# Data & Information – Solutions to Test 4

## Question 1

a) The above code can be vulnerable to (stored) Cross-Site Scripting if user input is stored and (later) output to the user without sanitization.
An attacker could exploit this by sending a private message containing malicious html / a malicious script in order to hijack a victim's session via for example cookie stealing.

b) The above code is vulnerable to SQL injection since user input to the query is not sanitized nor parameterized/prepared.
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*.
SQL injection can be prevented by using prepared statements. Only input sanitization is not enough.

c) The above approach is insufficiently secure because it stores credential hashes using an insecure hashing function and it doesn't use a key-derivation function / iterated hash function to slow down / mitigate brute-force or dictionary attackers.
Ideally these credentials should be stored using a memory-hard key derivation function / password hashing scheme.

d) This function is insecure because it uses a non-cryptographically secure pseudorandom number generator (PRNG), namely the default Java PRNG which is then seeded with the userid, which is known to the attacker, and the token length, which is static, thus violating the secrecy of the PRNG seed. An attacker can thus trivially obtain the reset token for any user (provided they know the userid) by simply executing the above function with the ID and even simply bruteforce userids and reset everyone's credentials.
Note that here the core of the matter is that:
- A non-cryptographically secure PRNG was used
- Even if a secure PRNG was used, the seed is known to the attacker

## Question 2

a)
```
SELECT XMLELEMENT(NAME country, XMLATTRIBUTES(m.name AS movie, r.runtime),
        r.country)
FROM runtime AS r, movie AS m
WHERE r.mid = m.mid
```

b)
```
SELECT XMLELEMENT(NAME actor, XMLATTRIBUTES(p.name),
         XMLAGG(XMLELEMENT(NAME movie, XMLATTRIBUTES(a.role),m.name)))
FROM person p, acts a, movie m
WHERE p.pid = a.pid AND a.mid = m.mid
GROUP BY p.name
```

c)

| json |
|---|
| {"name": "Monty Python and the Holy Grail", "country": "UK", "runtime": 89} |
| {"name": "Monty Python and the Holy Grail", "country": "USA", "runtime": 91} |
| {"name": "Twelve Monkeys", "country": "Australia", "runtime": 130} |
| … |

d) No.
Condition 1: only certain operators are supported
Condition 2: only top level is indexed.
Both conditions are not satisfied: (?, ?|, and @>, but not ->> and ->) and toplevel is "json" while the condition in the WHERE-clause is on "json ->> 'runtime').

e) Difference: '->' returns a JSON value (i.e., a value of type json or jsonb), '->>' returns a 'normal' value (i.e., a value of type TEXT).
We need to use '->>' in the WHERE-clause because otherwise we cannot compare with the value 89.

f)
```
WHERE json @> '{ "runtime": 89 }'::jsonb
```

g)

i)
```
SELECT unnest(xpath('//movie[.=''Psycho'']/parent::actor',x.xml))
FROM x
```

ii)
```
SELECT unnest(xpath('//actor[count(./movie)>1]/@name',x.xml))
FROM x
```

## Question 3

a) 1, 8, 0, elem, ‘actor’, NULL
2, 1, 1, attr, ‘name’, ‘Patricia Hitchcock’
3, 4, 1, elem, ‘movie’, NULL
4, 2, 2, attr, ‘role’, ‘Caroline’
5, 3, 2, text, NULL, ‘Psycho’
6, 7, 1, elem, ‘movie’, NULL
7, 5, 2, attr, ‘role’, ‘Barbara Morton’
8, 6, 2, text, NULL, ‘Strangers on a Train’

b)
```
SELECT DISTINCT a.pre
FROM edge m, edge r, edge a
WHERE m.type = 'elem' AND m.name='movie'
AND r.pre>m.pre AND r.post<m.post AND r.level=m.level+1
AND r.name='role' AND r.type='attr'
AND a.pre<m.pre AND a.post>m.post
AND a.name = 'actor' AND a.type='elem'
ORDER BY a.pre
```

c) No. The Dewey number contains the number of the parent in itself. Example: if a node has Dewey number 1.3.7, then the Dewey number of the parent is 1.3 (i.e., leave out the last part).