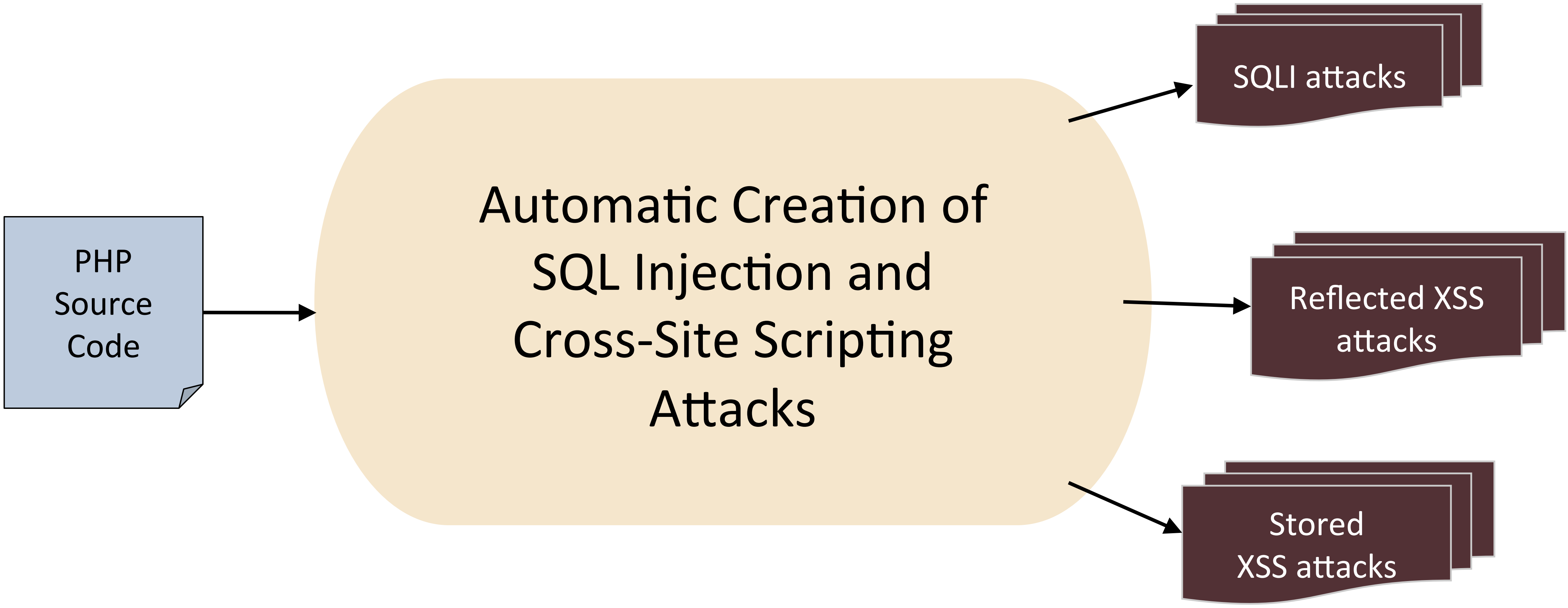
Detecting Vulnerabilities in Web Code with concolic execution

Suman Jana

*slides are adapted from Adam Kiezun, Philip J. Guo, Karthick Jayaraman, and Michael D. Ernst



Ardilla by Kiezun et al. [ICSE 2009]

Overview

Problem:

Finding security vulnerabilities (SQLI and XSS) in Web applications

Approach:

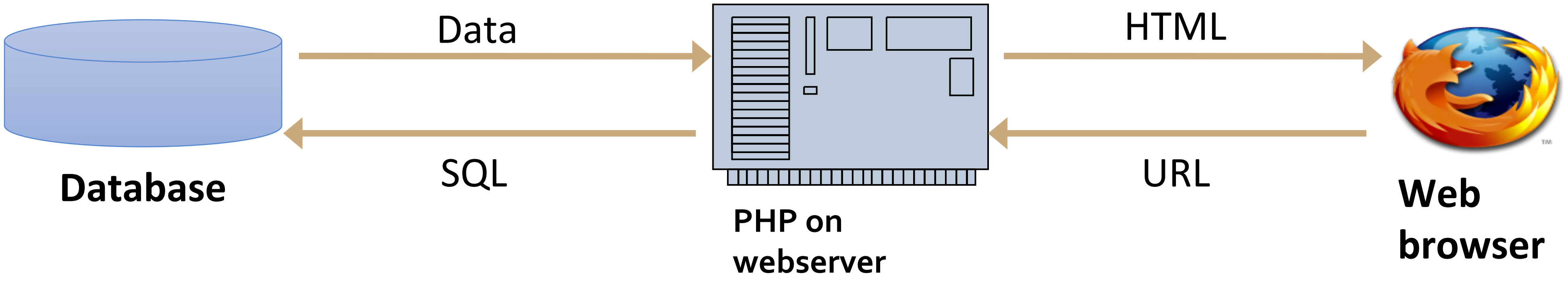
1. Automatically generate inputs
2. Dynamically track taint
3. Mutate inputs to produce exploits

Results:

- 60 unique new vulnerabilities in 5 PHP applications
- first to create 2nd-order XSS, no false positives

PHP Web applications

http://www.example.com/register.php?name=Bob&age=25



Example: Message board (add mode)

```
if ($_GET['mode'] == "add")
  addMessageForTopic();
else if ($_GET[ ‘mode’ ] == “display”)
  displayAllMessagesForTopic();
else
  die(“Error: invalid mode”);
```

```
$_GET[]:
  mode = “add”
  msg = “hi there”
  topicID = 42
  poster = “Bob”
```

Thanks for posting, Bob

```
function addMessageForTopic() {
  $my_msg =     $_GET['msg'];
  $my_topicID = $_GET['topicID'];
  $my_poster =  $_GET['poster'];

  $sqlstmt = ” INSERT INTO messages
    VALUES('$my_msg’ , '$my_topicID') ";

  $result = mysql_query($sqlstmt);
  echo "Thanks for posting, $my_poster”; }
```

Example: Message board (display mode)

```
if ($_GET['mode'] == "add")
  addMessageForTopic();
else if ($_GET[ ‘mode’ ] == “display”)
  displayAllMessagesForTopic();
else
  die(“Error: invalid mode”);
```

```
$_GET[]:
  mode = “display”
  topicID = 42545646546
```

Message: hi there

```
function displayAllMessagesForTopic() {
  $my_topicID = $_GET['topicID'];
  $sqlstmt = ” SELECT msg FROM messages WHERE
    topicID='$my_topicID’ ";
  $result = mysql_query($sqlstmt);

  while($row = mysql_fetch_assoc($result)) {
    echo "Message: " . $row['msg']; }
}
```

SQL injection attack

```
if ($_GET['mode'] == "add")
  addMessageForTopic();
else if ($_GET[ 'mode' ] == "display")
  displayAllMessagesForTopic();
else
  die("Error: invalid mode");
```

```
$_GET[]:
  mode = "display"
  topicID = 1' OR '1'='1
```

```
function displayAllMessagesForTopic() {
  $my_topicID = $_GET['topicID'];
  $sqlstmt = " SELECT msg FROM messages WHERE
    topicID='$my_topicID' ";
  $result = mysql_query($sqlstmt);

  while($row = mysql_fetch_assoc($result)) {
   echo "Message: " . $row['msg']; }
}
```

```
SELECT msg FROM messages WHERE topicID='1' OR '1'='1'
```

Reflected XSS attack

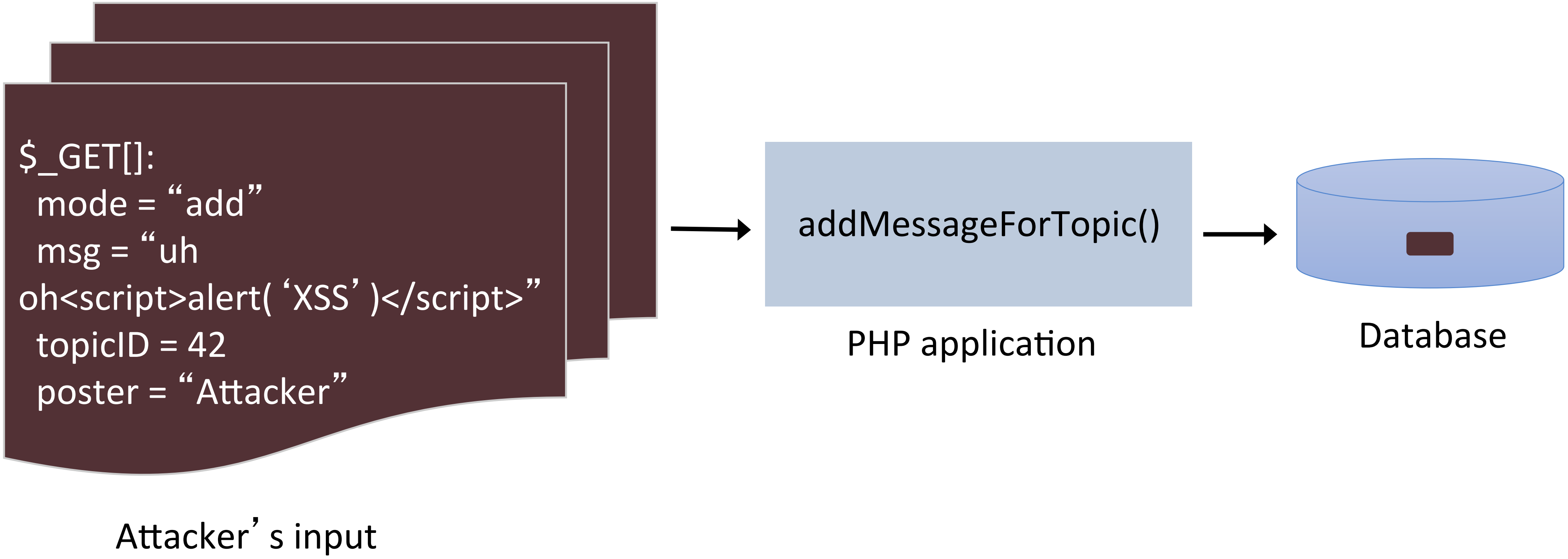
```
if ($_GET['mode'] == "add")
  addMessageForTopic();
```

```
$_GET[]:
  mode = “add”
  msg = “hi there”
  topicID = 42
  poster = “uh
oh<script>alert( ‘XSS’ )</script>”
```

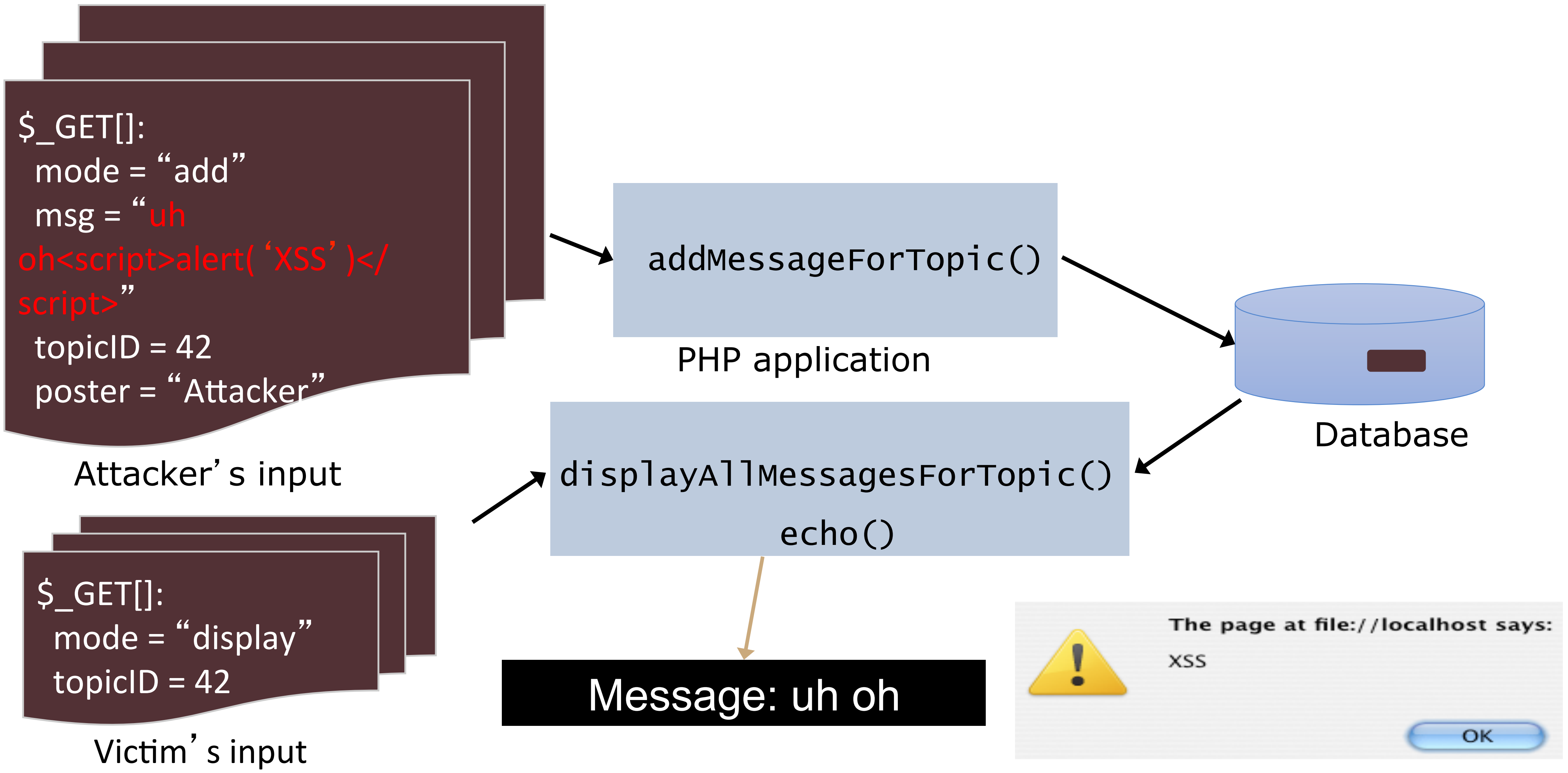
```
function addMessageForTopic() {
  $my_poster =  $_GET['poster'];
  [...]
  echo "Thanks for posting, $my_poster";
}
```

Thanks for posting, uh oh

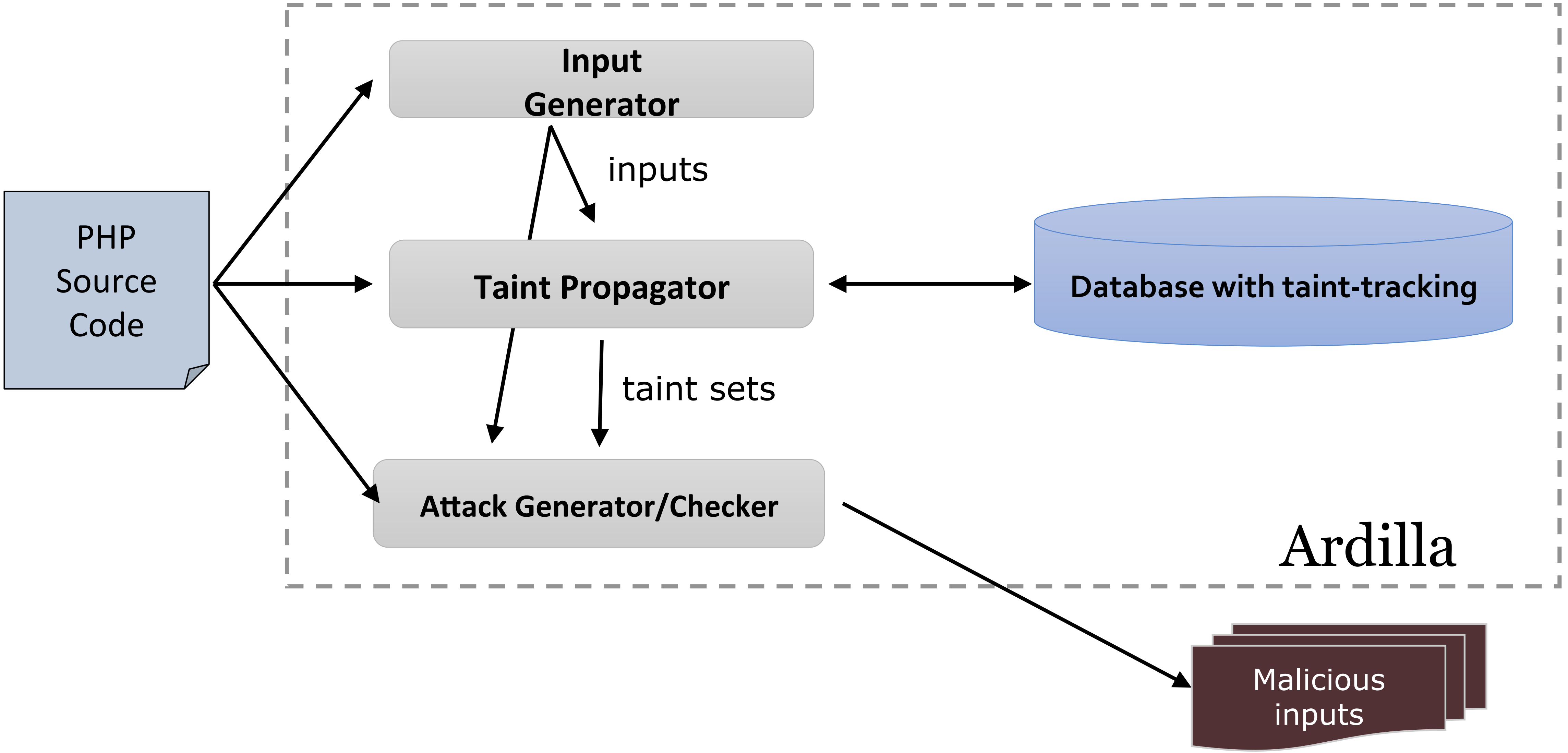
Stored XSS attack



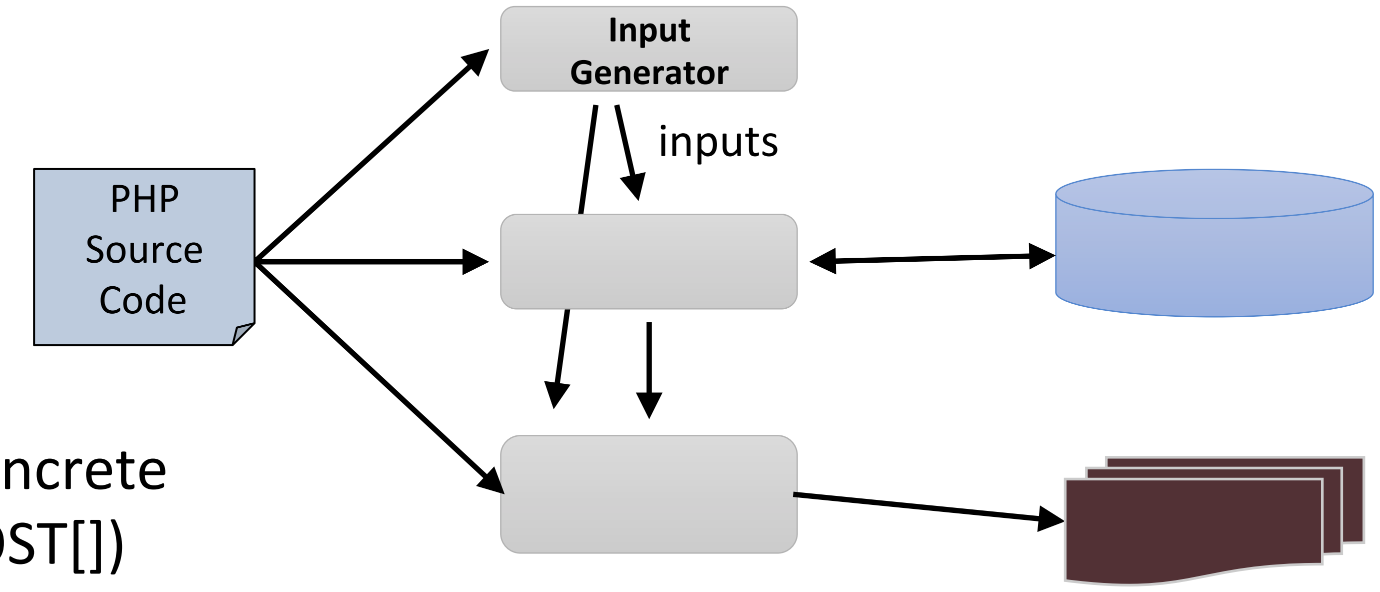
Stored XSS attack



Architecture



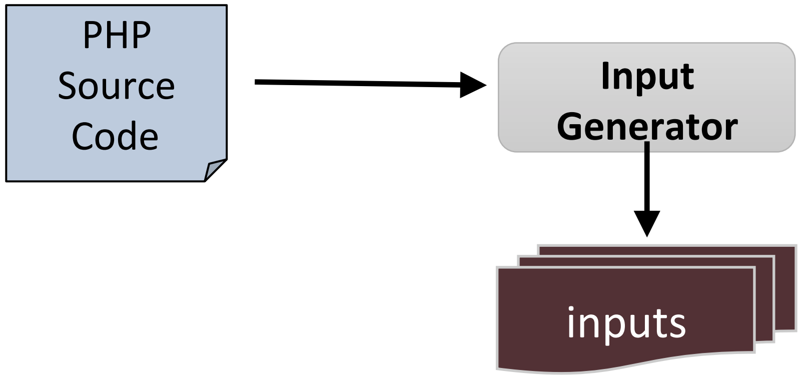
Input generation



Goal: Create a set of concrete inputs (_$GET[] & _$POST[]) based on concolic execution

Input generation: concolic execution

```
if ($_GET['mode'] == "add")
  addMessageForTopic();
else if ($_GET[‘mode’] == “display”)
  displayAllMessagesForTopic();
else
  die(“Error: invalid mode”);
```




```
$_GET[]:
  mode = “1”
  msg = “1”
  topicID = 1
  poster = “1”
```

```
$_GET[]:
  mode = “add”
  msg = “1”
  topicID = 1
  poster = “1”
```

```
$_GET[]:
  mode = “display”
  msg = “1”
  topicID = 1
  poster = “1”
```

Example: SQL injection attack

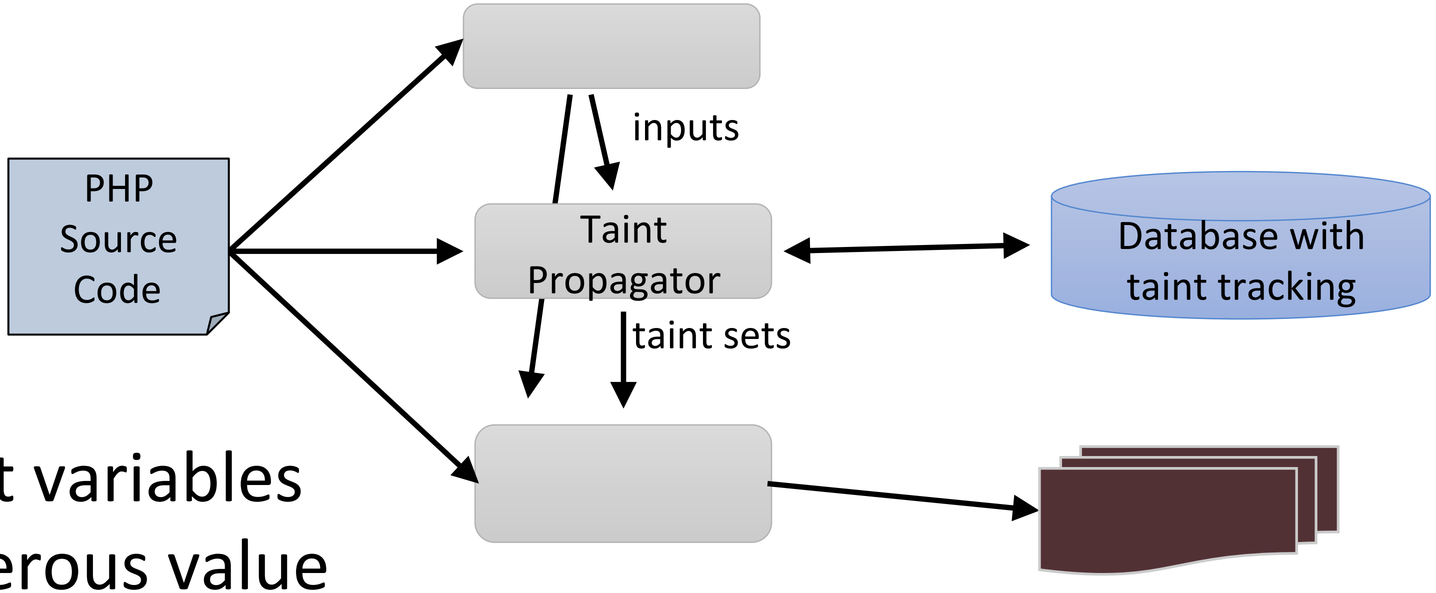
1. **Generate** inputs until program reaches an SQL statement

```
SELECT msg FROM messages WHERE topicID='$my_topicID‘
```

```
function displayAllMessagesForTopic() {
  $my_topicID = $_GET['topicID'];
  $sqlstmt = ” SELECT msg FROM messages WHERE
    topicID='$my_topicID’ ";
  $result = mysql_query($sqlstmt);
```

```
$_GET[]:
  mode = “display”
  msg = “1”
  topicID = 1
  poster = “1”
```

Taint propagation

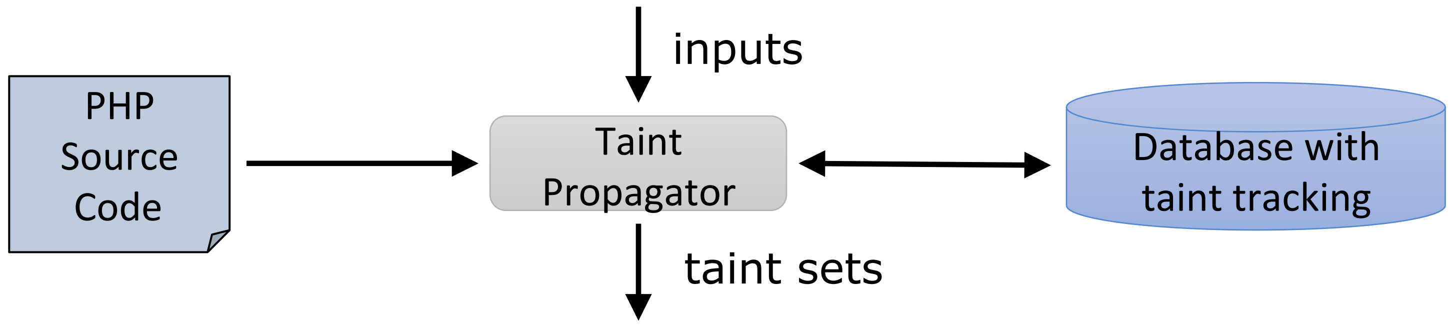


Goal: Determine which input variables affect each potentially dangerous value

Technique: Execute and track data-flow from input variables to *sensitive sinks*

Sensitive sinks: mysql_query(), echo(), print()

Taint propagation: data-flow



Each value has a **taint set**, which contains input *variables* whose values flow into it

```
function displayAllMessagesForTopic() {
  $my_topicID = $_GET['topicID'];
  $sqlstmt = " SELECT msg FROM messages WHERE
    topicID='$my_topicID' ";
  $result = mysql_query($sqlstmt); /* { 'topicID' } */
```

Sensitive sink

Taint set

Taint propagation

- Assignments: $my_poster = $_GET["poster"]
- String concatenation: $full_n = $first_n . $last_n
- PHP built-in functions: $z = foo($x, $y)
- Database operations (for stored XSS)

Example: SQL injection attack

1. **Generate** inputs until program reaches an SQL statement

SELECT msg FROM messages WHERE topicID='$my_topicID '

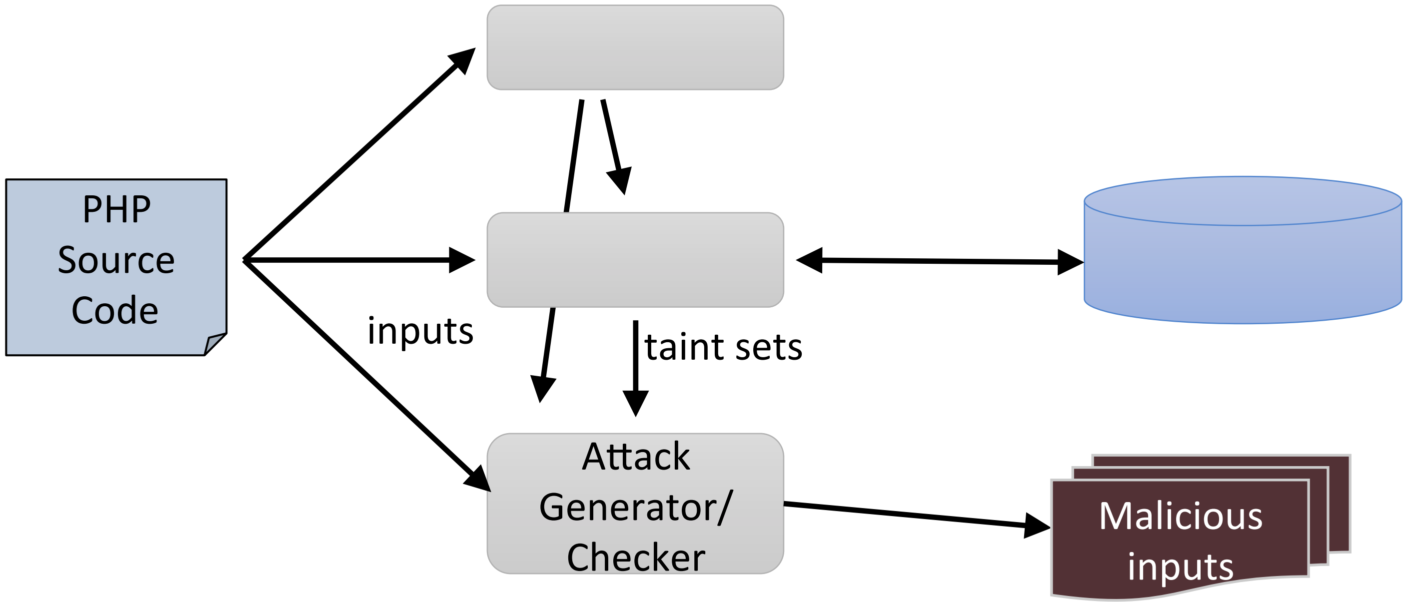
2. **Collect taint sets** for values in sensitive sinks: { ' topicID' }

```
function displayAllMessagesForTopic() {
  $my_topicID = $_GET['topicID'];
  $sqlstmt = "SELECT msg FROM messages WHERE topicID='$my_topicID' ";
  $result = mysql_query($sqlstmt); /* { 'topicID' } */
```

Sensitive sink

Taint set

Attack generation and checking



Goal: Generate attacks for each sensitive sink

Technique: Mutate inputs into candidate attacks

- Replace tainted input variables with shady strings developed by security professionals:
  - e.g., "1' or '1' = '1", "<script>*code*</script>"

Alternative: Use a string constraint solver

Attack generation and checking

Given a program, an input i, and taint sets

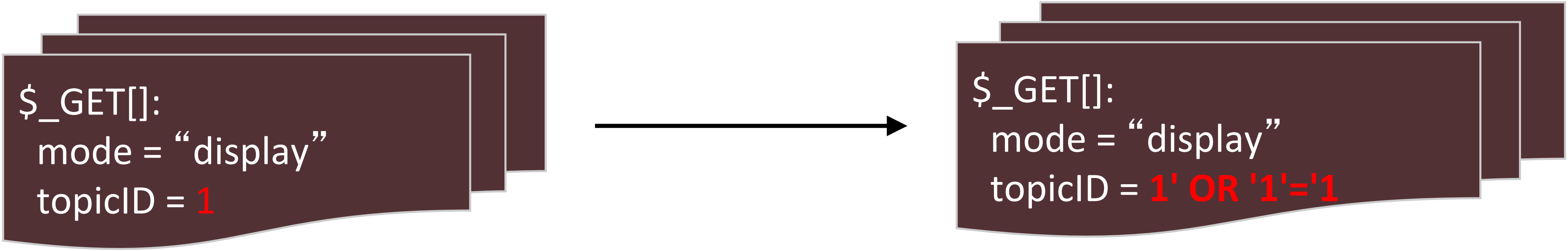
```
for each var that reaches any sensitive sink:
  res = exec(program, i)
for shady in shady_strings:
  mutated_input = i.replace(var, shady)
  mutated_res = exec(program, mutated_input)
if mutated_res DIFFERS FROM res:
  report mutated_input as attack
```

Attack generation

Attack checking

Attack generation: mutating inputs

```
res = exec(program, i)
for shady in shady_strings:
  mutated_input = i.replace(var, shady)
  mutated_res = exec(program, mutated_input)
  if mutated_res DIFFERS FROM res:
    report mutated_input as attack
```



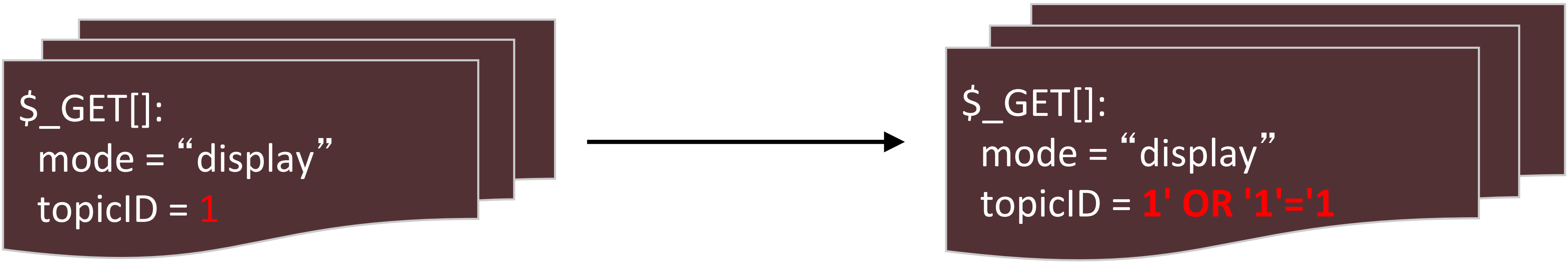
Example: SQL injection attack

1. **Generate** inputs until program reaches an SQL statement

SELECT msg FROM messages WHERE topicID='$my_topicID '

2. **Collect taint sets** for values in sensitive sinks: { ’ topicID’ }

3. **Generate** attack candidate by picking a shady string



Attack checking: diffing outputs

```
res = exec(program, i)
for shady in shady_strings:
  mutated_input = i.replace(var, shady)
  mutated_res = exec(program, mutated_input)
  if mutated_res DIFFERS FROM res:
    report mutated_input as attack
```

What is a significant difference?

- For SQLI: compare SQL parse tree *structure*
- For XSS: compare HTML for additional script-inducing elements (<script>)

Avoids false positives from input sanitizing and filtering

Example: SQL injection attack

1. **Generate** inputs until program reaches an SQL statement

   SELECT msg FROM messages WHERE topicID='$my_topicID ‘

2. **Collect taint sets** for values in sensitive sinks: { ’ topicID’ }
3. **Generate** attack candidate by picking a shady string
4. **Check** by mutating input and comparing SQL parse trees:

   innocuous: SELECT msg FROM messages WHERE topicID= ‘1’

   mutated: SELECT msg FROM messages WHERE topicID= ‘1’ OR ‘1’ = ‘1’

5. **Report** an attack since SQL parse tree *structure* differs

Experimental results

Name	Type	LOC	SourceForge Downloads
SchoolMate	School administration	8,181	6,765
WebChess	Online chess	4,722	38,457
FaqForge	Document creator	1,712	15,355
EVE activity tracker	Game player tracker	915	1,143
geccBBlite	Bulletin board	326	366

Vulnerability Kind	Sensitive sinks	Reached sensitive sinks	Unique attacks
SQLI	366	91	**23**
1^{st}-order XSS	274	97	**29**
2^{nd}-order XSS	274	66	**8**

Main limitation: input generator

Total: **60**

Comparison with other approaches

Defensive coding:
+ : can completely solve problem if done properly
- : must re-write existing code

Static analysis:
+ : can potentially prove absence of errors
- : false positives, does not produce concrete attacks

Dynamic monitoring:
+ : can prevent all attacks
- : runtime overhead, false positives affect app. behavior

Random fuzzing:
+ : easy to use, produces concrete attacks
- : creates mostly invalid inputs

Summary

- Automatically create SQLI and XSS attacks
- Technique
  - Dynamically track taint through both program and database
  - Input mutation and output comparison
- Implementation and evaluation
  - Found 60 new vulnerabilities, no false positives