Data & Information – Test 4: Solutions

18 June 2018, 13:45–15:15

Question 1: Security (40 points)

1.1

Users of a Web platform are given the feature to write private messages to other users. The following PHP code snippet is part of a script that implements this functionality. The fields *from* and *message* in the variable *$_PM* are input given by the sender. When a users receives a message, these fields will be displayed in the browser by the following code:

```
echo "New private message from " . $_PM['from'] . " : [" . $_PM['message'] . "]";
```

a) What kind of vulnerability could this code yield?

b) How could an attacker exploit this vulnerability

c) What could an attacker achieve by exploiting this vulnerability?

Answers:

a) (stored) Cross-Site Scripting

b) An attacker could send a private message containing malicious html / a malicious script (javascript), which is then executed in the victim's browser / BOM.

c) session hijacking, cookie stealing, information stealing, downloading malware, steal login credentials, execute malicious (javascript) code

1.2

The following code is part of an authentication script presenting the fields *username* and *password* to a user who wants to log in. After these fields are entered the script will check if there are any entries in the *$result* array and if so, log the user in as the first entry.

```
$user = $_POST['username'];
$pass = $_POST['password'];
$query = "SELECT * FROM users WHERE username = ' " . $user . " '
                        AND password = ' " . secure_hash($pass) . " ';";
$result = mysql_query($query);
```

a) Is the above code insecure and if so, why?

b) Give an example attacker input (the fields *username* and *password*) which will log the attacker in as the user "*admin*" without the attacker having to know the correct password. The user "*admin*" is the first user in the table *users*.

c) What is the best way, a developer can prevent such an attack?

Answers:

a) The code is vulnerable to SQL injection since user input to the query is not sanitized nor parameterized/prepared.

b) The above can be exploited in multiple fashions to log in as *admin* as long as the crafted query yields a set which contains the *admin* user. Some examples are:

- By crafting a query which ignores the password via commenting: for example setting the *user* field to be

  - admin' -- -

The password field can be anything here.

- By crafting a query which ignores the password via logic: for example setting the *user* field to be
  - admin' OR '1' = '0

  The password field can be anything here and 1 and 0 can be anything as long as they are different. The idea is to craft a contradiction to be part of the final AND clause resulting in *username = 'admin' OR ('1' = '0' AND password = '…')*

- By crafting a completely tautological query: for example setting the *user* field to be
  - (anything)' OR 1 = 1 -- -

  Here *(anything)* is a placeholder for anything (including an empty string) and 1 can be anything as long as the left and right sides of the equation are equal. The *password* can be anything here.

The above are just examples of valid injections. Note, however, that here the attacker can't use the output of *secure_hash($pass)* for injection and as such the injection must come completely from the input to *username*.

c) SQL injection can be prevented by using **prepared statements**. Only input sanitization is not enough.

1.3

A website implements advertisements of an external provider. This provider is malicious and delivers the following code within one of its ads:

```
<img src="http://serious-bank.com/transfer.php?IBAN=NL1337h4x0r&amount=1000"
                                                    width="0" height="0" border="0">
```

The *Serious Bank* is a reliable bank with a disastrous IT development department that does not implement any security mechanisms and website actions are generally handled via GET requests.

a) What kind of vulnerability may be in this scenario?

b) What would happen if a user is presented the malicious ad with the given code? In what case would this attack be successful and in which case would it fail?

c) How can such an attack be prevented by *Serious Bank*?

Answers:

a) Cross-Site Request Forgery (XSRF)

b) The link of the image src would be triggered, attempting to make a transfer of 1.000€ to NL1337h4x0r
   – successful, if the user is logged in at serious-bank.com
   – fail, if the user is not logged in

c) random CSRF-token with every response, expects same token in matching request as hidden field

Question 2: XML & JSON (40 points)

a) [SQL/XML] Given the SQL-query below, adapt it using the SQL/XML standard such that it produces the result as XML. The result should have one row per actor, which contains one column 'xml' containing an element "actor" with attributes "pid" and "moviecount" and as contents the name of the actor.

```
SELECT p.pid, p.name, COUNT(m.mid) AS moviecount
FROM acts a, person p, movie m
WHERE a.pid = p.pid
  AND a.mid = m.mid
GROUP BY p.pid, p.name
```

Answer:

```
SELECT XMLELEMENT(NAME actor,
       XMLATTRIBUTES(p.pid, COUNT(m.mid) AS moviecount),
       p.name) AS xml
FROM acts a, person p, movie m
WHERE a.pid = p.pid AND a.mid = m.mid
GROUP BY p.pid, p.name
```

b) [SQL/XML] Adapt the query of the previous question, such that the name of the actor is also an attribute and that the contents of the actor-element contain the names of the movie (s)he played in.

Answer:

```
SELECT XMLELEMENT(NAME actor,
       XMLATTRIBUTES(p.pid, p.name, COUNT(DISTINCT m.mid) AS moviecount),
       XMLAGG(XMLELEMENT(NAME movie, m.name))) AS xml
FROM acts a, person p, movie m
WHERE a.pid = p.pid AND a.mid = m.mid
GROUP BY p.pid, p.name
```

c) [XML querying] A table 'x' contains an attribute 'xml' which is of type 'xml' and contains the actor elements as in the example result of 2(b) above.

i. Give the XPath that produces all names of actors who only played in the movie "Godfather, The".

ii. What is the result of the query "`SELECT xpath('/movie/@name', x.xml) FROM x`" if the table 'x' contains only the three rows as shown in 2(b) above. Describe both the contents as well as the type(s) of the resulting column(s).

Answer:

(i) various alternatives (all are correct)

```
/actor[@moviecount=1 and ./movie="Godfather, The"]/@name
/actor[@moviecount=1][./movie="Godfather, The"]/@name
//movie[.="Godfather, The]/parent::actor[@moviecount=1]/@name
//@name[parent::actor[@moviecount=1 and ./movie="Godfather, The"]]
```

(ii) It should be a table with three rows and 1 column. The type of the column is "array of xml"

Each row contains an array with one XML-node in it. This node is an attribute node with name "name" and as value the name of the actor.

xpath
{name="Marlon Brando"}
{name=" Robert Duvall"}
{name=" Gianni Russo"}

d) [JSON construction] Given the SQL-query below which produces a list of JSON objects with attributes 'id', 'name', and 'roles'. Adapt the query, such that the 'roles' attribute is not an array of strings (array of role names), but an array of json-objects each containing both the role name as well as the movie name.

```
SELECT json_build_object('id',p.pid, 'name',p.name,
                         'roles',jsonb_agg(a.role))
FROM acts a, person p
WHERE a.pid = p.pid
GROUP BY p.pid
```

Answer:

```
SELECT json_build_object('id',p.pid, 'name',p.name, 'roles',jsonb_agg(
json_build_object('role',a.role, 'movie',m.name)))
FROM acts a, person p, movie m
WHERE a.pid = p.pid AND a.mid = m.mid
GROUP BY p.pid
```

e) [JSON querying] Given the SQL-query below which obtains the "id" of all actors with name "Robert Duvall" from a table "x" which has an attribute "json" containing json-objects as in the example result of the previous question. Rewrite the WHERE-clause such that the result of the query remains the same, but it now uses the "@>" operator instead of the "->>" operator.

```
SELECT x.json->>id
FROM x
WHERE x.json->>name = "Robert Duvall"
```

Answer:

```
SELECT x.json ->> id
FROM x
WHERE x.json @> {name: "Robert Duvall"}
```

f) [JSON] For each of the statements below, say whether they are true or false.

i. A JSON array can contain JSON scalar values

ii. A JSON array can contain JSON objects

iii. A JSON array can contain JSON arrays

Answer: All are true.

Question 3: Tree-shaped data / Pathfinder (20 points)

a) [Pathfinder] Given the small XML-document below, assign to each of the nodes its pre-order and post-order rank. Write down the full resulting table according to the Pathfinder document table structure: pre, post, level, kind, name, value.

```
<actor pid="6526" name="Marlon Brando" moviecount="4">
   <movie>Apocalypse Now</movie>
   <movie>On the Waterfront</movie>
   <movie>Godfather, The</movie>
   <movie>Streetcar Named Desire, A</movie>
</actor>
```

Answer

pre	post	level	kind	name	value
1	12	0	elem	actor	
2	1	1	attr	pid	6526
3	2	1	attr	name	Marlon Brando
4	3	1	attr	moviecount	4
5	5	1	elem	movie	
6	4	2	text		Apocalypse Now
7	7	1	elem	movie	
8	6	2	text		On the Waterfront
9	9	1	elem	movie	
10	8	2	text		Godfather, The
11	11	1	elem	movie	
12	10	2	text		Streetcar Named Desire, A

b) [Dewey numbering] Given an XML node with Dewey number 1.4.3. Which of the following statements is true?

i. The parent of this node has Dewey number 4.3.

ii. This node has 2 preceding siblings.

iii. It is possible to determine the level of the node from its Dewey number.

Answer: (i) is false; the other two are true.

c) [Pathfinder] Translate the XPath
`//movie[text()=”Apocalypse Now”]/parent::actor/@name`
according to the Pathfinder approach to an SQL-query that produces the right result given the table of the previous question. The result of the query on this table should be the pre-order rank of the resulting ‘name’ attributes. According to XPath semantics, the result should be duplicate-free and in document order. Obviously, the query should also work on any XML- fragment that is structured in this way.

Answer

```
SELECT DISTINCT n.pre
FROM edge m, edge, t, edge a, edge n
WHERE m.name = “movie” AND m.kind = “elem”                    -- // movie
AND t.elem = “text” AND t.value=”Apocalypse Now”              -- text()=‘Apocalypse Now’
AND t.pre > m.pre AND t.post < m.post AND t.level = m.level+1
AND a.name = “actor” and a.kind = “elem”                      -- /parent::actor
AND a.pre < m.pre AND a.post > m.post AND a.level = m.level-1
AND n.name = “name” and n.kind = “attr”                       -- /@name
AND n.pre > a.pre AND n.post < a.post AND n.level = a.level+1
ORDER BY n.pre
```

d) [Pathfinder] PostgreSQL stores XML and JSON values in a column of a table as a special type. With the Pathfinder approach such values are shredded, i.e., stored in a table with a fixed set of attributes. What is the main purpose of doing so? Explain your answer.

Answer: Indexing and efficiency of queries.