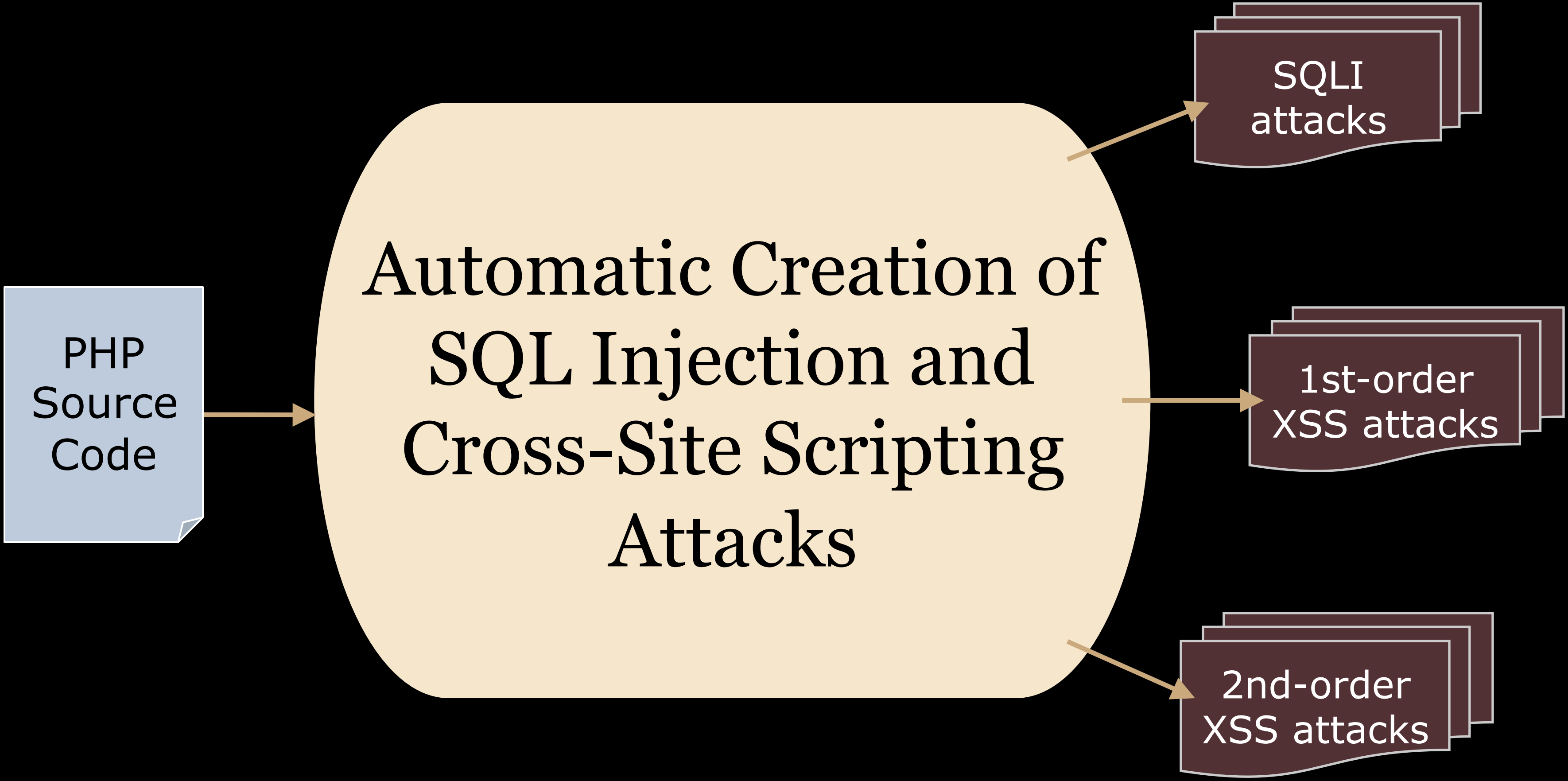


Automatic Creation of SQL Injection and Cross-Site Scripting Attacks

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

Presenter: Ruichao Qiu

Overview

Problem:

Finding security vulnerabilities (SQLI and XSS) in Web applications

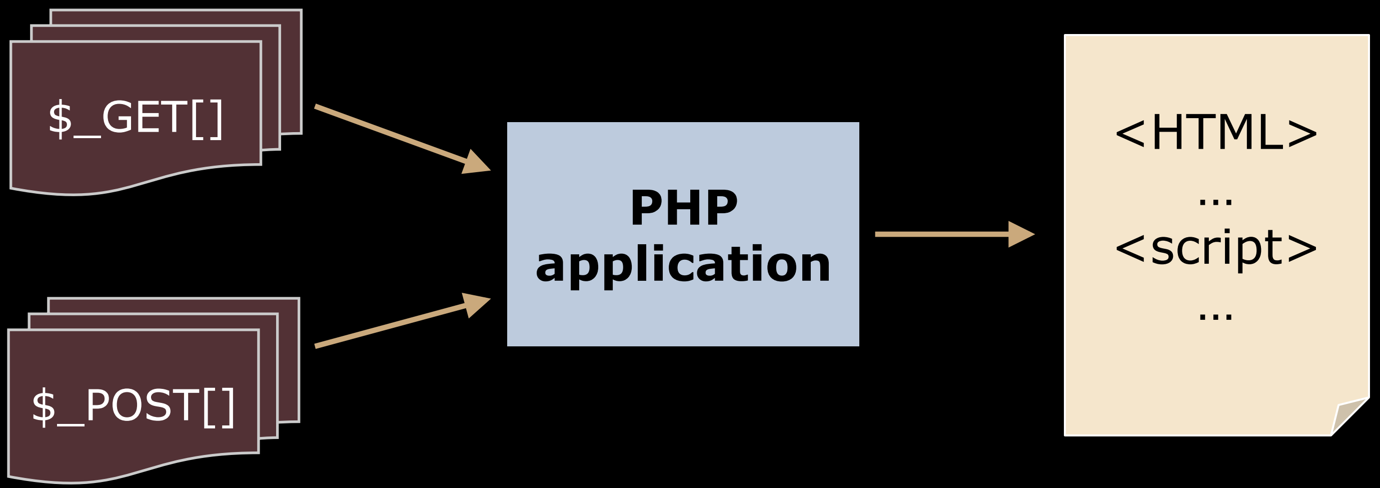
Approach:

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

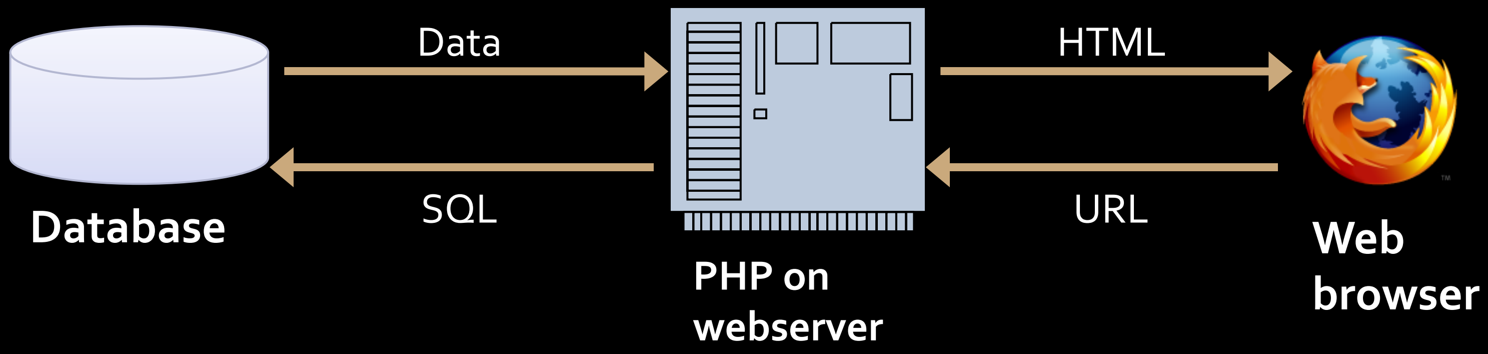
Results:

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

PHP Web applications



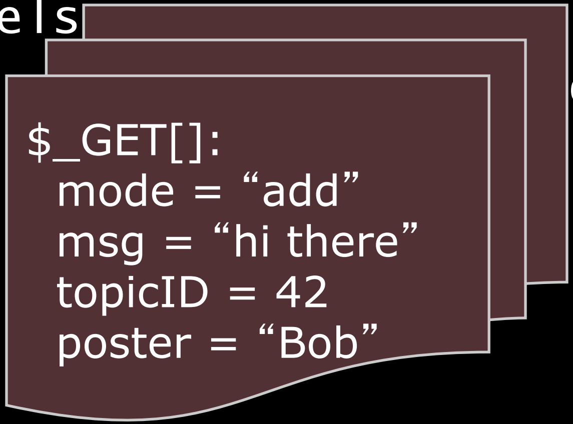
http://www.example.com/register.php?name=Bob&age=25



Example: Message board (add mode)

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

   displayAllMessagesForTop
   ic();
els
                       d
```



```
$_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”)

   displayAllMessagesForTop
   ic();
else
  di                d
```

```
$_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
```

Terminology Definition

- SQL Injection
  - User input for database statement
  - Structure of the SQL query changed
  - Get unauthorized access to data

SQL injection attack

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

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

   displayAllMessagesForTop
   ic();
else
  die(“Error: invalid
   mode”);
```

```
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'
```

Terminology Definition

- First-order XSS
  - Pass tainted data into function
  - Display HTML with attacker’s code
  - Steal browser cookies

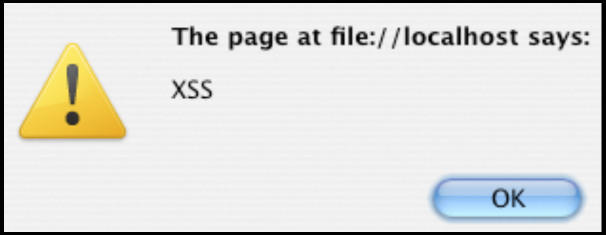
First-order XSS attack

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

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

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

Thanks for posting, A



The page at file://localhost says:

XSS

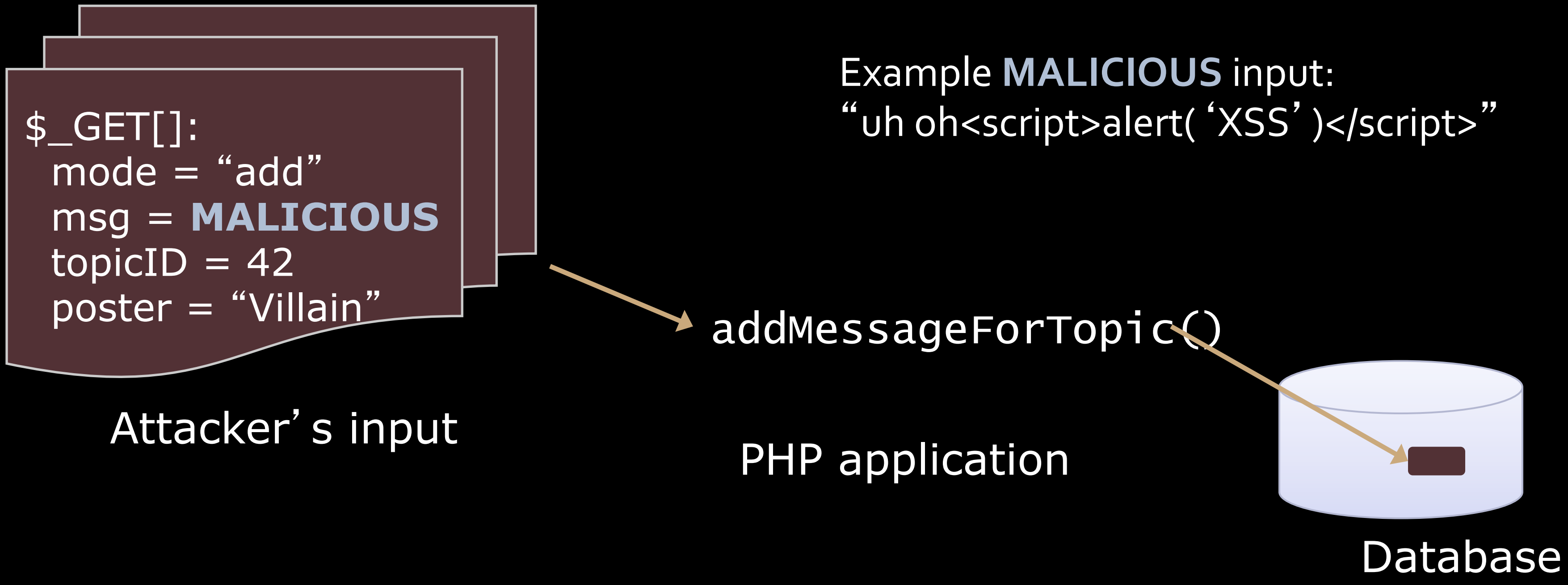
OK

Terminology Definition

- Second-order XSS
  - Store attacker’s input in database
  - Execute attacker’s code in HTML page
  - Affect multiple victim users

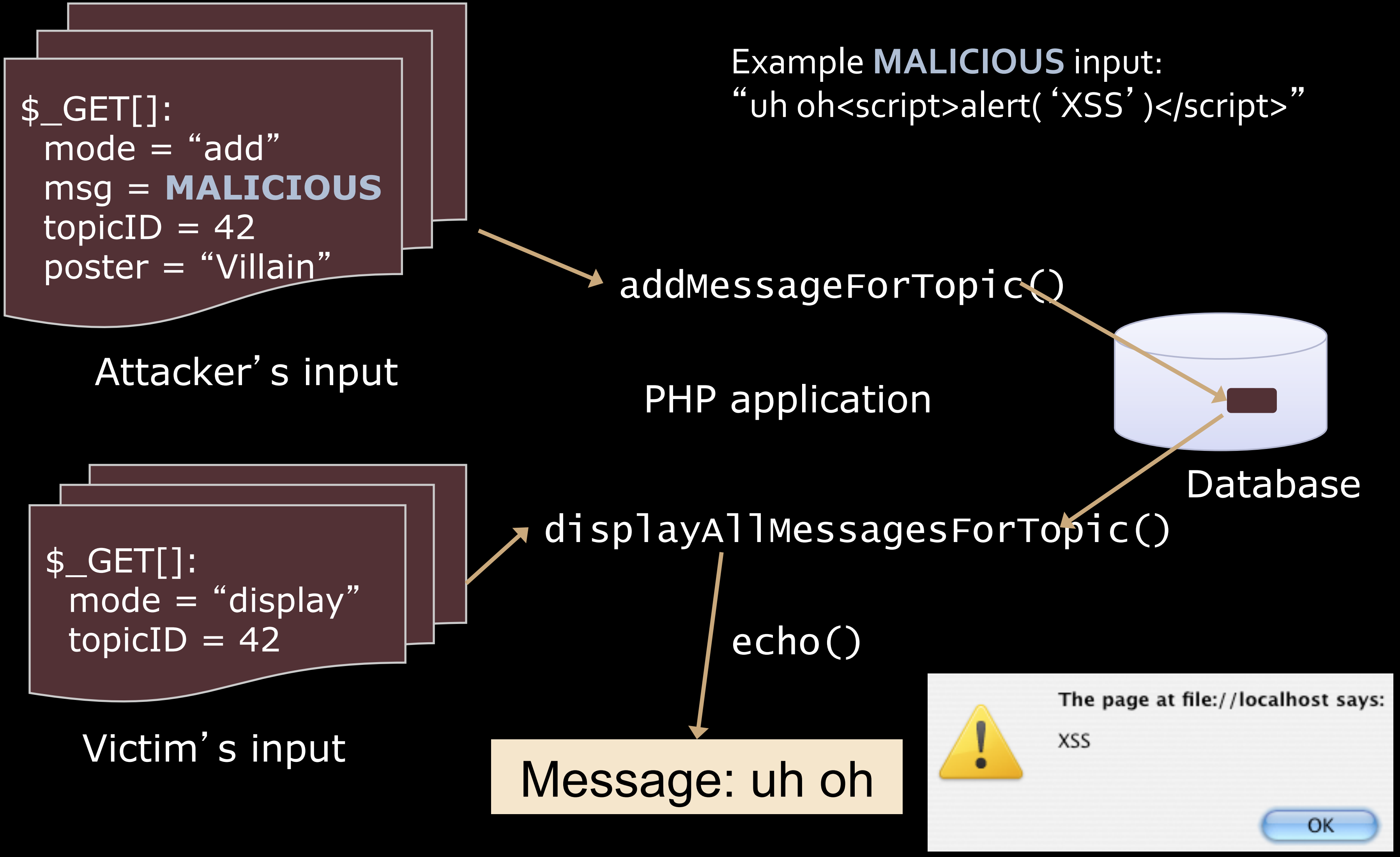
Second-order XSS attack

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

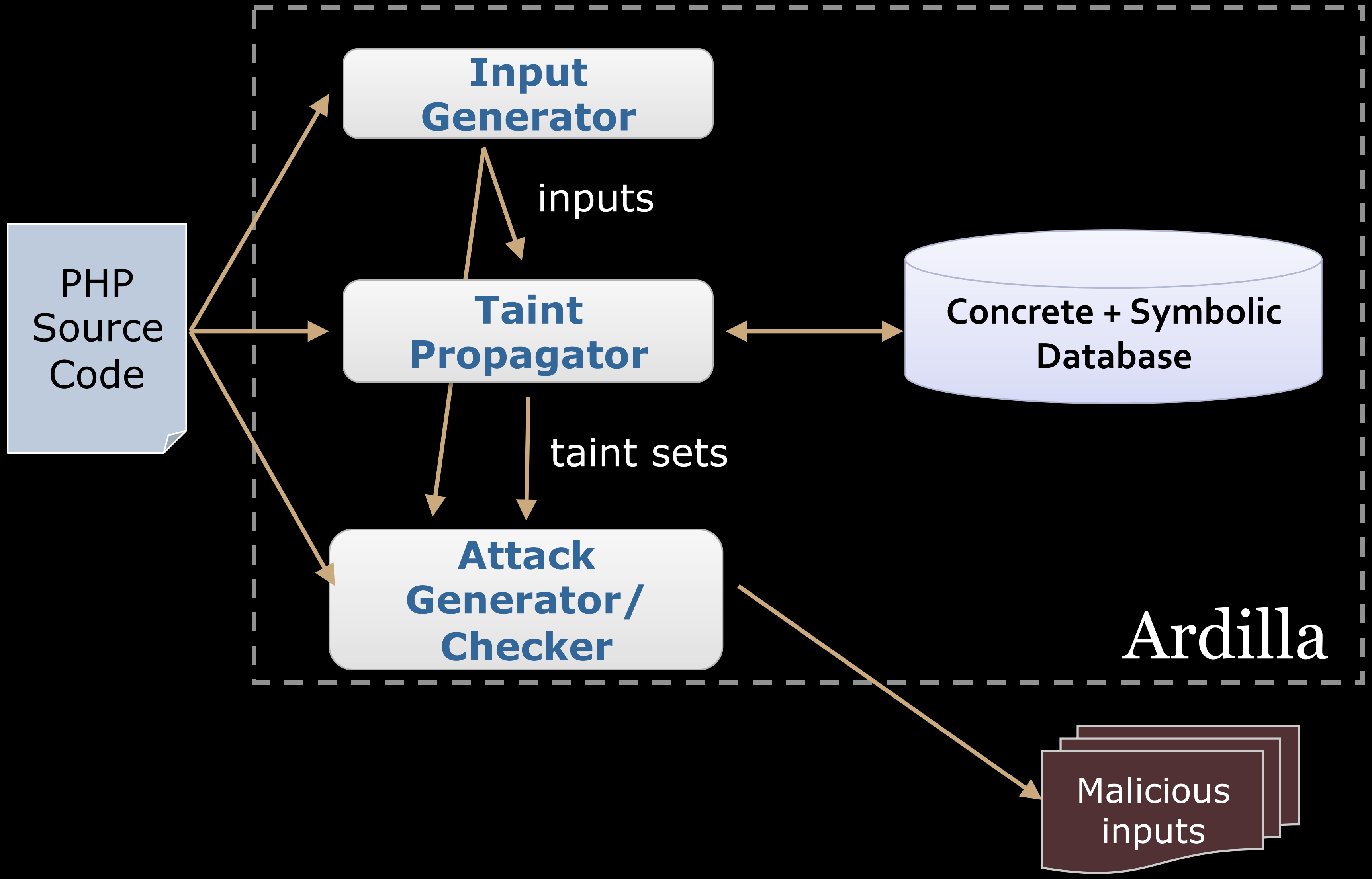


```
$_GET[]:
  mode = “add”
  msg = MALICIOUS
  topicID = 42
  poster = “Villain”
```

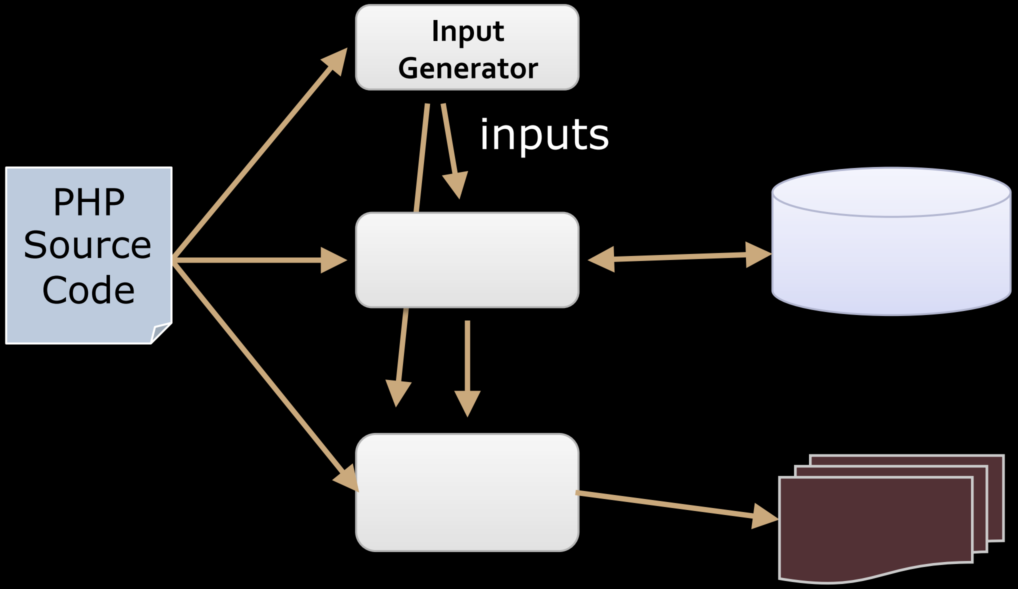
Second-order XSS attack



Architecture



Input generation



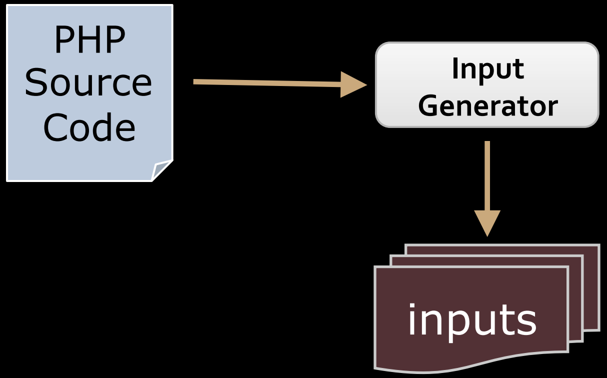
Goal: Create a set of concrete inputs
(_$GET[] & _$POST[])

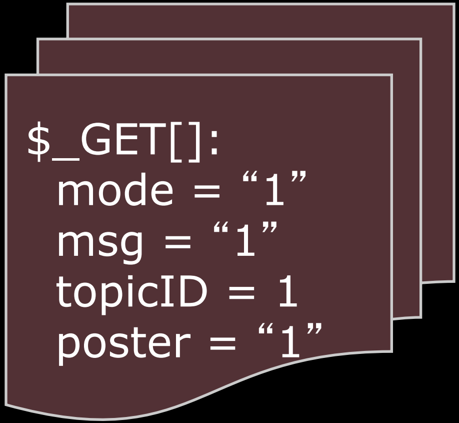
Use Apollo generator (Artzi et al. '08)

Input generation:

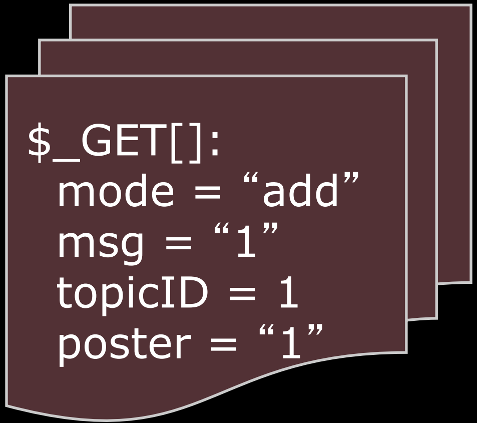
```
if ($_GET['mode'] ==
   "add")
  addMessageForTopic();
else if ($_GET[‘mode’] ==
   “display”)

   displayAllMessagesForTop
   ic();
else
  die(“Error: invalid
   mode”);
```

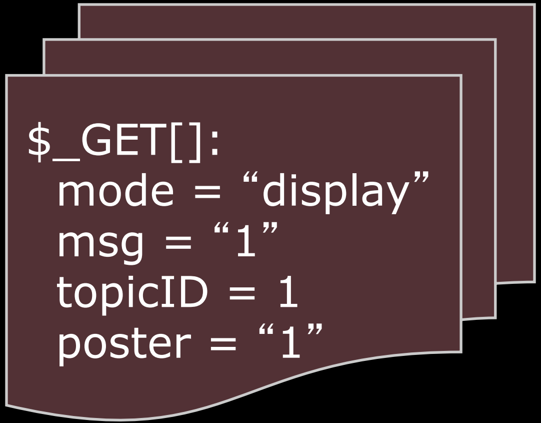




```
$_GET[]:
  mode = “1”
  msg = “1”
  topicID = 1
  poster = “1”
```

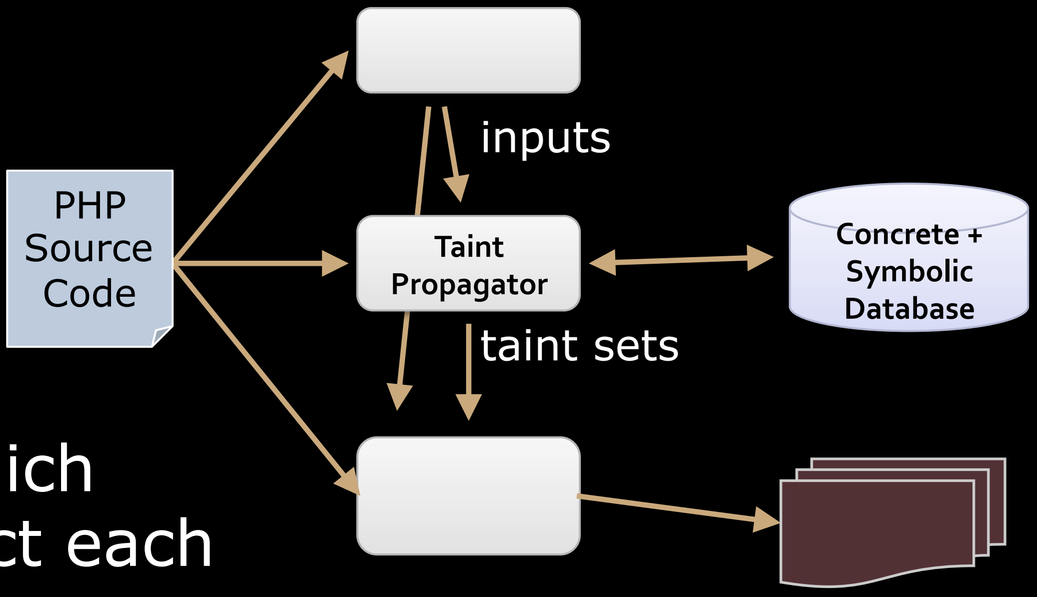


```
$_GET[]:
  mode = “add”
  msg = “1”
  topicID = 1
  poster = “1”
```



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

Taint propagation



Goal: Determine which input variables affect each potentially dangerous value

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

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

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:

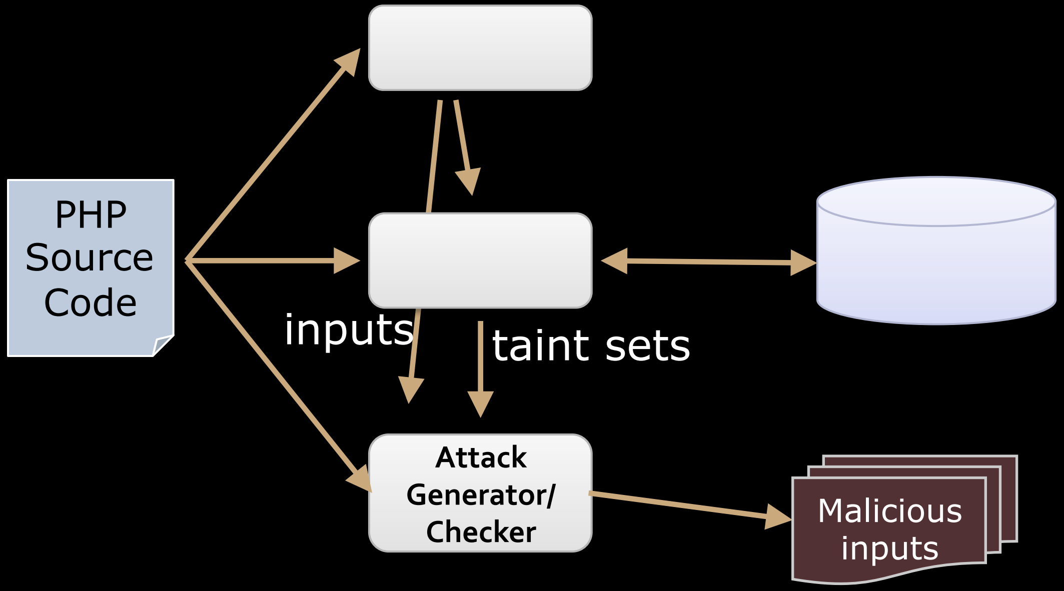
```
    { ’topicID’ }
```

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

Sensitive sink

Taint set

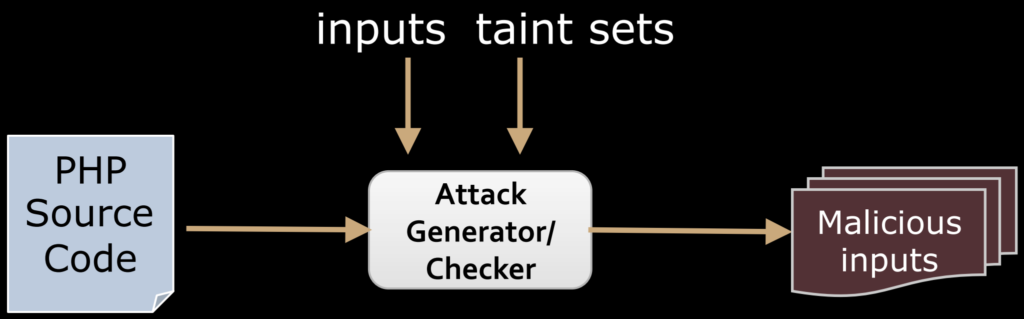
Attack generation and checking



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>"

Attack generation and checking



Given a program, an input i, and taint sets

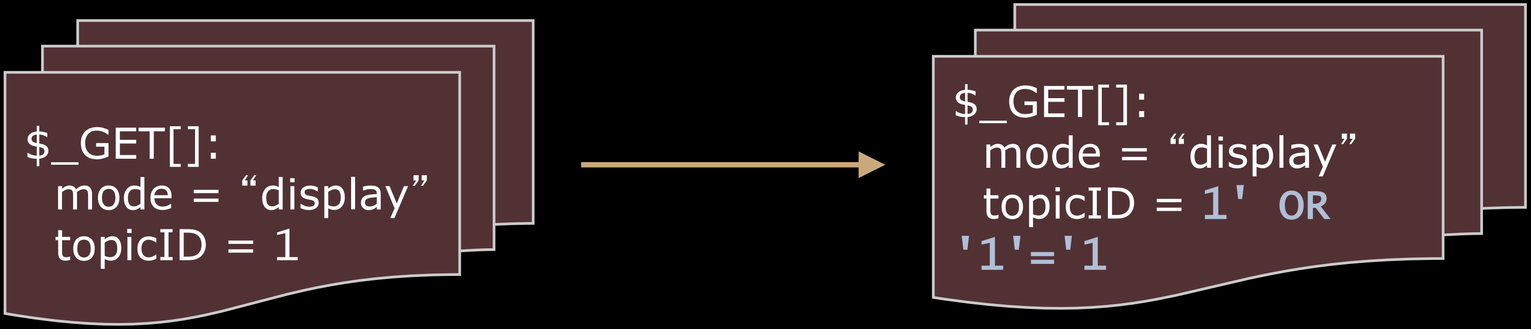
```
for each var that reaches any sensitive sink:
  res = exec(program, i)
  for shady in shady_strings:
    mutated_input = i.replace(var, shady)
    mutated_res = exec(program, mutated_input)
    if mutated_res DIFFERS FROM res:
      report mutated_input as attack
```

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
```



Attack checking: diffing outputs

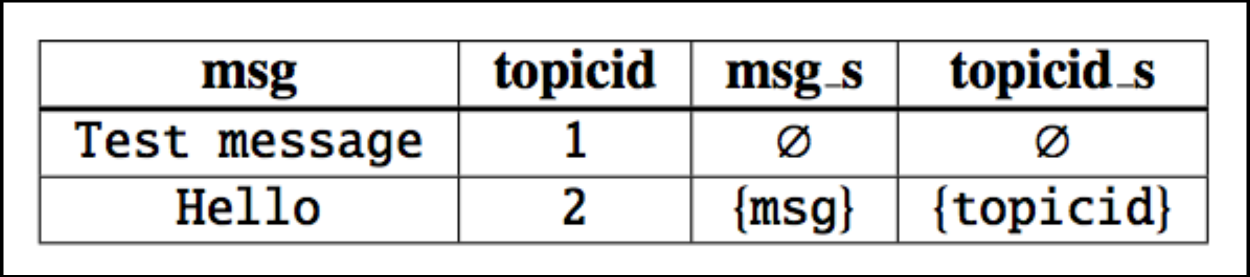
```
res = exec(program, i)
for shady in shady_strings:
  mutated_input = i.replace(var, shady)
  mutated_res = exec(program, mutated_input)
  if mutated_res DIFFERS FROM res:
    report mutated_input as attack
```

What is a significant difference?

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

Concrete + Symbolic Database

- Database: shared state enables data exchange
- A duplicate of concrete database
- Additional columns for symbolic data (taint set)



msg	topicid	msg_s	topicid_s
Test message	1	∅	∅
Hello	2	{msg}	{topicid}

Concrete + Symbolic Database

- Rewrite SQL statement

```
SELECT msg FROM messages WHERE topicid = '2'
```

```
SELECT msg, msg_s FROM messages WHERE topicid = '2'
```

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
1st-order XSS	274	97	29
2nd-order XSS	274	66	8

Total: **60**

Automatic Creation of SQL Injection and Cross-Site Scripting Attacks

- Contributions
  - Automatically create SQLI and XSS attacks
  - First 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