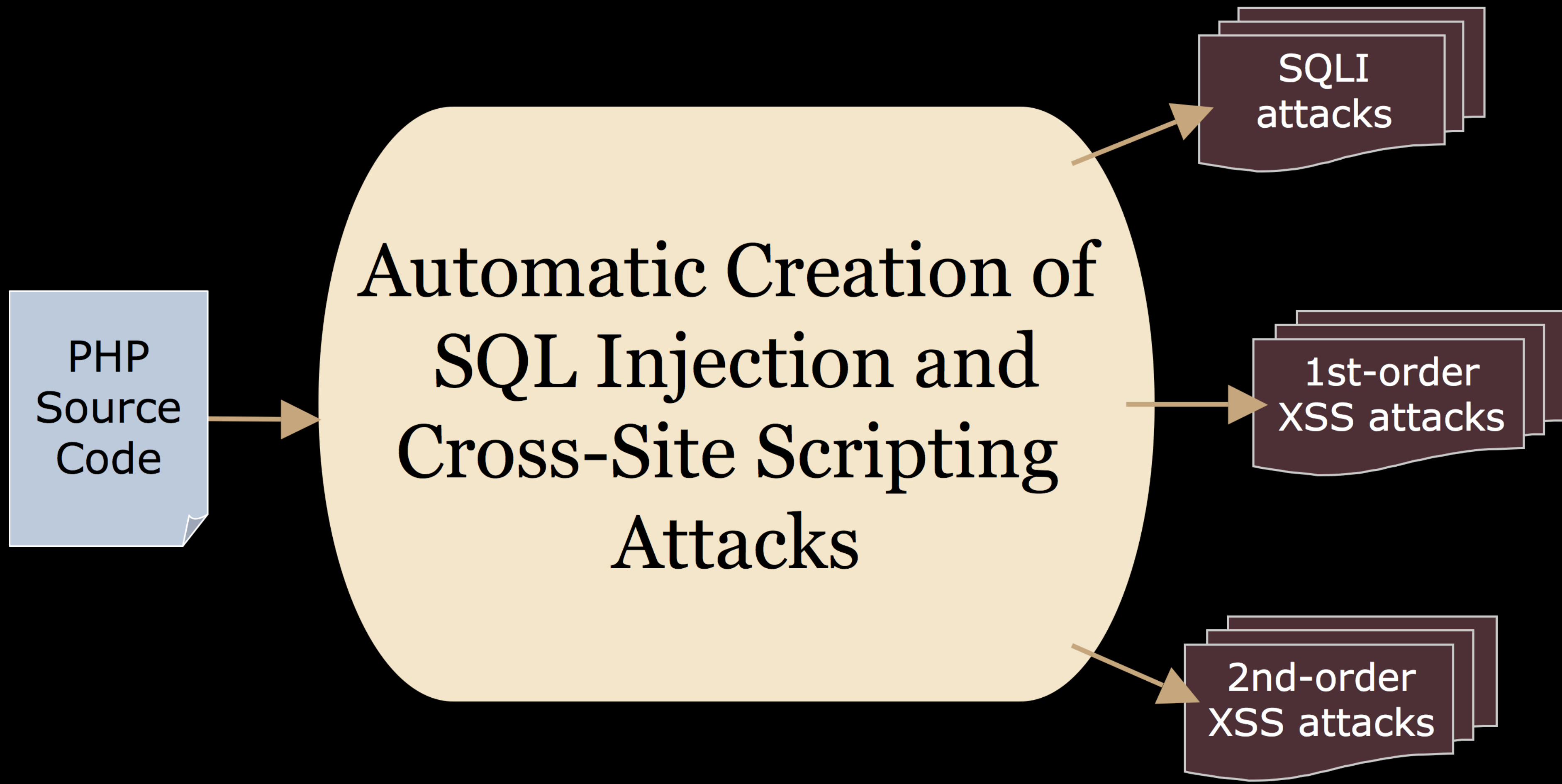
Slides for a 25-minute conference presentation on this paper:

Adam Kiezun, Philip J. Guo, Karthick Jayaraman, Michael D. Ernst. Automatic Creation of SQL Injection and Cross-site Scripting Attacks. In Proceedings of the 2009 IEEE International Conference on Software Engineering (ICSE), May 2009.

Created by Philip J. Guo
pg@cs.stanford.edu

Automatic Creation of SQL Injection and Cross-Site Scripting Attacks



Adam Kiezun, Philip J. Guo, Karthick Jayaraman, Michael D. Ernst

International Conference on Software Engineering
May 20, 2009

Overview

Problem:

Finding security vulnerabilities (SQLI and XSS) in Web applications

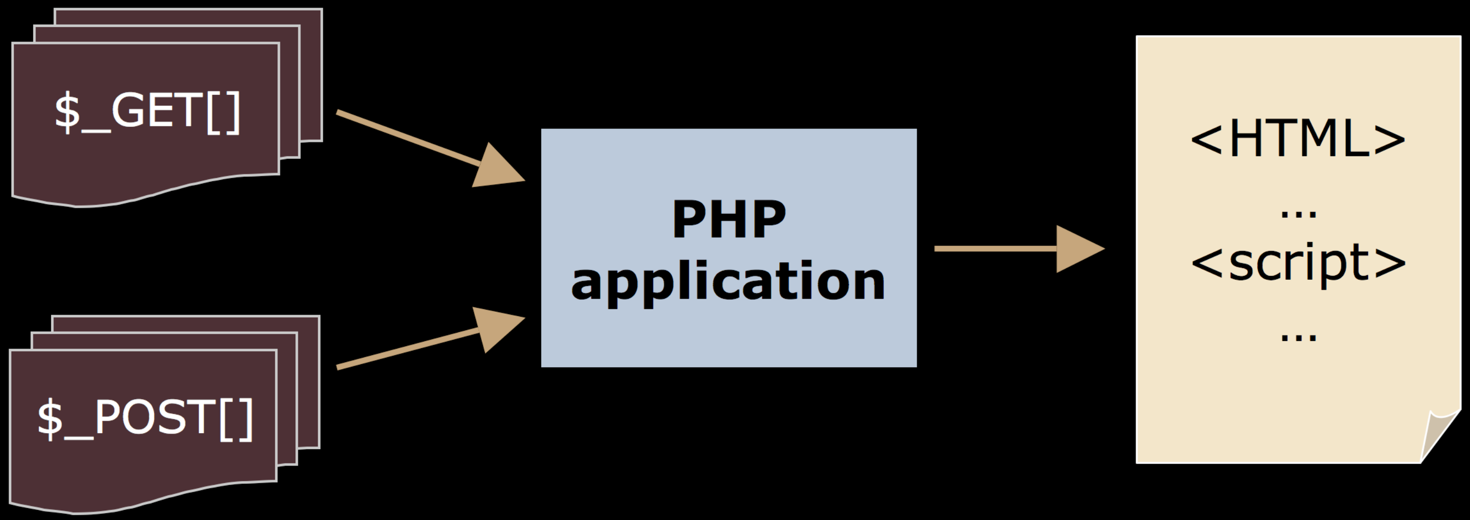
Approach:

1. Automatically generate inputs
2. Dynamically track taint (through program and DB)
3. Mutate inputs to produce exploits

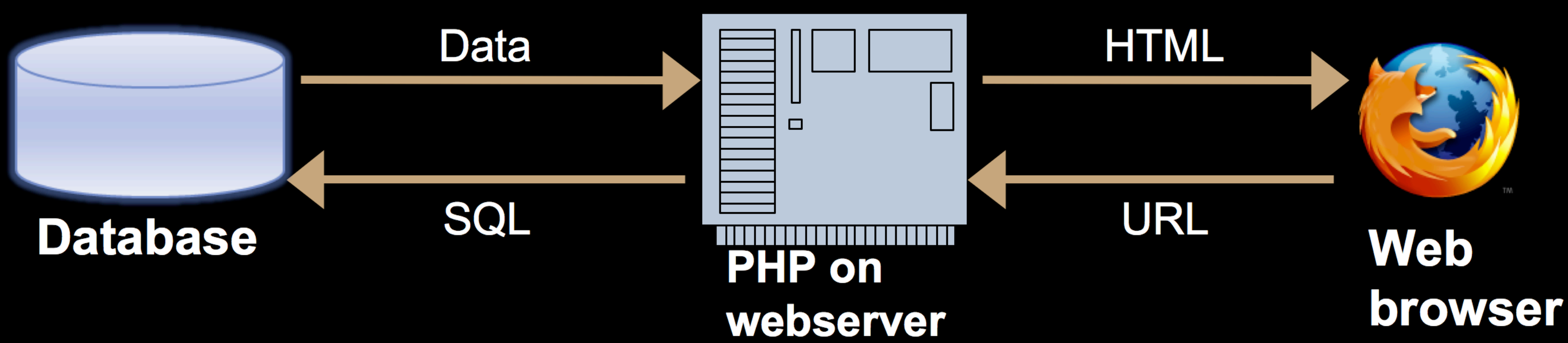
Results:

60 unique new vulnerabilities in 5 PHP applications, 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 = 42
```

```
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'];
}}
```

Message: hi there

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'
```

First-order XSS attack

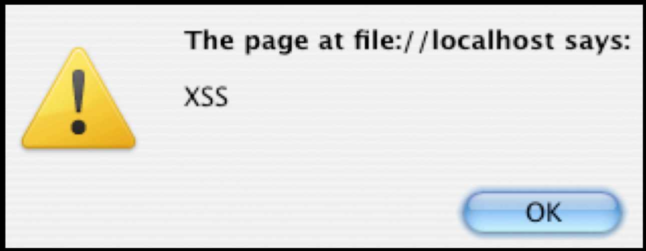
```
if ($_GET['mode'] == "add")
  addMessageForTopic();
```

```
$_GET[]:
  mode = "add"
  msg = "hi there"
  topicID = 42
  poster = MALICIOUS
```

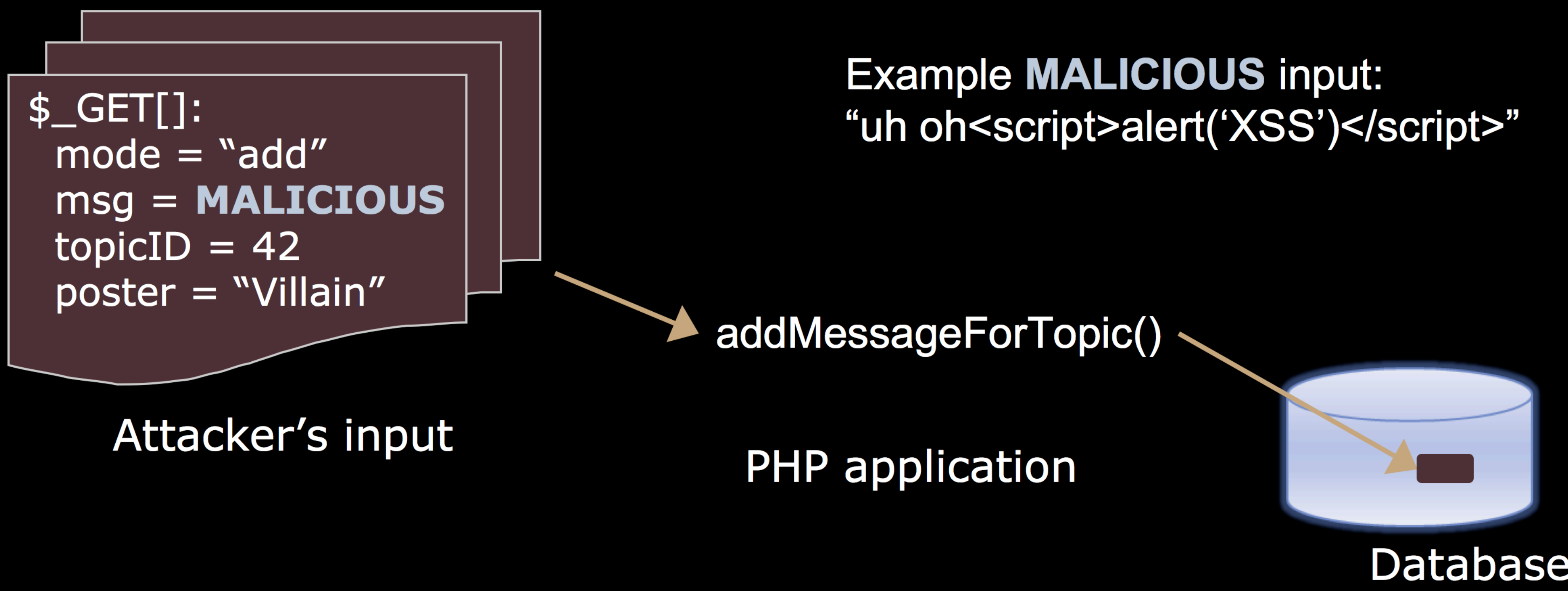
Example **MALICIOUS** input:
“uh oh<script>alert(‘XSS’)</script>”

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

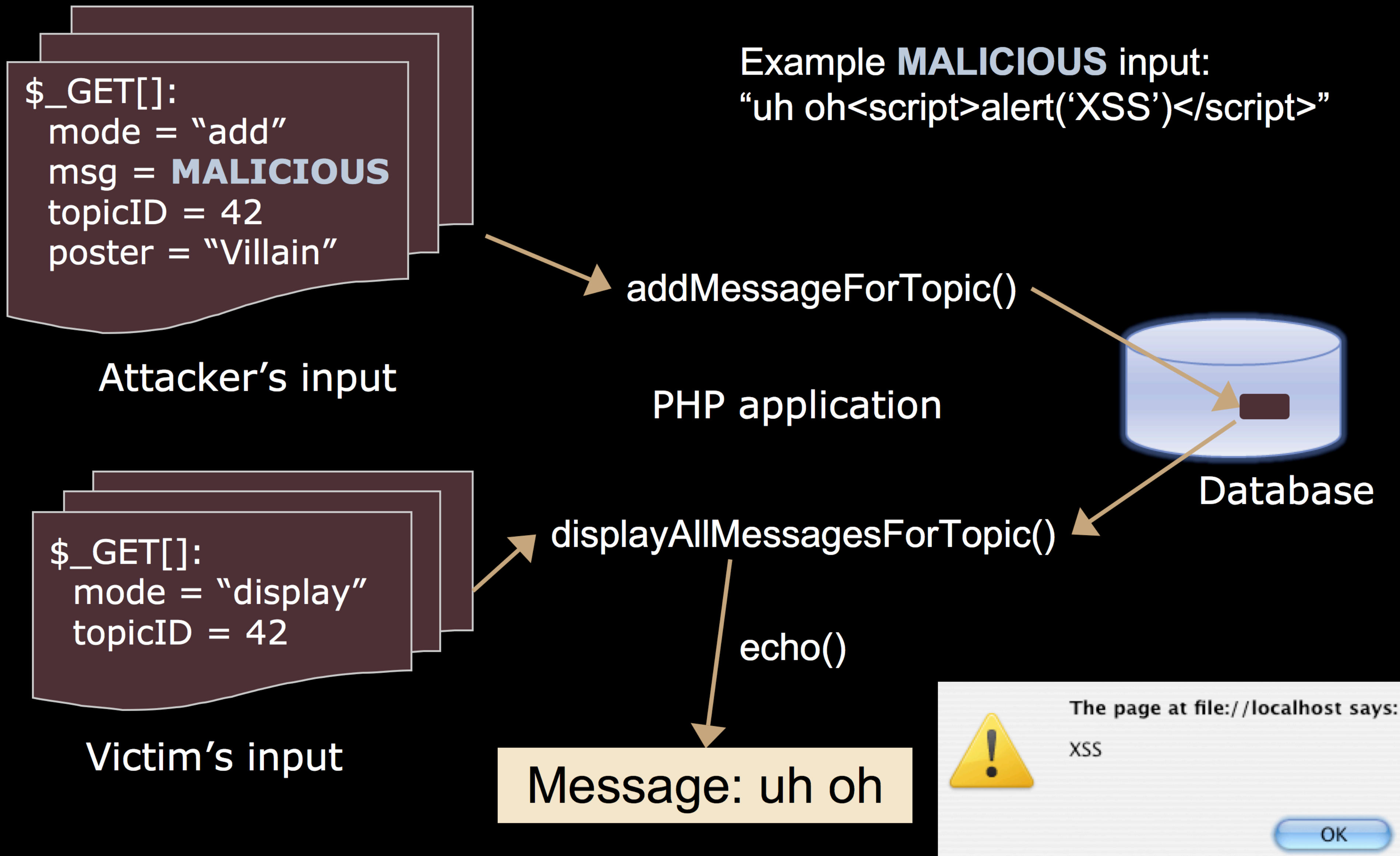
Thanks for posting, uh oh



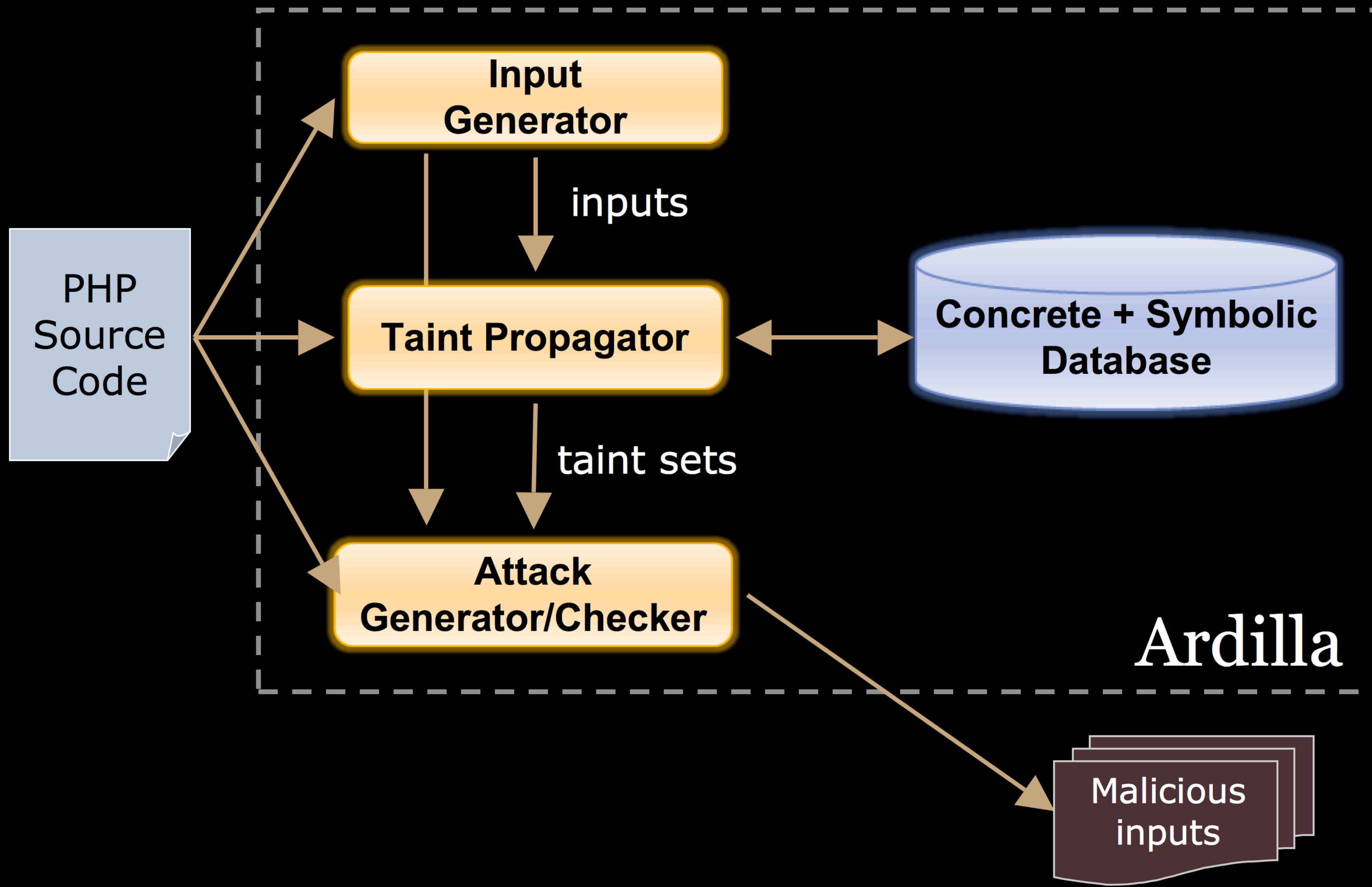
Second-order XSS attack



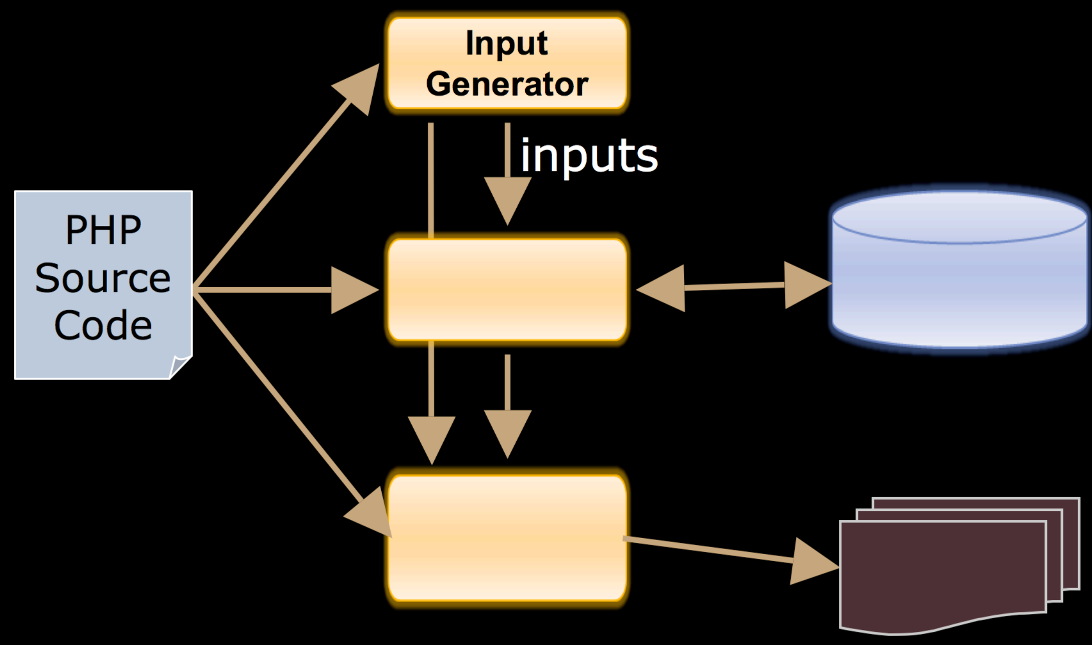
Second-order XSS attack



Architecture



Input generation

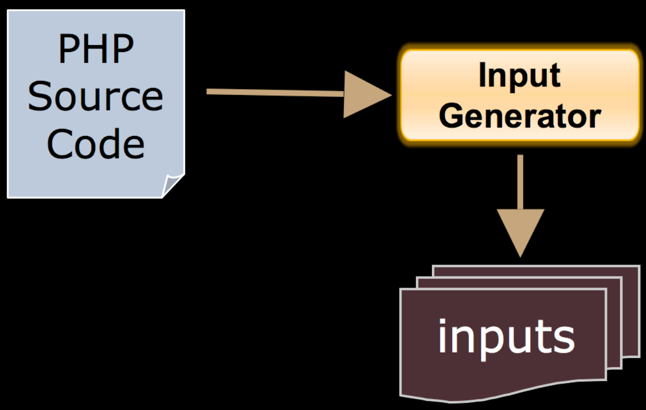


Goal: Create a set of concrete inputs
(_$GET[] & _$POST[])

We use Apollo generator (Artzi et al. ’08), based on **concolic execution**

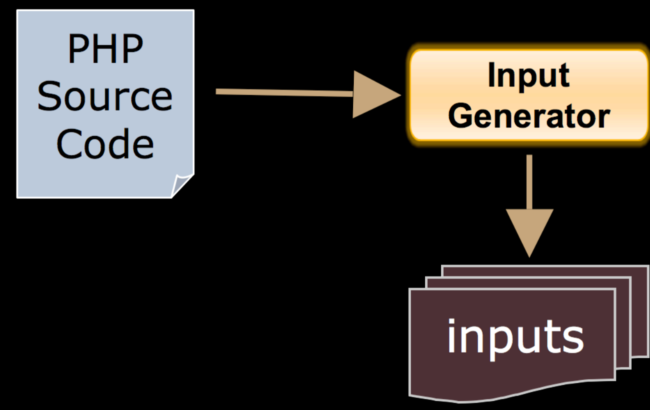
Input generation: concolic execution

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



Input generation: concolic execution

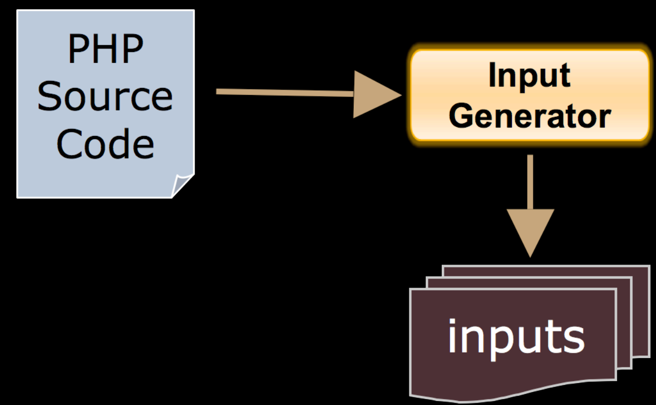
```
if ($_GET['mode'] == "add")
  addMessageForTopic();
else if ($_GET['mode'] == "display")
  displayAllMessagesForTopic();
else
  die("Error: invalid mode");
```




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

Input generation: concolic execution

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

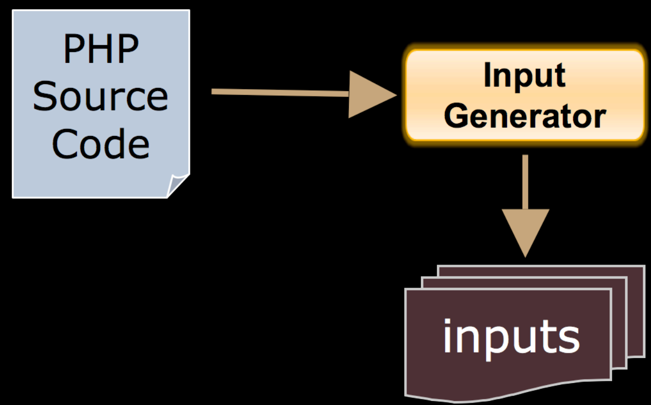



```
$_GET[]:
  mode = "1"
  msg = "1"
  topicID = 1
  poster = "1"
```

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

Input generation: concolic execution

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

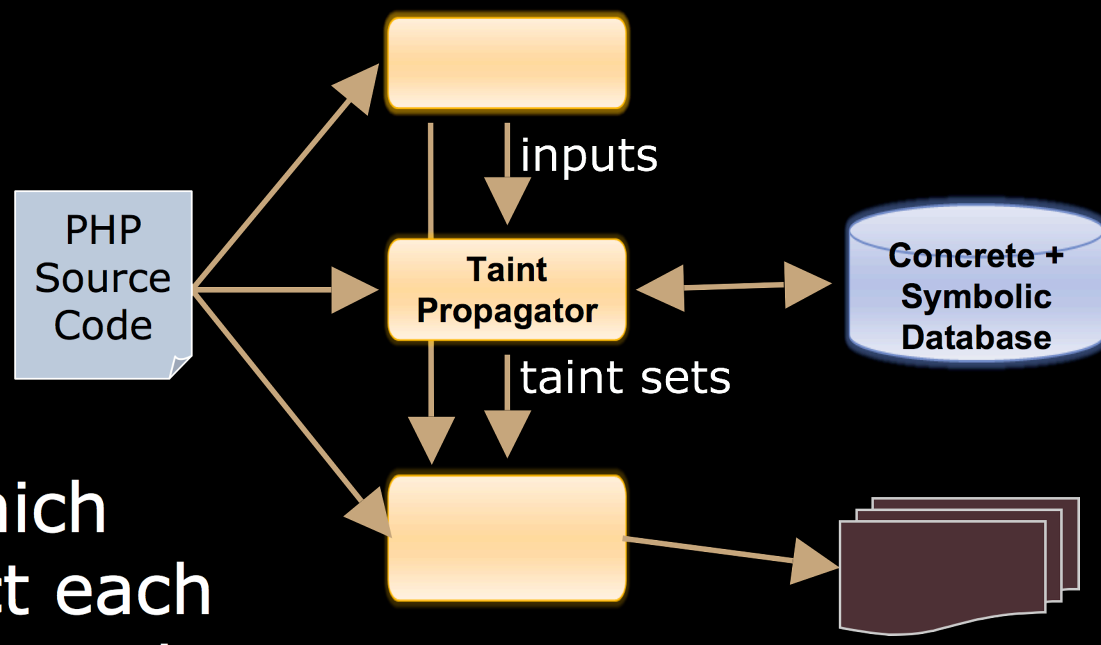



```
$_GET[]:
  mode = "1"
  msg = "1"
  topicID = 1
  poster = "1"
```

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

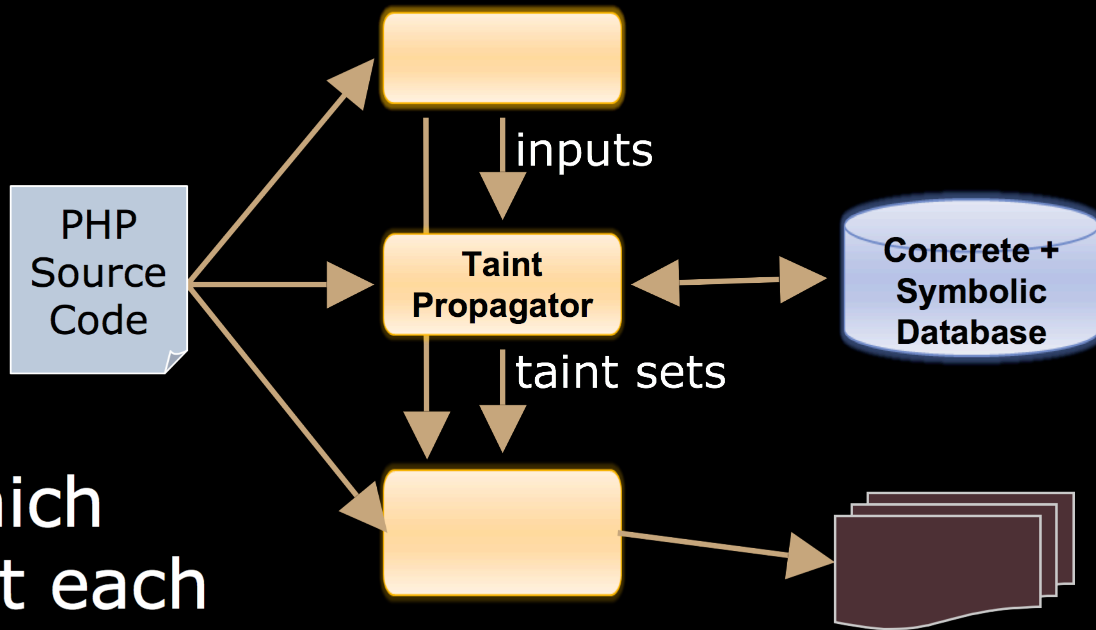
```
$_GET[]:
  mode = "display"
  msg = "1"
  topicID = 1
  poster = "1"
```

Taint propagation



Goal: Determine which input variables affect each potentially dangerous value

Taint propagation



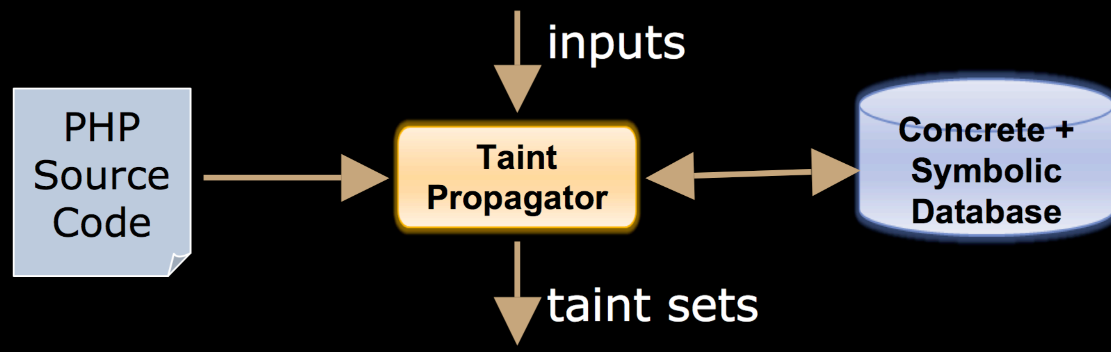
Goal: Determine which input variables affect each potentially dangerous value

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

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

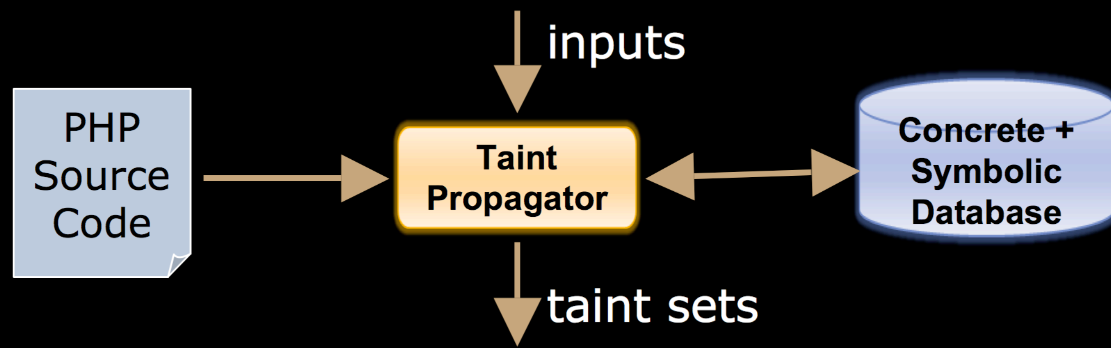
Taint propagation: data-flow

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



Taint propagation: data-flow

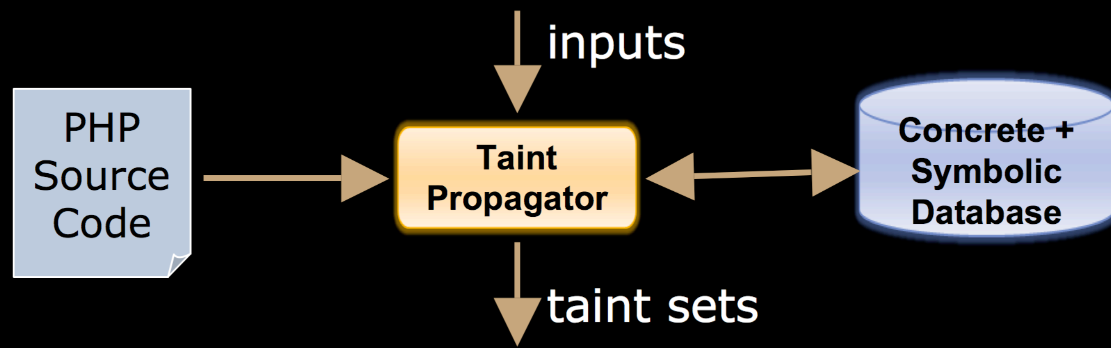
Each value has a **taint set**, which contains input *variables* whose values flow into it




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

Taint propagation: data-flow

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

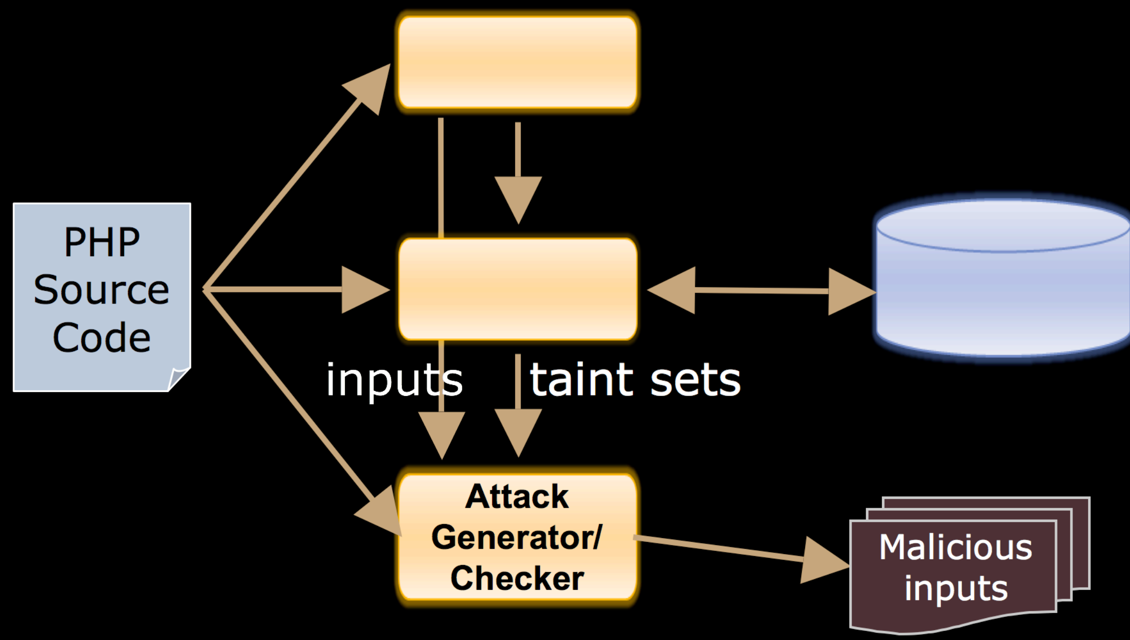



```
function displayAllMessagesForTopic() {
  $my_topicID = $_GET['topicID'];
  $sqlstmt = ” SELECT msg FROM messages
    WHERE topicID='$my_topicID’ ";
  $result = mysql_query($sqlstmt); /* {‘topicID’} */
```

Taint propagation

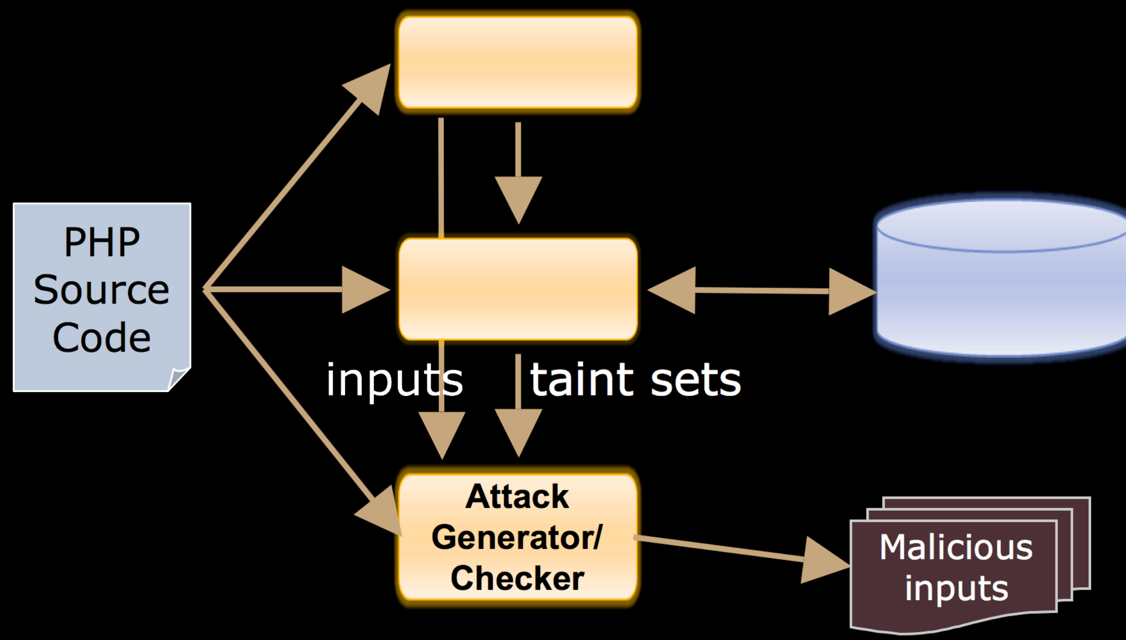
- Assignments: $my_poster = $_GET[“poster”]
- String concatenation: $full_n = $first_n . $last_n
- PHP built-in functions: $z = foo($x, $y)
- Database operations (for 2nd-order XSS)

Attack generation



Goal: Generate attacks for each sensitive sink

Attack generation



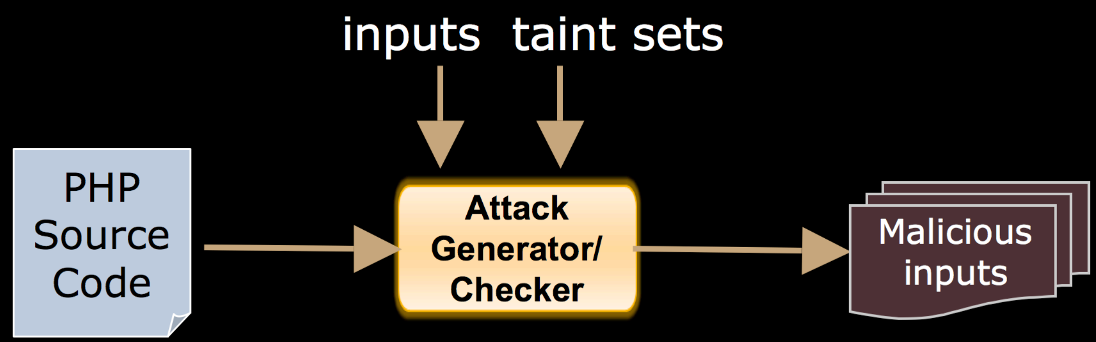
Goal: Generate attacks for each sensitive sink

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>"

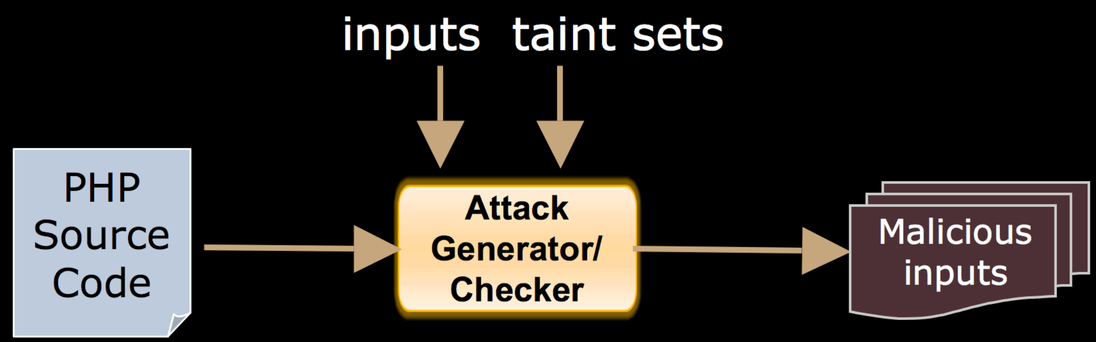
Alternative: String constraint solver (Kiezun et al. '09)

Attack generation



Given a program and an input i

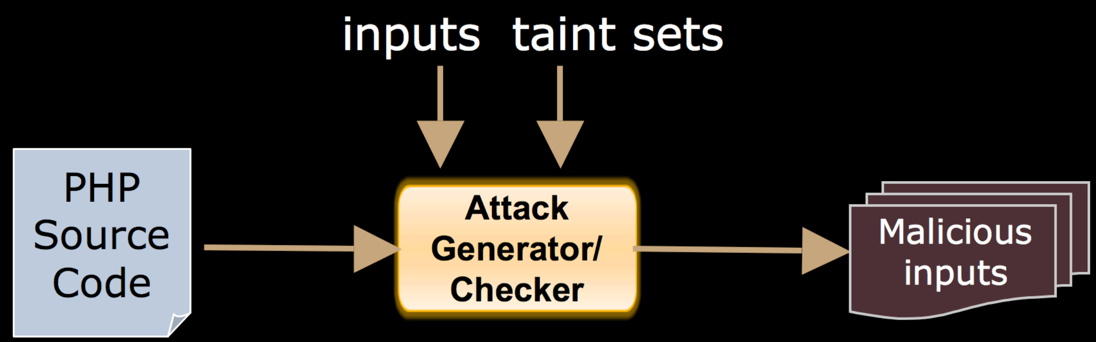
Attack generation



Given a program and an input i

for each var that reaches any sensitive sink:
  res = exec(program, i)

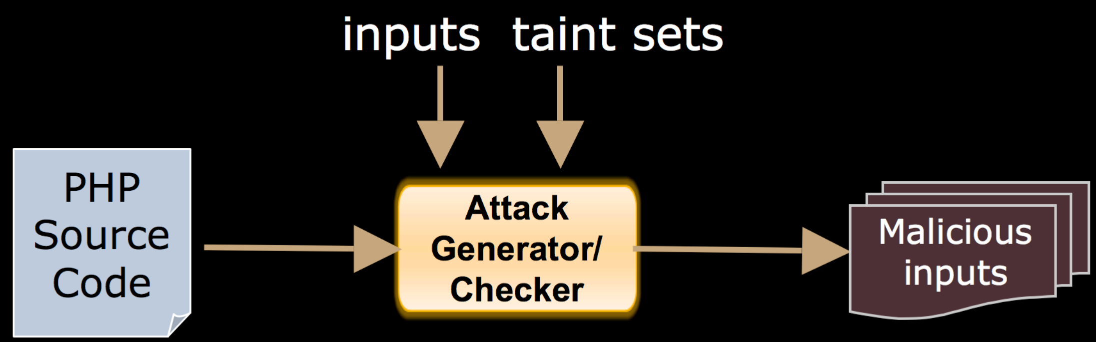
Attack generation



Given a program and an input i

```
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)
```

Attack generation

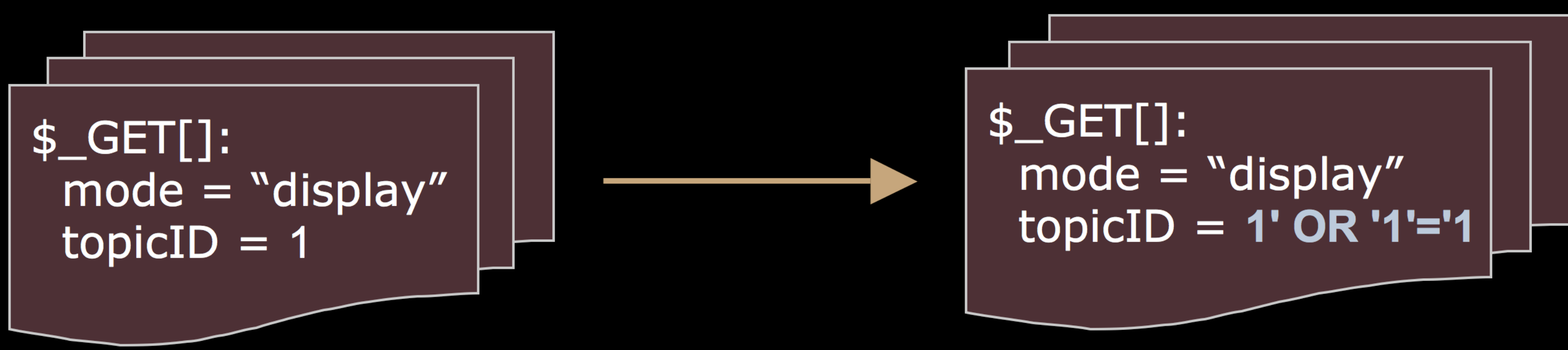


Given a program and an input i

```
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: 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
```



Attack generation: 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></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

Kind	Sensitive sinks	Reached sensitive sinks	Tainted sensitive sinks	Unique attacks
SQLI	366	91	76	**23**
1st-order XSS	274	97	78	**29**
2nd-order XSS	274	66	12	**8**

Total: **60**

Comparison with previous work

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

Automatic Creation of SQL Injection and Cross-Site Scripting Attacks

- Contributions
  - Automatically create SQLI and XSS attacks
  - First known technique for 2nd-order XSS
- Technique
  - Dynamically track taint through both program and database
  - Input mutation and output comparison
- Implementation and evaluation
  - Found 60 new vulnerabilities, no false positives