Università degli Studi di Milano
Facoltà di Scienze Matematiche, Fisiche e Naturali
Dipartimento di Informatica e Comunicazione

On Race Vulnerabilities in Web Applications

Roberto Paleari Davide Marrone Danilo Bruschi
Mattia Monga

DIMVA 2008

Introduction

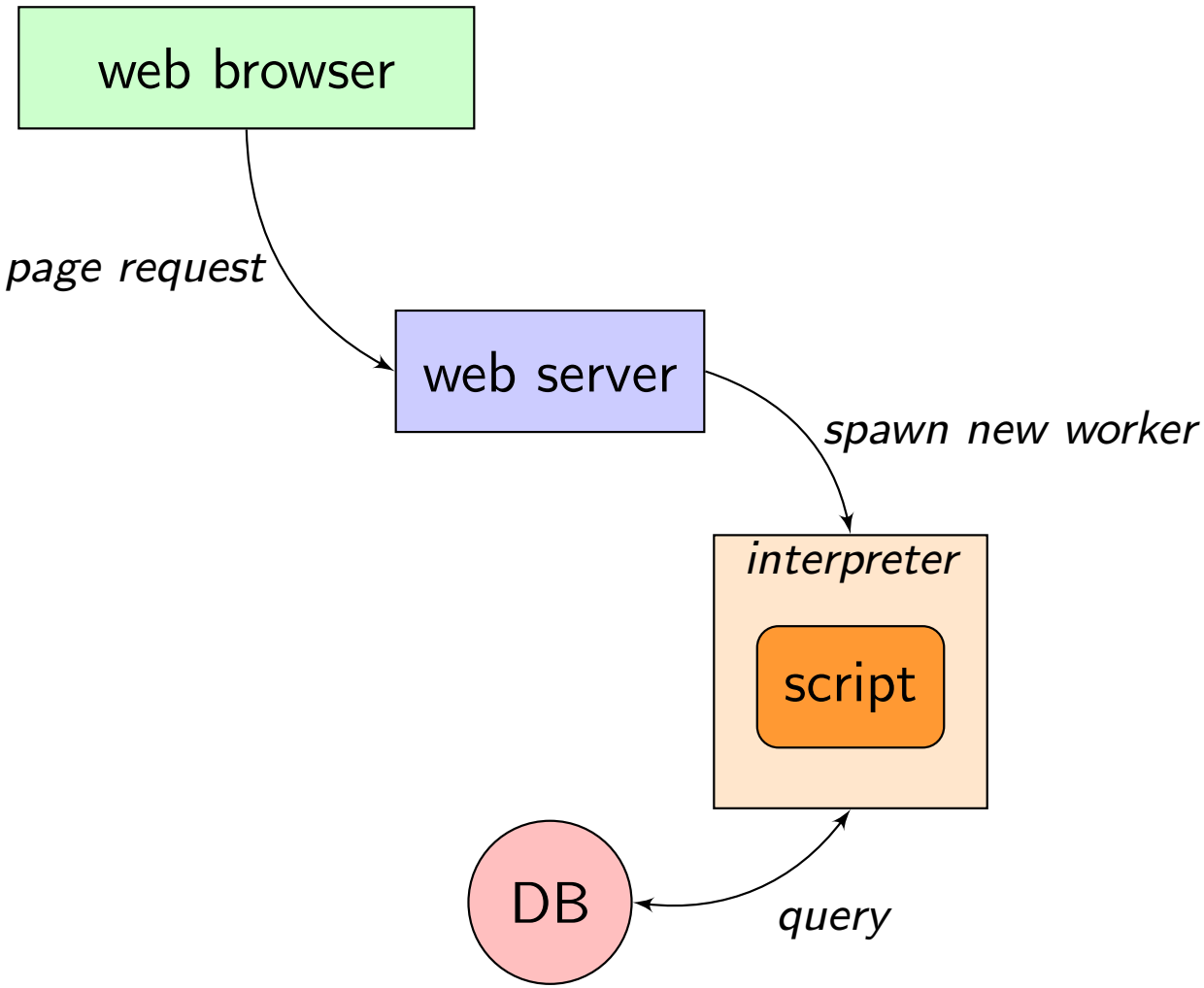
Web applications

- many applications adopt the web paradigm: client-server model + HTTP protocol
- web servers are augmented with modules for the execution of server-side code

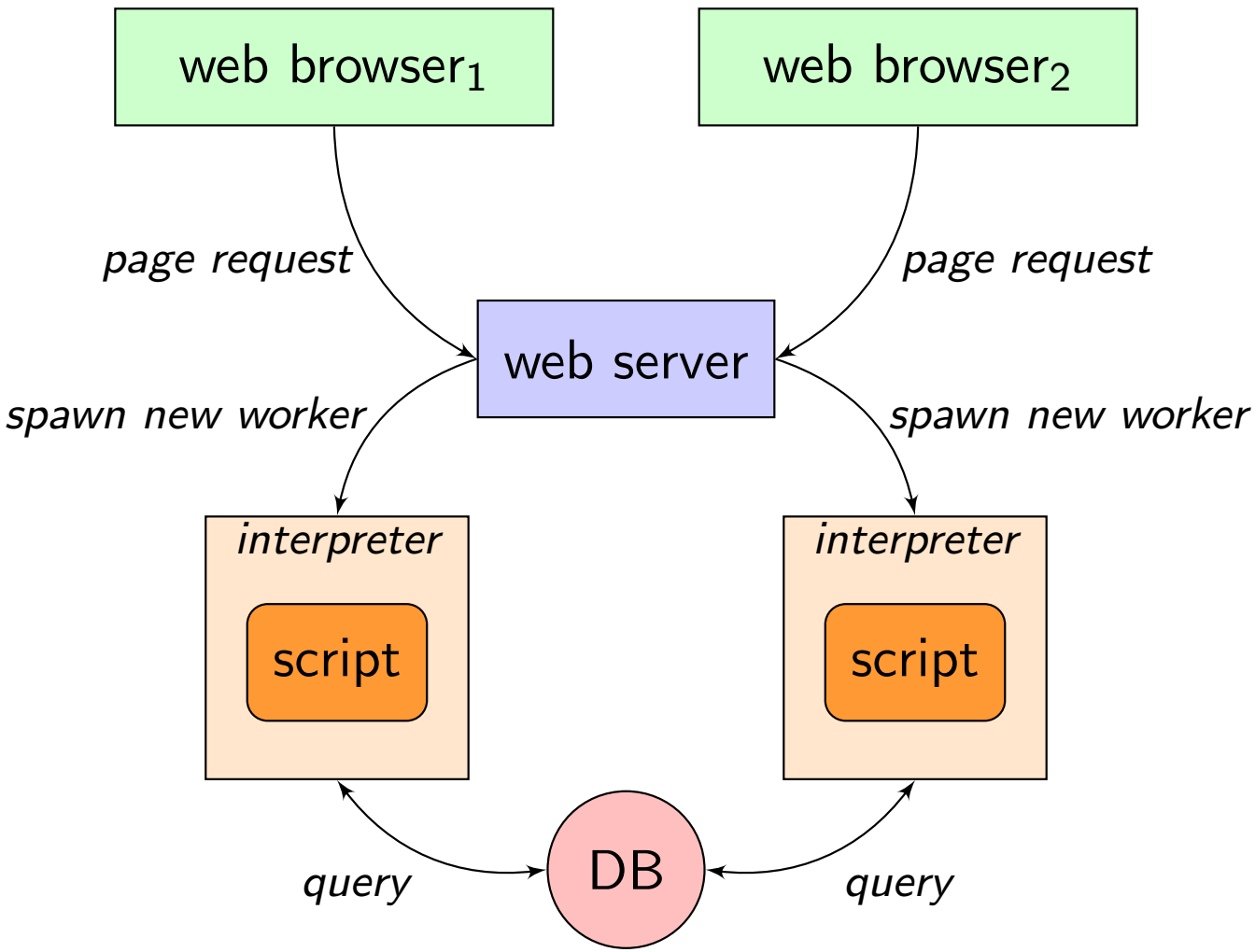
Security issues

- web applications are known to be subject to different attacks (*e.g.*, SQLI, XSS, command injection)
- $\sim$ 60% of software vulnerabilities are specific to web applications

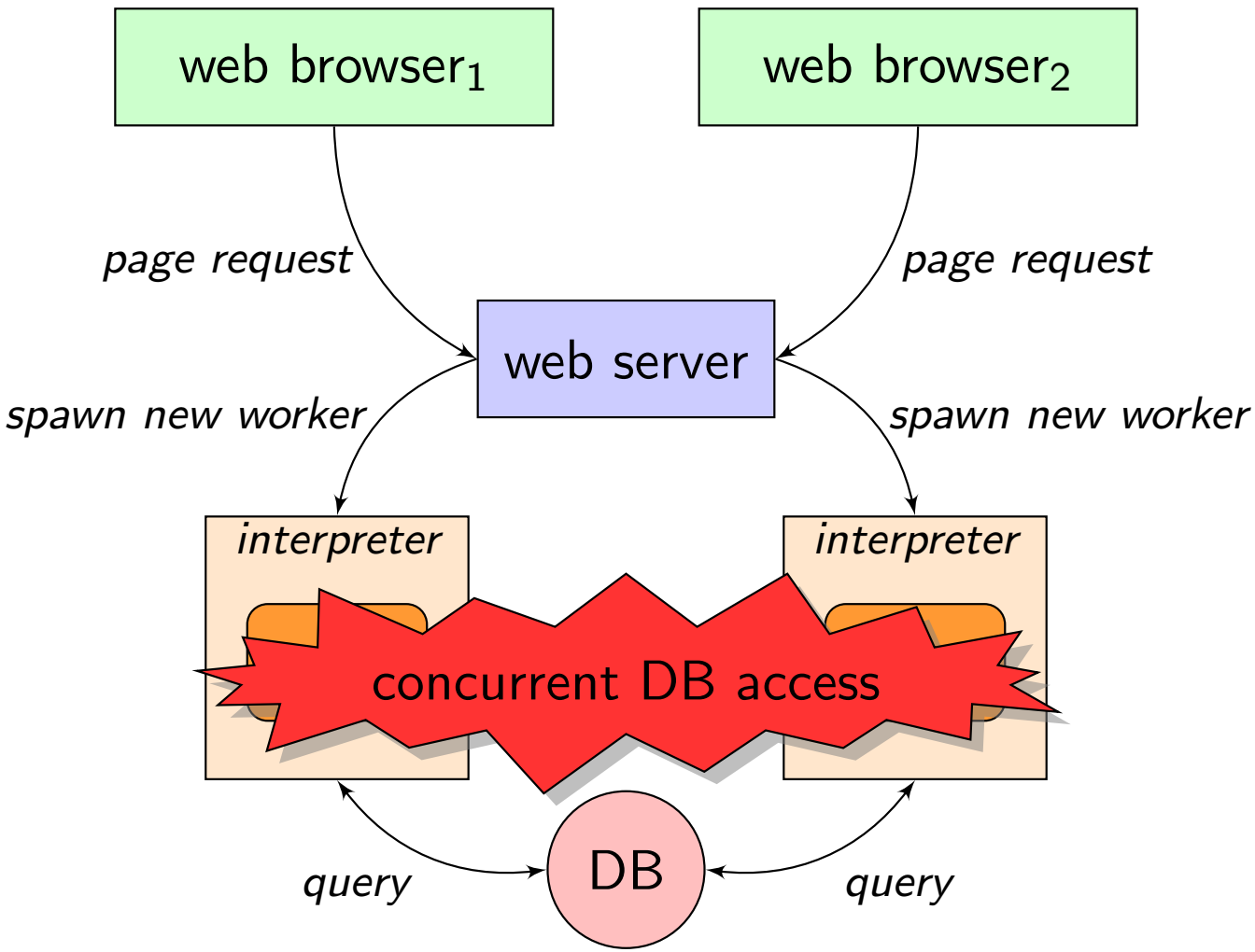
Web application framework
Single request



Web application framework
Multiple parallel requests



Web application framework
Multiple parallel requests



Concurrency in web applications

- web apps are made of different scripts that perform well-defined, *sequential* tasks
- scripts usually access some shared resources (*e.g.*, a database)
- multiple script instances can be executed concurrently
- race conditions are well known, but their impact on web applications has not been investigated

Problem

web programmers do not conceive their applications as multi-threaded or multi-process entities

- unexpected parallelism can lead to unforeseen interactions
- parallelism can be controlled client-side
- synchronization primitives are seldom used, and their efficacy is often system-dependent

Concurrency in web applications

- web apps are made of different scripts that perform well-defined, *sequential* tasks
- scripts usually access some shared resources (*e.g.*, a database)
- multiple script instances can be executed concurrently
- race conditions are well known, but their impact on web applications has not been investigated

Problem

web programmers do not conceive their applications as multi-threaded or multi-process entities

- unexpected parallelism can lead to unforeseen interactions
- parallelism can be controlled client-side
- synchronization primitives are seldom used, and their efficacy is often system-dependent

Motivating example

```
1   $res = mysql_query("SELECT credit FROM Users WHERE id=$id");
2   $row = mysql_fetch_assoc($res);
3   if($row['credit'] >= 800) {
4       <execute the requested operation>
5       $new_credit = $row['credit'] - 800;
6       $res = mysql_query("UPDATE Users SET credit=$new_credit" .
                           "WHERE id=$id");
    }
```

P_1		P_2	
Line	*Data*	*Line*	*Data*

Database	
ID	*Credit*
50	2500
92	820
123	**1000**
205	1200
...	...

Motivating example

```
1   $res = mysql_query("SELECT credit FROM Users WHERE id=$id");
2   $row = mysql_fetch_assoc($res);
3   if($row['credit'] >= 800) {
4       <execute the requested operation>
5       $new_credit = $row['credit'] - 800;
6       $res = mysql_query("UPDATE Users SET credit=$new_credit" .
                           "WHERE id=$id");
    }
```

P_1		P_2	
Line	*Data*	*Line*	*Data*
2	(id: 123, credit: 1000)	$\perp$	$\perp$

Database	
ID	*Credit*
50	2500
92	820
123	**1000**
205	1200
...	...

Motivating example

```
1   $res = mysql_query("SELECT credit FROM Users WHERE id=$id");
2   $row = mysql_fetch_assoc($res);
3   if($row['credit'] >= 800) {
4       <execute the requested operation>
5       $new_credit = $row['credit'] - 800;
6       $res = mysql_query("UPDATE Users SET credit=$new_credit" .
                           "WHERE id=$id");
    }
```

P_1		P_2	
Line	*Data*	*Line*	*Data*
2	(id: 123, credit: 1000)	$\perp$	$\perp$
4	(id: 123, credit: 1000)	1	$\perp$

Database	
ID	*Credit*
50	2500
92	820
123	**1000**
205	1200
...	...

Motivating example

```
1   $res = mysql_query("SELECT credit FROM Users WHERE id=$id");
2   $row = mysql_fetch_assoc($res);
3   if($row['credit'] >= 800) {
4       <execute the requested operation>
5       $new_credit = $row['credit'] - 800;
6       $res = mysql_query("UPDATE Users SET credit=$new_credit" .
                           "WHERE id=$id");
    }
```

P_1		P_2	
Line	*Data*	*Line*	*Data*
2	(id: 123, credit: 1000)	$\perp$	$\perp$
4	(id: 123, credit: 1000)	1	$\perp$
4	(id: 123, credit: 1000)	2	(id: 123, credit: 1000)

Database	
ID	*Credit*
50	2500
92	820
123	**1000**
205	1200
...	...

Motivating example

```
1   $res = mysql_query("SELECT credit FROM Users WHERE id=$id");
2   $row = mysql_fetch_assoc($res);
3   if($row['credit'] >= 800) {
4       <execute the requested operation>
5       $new_credit = $row['credit'] - 800;
6       $res = mysql_query("UPDATE Users SET credit=$new_credit" .
                           "WHERE id=$id");
    }
```

P_1		P_2	
Line	*Data*	*Line*	*Data*
2	(id: 123, credit: 1000)	$\perp$	$\perp$
4	(id: 123, credit: 1000)	1	$\perp$
4	(id: 123, credit: 1000)	2	(id: 123, credit: 1000)
5	(id: 123, credit: 1000)	4	(id: 123, credit: 1000)

Database	
ID	*Credit*
50	2500
92	820
123	**1000**
205	1200
...	...

Motivating example

```
1   $res = mysql_query("SELECT credit FROM Users WHERE id=$id");
2   $row = mysql_fetch_assoc($res);
3   if($row['credit'] >= 800) {
4       <execute the requested operation>
5       $new_credit = $row['credit'] - 800;
6       $res = mysql_query("UPDATE Users SET credit=$new_credit" .
                           "WHERE id=$id");
    }
```

P_1		P_2	
Line	*Data*	*Line*	*Data*
2	(id: 123, credit: 1000)	$\perp$	$\perp$
4	(id: 123, credit: 1000)	1	$\perp$
4	(id: 123, credit: 1000)	2	(id: 123, credit: 1000)
5	(id: 123, credit: 1000)	4	(id: 123, credit: 1000)

Database	
ID	*Credit*
50	2500
92	820
123	**200**
205	1200
...	...

Motivating example

```
1   $res = mysql_query("SELECT credit FROM Users WHERE id=$id");
2   $row = mysql_fetch_assoc($res);
3   if($row['credit'] >= 800) {
4       <execute the requested operation>
5       $new_credit = $row['credit'] - 800;
6       $res = mysql_query("UPDATE Users SET credit=$new_credit" .
                           "WHERE id=$id");
    }
```

P_1		P_2	
Line	*Data*	*Line*	*Data*
2	(id: 123, credit: 1000)	$\perp$	$\perp$
4	(id: 123, credit: 1000)	1	$\perp$
4	(id: 123, credit: 1000)	2	(id: 123, credit: 1000)
5	(id: 123, credit: 1000)	4	(id: 123, credit: 1000)

Database	
ID	*Credit*
50	2500
92	820
123	**200**
205	1200
...	...

Race conditions in web applications

Case studies

- tested several open-source apps
- 2 real-world closed-source SMS[a] applications, both found to be vulnerable (. . . without having access to their source code!)

[a] Text messages for mobile phones.

How to dynamically spot race conditions?

- we focus on LAMP applications
- interactions between multiple instances of the *same* script
- limited to races on database accesses

Race conditions in web applications

Case studies

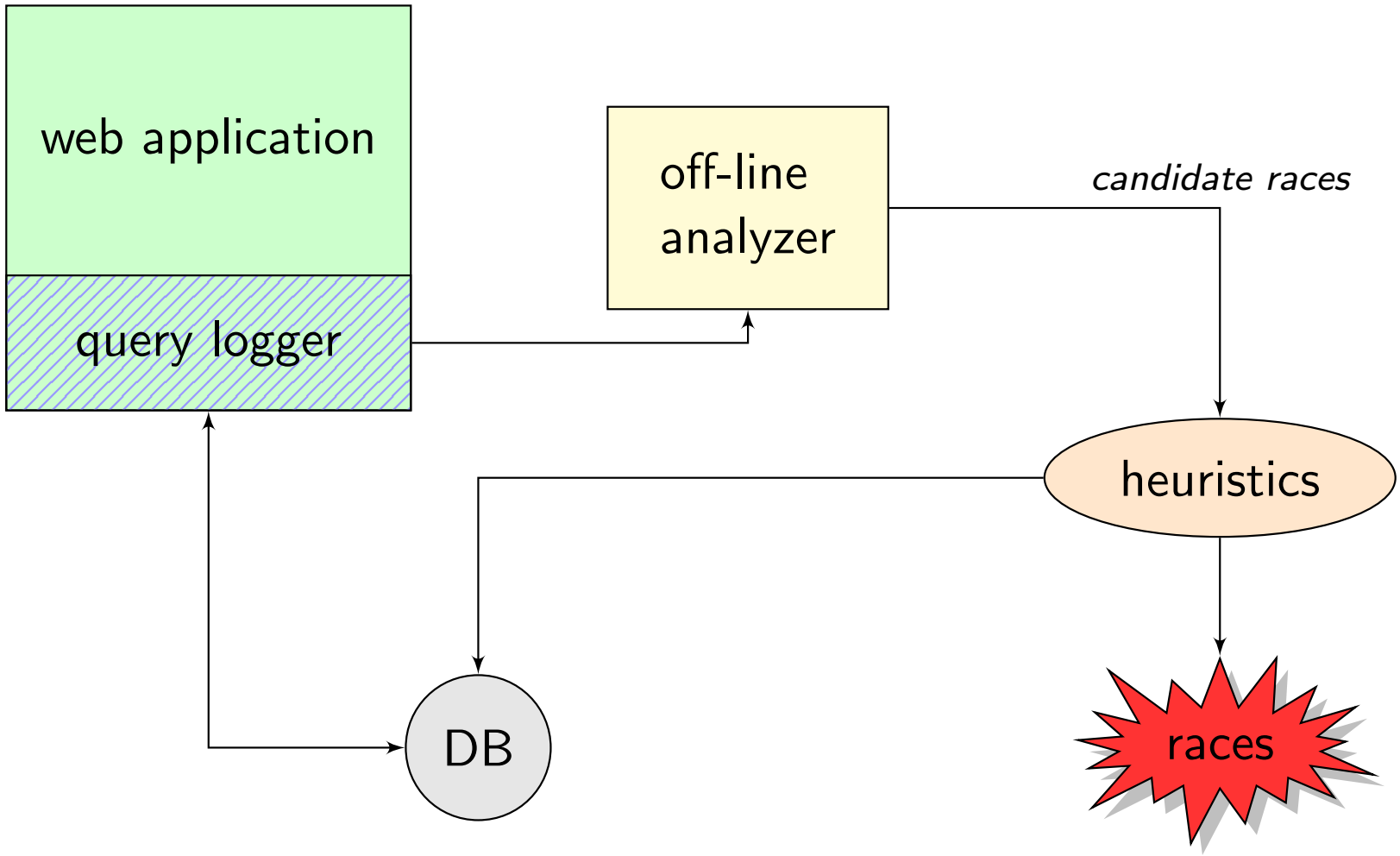
- tested several open-source apps
- 2 real-world closed-source SMS[a] applications, both found to be vulnerable (. . . without having access to their source code!)

[a] Text messages for mobile phones.

How to dynamically spot race conditions?

- we focus on LAMP applications
- interactions between multiple instances of the *same* script
- limited to races on database accesses

How to spot race conditions?
Framework overview



Off-line analysis
Candidate race conditions

Idea

- monitor application's SQL queries
- *interdependent* queries could lead to race conditions

Off-line analysis algorithm

- logged queries: $Q = \langle q_1, q_2, \ldots, q_n \rangle$
- $\forall q \in Q$ compute schema objects in $\mathrm{use}(q)$ and $\mathrm{def}(q)$
- *candidate* races are
  $(q_i, q_j) \in Q^2 : i < j \wedge \mathrm{use}(q_i) \cap \mathrm{def}(q_j) \neq \emptyset$

A simple example:

Query$_1$
SELECT id, credit
FROM Users
WHERE id = 123;

Query$_2$
UPDATE Users
SET credit = 100
WHERE id = 123;

= *used*

= *defined*

disjoint WHERE clauses can lead to false positives:

$Query_1$
SELECT id
FROM Sessions
WHERE *expiry_time* <= *123*;

$Query_2$
DELETE
FROM Sessions
WHERE *expiry_time* > *123*;

Solutions

- dynamically query the DB for a conjunction of WHERE clauses ($\rightarrow$ efficient, but *not* sound)
- constraint solver ($\rightarrow$ sound, but expensive and does not support all SQL constructs)

Attribute-relation bindings

- it is not always apparent to which relation an attribute belongs (*e.g.*, **SELECT** a_1, a_2 **FROM** T_1, T_2)
- $\Rightarrow$ actively query the application database

Annotations

- synchronization attempts can lead to false positives
- $\Rightarrow$ annotations to avoid reporting a race between a pair of SQL queries

Evaluation

Application	Category	Queries	FP	TP
Joomla! 1.5RC4	CMS	4086	0	55 (2)
phpBB 3.0.0	forum	2236	0	35 (4)
WordPress 2.3.2	blog/CMS	3638	0	47 (4)
Zen Cart 1.3.8a	shopping cart	35194	0	46 (1)

What kind of vulnerabilities did we find?

Highly application-dependent. Some examples:

- bypass brute force checks
- vote multiple times with parallel vote requests
- circumvent anti-flooding features

Limitations

Dynamic, database-level analysis

- analysis is completely dynamic
- we only take into account database queries

Lack of support for synchronization primitives

Why?

- during evaluation, we found really *few* synchronization attempts!
- lack of a set of "standard" synchronization primitives
- future work ;-)

Conclusions

Contributions

- study of the impact of race conditions on web applications
- novel detection technique
- working experimental prototype for LAMP applications

Future work

- consider interactions between different scripts
- employ program analysis techniques to extract more information about the application's logic
- take into account synchronization primitives

Thanks for the attention!

Questions?