Kódolás és IT biztonság

(VIHIBB01)

Laboratóriumi gyakorlat

Szoftverek biztonsági tesztelése

Szerző:
Gazdag András és Koltai Beatrix

2023. szeptember 27.

Tartalomjegyzék

1. Oktatási célok

Ezen a laboratóriumi gyakorlaton a széles körben elterjedt biztonsági tesztelési technikákat mutatjuk be. A megismert statikus és dinamikus tesztelési technikák hatékony eszközök a szoftverekben előforduló hibák azonosítására, és ezáltal az általános biztonság növelésére. A feladatok során a korábban, az input validációs mérésen megismert alkalmazás tesztelését kell elvégezni. Ennek során egy hibákat tartalmazó implementációból kell kiindulni, és az egyes módszerek segítségével folyamatosan javítani az alkalmazást. A laboratóriumi gyakorlat elvégzésével a résztvevő képessé válik a legfontosabb tesztelési technikák használatára szoftverfejlesztés során.

2. Háttéranyag

A szoftvertesztelés egy kiemelten fontos terület az implementáció során előforduló hibák kiszűrésére. Megfelelő tesztelések használatával jelentősen növelhető az elkészült termék minősége, és számottevően csökkenthető a hibákból adódó felhasználói problémák és egyéb anyagi veszteségek mértéke.

A szoftvertesztelésnek több célja is van:

- ***Funkcionális tesztelés*** során azt ellenőrizzük, hogy a szoftver a meghatározott specifikációnak megfelelően működik-e. Ebben az esetben a követelményekből szisztematikusan levezetett tesztek segítségével nagy bizonyossággal megállapíthatjuk, hogy az elkészült szoftver teljesíti az összes elvárást.

- Ezzel szemben ***biztonsági tesztelés*** esetén arról szeretnénk megbizonyosodni, hogy az alkalmazás a funkcionális tesztelés során nem ellenőrzött, összes többi esetben is elfogadhatóan működik. Mivel kimerítő tesztelés a végtelen lehetséges esetszám miatt nem megvalósítható, ezért sikeres biztonsági tesztelést végezni egy jóval nehezebb és több szakértelmet ígénylő tevékenység. Szerencsére a statisztikák alapján a biztonsági hibák döntő többsége hasonló okokra vezethető vissza, így a leghasznosabb tesztelési módszerek elsajátításával hatékony eredmények érhetőek el ezen a területen is. A biztonsági tesztelési technikákat *statikus* vagy *dinamikus* elemzésekként csoportosíthatjuk.

A hatékony biztonsági teszeléshez nagyban hozzá tud járulni, ha típusos nyelvet használunk a fejlesztés során. Első sorban a statikus elemzők számára

jelent ez előnyt, mivel a típusok elemzésével pontosabb információkkal tudják segíteni a fejlesztést. Ennek ellenére, a típusos nyelvek használata egy régre visszanyúló vita a szakmában, amiről a mai napig nincs egyértelmű álláspont.

2.1. Típusok alkalmazása

2.1.1. Jó-e vagy rossz a dinamikus típusosság?

Erre a kérdésre nincs végleges válasz[1]. A projekttől és annak céljaitól függ, hogy egy dinamikus típusokat használó nyelv megfelelő választás-e a probléma megoldására. A Python például egy *dinamikusan* típusos nyelv, nem pedig *statikusan* típusos. Ez azt jelenti, hogy a Python-interpreter nem ismeri az objektumok típusát a tényleges kódfuttatás előtt.

- A dinamikus típusokat legtöbbször script nyelvekben használják, mint például a Ruby, JavaScript, MATLAB, stb.
- A spektrum másik oldalán a statikusan típusos, általában fordított, nem pedig interpretált nyelvek találhatóak. A statikusan típusos nyelvek közé tartozik például a Fortran, a C, a C++ vagy a Java.

Mivel a Python dinamikusan típusos nyelv, a Python-értelmezőnek nem kell ismernie a kezelendő objektumok típusát az inicializálásuk előtt. Egy objektum típusa függhet például egy olyan értéktől, amelyet csak futásidőben tudunk meg, ezért a Python-értelmező csak dinamikusan következtet a típusra. A Python esetében a dinamikus típuskezelés nagy mozgásteret ad a programozónak. Azonban ennek a rugalmasságnak is van néhány hátulütője. Tipikusan mikor már elég nagy méretű a kódbázis, akkor hasznossá válik a típusok bevezetése, amely számos előnnyel járhat. Ezek közül mutatunk be párat a következő néhány példa segítségével.

2.1.2. A típusinformáció dokumentációként is szolgál

A típusok dinamikus jellege miatt a Python függvények paraméterei bármilyen objektumot elfogadhatnak, függetlenül annak típusától. A következő kódrészlet emiatt tökéletesen helyes:

```
1 def append_to_container(c, e):
2     # Code of the function
```

[1] https://cerfacs.fr/coop/python-typing

Egy ilyen kódrészlet láttán felmerül a kérdés, hogy meg lehet-e 100%-os bizonyossággal mondani, hogy hogyan kell használni ezt a függvényt anélkül, hogy látnánk a függvény megvalósítását? Nem. Itt van két "érvényes", de egymással összeegyeztethetetlen módja a függvény használatának:

```
# First possibility
c = [1, 2]
e = 3
append_to_container(c, e)
    # c is modified in-place in the function and the
    # function does not return any value

# Second possibility
c = [1, 2]
e = 3
new_c = append_to_container(c, e)
    # c is copied internally by the function and a modified
    # copy is returned.
```

Az egyik esetben a függvény helyben változtatja meg a bemeneti paraméterként kapott tömböt, másik esetben viszont visszatér a módosítottal. Az `append_to_container` függvény fejlécéből nem lehet kitalálni, hogy melyik a helyes használat a kettő közül. Erre a problémára egy lehetséges megoldás az lenne, ha a függvényt dokumentációval látnánk el, például egy docstringgel:

```
def append_to_container(c, e):
    """Modify c in-place to append e at the end of the
   container."""
    # Code of the function
```

Ez viszont további problémát vet fel: a docstringek (és általában a dokumentációk) gyakran elavulnak, nincsenek karbantartva, vagy egyszerűen csak nincsenek megfelelően megírva és nem tartalmazzák a megfelelő információt. Erre példa a következő részlet:

```
# Example of a useless docstring. The docstring only repeats
# informations already in function or arguments names.
def append_to_container(c, e):
"""Append element e to the contained c."""
    # Code of the function
```

A docstringek alkalmazásának a legnagyobb hátránya viszont, hogy azok nem elemezhetőek automatizált eszközökkel, például integrált fejlesztőkörnyezetekkel (IDE) vagy kódelemző programokkal, kivéve, ha nagyon szigorú formázási szabályokhoz igazodnak (pl Sphinx formátum, Google formátum vagy Numpy formátum).

2.1.3. Az IDE számos funkciója típusokon alapul

A következő funkciók működéséhez kifejezetten előnyös, ha típusokkal van ellátva a kód:

- Dokumentáció megjelenítése tooltippekben.
- Hibák vagy figyelmeztetések megjelenítése, ha az adott típus nem egyezik az elvárt típussal.
- A "Go to type definition" funkció alkalmazása egy új kódbázis felfedezésekor.

2.1.4. A hibák érthetőbbek lesznek

Ha előzetesen tudjuk, hogy egy változónak milyen típusúnak kell lennie, akkor futás közben, a változó értékadásakor ellenőrizhetjük, hogy a változóhoz rendelt objektum megfelelő típusú-e.

```
1 def count_bit_number(i):
2     return i.bit_length()
3
4 a = count_bit_number(3)  # result is 2
5 b = count_bit_number(1.0) # error
```

A fenti kódrészlet utolsó sora hibát okoz:

```
1 AttributeError: 'float' object has no attribute 'bit_length'
```

ahol, ha a Python tisztában lett volna azzal, hogy a `count_bit_number` függvény csak egész számokat fogad el bemenetként, a hibaüzenet valami olyasmi lehetett volna, mint például

```
1 TypeError: 'count_bit_number' got a 'float', expected 'int'
```

ami már sokkal könnyebben érthető.

2.1.5. Python kiegészítése típusokkal

Az előző részben láttuk, hogy mikor és miért éri meg típusokat használni. Ahelyett viszont, hogy a Python típus kezelési rendszerét egy az egyben megváltoztatták volna (ami rengeteg nemkívánatos következménnyel és bonyodalommal járt volna), a Python fejlesztői a típusmegjelölések bevezetése mellett döntöttek (type annotations, típus annotációk), a PEP 484 keretében.

Fontos, hogy az *annotáció* elnevezés arra utal, hogy a Python-értelmező a típusinformációkat nem használja fel. A típus annotációk a Python-értelmező számára olyanok, mint a megjegyzések, nem kerülnek feldolgozásra.

A típus annotációk használatáról jó összefoglaló olvasható a mypy könyvtár oldalán valamint a hivatalos python dokumentációban.

2.2. Statikus tesztelés

A *statikus* elemzések nem hajtják végre a programokat, csak azok forráskód szintű utasításait vagy gépi kódját "olvassák" végig és értelmezik. Az elemzés az utasítások egy memóriabeli reprezentációján történik. Ezek a technikák jól skálázódnak és nagy kódbázist is képesek kezelni. Hátrányuk, hogy nem férnek hozzá futási idejű információkhoz, ezért gyakran adnak hamis pozitív válaszokat: olyan kódrészleteket is sérülékenynek jelölhetnek, amik a valós életben sosem futnának le vagy sosem lehetne hibát kihasználni bennük.

2.2.1. Mypy

A Mypy egy statikus típusellenőrző Pythonhoz, amely képes a típusinformációk alapján hibákat azonosítani. A típusellenőrzők segítenek biztosítani, hogy a változókat és függvényeket helyesen használjuk-e a kódban. Ezzel a problémák egy részét már futtatás nélkül, statikus elemzők segítésével is ki lehet szűrni. A következő példa bemutat egy lehetséges hibát, ami a Mypy segítségével könnyen megtalálható:

```
1 number = input("What is your favourite number?")
2 # error: Unsupported operand types for + ("str" and "int")
3 print("It is", number + 1)
```

A típusjavaslatok hozzáadása nem zavarja a program futtatását. Ezekre az információkra úgy érdemes gondolni, mint a megjegyzésekre. A Python-

értelmező mindig képes az alkalmazást futtatni, még akkor is, ha a Mypy hibát jelez, bár ilyen esetben valószínűleg egy futásidejű probléma fog jelentkezni.

A Mypy-t úgy tervezték, hogy fokozatosan lehessen hozzáadni a kódhoz típusinformációkat, nem szükséges a kód egészét egyben annotálni. Emiatt a gyakorlatban könnyen hasznosítható. Támogat olyan funkciókat, mint a típuskövetkeztetés (type inference), a generikusok (generics), a hívható típusok (callable types), a tuple típusok (tuple types), a union típusok (union types), a structural subtyping, és még sok más. A Mypy használatával a programok könnyebben érthetővé és karbantarthatóvá válnak.

2.3. Dinamikus tesztelés

A *dinamikus* tesztek végrehajtás közben elemzik a szoftvert. Emiatt hozzáférnek a futásidejű információkhoz, így jóval pontosabb elemzést tudunk velük végezni. A dinamikus tesztek jellegéből fakadóan egy-két dologra figyelni kell:

- nehezen skálázódnak, mivel a teszteket gyakran manuálisan kell megírni.
- ha nem fedjük le a bemenetekkel az összes lehetséges lefutást, a nem futtatott kódrészletnek a biztonságáról nem tudunk semmit megállapítani.
- a statikus elemzéshez hasonlóan, ez is adhat hamis negatív eredményt, és nem garantálható, hogy az összes sérülékenységet meg lehet vele találni.

2.3.1. Pytest

A Pytest egy dinamikus tesztelésre készített Python alapú tesztelési keretrendszer, amely tesztkódok írására és végrehajtására szolgál. A Pytest előnyei a következők:

- A Pytest ingyenes és nyílt forráskódú.
- Egyszerű a szintaxisa, emiatt a Pytestet nagyon könnyű elkezdeni használni.

- A Pytest több tesztet is képes párhuzamosan futtatni, ami csökkenti a tesztcsomag végrehajtási idejét.
- A Pytest automatikusan megtalálja a test_*.py kezdetű vagy **_test.py végződésű fájlokat.
- A Pytest lehetővé teszi, hogy a teljes tesztcsomagnak csak egy részhalmazát futtassuk vagy kihagyjuk a végrehajtás során.

A tesztek elkészítése során a fájlneveknek "test"-el kell kezdődniük vagy végződniük, mint például `test_pelda.py` vagy `pelda_test.py`, ahhoz, hogy a Pytest automatikusan megtalálja őket. Ha a tesztek egy osztály metódusaiként vannak definiálva, az osztály nevének "Test"-tel kell kezdődnie (például: `TestOsztaly`. Ilyen esetben, az osztálynak nem lehet `__init__` metódusa. Az osztályon belül a függvények neveinek szintén "test_"-tel kell kezdődniük. Az olyan metódusok, amelyek neve nem felel meg ennek a mintának, nem lesznek tesztként végrehajtva.

```
# Tesztelendo fuggveny a muveletek.py fajlban
def osszeg(szam1, szam2):
    """Visszaadja ket szam osszeget"""
    return szam1 + szam2
```

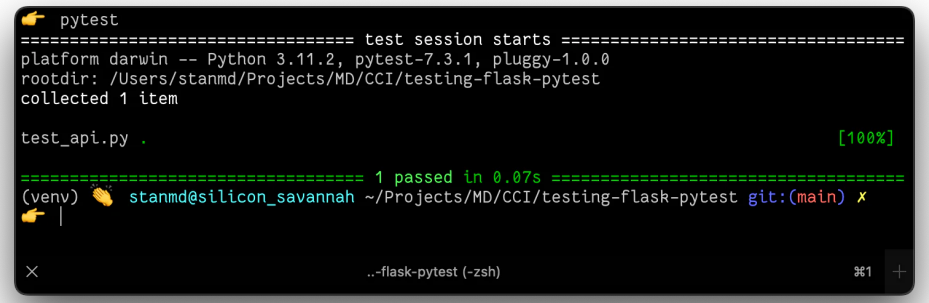
```
# Teszteset definicioja a test_peldak.py fajlban
import pytest

def test_osszeg():
    assert osszeg(1, 2) == 3
```

Példa 1. Pytest használata

Ezután a `pytest` paranccsal tudjuk lefuttatni a teszteket, vagy az egyes teszteket külön-külön, például a `pytest test_peldak.py` paranccsal.

Ennek hatására a pytest automatikusan megtalálja a teszteket a mappában, vagy a megadott fájlon belül. Megkeresi a `test` kezdetű fájlokat és az azokban található `test` kezdetű függvényeket. Az 1. képen[2] is látszik, hogy a Pytest a sikeres tesztet zöld ponttal jelöli, a sikertelen tesztet piros F-vel. Ezen felül jelzi, hogy hány teszt ment át vagy volt sikertelen.

[2] https://circleci.com/blog/testing-flask-framework-with-pytest/#c-consent-modal



1. ábra. Pytest futtatása

```python
import pytest
@pytest.fixture
def client():
    """Configures the app for testing

    Sets app config variable ``TESTING`` to ``True``

    :return: App for testing
    """

    #app.config['TESTING'] = True
    client = app.test_client()

    yield client
```

Példa 2. Tesztelhető komponens használata

A fenti példában látható `@pytest.fixture` annotáció megmondja a pytestnek, hogy a következő függvény létrehoz (a yield parancs segítségével) egy tesztelésre szánt alkalmazást. Ebben az esetben a függvény nem csinál túl sok mindent, de akár ideiglenes adatbázisfájlokat is konfigurálhat, vagy beállíthat konfigurációkat a teszteléshez (pl. a kikommentelt `app.config` sort)[3].

[3] https://codethechange.stanford.edu/guides/guide_flask_unit_testing

3. Feladatok

A laborgyakorlat egy vezetett és két önálló feladatból áll. Mindkét fázisban először egy statikus, majd egy dinamikus tesztelési technika bemutatása és kipróbálása a cél.

3.1. Vezetett rész

A példa alkalmazáson keresztül demonstráljuk a `mypy` valamint a `pytest` használatát.

3.2. Feladat: Mypy

A feladat során a cél, hogy az előző mérésből megismert `CIFF` osztályt lássa el olyan típusinformációkkal, amik segítik a fejlesztést. Ehhez hajtsa végre a következő lépéseket:

- A `ciff.py` fájlban az `__init__` függvény paramétereit és változóit lássa el típus információkkal!
- A projekt mappából futtassa a `mypy` alkalmazást a következő paranccsal:

```
mypy --config-file mypy.ini src/ciff.py
```

- Ha a `mypy` jelez valamilyen hibát, akkor javítson az annotációkon, amíg hiba nélkül le nem fut az ellenőrzés!

3.3. Feladat: Pytest

A feladat során a `pytest`et alkalmazva dinamikus tesztelést kell végrehajtani:

- Első lépésként futtassa le a teszteket, és a kimenet alapján értékelje, hogy a tesztek helyes vagy helytelen eredményt adnak. A tesztek futtatása így lehetséges:

```
pytest test_ciff.py
```

- Mint láthatja, egy teszt hibás eredményt adott: az invalid5.ciff fájl feldolgozása során az alkalmazás helyesnek tekintette a fájlt, pedig az valójában nem az. Javítsa ki a `ciff.py` fájlban a hibát: a hibaüzenet alapján implementálja a hiányzó ellenőrzést!

- Futtassa újra a teszteket, és ellenőrizze, hogy most már minden teszt helyes eredményt ad-e!

- Vizsgálja meg a tesztesetek futása során keletkezett log üzeneteket, hogy biztos minden eset megfelelően működik-e! Ha talál olyan esetet, ami bár helyes eredménnyel tér vissza, de mégsem hibátlanul fut le, akkor keresse meg szintén a `ciff.py` fájlban a problémát, és implementáljon további ellenőrzéseket.

- Az összes probléma kijavítása után futtassa ismét a `pytest` parancsot a korábbiakhoz hasonló módon, és ellenőrizze, hogy most már minden teszt megfelelően működik!