, 2015. https://source.android.com/devices/storage.

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

[56] Pxb1988. pxb1988/dex2jar - github, 2015. https://github.com/pxb1988/dex2jar.

[57] Java decompiler, 2015. http://jd.benow.ca/.

[58] Tomas Varaneckas. Jad java decompiler download mirror, 2015. http://varaneckas.com/jad/.

[59] Google. Proguard, 2015. https://developer.android.com/tools/help/proguard.html.

[60] Free Software Foundation Inc. Bash reference manual, 2015. https://www.gnu.org/software/bash/manual/bashref.html.

[61] Free Software Foundation Inc. Bash - gnu project - free software foundation, 2015. http://ftp.gnu.org/gnu/bash/.

[62] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity: Principles, implementations, and applications. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, pages 340–353, November 2005.

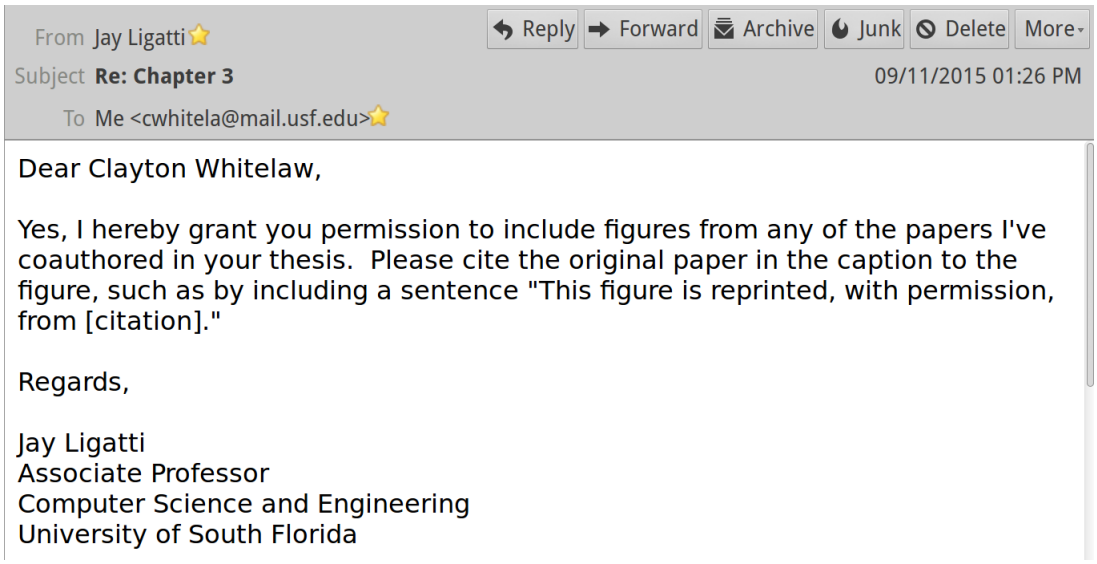
[63] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold. An overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–355, 2001.

[64] Terence Parr. Antlr, 2015. http://www.antlr.org/.

[65] Terence Parr. antlr/grammars-v4 github, 2015. https://github.com/antlr/grammars-v4.

APPENDIX A : COPYRIGHT PERMISSIONS

Below is permission for the use of Figure 2.1 in Chapter 2.



From Jay Ligatti
Subject **Re: Chapter 3**
To Me <cwhitela@mail.usf.edu>

Reply | Forward | Archive | Junk | Delete | More

09/11/2015 01:26 PM

Dear Clayton Whitelaw,

Yes, I hereby grant you permission to include figures from any of the papers I've coauthored in your thesis. Please cite the original paper in the caption to the figure, such as by including a sentence "This figure is reprinted, with permission, from [citation]."

Regards,

Jay Ligatti
Associate Professor
Computer Science and Engineering
University of South Florida