Generating String Inputs Using Constrained Symbolic Execution

Pieter Hooimeijer
University of Virginia
The Problem: SQL Injection
Motivation Montage

HI, THIS IS YOUR SON’S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.

OH, DEAR - DID HE BREAK SOMETHING?
IN A WAY-

DID YOU REALLY NAME YOUR SON Robert)? DROP TABLE Students;-- ?

OH, YES. LITTLE BOBBY TABLES, WE CALL HIM.

WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.
Mitre Corp. data reported on http://www.attrition.org/
Motivation Montage

www.xkcd.com

Report Frequency

Mitre Corp. data; reported on http://www.attrition.org/
Motivation Montage

“String variables have lost their innocence…” [Thiemann05]
/$userid$ is untrusted

if (!eregi('[0-9]+', $userid)) {
    unp_msg('You entered an invalid user ID.');
    exit;
}

$user$ = $DB->query("SELECT * FROM `unp_user`.
    "WHERE userid='{$userid}'");

if (!DB->is_single_row($user)) {
    unp_msg('You entered an invalid user ID.');
    exit;
}
Motivation Montage

// $userid is untrusted

if (!eregi('[0-9]+', $userid)) {
    unp_msg('You entered an invalid user ID.');
    exit;
}

$user = $DB->query("SELECT * FROM `unp_user` WHERE userid='".$userid.'"');

if (!$DB->is_single_row($user)) {
    unp_msg('You entered an invalid user ID.');
    exit;
}

Matches any string that contains a sequence of digits...
```
// $userid is untrusted

if (!eregi('[0-9]+', $userid)) {
    unp_msg('You entered an invalid user ID.');
    exit;
}

$user = $DB->query(
    'SELECT * FROM `unp_user` WHERE userid=1;
    DROP TABLE unp_user;
    --'
) .

if (!$DB->is_single_row($user)) {
    unp_msg('You entered an invalid user ID.');
    exit;
}
```

Mitre Corp. data; reported on http://www.attrition.org/
“String variables have lost their innocence...”
[Thiemann05]
The Plan

- Wassermann and Su '07:
  - detect SQL Command Injection Vulnerabilities in real PHP code
  - Input: PHP code
  - Output: Context-Free Grammar

- Plan: Extend this to generate *actual attack inputs*
Why?
Why?

Our Output:

STRING: 004 =\$251 =\$1486.
  == """" (len 0)

STRING: 013 =\$558 =\$559 =\$561.
  '99 == ""99"" (len 3)

ANSWER: $$558 ==
  '99 == ""99"" (len 3)
Up Next

1. Describe Wassermann and Su '07

2. How to run PHP code backwards
WSU: An Example

Some Code:

```c
x = 'z';

while(n < 5) {
    x = '(' . x . ')';
    n ++;
}
```

- We want a context-free grammar to model `x`
- Suppose we don't know anything about `n`
Some Code:

```plaintext
x = 'z';

while (n < 5) {
    x = '(' . x . ')';
    n ++;
}
```

Grammar:

```
A -> z
```
WSU: An Example

Some Code:

```c
x = 'z';

while(n < 5) {
    x = '('* . x . ')';
    n ++;
}
```

Grammar:

```
A  ->  z
```
Some Code:

```c
x = 'z';

while(n < 5) {
  > x = '(' . x . ')';
  n ++;
}
```

Grammar:

```grammar
A -> z
B -> (A)
```
The Nugget

Some Code:

```java
x = 'z';

while (n < 5) {
    x = '(' + x + ')';
    n ++;
}
```

Grammar:

```
A -> z
B -> (A)
C -> A | B
```
WSU: Yet More Example

Some Code:

```c
x = 'z';

while(n < 5) {
    x = '(' . x . ')';
    n ++;
}>
```
Some Code:

```c
char x = 'z';

while(n < 5) {
    x = '(' . x . ')';
    n ++;
}
```

Grammar:

```
X -> C
A -> z
B -> (C)
C -> A | B
```
Is that good?

• Can model home-grown string sanitizing functions using finite state transducers [Minamide05]

• Does not require programmer assistance; always terminates

• 20% False positives; output may be difficult to interpret
Ur instrucshuns
unningessary

r unnesussary
Before:

Source → Wassermann & Su → Context-Free Grammar
After:

Source → Wassermann & Su → Context-Free Grammar + Program Locations → String(s) → Execution Path(s) → Actual Inputs
After:

Source → Wasserman & Su → Context-Free Grammar + Program Locations

Contributions

String(s) → Execution Path(s) → Actual Inputs
Contributions

• Add a mapping from context-free grammar back to the source

• Use mapping to find bad strings and execution paths

• Use symbolic execution to reverse string operations along a path
Up Next:

Source → Wassermann & Su → Context-Free Grammar + Program Locations

String(s) → Execution Path(s) → Actual Inputs
\texttt{p = 'y';}

\texttt{if (myth()) { }
    \texttt{p = 'xyzz'. p; }
\texttt{}}

\texttt{print p;}

\texttt{P \rightarrow Q}
\texttt{P \rightarrow R}
\texttt{Q \rightarrow xyzzzR}
\texttt{R \rightarrow y}
Grammar Annotations

Source → Wassermann & Su → Context-Free Grammar + Program Locations

Actual Inputs

\[
\begin{align*}
p = 'y'; \\
\text{if (myth())} & \{ \\
\quad p = 'xyzz' . p; \\
\}
\end{align*}
\]

\[
\begin{align*}
P & \rightarrow Q \quad \text{[TRUE]} \\
P & \rightarrow R \quad \text{[!myth]} \\
Q & \rightarrow xyzzR \quad \text{[myth]} \\
R & \rightarrow y \quad \text{[TRUE]}
\end{align*}
\]
Up Next:

Source → Wassermann & Su → Context-Free Grammar + Program Locations

- String(s)
- Execution Path(s)

Actual Inputs
How to Find Inputs

1. Create a **dependency graph**

2. Detect **implicit cycles**

3. Solve constraints
Creating a Dependency Graph

1   $userid = $_POST['uid'];
2   if (!eregi('[0-9]+', $userid)) {
3       exit;
4   }
5
6   query("SELECT * FROM `unp_user`.
7     "WHERE userid=" . $userid");

Path: 1-2-4-6
Creating a Dependency Graph

Path: 1-2-4-6

```php
$userid = $_POST['uid'];
if (!eregi('[0-9]+', $userid)) {
    exit;
}
query("SELECT * FROM `unp_user`
    WHERE userid=" . $userid);
```
$userid = $_POST['uid'];
if (!eregi('[0-9]+', $userid)) {
    exit;
}
query("SELECT * FROM `unp_user` WHERE userid=": $userid);
SELECT * FROM `unp_user` WHERE userid=1 OR 1=1
Solving Constraints

SELECT ... WHERE userid ...

SELECT * FROM `unp_user` WHERE userid=1 OR 1=1
Solving Constraints

SELECT ... WHERE userid = ...

SELECT * FROM `unp_user` WHERE userid=1 OR 1=1
Solving Constraints

SELECT ... WHERE userid = ...

SELECT * FROM `unp_user` WHERE userid=1 OR 1=1
Solving Constraints

SELECT * FROM `unp_user` WHERE userid=1 OR 1=1
Solving Basic Constraints

SELECT ... WHERE userid = Σ*[0-9]+Σ*

SELECT * FROM `unp_user` WHERE userid=1 OR 1=1
Problem: Backward Propagation

How do we map $1 \ OR \ 1=1$ back onto $u$?

SELECT ... WHERE userid = ...

SELECT * FROM `unp_user` WHERE userid=1 OR 1=1
// a and b are user inputs

if (!$ereg('o(pp)+', a)) { exit; }
if (!$ereg('p*q', b)) { exit; }

\[ d = a . b; \] // concat
if (!$ereg('oppppq', d) { exit; }
Concat-Intersect Example I

$L_1 = o(pp)^+$

$L_2 = p^*q$

$L_3 = op^4q$
Solution I: $L_1' = opp$
$L_2' = ppq$
Solution I: $L_1' = opp \quad L_2' = ppq$

Solution II: $L_1' = oppppp \quad L_2' = q$
## Evaluation

- Found inputs for 17/22 vulnerabilities

<table>
<thead>
<tr>
<th>Name</th>
<th>Version</th>
<th>Description</th>
<th>files Total</th>
<th>loc Total</th>
<th>Vulnerable</th>
</tr>
</thead>
<tbody>
<tr>
<td>e107</td>
<td>0.7.5</td>
<td>content management</td>
<td>741</td>
<td>132,862</td>
<td>N/A</td>
</tr>
<tr>
<td>eve</td>
<td>1.0</td>
<td>activity tracker</td>
<td>8</td>
<td>905</td>
<td>3</td>
</tr>
<tr>
<td>tiger</td>
<td>1.0 beta 39</td>
<td>news management</td>
<td>30</td>
<td>6,701</td>
<td>0</td>
</tr>
<tr>
<td>utopia</td>
<td>1.3.0</td>
<td>news management</td>
<td>24</td>
<td>5,438</td>
<td>5</td>
</tr>
<tr>
<td>warp</td>
<td>1.2.1</td>
<td>content management</td>
<td>44</td>
<td>24,365</td>
<td>14</td>
</tr>
</tbody>
</table>

Min: 1s  
Avg: 86s  
Med: 36s  
Max: 697s
Conclusion

- We presented a general constraint-solving approach for string variables
- It can find inputs for SQL injections within a reasonable time
- We used a three-stage algorithm:
  1. Generate annotated grammar
  2. Search for strings and associated paths
  3. Solving (cyclic) constraints over strings