Defining Injection Attacks
Technical Report #CSE-TR-081114

Donald Ray and Jay Ligatti

University of South Florida
Department of Computer Science and Engineering
{dray3,ligatti}@cse.usf.edu

Abstract. This paper defines and analyzes injection attacks. The definition is based on the *NIE property*, which states that an application's untrusted inputs must only produce Noncode Insertions or Expansions in output programs (e.g., SQL queries). That is, when applications generate output programs based on untrusted inputs, the NIE property requires that inputs only affect output programs by inserting or expanding noncode tokens (e.g., string and float literals, lambda values, pointers, etc). This paper calls attacks based on violating the NIE property *BroNIEs* (i.e., Broken NIEs) and shows that all code-injection attacks are BroNIEs. In addition, BroNIEs contain many malicious injections that do not involve injections of code; we call such attacks *noncode*-injection attacks. In order to mitigate both code- and noncode-injection attacks, this paper presents an algorithm for detecting and preventing BroNIEs.

Key words: injection attacks, formal methods, language-based security

1 Introduction

According to multiple sources, the most commonly reported software attacks are injection attacks, including SQL injections, cross-site scripting, and OS-command injections [1–3]. MITRE-SANS considers these attacks to result from three of their top four "most dangerous software errors" [1].

Applications vulnerable to injection attacks generate output programs based on untrusted inputs. By providing a malicious input, an attacker can cause the application to output a malicious program. A classic example involves a simple web application for a bank; the application inputs a password and returns the balance of accounts with the given password. On a typical, benign input such as `123456`, the banking application outputs the following program (throughout this paper, input symbols injected into the output program are underlined).

`SELECT balance FROM accts WHERE pw='`<u>`123456`</u>`'`

Unfortunately, if this application does not validate its input, it can be manipulated into creating malicious programs. For example, on input `' OR 1=1 --`, the application outputs the following program.

```
SELECT balance FROM accts WHERE pw=‘’ OR 1=1 --’
```

This output program circumvents the password check and returns the balances of all accounts because (1) the 1=1 subexpression is a tautology, making the entire `WHERE` clause true, and (2) in SQL, the `--` sequence begins a comment, which removes the final apostrophe and makes the program syntactically valid. Because some of the injected symbols are code symbols (i.e., disjunction and equality operators), this is an example of a *code*-injection attack.

Due to their prevalence, much research has been performed to define code-injection attacks formally (e.g., [4–9]). Many additional papers have described tools for detecting and preventing code-injection attacks (e.g., [10–19]).

Interestingly, a related class of attacks exists but has not yet, as far as we're aware, been explored. These related attacks have many of the symptoms of code-injection attacks but don't involve injecting *code*. Because these attacks are performed by injecting noncode symbols, we call them *noncode-injection attacks*. Although noncode-injection attacks don't involve injecting malicious code, they may cause other parts of the output program to execute maliciously.

For example, the following web application[1] is vulnerable to noncode-injection attacks.

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

On a benign input such as `Hello!`, this application outputs the following.

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

This output program sets `$data` to a value, calls the `securityCheck()` function, appends a string to `$data`, and then invokes the `f` function. However, if an attacker enters the string \, the application outputs the following program.

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

No code has been injected in this alternative output program; the injected \ is part of a (noncode) string literal. However, because the injected symbol escapes the apostrophe that would have terminated the first string literal, the string literal continues until the next (non-escaped) apostrophe. As a result, the call to `securityCheck` is bypassed, the function `f` is updated to be the `exit` function, and finally, because `#` begins a comment that continues until the line break, the `exit` function (i.e., `f`) is invoked, shutting down the web server and causing a denial of service.

[1] We're grateful to Mike Samuel of Google for creating the first version of this example.

1.1 Related Work

Although much effort has been made to understand and prevent injection attacks, the previous work in this area has focused on *code*-injection attacks.

For example, Halfond et al. and Nguyen-Tuong et al. detect code-injection attacks based on whether keywords or operator symbols have been injected into output programs [7, 8]. False positives and negatives may arise with these techniques [4]; for example, these works consider injections of string literals to be attacks (because the beginning and ending apostrophes in string literals are classified as operators) and consider injections of function names to be non-attacks.

Other works detect code-injection attacks based on whether input symbols span tokens, or parse trees, in output programs. For example, Xu et al. detect code injections based on whether untrusted input spans tokens in the output program [9], while SqlCheck detects code injections based on whether untrusted input, injected into an output program P, spans subtrees in P's parse tree [5]. False positives and negatives again may arise with these techniques [4].

Candid detects code-injection attacks based on whether a second copy of the application, which is only given strings of a's (or 1s) as input but is forced to follow the same control-flow path as the original application, outputs a syntactically different program [6]. Again, false positives and negatives may arise with this technique [4].

To summarize the related work discussed to this point, all exhibit false positives and negatives when detecting code-injection attacks. Furthermore, all of these related works exhibit false positives and negatives when detecting general, including noncode, injection attacks (all of the example code-injection-attack false positives and negatives given in Figure 2 of [4] are also general-injection-attack false positives and negatives).

Still other work detects code-injection attacks based on whether untrusted inputs get used as non-values (i.e., non-normal-form terms) in output programs [4]. Although we believe that this technique detects code-injection attacks precisely (i.e., lacks false positives and negatives), it is tailored to code-injection attacks and cannot detect noncode-injection attacks, because it considers any symbol injected into a noncode value as safe. Hence, false negatives arise when using the techniques of [4] to detect noncode-injection attacks. In contrast, the present paper presents definitions and techniques for precisely detecting general injection attacks, including noncode-injection attacks.

In practice, a commonly recommended technique for preventing injection attacks is to use *parameterized queries* [20], wherein applications create output programs containing placeholders, or "holes", for untrusted inputs. For example, an application might create an output program that has a single placeholder for a string literal; to output a program, the application provides a string to fill that placeholder. In this way, parameterized queries can limit injections to filling in holes for string, numeric, or other kinds of literals, thus preventing both code- and noncode-injection attacks. While effective at preventing injection attacks, parameterized queries have significant disadvantages:

- Although parameterized queries are a standard feature of many SQL dialects [21–24], they are not supported by other common output-program languages such as HTML or bash. An output-program language must provide support for parameterized queries before an application can use them.
- It's the responsibility of application programmers to use parameterized queries. Unfortunately, parameterized queries are of little use to programmers that are either ignorant of, or apathetic to, their benefits. Whatever the reason, many application programmers are not doing so, as evidenced by the prevalence of injection vulnerabilities [1–3].
- Once the decision to use parameterized queries is made, modifying existing applications to use them is a manual, time-consuming process. Application programmers must find all possible ways for programs to be output, and replace those programs with new versions containing the appropriate placeholders. If even one output program is not replaced, the application will remain vulnerable to injection attacks.

1.2 Summary of Contributions and Roadmap

This paper defines and proves properties of a broad class of injection attacks, one that includes both code- and noncode-injection attacks. The definition of injection attacks is based on whether untrusted inputs affect output programs in any way besides inserting or expanding noncode tokens (such as string, integer, or float literals). When untrusted inputs only insert or expand noncode tokens in an output program, we say that the output program satisfies the *NIE property* (Noncode Insertion or Expansion). Intuitively, the NIE property restricts untrusted inputs to filling in "holes" in output programs that are reserved for noncode tokens.

The NIE property thus restricts untrusted inputs in ways that are similar to parameterized queries; both techniques require untrusted inputs to fill noncode-token "holes" in output programs. With parameterized queries, the holes for untrusted-input tokens are manually specified by application programmers; with this paper's techniques, all holes for untrusted-input tokens are automatically confined to being noncode (e.g., a string or numeric literal). Because this paper's techniques are widely applicable and require no modifications to existing applications, the techniques avoid the disadvantages of parameterized queries. However, the tradeoff here is that this paper's techniques rely on runtime monitoring for injection-attack detection, implying higher runtime overhead than parameterized queries.

This paper considers inputs that violate the NIE property to have "broken" the NIE property and calls such attacks BroNIEs. BroNIEs are therefore considered a general class of injection attacks, one that includes both code- and noncode-injection attacks.

After defining BroNIEs in Section 2, this paper presents several examples in Section 3. Section 4 explores the implications of the definitions in Section 2 and shows that (1) BroNIEs are indeed a strict superset of code-injection attacks, (2) all applications that blindly copy-and-paste an untrusted input into a SQL output program are vulnerable to BroNIEs, and (3) a linear-time algorithm exists for precisely (i.e., soundly and completely) detecting BroNIEs. Section 5 concludes with a brief discussion.

2 Definitions

This section formalizes criteria for determining when a BroNIE has occurred. Section 2.1 presents preliminary notation and assumptions. Then, because BroNIEs occur when injected symbols affect output programs beyond inserting or expanding noncode tokens, it is critical to know which symbols are *injected* and which tokens are *noncode*; Sections 2.2 and 2.3 present these subdefinitions. Finally, Sections 2.4 and 2.5 use the subdefinitions of injected symbols and noncode tokens to define code-injection attacks and BroNIEs.

2.1 Notation and Assumptions

An application vulnerable to BroNIEs outputs programs in some language L (e.g., SQL) that has a finite concrete-syntax alphabet Σ_L (e.g., the set of printable Unicode characters). These output programs, which we also call L-programs, are finite sequences of Σ_L symbols that each form an element of L. For L-program $p = \sigma_1\sigma_2..\sigma_n$, let $|p| = n$ and $p[i] = \sigma_i$. That is, $|p|$ denotes the length of p, and $p[i]$ denotes the i^{th} symbol of p. For a sequence S, the replacement of item t with item t' (i.e., the substitution of t' for t in S) is denoted $[t'/t]S$.

This paper makes a few assumptions about output-program languages. All output-program languages under consideration have well-defined functions for:

- Computing the free variables of program terms. Terms are called *open* if they contain free variables (e.g., `1+x`), and are otherwise *closed* (e.g., `1+2`).
- Testing whether program terms are *values*. Values are the "fully evaluated" terms of a programming language, such as literals, pointers, objects, lists and tuples of other values, lambda terms, etc.
- Tokenizing output programs. Function $tokenize_L(\sigma_1..\sigma_n)$ returns the sequence of tokens within the string $\sigma_1..\sigma_n$ (assuming $\sigma_1..\sigma_n$ is lexically valid; otherwise, $tokenize_L(\sigma_1..\sigma_n)$ returns the empty sequence). A token of *kind* τ composed of symbols $\sigma_i..\sigma_j$ is represented as $\tau_i(\sigma_i..\sigma_j)_j$. For example, for program $q =$ `SELECT * FROM orders WHERE s='' OR i<3`, $tokenize_{SQL}(q)$ returns tokens $SELECT_1(\texttt{SELECT})_6$, $STAR_8(\texttt{*})_8$, $FROM_{10}(\texttt{FROM})_{13}$, $ID_{15}(\texttt{orders})_{20}$, $WHERE_{22}(\texttt{WHERE})_{26}$, $ID_{28}(\texttt{s})_{28}$, $EQUALS_{29}(\texttt{=})_{29}$, $STRING_{30}(\texttt{''})_{31}$, $OR_{33}(\texttt{OR})_{34}$, $ID_{36}(\texttt{i})_{36}$, $LESS_{37}(\texttt{<})_{37}$, and $INT_{38}(\texttt{3})_{38}$. Predicate $TR_L(p, i)$ (tokenizer-removed) holds iff i is not within the bounds of any token in $tokenize_L(p)$. For example, $TR_{SQL}(q, i)$ holds for all $i \in \{7, 9, 14, 21, 27, 32, 35\}$.

This paper omits the L subscript from the $tokenize_L$ and TR_L functions when the output-program language is clear from context. Also, because all tokens are labeled with begin and end indices, it's trivial to convert a set of nonoverlapping tokens into the equivalent sequence of tokens (and vice versa). This paper therefore treats sequences of tokens as sets, and vice versa, as convenient.

2.2 Defining Injection

Applications vulnerable to BroNIEs output programs based on inputs from trusted and untrusted sources. *Injected* symbols are those that originate from untrusted sources and propagate unmodified through an application into its output program. A BroNIE occurs when injected symbols affect output programs in any way besides inserting or expanding noncode tokens.

We rely on the well-studied concept of taint tracking [7–9, 18, 19, 25, 26] to determine which output-program symbols originate from untrusted sources and are therefore injected. At a high level, a *taint-tracking application* works by replacing all symbols input from untrusted sources with their tainted versions and preserving these taint metadata during all copy and output operations. Provided that the application only taints symbols when they are being input from an untrusted source, and never untaints symbols, the injected symbols in the output program are exactly those that are tainted.

As we have been underlining injected symbols, we use the same notation to mark tainted symbols.

Definition 1 ([4]). *For all alphabets Σ, the* tainted-symbol alphabet $\underline{\Sigma}$ *is* $\{\sigma \mid \sigma \in \Sigma \vee (\exists \sigma' \in \Sigma : \sigma = \underline{\sigma'})\}$.

Next, language L is augmented to allow programs to contain tainted symbols.

Definition 2 ([4]). *For all languages L with alphabet Σ, the* tainted output language $\underline{L}$ *with alphabet* $\underline{\Sigma}$ *is* $\{\sigma_1..\sigma_n \mid \exists \sigma'_1..\sigma'_n {\in} L\colon \forall i {\in} \{1..n\}\colon (\sigma_i{=}\sigma'_i \vee \sigma_i{=}\underline{\sigma'_i})\}$.

Finally, an output-program symbol is injected if and only if it is tainted.

Definition 3 ([4]). *For all alphabets Σ and symbols $\sigma \in \underline{\Sigma}$, the predicate injected(σ) is true iff $\sigma \notin \Sigma$.*

For example, taint-tracking application `output('Hello' + input() + '!')` on untrusted input `World` will output the program `Hello` <u>`World`</u>`!` because the initial input is replaced by its tainted version <u>`World`</u>, and the taint metadata are preserved by the string-concatenation and output operations.

Because taint tracking is a well-studied technique, the remainder of this paper assumes that applications can track taints and thus output programs in which the injected (i.e., tainted) symbols are underlined.

2.3 Defining Noncode

Intuitively, noncode symbols in output programs are those that are dynamically passive; they specify no computation to be performed during execution. Code symbols, on the other hand, are dynamically active; they specify computation that could be performed during execution.

This paper considers a program's noncode symbols to be exactly those that are either (1) removed by the tokenizer or (2) within a closed value:

1. Although previous work sometimes allowed tokenizer-removed symbols such as whitespace or comments to be code [4], we believe that, because tokenizer-removed symbols are dynamically passive and cannot specify computation, it is more intuitive and accurate to consider such symbols noncode.
2. Closed values are operationally irreducible and thus specify no dynamic computation. Typical values include literals, pointers, objects, and tuples of other values. Open values are excluded because they specify the dynamic computation of substituting a term for a free variable during execution; closed values specify no such substitution operations.

When $p[i]$ is noncode (where p is an output program), we write $Noncode(p, i)$. Otherwise, we write $Code(p, i)$.

Definition 4. *For all L-programs $p = \sigma_1..\sigma_n$ and position numbers $i \in \{1..|p|\}$, predicate $Noncode(p, i)$ holds iff $TR_L(p, i)$ or there exist low and high symbol-position numbers $l \in \{1..i\}$, $h \in \{i..|p|\}$ such that $\sigma_l..\sigma_h$ is a closed value in p.*

Tokens composed entirely of noncode symbols are called *noncode tokens*. The set of all noncode tokens in a program p is $noncodeToks(p)$.

2.4 An Aside: Defining *Code*-injection Attacks

Using the predicates for determining which symbols are injected (Definition 3), and which are noncode (Definition 4), we can define Code-Injection Attacks on Output programs (CIAOs) as occurring exactly when an output program contains an injected code symbol.

Definition 5 ([4]). *A* CIAO *occurs exactly when a taint-tracking application outputs $\underline{L}$-program $p = \sigma_1..\sigma_n$ such that $\exists i \in \{1..n\} : (injected(\sigma_i) \ \wedge \ Code(p, i))$.*

2.5 Defining BroNIEs

Because BroNIEs occur when injected symbols affect output programs beyond inserting or expanding noncode tokens, they can be detected by observing how a program's sequence of tokens is affected by the removal of its injected symbols. Intuitively, removing all injected symbols from the output program should only affect the sequence of tokens in the following ways:

1. Some noncode tokens may no longer be present.
2. Some noncode tokens may become smaller but should not change kind (e.g., string literals should not become integer literals).

To formalize this intuition, we need to consider the sequence of tokens obtained by removing all injected symbols from an output program. The injected symbols cannot simply be deleted; doing so would affect the indices of tokens that follow the injected symbols. Instead, each injected symbol is replaced with an ε (the empty string). The sole purpose of an ε symbol is to hold the place of an injected symbol; ε's are otherwise ignored. Because the resulting string contains only uninjected symbols, it can be considered a *template* of the program.

Definition 6. *The* template *of a program p, denoted $[\varepsilon/\underline{\sigma}]p$, is obtained by replacing each injected symbol in p with an ε.*

For example, let program $r = \underline{\texttt{1}}\texttt{2}\underline{\texttt{3}}\texttt{+1}$. Then $[\varepsilon/\underline{\sigma}]r$ is $\varepsilon\texttt{2}\varepsilon\texttt{+1}$ (which is equivalent to `2+1`) and contains the tokens $INT_2(\texttt{2})_2$, $PLUS_4(\texttt{+})_4$, and $INT_5(\texttt{1})_5$.

The definition of BroNIEs also relies on notions of token insertion and expansion. Token insertion is straightforward, but we need to be clear about the meaning of token expansion: injected symbols may expand noncode tokens by increasing their ranges of indices and corresponding strings of program symbols.

Definition 7. *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'.

Returning to the example above, token $INT_2(\texttt{2})_2$ in $[\varepsilon/\underline{\sigma}]r$ can be expanded into token $INT_1(\underline{\texttt{1}}\texttt{2}\underline{\texttt{3}})_3$ in r. That is, $INT_2(\texttt{2})_2 \preceq INT_1(\underline{\texttt{1}}\texttt{2}\underline{\texttt{3}})_3$.

We can now formally specify when an output program exhibits only noncode insertion or expansion (NIE). Given a program p and its template $[\varepsilon/\underline{\sigma}]p$, it should be possible to get to the sequence of tokens in p from the sequence of tokens in $[\varepsilon/\underline{\sigma}]p$ by only inserting or expanding noncode tokens. If the sequence of tokens in p can be reached from the sequence of tokens $[\varepsilon/\underline{\sigma}]p$ in this way, we say that p satisfies the NIE property; otherwise it exhibits a BroNIE.

Definition 8. *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),*
- $\{t_1..t_n\} \subseteq tokenize([\varepsilon/\underline{\sigma}]p)$ *(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*
- $tokenize(p) = ([t'_1/t_1]..[t'_n/t_n]tokenize([\varepsilon/\underline{\sigma}]p)) \cup I$.

Definition 9. *A* BroNIE *(Broken NIE) occurs exactly when a taint-tracking application outputs a program that violates the NIE property.*

3 Examples

Let us consider several examples of how Definition 9 classifies programs as either attacks or non-attacks. Although all but one of this section's examples are presented in SQL, the underlying concepts apply to other languages as well.

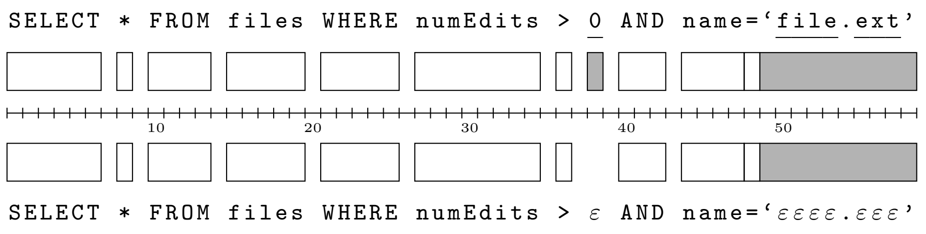

SELECT * FROM files WHERE numEdits > 0 AND name='file.ext'
10
20
30
40
50
SELECT * FROM files WHERE numEdits > ε AND name='εεεε.εεε'

Fig. 1: A depiction of how the Example-1 program (top) and its template (bottom) are tokenized. Noncode tokens are shaded.

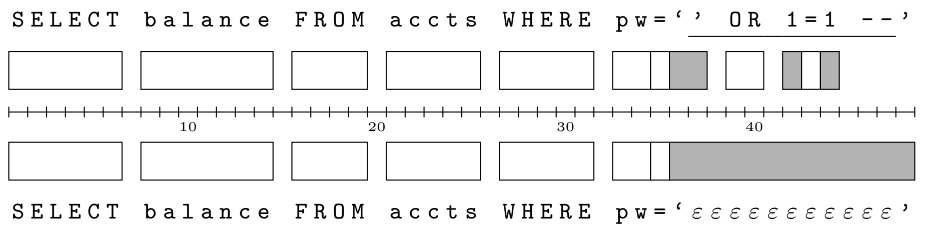

SELECT balance FROM accts WHERE pw='' OR 1=1 --'
10
20
30
40
SELECT balance FROM accts WHERE pw='εεεεεεεεεεεε'

Fig. 2: A depiction of how the Example-2 program (top) and its template (bottom) are tokenized. Noncode tokens are shaded.

Example 1. Consider the following simple output program.

```
SELECT * FROM files WHERE numEdits > 0 AND name='file.ext'
```

This query returns files named `file.ext` that have been edited at least once. Figure 1 shows how this program and its template are tokenized. This program does not exhibit a CIAO; all injected symbols are part of integer or string values. Neither is it a BroNIE; the injected symbols only cause noncode insertion and expansion, as depicted in Figure 1. More formally, the sequence of tokens in the template can be made into the sequence of tokens in the program by inserting the noncode token $INT_{38}(\texttt{0})_{38}$ and expanding the token $STRING_{49}(\texttt{'.'})_{58}$ into the (noncode) token $STRING_{49}(\texttt{'\underline{file}.\underline{ext}'})_{58}$. Hence, the definition of BroNIEs matches our intuition that this program does not exhibit an attack.

Example 2. Turning our attention to programs that do exhibit injection attacks, let's return to the first malicious output program presented in Section 1.

```
SELECT balance FROM accts WHERE pw='' OR 1=1 --'
```

This query returns all balances from the `accts` table because the `WHERE` clause is a tautology. The application that output this program tries to access only balances for which a password is known, but the malicious input circumvents the password check.

Figure 2 shows how this program and its template are tokenized. The program exhibits a CIAO and a BroNIE: a CIAO because the injected `O`, `R`, and `=` symbols

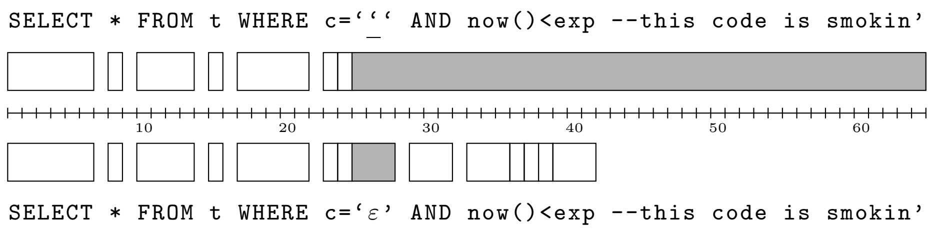

SELECT * FROM t WHERE c=‘‘‘ AND now()<exp --this code is smokin’
10
20
30
40
50
60
SELECT * FROM t WHERE c=‘ε’ AND now()<exp --this code is smokin’

Fig. 3: A depiction of how the Example-3 program (top) and its template (bottom) are tokenized. Noncode tokens are shaded.

are code, and a BroNIE because the injected symbols insert code tokens and *contract* the $STRING_{36}(‘’)_{48}$ token into $STRING_{36}(‘’)_{37}$.

Example 3. Because injections that cause CIAOs must also cause BroNIEs (as will be shown in Theorem 1), the remainder of this section focuses on non-CIAO examples of BroNIEs, such as the following.

```
SELECT * FROM t WHERE c=‘‘‘ AND now()<exp --this code is smokin’
```

The application that outputs this program attempts to find all unexpunged records with a given column value. For instance, the application could be querying a table of juvenile crimes (such as truancy) that can legally only be displayed when the offenders are not yet adults. However, the application appends a seemingly harmless, boastful comment to all output queries that allows an injected apostrophe to circumvent the expungement check by escaping the string literal's terminator (in SQL, apostrophes within string literals are escaped by a second apostrophe). If the attacker has control over the column being queried (e.g., it could be a comment column), then he or she can illegally access records after they have been expunged.

Figure 3 presents the tokenizations of this example program and its template. This program does not exhibit a CIAO; the injected symbol is part of a noncode (string) token. On the other hand, this program does exhibit a BroNIE; the injected symbol violates the NIE property by deleting code tokens.

Example 4. The following output program also contains a non-CIAO BroNIE.

```
INSERT INTO users VALUES (‘evilDoer’, TRUE)--’, FALSE)
```

This program creates a new user by inserting a record into the `users` table. Each record contains a field for the username and a boolean flag indicating whether the user has administrator privileges. By hardcoding a `FALSE` value for the second element of new-user records, the application that output this program attempts to ensure that all accounts it creates do not have administrator privileges. However, in this case an attacker has supplied a malicious username that causes the application to create an administrator-privileged account.

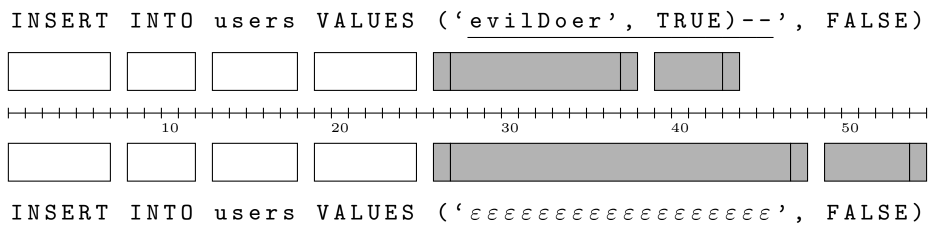

INSERT INTO users VALUES (‘evilDoer’, TRUE)--’, FALSE)
10
20
30
40
50
INSERT INTO users VALUES (‘εεεεεεεεεεεεεεεεεε’, FALSE)

Fig. 4: A depiction of how the Example-4 program (top) and its template (bottom) are tokenized. Noncode tokens are shaded. The tokens for commas and parentheses are considered noncode because they are within a tuple value; tuple terms are values when all subterms are values.

Figure 4 shows how this program and its template are tokenized. None of the injected symbols in the program are code symbols, so it does not exhibit a CIAO. However, it does exhibit a BroNIE; the injected symbols cause noncode contraction and deletion. For example, the last three noncode tokens in the template are not present in the output program. Thus, Definition 9 correctly considers this output program to be an attack.

Example 5. The following output program will serve as a final example of a non-CIAO BroNIE in SQL.

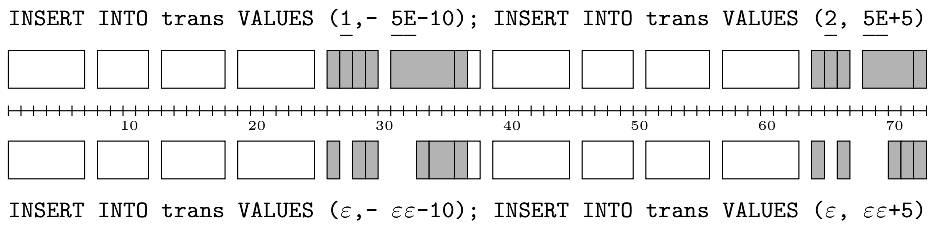
```
INSERT INTO trans VALUES (1,- 5E-10);
 INSERT INTO trans VALUES (2, 5E+5)
```

Here, an intern-authored application outputs programs to handle money transfers from one account (e.g., account number 1) to another (e.g., account number 2). The application applies a service charge of $10 to the paying account, but is currently running a special that transfers an extra $5 into the receiving account. However, by appending an ‘E’ to the money transfer amount, a malicious user can drastically affect the transfer process—in the above program, the paying account is only charged $0.0000000005 while the receiving account is credited $500,000.

Figure 5 shows that this output program exhibits a non-CIAO BroNIE. The output program does not exhibit a CIAO because no code has been injected; only components of float literals have been injected. On the other hand, the output program does exhibit a BroNIE because the injections delete the $MINUS_{33}(\texttt{-})_{33}$ and $PLUS_{70}(\texttt{+})_{70}$ tokens and transform two INT tokens into $FLOAT$ tokens (recall that Definition 7 does not allow tokens of one kind to expand into tokens of another kind).

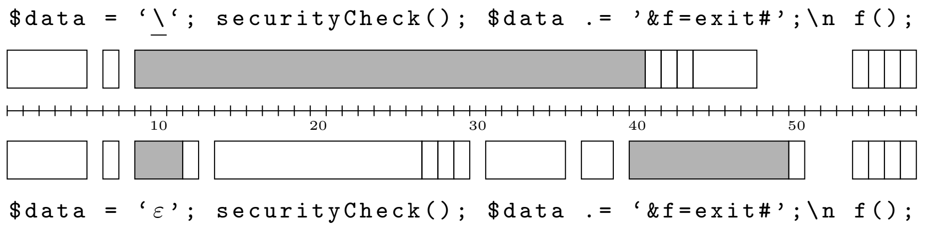
Example 6. To demonstrate the effectiveness of these techniques in other languages, we return to the following output program from Section 1.

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



```
INSERT INTO trans VALUES (1,- 5E-10); INSERT INTO trans VALUES (2, 5E+5)
INSERT INTO trans VALUES (ε,- εε-10); INSERT INTO trans VALUES (ε, εε+5)
```

Fig. 5: A depiction of how the Example-5 program (top) and its template (bottom) are tokenized. Noncode tokens are shaded. The tokens for commas and parentheses are considered noncode because they are within a tuple value; tuple terms are values when all subterms are values.



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

Fig. 6: A depiction of how the Example-6 program (top) and its template (bottom) are tokenized. Noncode tokens are shaded.

Because apostrophes within string literals are escaped by backslashes in this language, the injected symbol bypasses the call to the `securityCheck` function, garbles the contents of the `data` variable, and causes the `exit` function to be invoked instead of the `f` function.

The tokenizations of this program and its template are depicted in Figure 6. The program does not exhibit a CIAO; the only injected symbol is part of a string literal (i.e., a noncode value). However, the program does exhibit a BroNIE; the injected symbol deletes code and noncode tokens, and inserts code tokens, thus violating the NIE property.

4 Analysis of the BroNIE Definition

This section explores implications of the definitions in Section 2.

The first theorem formalizes our claim that all code-injection attacks are also BroNIEs. In other words, every application vulnerable to CIAOs is also vulnerable to BroNIEs

Theorem 1. *If a program exhibits a CIAO, then it exhibits a BroNIE.*

Proof. Let p be an output program that exhibits a CIAO. Then at least one of p's code symbols is injected; let σ_c be one of these injected code symbols and t be the token containing σ_c. Program p satisfies the NIE property iff the sequence of tokens in $[\varepsilon/\underline{\sigma}]p$ can be made equal to the sequence of tokens in p, subject to the constraints of Definition 8. Observe that t cannot be in $[\varepsilon/\underline{\sigma}]p$; no token in $[\varepsilon/\underline{\sigma}]p$ can have the same text as t unless its begin and/or end indices are different because, in $[\varepsilon/\underline{\sigma}]p$, σ_c will be replaced by an ε. Because t is a code token (it contains the code symbol σ_c), Definition 8 does not allow t to be inserted into the token sequence of $[\varepsilon/\underline{\sigma}]p$, nor for a token in $[\varepsilon/\underline{\sigma}]p$ to be expanded into t. Hence, the sequence of tokens in $[\varepsilon/\underline{\sigma}]p$ cannot be made equal to the sequence of tokens in p by only inserting and/or expanding noncode tokens, so p exhibits a BroNIE. $\square$

The second theorem provides a formal basis for the widely-accepted rule of thumb that it is unsafe to use unvalidated input during query construction [20, 27]. Theorem 2 states that if an application always includes an untrusted input (i_m) verbatim in its output (without even inspecting the input), and the same application has some input ($v_1, .., v_n$) for which it outputs a valid SQL program, then there exists a way to construct an attack input (a_m) such that the application's output will exhibit a CIAO and therefore a BroNIE. This theorem is a generalization of Theorem 9 in [4], which was limited to CIAOs in an idealized subset of SQL called SQL Diminished; in contrast, Theorem 2 below applies to full SQL.

Theorem 2. *For all n-ary functions A and (n-1)-ary functions A' and A'', if $\forall i_1, .., i_n\colon A(i_1, .., i_n) = A'(i_1, .., i_{m-1}, i_{m+1}, .., i_n)\underline{i_m}A''(i_1, .., i_{m-1}, i_{m+1}, .., i_n)$, where $1 \leq m \leq n$, and $\exists v_1, .., v_n\colon (v_m \in \Sigma^+_{SQL} \wedge A(v_1, .., v_n) \in SQL)$, then $\exists a_1, .., a_n\colon A(a_1, .., a_n) \in SQL$ and $A(a_1, .., a_n)$ exhibits a CIAO and a BroNIE.*

Proof. Observe that changing v_m to any a_m, without changing any of the other inputs to A, will cause A to output the same program but with a_m instead of v_m, because A' and A'' are independent of i_m. Now construct $a_m = v_m A''(v_1, .., v_{m-1}, v_{m+1}, .., v_n)\backslash n;\ \mathit{drop\ table\ t};\ A'(v_1, .., v_{m-1}, v_{m+1}, .., v_n)v_m$. This construction causes $A(v_1, .., v_{m-1}, a_m, v_{m+1}, .., v_n)$ to output the string $p = A(v_1, .., v_n)\underline{\backslash n;\ \mathit{drop\ table\ t};}A(v_1, .., v_n)$. Because $A(v_1, .., v_n) \in SQL$, so too is output-program p. By Definition 5, p exhibits a CIAO due to the injected `DROP` statement, as well as any other code symbols in a_m. By Theorem 1, p also exhibits a BroNIE. $\square$

It is also straightforward to prove, using the same techniques from [4], that neither static nor black-box mechanisms can precisely prevent BroNIEs. That is, precise detection of BroNIEs requires dynamic, white-box mechanisms.

4.1 An Algorithm for Precisely Detecting BroNIEs.

Given that applications commonly fail to validate untrusted inputs [27], it would be beneficial to have mechanisms for automatically preventing injection attacks.

Algorithm 1 : A BroNIE-preventing mechanism.

```
**Input:** Taint-tracking application $A$ and inputs $T$, $U$ (trusted, untrusted)
**Ensure:** $A$'s output is executed iff it doesn't exhibit a BroNIE
$Output \leftarrow A\ (T, \text{Taint}(U))$
$pgmTokens \leftarrow$ tokenize($Output$)
$temTokens \leftarrow$ tokenize($[\varepsilon/\underline{\sigma}]Output$)
MarkNoncodeToks($pgmTokens$)
$i \leftarrow j \leftarrow 1$
while $i \leq pgmTokens$.length and $j \leq temTokens$.length do
    if $pgmTokens[i] = temTokens[i]$ then
        $i \leftarrow i + 1$
        $j \leftarrow j + 1$
    else if $pgmTokens[i]$.isNoncode and $temTokens[i] \preceq pgmTokens[i]$ then
        $i \leftarrow i + 1$
        $j \leftarrow j + 1$
    else if $pgmTokens[i]$.isNoncode then
        $i \leftarrow i + 1$
    else
        throw $BronieException$
    end if
end while
// Handle any trailing noncode tokens in the program
while $i \leq pgmTokens$.length and $pgmTokens[i]$.isNoncode do
    $i \leftarrow i + 1$
end while
if $i > pgmTokens$.length and $j > temTokens$.length then
    Execute($Output$)
else
    throw $BronieException$
end if
```

At a high level, BroNIEs can be precisely and automatically prevented by:

– instrumenting the target application with a taint-tracking mechanism,
– interposing between the target application and the environment that evaluates the application's output programs,
– detecting whether output programs satisfy the NIE property, and
– only executing programs that do not exhibit BroNIEs.

Algorithm 1 is a psuedocode implementation of this mechanism. The algorithm detects BroNIEs by iterating through the sequences of tokens in the output program and its template while ensuring that the two token streams can be made equal subject to the constraints of Definition 8 (i.e., obtaining the sequence of tokens in the output program by only inserting noncode tokens into, or expanding noncode tokens within, the sequence of tokens in the program's template). Algorithm 1 relies on auxiliary functions for tainting untrusted inputs, tokenizing output programs, and marking which tokens are noncode.

4.2 Analysis of the BroNIE-detection Algorithm

The following theorems relate to the correctness and execution time of Algorithm 1.

Theorem 3. *Algorithm 1 executes output-program p iff p does not exhibit a BroNIE.*

Proof. We prove the if direction; the only-if direction is similar. By assumption, p satisfies the NIE property. Hence, the sequence of tokens in $[\varepsilon/\underline{\sigma}]p$ can be made equal to the sequence of tokens in p by expanding tokens in $[\varepsilon/\underline{\sigma}]p$ into noncode tokens in p, and/or inserting noncode tokens in p into $[\varepsilon/\underline{\sigma}]p$. Variables i and j are initialized to 1. For every token present in p and $[\varepsilon/\underline{\sigma}]p$, both i and j will be incremented by the first branch of the `if` statement (Lines 7–9). For every token in $[\varepsilon/\underline{\sigma}]p$ that needs to be expanded into a noncode token in p, i and j will be incremented by the second branch of the `if` statement (Lines 10–12). For every noncode token in p that needs to be inserted into $[\varepsilon/\underline{\sigma}]p$, i will be incremented by either the third branch of the `if` statement (Lines 13–14) or the second `while` loop (Lines 20–22). After both `while` loops have completed, both i and j will be greater than the lengths of their token sequences, so p will be executed on Line 24. $\square$

To examine the running time of Algorithm 1, we exclude the time taken to execute the target application (i.e., Line 1), as this operation is independent from the process of detecting BroNIEs. The parts of Algorithm 1 that are directly related to BroNIE detection are the calls to *tokenize* (Lines 2 and 3), the invocation of $MarkNoncodeToks$ (Line 4), the initialization of the i and j variables (Line 5), the two `while` loops (Lines 6–22), and the final `if` statement (Lines 23–27). To the best of our knowledge, every commonly-used programming language requires no more than linear time to partition tokens as code or noncode, so the proof of Theorem 4 assumes that the language-dependent $MarkNoncodeToks$ function runs in time linear in the number of tokens.

Theorem 4. *The BroNIE-detection part of Algorithm 1 (i.e., Lines 2–27) executes in $O(n)$ time, where n is the length of the output program.*

Proof. Each call to *tokenize* executes in $O(n)$ time and produces $O(n)$ tokens. After the program tokens are marked as either code or noncode (taking $O(n)$ time), the two `while` loops execute; each iteration takes constant time, and each loop executes $O(n)$ times. Afterward, if a BroNIE has not yet been detected, deciding whether a BroNIE has occurred takes constant time. Thus, Algorithm 1 runs in $O(n)$ time. $\square$

5 Discussion

This paper has presented BroNIEs, a general class of injection attacks in which injected symbols affect output programs beyond inserting or expanding noncode tokens. BroNIEs include not only all code-injection attacks on output programs (CIAOs), but also noncode-injection attacks on output programs.

In practice, precise detection of BroNIEs can be accomplished by using a taint-tracking mechanism (e.g., [7–9, 18, 19, 25, 26]) in conjunction with Algorithm 1. DIGLOSSIA [19] appears to be a good candidate framework for detecting and preventing BroNIEs in practice—DIGLOSSIA (which has not yet been publicly released) already detects and prevents CIAOs, so we expect that it could be modified to use Algorithm 1 to detect and prevent BroNIEs instead. Because DIGLOSSIA performs taint tracking through modified system libraries, existing applications need not be manually rewritten.

We believe that the definition of BroNIEs more closely matches our intuition of malicious injections than just *code* injections, due to the capability of attackers to cause malicious behavior by injecting noncode symbols (as demonstrated in Section 3).

Furthermore, it appears that the definition of BroNIEs imparts similar advantages as parameterized queries, without the significant disadvantages of requiring application-programmer compliance (which has historically been unreliable). Parameterized queries and BroNIE-preventing mechanisms both limit injections to inserting or expanding noncode tokens. However, whereas applications must be manually modified to use parameterized queries, BroNIEs can be automatically prevented by instrumenting applications and/or system libraries with a taint-tracking mechanism and running a lightweight BroNIE-detection algorithm prior to executing output programs.

References

1. The MITRE Corporation: CWE/SANS Top 25 Most Dangerous Software Errors. (2011) `http://cwe.mitre.org/top25/archive/2011/2011_cwe_sans_top25.pdf`.
2. Open Sourced Vulnerability Database: OSVDB: Open Sourced Vulnerability Database. (2014) `http://osvdb.org/`.
3. The OWASP Foundation: OWASP Top 10 - 2013. (2013) `http://owasptop10.googlecode.com/files/OWASPTop10-2013.pdf`.
4. Ray, D., Ligatti, J.: Defining code-injection attacks. In: Proceedings of the Symposium on Principles of Programming Languages (POPL). (2012) 179–190
5. Su, Z., Wassermann, G.: The essence of command injection attacks in web applications. In: Proceedings of the Symposium on Principles of Programming Languages (POPL). (2006) 372–382
6. Bisht, P., Madhusudan, P., Venkatakrishnan, V.N.: CANDID: Dynamic candidate evaluations for automatic prevention of SQL injection attacks. Transactions on Information and System Security (TISSEC) **13**(2) (2010) 1–39
7. Halfond, W., Orso, A., Manolios, P.: Wasp: Protecting web applications using positive tainting and syntax-aware evaluation. Transactions on Software Engineering (TSE) **34**(1) (2008) 65–81
8. Nguyen-Tuong, A., Guarnieri, S., Greene, D., Shirley, J., Evans, D.: Automatically hardening web applications using precise tainting. In: Proceedings of the International Information Security Conference (SEC). (2005) 372–382

9. Xu, W., Bhatkar, S., Sekar, R.: Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In: Proceedings of the USENIX Security Symposium. (2006) 121–136
10. Bravenboer, M., Dolstra, E., Visser, E.: Preventing injection attacks with syntax embeddings. Science of Computer Programming (SCP) **75**(7) (2010) 473–495
11. Hansen, R., Patterson, M.: Stopping injection attacks with computational theory. In: Black Hat Briefings Conference. (2005)
12. Oracle: How to write injection-proof PL/SQL. An Oracle White Paper (2008)
13. Jovanovic, N., Kruegel, C., Kirda, E.: Precise alias analysis for static detection of web application vulnerabilities. In: Proceedings of the Workshop on Programming Languages and Analysis for Security. (2006) 27–36
14. Buehrer, G., Weide, B.W., Sivilotti, P.A.G.: Using parse tree validation to prevent SQL injection attacks. In: Proceedings of the Workshop on Software Engineering and Middleware (SEM). (2005) 106–113
15. Kieżun, A., Guo, P.J., Jayaraman, K., Ernst, M.D.: Automatic creation of SQL injection and cross-site scripting attacks. In: Proceedings of the International Conference on Software Engineering (ICSE). (2009) 199–209
16. Luo, Z., Rezk, T., Serrano, M.: Automated code injection prevention for web applications. In: Proceedings of the Conference on Theory of Security and Applications (TOSCA). (2011) 186–204
17. Magazinius, J., Rios, B.K., Sabelfeld, A.: Polyglots: Crossing origins by crossing formats. In: Proceedings of the Conference on Computer and Communications Security (CCS). (2013) 753–764
18. Pietraszek, T., Berghe, C.V.: Defending against injection attacks through context-sensitive string evaluation. In: Proceedings of Recent Advances in Intrusion Detection (RAID). (2005) 124–145
19. Son, S., McKinley, K.S., Shmatikov, V.: Diglossia: detecting code injection attacks with precision and efficiency. In: Proceedings of the Conference on Computer and Communications Security (CCS). (2013) 1181–1192
20. The Open Web Application Security Project (OWASP): SQL injection prevention cheat sheet (2012) `https://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet`.
21. The PostgreSQL Global Development Group: PostgreSQL 9.3.4 Documentation. (2014) `http://www.postgresql.org/docs/9.3/static/index.html`.
22. Microsoft: Transact-SQL Reference (Database Engine). (2014) `http://msdn.microsoft.com/en-us/library/bb510741(v=sql.120).aspx`.
23. Oracle: MySQL 5.7 Reference Manual. (2014) `https://dev.mysql.com/doc/refman/5.7/en/index.html`.
24. Oracle: Oracle Database SQL Language Reference, 12c Release 1 (12.1). (2013) `http://docs.oracle.com/cd/E16655_01/server.121/e17209.pdf`.
25. Dalton, M., Kannan, H., Kozyrakis, C.: Raksha: A flexible information flow architecture for software security. In: Proceedings of the International Symposium on Computer Architecture (ISCA). (2007) 482–493
26. Clause, J., Li, W., Orso, A.: Dytan: a generic dynamic taint analysis framework. In: Proceedings of the International Symposium on Software Testing and Analysis (ISSTA). (2007) 196–206
27. Scholte, T., Robertson, W., Balzarotti, D., Kirda, E.: An empirical analysis of input validation mechanisms in web applications and languages. In: Proceedings of the Symposium on Applied Computing (SAC). (2012) 1419–1426