Kódolás és IT biztonság

(VIHIBB01)

Laboratóriumi gyakorlat

Input validáció

Szerző:
Futóné Papp Dorottya és Gazdag András

2023. szeptember 20.

Tartalomjegyzék

1. Oktatási célok **2**

2. Háttéranyag **2**

2.1. Adatok feldolgozása 2

2.2. Python property-k és dekorátorok 3

2.3. CrySyS Image File Format 7

2.4. Python kód debuggolása 9

3. Feladatok **11**

3.1. Vezetett rész 11

3.2. Önálló rész 13

1. Oktatási célok

A gyakorlat során input validációs kódot kell készíteni egy egyedi fájlformátumhoz. A gyakorlat célja, hogy bemutassa, milyen fontos meggyőződni arról, hogy az alkalmazásnak adott bemenetek nem tudnak meghibásodást okozni. Az egyedi fájlformátum, a CrySyS Image File Format (CIFF), specifikációját kapja kézhez, valamint a megjelenítését naívan (ellenőrzések nélkül) implementáló alkalmazást. A feladata az, hogy implementálja a szükséges ellenőrzéseket az alkalmazásba úgy, hogy hibás bemeneteket ne jelenítsen meg az alkalmazás. A gyakorlat elvégzésével képessé válik arra, hogy kiemelje egy specifikációból az implicit és explicit előfeltételezéseket a bemenő adatokkal szemben, valamint képessé válik olyan kód implementálására, ami ellenőrzi az előfeltételezések fennállását bármilyen bemenet esetén.

2. Háttéranyag

2.1. Adatok feldolgozása

Ahhoz, hogy kevesebb bug legyen az alkalmazásainkban és a kompromittálások súlyossága csökkenjen, figyelembe kell venni az alkalmazások *támadási felületét*. Ez a fogalom rávilágít az alkalmazás számára értékes dolgokra és arra, hogy az alkalmazás hogyan lép interakcióban a felhasználókkal és más folyamatokkal. A támadási felületet az alábbiak alkotják:

- Az alkalmazás számára értékes és/vagy szenzitív adat és az ezt védő kódszegmensek, valamint
- azok a végrehajtási ágak, amiken keresztül adat és/vagy parancs érkezik vagy távozik, és az ezeket védő kódrészletek.

A támadási felület figyelembe vétele az első lépés ahhoz, hogy input validációt valósítsunk meg az alkalmazásokban. A programozó számára alapvetőnek kell lennie, hogy a felhasználóktól érkező bármely bemenetet akár a támadó is adhatja, ezért a bemenet megbízhatatlan. A megbízhatatlan bemenet kezeléséhez három nagyobb megközelítés létezik:

- Normalizálás / kanonikalizálás: a bemenet átalakítása annak legegyszerűbb és elvárt formájára. Például, amikor a bemeneti karaktersorozatokat átkonvertáljuk egy előre meghatározott kódolásra (pl. Unicode), akkor normalizáljuk ezeket a bemeneteket, mivel a karaktersorozat bináris reprezentációja megváltozik (az eredeti bitek elvesznek, a szemantika viszont megmarad). Egy másik példa az XML dokumentumok esete, ahol az üres tageket kétféleképpen is leírhatjuk: `<tag/>` vagy `<tag></tag>`. Amikor megváltoztatjuk a bemenetet, hogy csak az egyik leírás szerepelhessen, átalakítást végzünk egy ekvivalens formára.

- Szűrés: bizonyos elemek törlése megadott kritérium alapján. Például, a `<` és `>` karakterek törlése webes formok bemeneteiből egy olyan szűrési lépés, amivel bizonyos webes támadásokat (pl. cross-site scripting) próbálunk kivédeni. A szűrést mindig normalizálás / kanonikalizálás után végezzük el!

- Validálás: Annak ellenőrzése, hogy az adat struktúrája és szemantikája ésszerű-e. Például, ha a bemenet egy karaktersorozat, ami egy `YYYY-MM-DD` formátumú dátumot hivatott reprezentálni, akkor a validálás során

  – szintaktikai ellenőrzést (pl. a bemenet pontosan két `-` karaktert tartalmaz, ami a bemenetet három számmá bontja szét és a számok a megadott hosszra paddelve vannak), és

  – szemantikai ellenőrést (pl. a három szám egyike sem negatív, a második szám 1 és 12 között van a 12 hónapnak megfelelően és a harmadik szám értéke az adott hónap napjainak számába esik) is

  végre kell hajtani. A szűréshez hasonlóan a validációt is meg kell előznie a normalizálásnak!

A robusztus és biztonságos kód írásához mind a három megközelítést alkalmazni kell.

2.2. Python property-k és dekorátorok

A Python nyelv alapvetően nem rendelkezik olyan hozzáférés módosító kulcsszavakkal, mint amilyen a `private` vagy a `protected`. Ezért a Python programozók egy elnevezési konvenciót használnak: ha egy attribútum neve

aláhúzással kezdődik, pl. `class._x`, akkor az attribútumot privátként kezelik, egyébként publikusként. Ez a konvenció azonban nem adja meg ugyanazt a szintű adat beágyazást a Python osztályok számára, mint amit a Java vagy C# nyelvek nyújtanak.

Az adat beágyazás eléréséhez használhatunk getter és setter metódusokat, ahogy azt az alábbi példa is mutatja.

```
class Example:
    def __init__(self, v):
        """
        Constructor
        """
        self._value = v

    def get_value(self):
        """
        Getter method
        """
        return self._value

    def set_value(self, v):
        """
        Setter method
        """

        # ...
        # input validation
        # ...

        self._value = v
```

Azonban ennek a megoldásnak az a hátránya, hogy a beágyazott adatokkal végzett számításokat nehéz olvasni. Vegyük például azt az esetet, amikor a `_value` attribútum egész számokat tartalmaz. Ahhoz, hogy két `_value`-t összeadjuk az `Example` osztály két példányából, a következő kódra van szükség:

```
a = Example(1)
b = Example(2)
result = a.get_value() + b.get_value()
```

Minél összetettebb a számítás, annál nehezebben olvasható a kód. Az osztály akár implementálhatna egy `add(self, Example)` metódust is, hogy jobban megfeleljen az objektum orientált programozás szabályainak, de ez a megoldás sem segítene az olvashatóságon.

A probléma megoldását a *property-k* adják. Ezzal a konstrucióval a programozók tarthatják az elnevezési konvenciókat és adat beágyazást is implementálhatnak. Property-k létrehozásához a speciális `property()` függvényt kell használni, ami argumentumként három függvényt (egy getter, egy setter és egy deleter) és egy dokumentációs sztringet vár. A property-t aztán az attribútum helyett lehet használni a kódban.

```
class Example:
    def __init__(self, v):
        """
        Constructor
        """
        self._value = v

    def get_value(self):
        """
        Getter method
        """
        return self._value

    def set_value(self, v):
        """
        Setter method
        """

        # ...
        # input validation
        # ...

        self._value = v

    value = property(get_value, set_value, None, "")


a = Example(1)
b = Example(2)
result = a.value + b.value
```

Amikor a property-t olvassuk, az argumentumként megadott getter metódus fut le. Amikor a property-t írjuk, akkor a megadott setter metódus hívódik meg.

A `property()` függvény egy ún. *dekorátor* függvény: olyan függvény, ami argumentumként másik függvényt vár, módosítja annak végrehajtását és visszatér vele. A dekorátor függvényeket közvetlenül meghívhatjuk, ahogy azt az előző példában láthattuk. Azonban a speciális `@` szimbólum használatával úgy dekorálhatunk függvényeket, hogy nem kell explicit meghívni a dekorátor függvényt. Az előző példát ennek megfelelően az alábbiként is írhatjuk:

```
class Example:
    def __init__(self, v):
        """
        Constructor
        """
        self._value = v

    @property
    def value(self):
        """
        Getter
        """
        return self._value

    @value.setter
    def value(self, v):
        """
        Setter
        """

        # ...
        # input validation
        # ...

        self._value = v
```

A `value(self)` metódust úgy dekoráltuk, hogy az az ugyanazon nevű property getter metódusa legyen. Az osztály tartalmazza továbbá a `value(self, t)` metódust (vegyük észre az operátor túlterhelést!), amit a `value` property setterévé dekoráltunk. A property felhasználása nem változik, az előzőeknek megfelelően lehet használni.

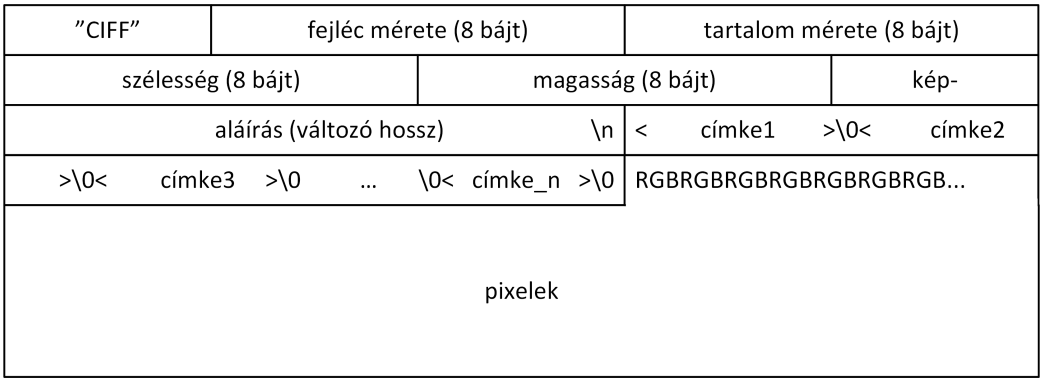
A Python három beépített dekorátort ismer, amikor a `@` karaterrel használhatunk. Ezek a következőek:

- `property` - property-k létrehozása, ahogy az előbbiekben tárgyaltuk,
- `staticmethod` - statikus metódus létrehozása, ami nem fér hozzá semmilyen `self` változóhoz, és
- `classmethod` - úgy módosítja a metódust, hogy az első paraméterként nem a `self` objektumot kapja, hanem az osztályt.

Saját dekorátorokat is létrehozhatunk, ezek tárgyalása azonban túlmutat ezen a laboratóriumi gyakorlaton. További információt a https://realpython.com/primer-on-python-decorators/ oldalon találhatunk.

2.3. CrySyS Image File Format

A CrySyS Image File Format (CIFF) egy egyedi, tömörítetlen képformátum. Egyedisége miatt nem létezik olyan szabadon hozzáférhető könyvtár vagy modul, ami a feldolgozását implementálná. A fájlformátum áttekintő ábráját az 1. ábra mutatja.



1. ábra. CrySyS Image File Format

A CIFF specifikációnak megfelelő fájlok egy fejléccel kezdődnek. A fejlév az alábbi részekből tevődik össze:

- *Magic:* 4 karakter, amik együttesen a "CIFF" szócskát adják.
- *Fejléc mérete:* Egy 8 bájt hosszú egész szám, aminek az értéke megadja a fejléc méretét (minden mezőt figyelembe véve), vagyis a fájl első `fejléc mérete` számú bájtjai adják az egész fejlécet.
- *Tartalom mérete:* Egy 8 bájt hosszú egész szám, aminek az értéke megadja a fájl végén található pixelek méretét. Az értékének egyeznie kell a `szélesség * magasság * 3` értékkel.
- *Szélesség:* Egy 8 bájt hosszú egész szám, aminek az értéke a kép szélessége. Az értéke lehet 0, de ebben az esetben nem lehetnek pixelek a fájlban.
- *Magasság:* Egy 8 bájt hosszú egész szám, aminek az értéke a kép magassága. Az értéke lehet 0, de ebben az esetben nem lehetnek pixelek a fájlban.
- *Képaláírás:* változó hosszú ASCII karaktersorozat, ami `\n`-nel végződik és a képaláírást adja meg. Mivel a `\n` speciális karakter a fájl formátum számára, a képaláírás nem tartalmazhat ilyen karaktert.
- *Címkék:* Változó darab, változó hosszúságú ASCII karaktersorozat, amiket `\0` választ el. A címkék nem lehetnek többsorosak. Az utolsó címkét is `\0`-nak kell követnie.

A fejlécet a képet alkotó pixelek követik RGB formátumban, minden komponens egy bájtot teszt ki. A pixelek pontosan `tartalom mérete` bájtot kell, hogy kitegyenek.

A laboratóriumi gyakorlathoz egy olyan naív Python alkalmazást fog kapni, ami képes a specifikációnak megfelelő CIFF képeket megjeleníteni. Az alkalmazás két fájlból áll, `view.py` és `ciff.py`. Az első implementálja a grafikus felhasználói felületet, a második pedig a CIFF képeket reprezentáló osztályt. A feladata az, hogy a második fájlban található osztályban implementálja az input validációhoz szükséges ellenőrzéseket. Kis segítség az osztály kódjának megértéséhez:

- Az osztály több property-vel is rendelkezik, külön-külön a fejléc minden részéhez és a pixelek listájához is. A property-khez tartoznak getter és setter metódusok is.
- Az `is_valid` property jelzi az alkalmazás számára, hogy a `CIFF` osztály egy példánya egy érvényes CIFF kép adatait tartalmazza-e. A property getter metódusának boolean értéket kell visszaadnia.
- Az osztály tartalmaz egy statikus feldolgozó metódust, amit `@staticmethod` dekorátorral láttunk el, ez a `CIFF.parse_ciff_file()`. Ez a metódus a CIFF fájl beolvasásakor hívódik meg és a `CIFF` osztály egy példányát kell visszaadnia. A metódus a `struct` Python modult használja fel, hogy a megadott formátumnak megfelelően értelmezze bájtok egy sorozatát. A modul dokumentációja itt érhető el. A formátumot meghatározó karaktereket meg kell ismernie a gyakorlat előtt!

2.4. Python kód debuggolása

Python alkalmazások debuggolására két alapvető megközelítés létezik. Az első megközelítés a fejlesztőkörnyezetbe, pl. IDLE, VSCode, épített funkciókra épül. Az IDLE debuggoláshoz használahtó funkcióinak egy jó áttekintése itt érhető el. A laboratóriumi gyakorlaton való részvételhez kötelező ennek az oldalnak az átolvasása! VSCode környezetben pedig ez a leírás segíthet a hibakeresésben.

A második megközelítés egy kifejezetten debuggolásra készített Python modulra épül és olyan esetekben alkalmazható, amikor nincs grafikus felhasználói felület vagy fejlesztőkörnyezet. A modul neve `pdb`, Python Debugger, és használat előtt importálni kell a kódba. Importálás után breakpointot a `pdb.set_trace()` függvénnyel helyezhetünk el a kódba. A függvény megállítja a végrehajtást meghívásakor és lehetőséget nyújt a fejlesztőnek arra, hogy megvizsgáljon változókat, scope-on belüli függvényeket és metódusokat hívjon meg, valamint egyszerű Python utasításokat adjon ki. A modul különböző parancsokat is implementál, ezek közül a legismertebbek a következőek.

- `p` - print, változók és egyéb számítási eredmények kiíratása,
- `s` - step into, következő utasításként végrehajtandó függvénybe/metódusba lépés,

- `n` - next, következő sor végrehajtása,
- `u` - until, végrehajtás, amíg egy olyan sorhoz nem érünk, aminek a sorszáma nagyobb (kifejezetten ciklusok átlépéséhez hasznos), és
- `c` - continue, következő breakpointig vagy kilépésig folytatja a végrehajtást.

3. Feladatok

3.1. Vezetett rész

A labor gyakorlat elején először értelmezzük a specifikációt és írjuk össze, hogy milyen ellenőrzéseket kell implementálni!

- Minden beolvasott bájt után az implementációnak ellenőriznie kell, hogy elérte-e a fájl végét (end of file, EOF). Ehhez meg kell nézni, hogy az olvasás üres sztringet adott-e vissza.
- Magic: A karaktereknek a `CIFF` szócskát kell kiadniuk. A naiv implementáció jelenleg egyetlen művelettel dekódolja a 4 karaktert, de ha olyan teszt vektort kapna, ami csak 3 karakterből áll, összeomlana.
- Fejléc mérete: $\in [38; 2^{64} - 1)$. A 38-at úgy kapjuk meg, hogy kiszámoljuk a lehető legrövidebb, még érvényes fejléc méretét (üres képaláírás, nincsenek címkék). A $2^{64} - 1$) a legnagyobb szám, amit a *fejléc mérete* mezőben találhatunk. A naiv implementáció előjeles egészként olvassa be ezt a mezőt (a `struct` modul `q` format sztringjét használja `Q` helyett), emiatt a $2^{64} - 1$-t -1-nek értelmezi.
- Tartalom mérete: $\in [0; 2^{64} - 1)$, mivel elképzelhető, hogy a fájlban egyáltalán nincsenek pixelek, de az is lehet, hogy a lehető legtöbb pixelt tartalmazza a fájlt. A mező értékének meg kell egyeznie a `szélesség*magasság*3` értékkel. A naiv implementáció előjeles számként olvassa be (`struct` format sztring).
- Szélesség: $\in [0; 2^{64} - 1)$, ugyanazok a megfontolások érvényesek, mint a *tartalom mérete* esetén. A naiv implementáció előjeles számként olvassa be (`struct` format sztring).
- Height: $\in [0; 2^{64} - 1)$, ugyanazok a megfontolások érvényesek, mint a *tartalom mérete* esetén. A naiv implementáció előjeles számként olvassa be (`struct` format sztring).
- Képaláírás: Csak ASCII karaktereket tartalmazhat. A naiv implementáció megpróbálja így dekódolni a beolvasott bájtot, de a kódból hiányzik a hibakezelés (`try-except` arra az esetre, ha a bájtot nem lehet ASCII karakterként dekódolni. Ahhoz, hogy a képaláírás semmiképp ne tartalmazzon `\n` karaktert, az első ilyen karakter beolvasása

után úgy kell kezelni a fájlt, hogy elértük a képaláírás végét (a naiv implementáció ezt teszi).

- Címkék: Csak ASCII karaktereket tartalmazhat, a beolvasásnál itt is hiányzik a hibakezelés. A beolvasott karaktereket egyenként ellenőrizni kell, hogy egyik se legyen `\n`. A naiv implementáció úgy olvassa be a címkéket, hogy a `\0`-kat is beolvassa a címkék végére, de ezeket nem jeleníti meg a grafikus felületen. Ellenőriznünk kell, hogy a legutolsó címke végén is szerepel-e a `\0`!

- Pixelek: Pontosan annyi pixelt kell tartalmazni a fájlnak, amennyit a *tartalom mérete* mező mond. Ezt könnyen implementálhatjuk úgy, hogy rendszeresen ellenőrizzük, hogy elértük-e az EOF-ot: a fájl érvénytelen, ha hamarabb elérjük a fájl végét, mint kéne, vagy ha nincs EOF a megadott darabnyi pixel után.

Az önálló munkát segítendő, a magic karakterek ellenőrzéséhez egy példakód:

```
# read the magic bytes
magic = ciff_file.read(4)
# read may not return the requested number of bytes
# TODO: magic must contain 4 bytes. It not, raise Exception
if len(magic) != 4:
    raise Exception("Invalid magic: length")
bytes_read += 4
# decode the bytes as 4 characters
try:
    new_ciff.magic = magic.decode('ascii')
except Exception as e:
    raise Exception("Invalid magic: non-ASCII")
# TODO: the magic must be "CIFF". If not, raise Exception
if new_ciff.magic != "CIFF":
    raise Exception("Invalid magic: value")
```

3.2. Önálló rész

A labor teljesítéséhez két helyen lehet módosítani a `ciff.py` fájlt:

1. a `CIFF.is_valid` property-hez tartozó metódusban lehet implementálni a validációs ellenőrzéseket a beolvasást követően, vagy

2. a `CIFF.parse_ciff_file()` metódusban a beolvasással kapcsolatos hiányzó ellenőrzéseket és format sztring hibákat mindenképpen javítani kell, de itt meg lehet valósítani a teljes validációt is.

A naiv implementáció `CIFF.parse_ciff_file()` metódusában található egy kikommentezett `try-except` blokk. Ez a blokk bármilyen kivétel keletkezése esetén beállítja az `CIFF.is_valid` property-t `false` értékre. Amennyiben kikommentezi ezt a `try-except` blokkot, elég csak kivételeket dobnia a megfelelő helyeken és a kód automatikusan érvénytelennek fogja minősíteni a bemenetet. A munkát segítendő, a `CIFF.parse_ciff_file()` metódusban több helyen is `TODO:` kommentek találhatóak, amikben röviden le van írva a megvalósítandó validáció. A teszteléshez használható példa fájlok a `test-vectors` mappában találhatóak.

Az elkészült implementációról leadandó beszámolót a `moodle_submission.py` szkript készíti el. A szkript meghívja a `CIFF.parse_ciff_file()` metódust minden tesztvektorra és a `CIFF.is_valid` property értéke alapján kiírja, hogy az adott fájlt érvényes vagy érvénytelen bemenetnek érzékelte-e a kód. Amennyiben egy bemenetre összeomlana a kód, azt is jelzi a kimeneten. A munka végeztével futtassa a `moodle_submission.py` szkriptet és a kimenet alapján töltse ki a Moodle kérdéssort!