ANSWERS

F11.8 Writing secure programs

Cyber security · GCSE · OCR J277 1.4.2, AQA 8525 3.6.3, Edexcel 1CP2 5.3.2 · about 15 min

What this lesson is about

Validating input, access levels in code, and testing for security.

- Never trust input
- Access levels in code
- More defensive habits
- Testing for security

Questions

5 marks in all

1. How does a program reduce the risk of SQL injection? [1 mark]

- ☐ **A** Validating input and keeping it separate from commands
- ☐ **B** Running faster
- ☐ **C** Using a bigger database
- ☐ **D** Encrypting the screen

> **Answer:** A. Never drop untrusted input straight into a query.

2. What does 'fail safely' mean? [1 mark]

- ☐ **A** If something goes wrong, stop in a safe state and reveal nothing
- ☐ **B** Ignore all errors
- ☐ **C** Crash loudly with details
- ☐ **D** Keep running whatever happens

> **Answer:** A. BugBot stops its motors if the program crashes.

3. What is the principle of least access? [1 mark]

- ☐ **A** Give each user or program only the access its job needs
- ☐ **B** Give everyone admin access
- ☐ **C** Remove all access
- ☐ **D** Share one account

> **Answer:** A. It limits the damage a breach can do.

4. Why should you test with invalid and boundary input? [1 mark]

- ☐ **A** Attacks often use unexpected input
- ☐ **B** It makes the code shorter
- ☐ **C** Valid input never fails
- ☐ **D** It speeds up the program

> **Answer:** A. A secure program is attacked by its own tests first.

5. What does this program print? [1 mark]

```
def ok(t):
    return t.isdigit() and 0 <= int(t) <= 100
print(ok('50'), ok('999'), ok('go'))
```

Answer:

```
True False False
```

50 is valid; 999 is out of range; 'go' is not a number.

The task: an access checker

Write `allowed(role, action)` using the `can_do` table, returning True or False. For each `(user, role, action)` in `requests`, print `<user> (<role>) <action>: allowed` or `: denied`. Validate first: if the role is not in `can_do`, print `<user> (<role>) <action>: unknown role` instead. At the end print `denied or blocked: <n>`.

```
# the two lines every program starts with: the commands, then the robot
from bugbot import *
connect()

can_do = {"student": ["view"], "teacher": ["view", "edit"], "admin": ["view", "edit",
"delete"]}
# user, role, action
requests = [
    ("Sam", "student", "view"),
    ("Sam", "student", "edit"),
    ("Mr Lee", "teacher", "edit"),
    ("root", "admin", "delete"),
    ("ghost", "hacker", "delete"),
]
```

The hint students can ask for: Check the role exists before anything else, and say so if it does not. Otherwise look the role up and see whether the action is in the list of things it may do. Count everything that was not allowed.

A solution

```
from bugbot import *
connect()
can_do = {"student": ["view"], "teacher": ["view", "edit"], "admin": ["view", "edit",
"delete"]}
requests = [
    ("Sam", "student", "view"),
    ("Sam", "student", "edit"),
    ("Mr Lee", "teacher", "edit"),
    ("root", "admin", "delete"),
    ("ghost", "hacker", "delete"),
]

def allowed(role, action):
    return action in can_do.get(role, [])

blocked = 0
for user, role, action in requests:
    if role not in can_do:
        print(f"{user} ({role}) {action}: unknown role")
        blocked = blocked + 1
    elif allowed(role, action):
        print(f"{user} ({role}) {action}: allowed")
    else:
        print(f"{user} ({role}) {action}: denied")
        blocked = blocked + 1
print("denied or blocked:", blocked)
```

Any program that meets the task's checks is marked correct in the simulator; this is one way, not the only way.