

Kyberturvallisuus – ohjelmistokehitys

Oppijan arvioitu taitotaso: keskitaso tai asiantuntija

Kyberturvalliset käytännöt

Alla on joitain käytäntöjä kyberturvalliseen ohjelmistokehitykseen. Tämän materiaalin koodit on muotoiltu ilmaisella online-työkalulla (HighlightCode 2024).

Tietokannan käsittely

Syöttötietojen puhdistaminen on kiistatta tärkein osa tietokannan käsittelyä, kun pääsy ja riittävät oikeudet on myönnetty. Syötteen puhdistuksessa käyttäjän antamat tekstitiedot puhdistetaan niin, että ne eivät sisällä sopimattomia merkkejä tai sanoja. Tämän tarkoituksena on estää tietojen syöttöjärjestelmän väärinkäyttö ja siten estää SQL-injektio ja vastaavat haitalliset toiminnot. (SQL-lisäyksessä syöttöteksti sisältää koodia, jota voidaan käyttää esimerkiksi tietojen poistamiseen tietokannasta.)

Yksi tapa estää SQL-injektio Pythonissa ja SQLite3:ssa on parametrisoida syöte (kolme pistettä ovat koodin paikkamerkkejä):

```
1. import sqlite3
2. conn = sqlite3.connect('example.db')
3. cursor = conn.cursor()
4.
5. ...
6.
7. # Parameterised query where "query" and "user_input" are strings
8. cursor.execute(query, (user_input,))
9.
10....
11.
12. conn.close()
```

Muita työkaluja, kuten HTMLPurifier (PHP) tai Bleach (Python), voidaan käyttää HTML-syötteen jäsentämisessä.

Joissakin ohjelmointikirjastoissa voi olla myös olemassa olevia, helposti integroitavia työkaluja syötteen puhdistamisen lisäämiseksi ohjelmiin. Esimerkiksi Java-kehittäjät voivat käyttää OWASP ESAPI -kirjastoja, ja JavaScriptille on esimerkiksi komponentti nimeltä Validate.js, joka sisältää ennalta määritetyt validoijat tietojen syöttämisen puhdistamiseen.



Käyttäjän käyttöoikeudet ja oikeudet

Ohjelmistokehittäjä voi määrittää käyttöoikeudet useille ohjelmistojärjestelmän osille, kuten tiedostoille, kansioille, verkkoporteille ja tietokannoille. Käyttöoikeudet voivat sisältää lukemisen, kirjoittamisen ja suorittamisen.

Poikkeusten käsittely

Poikkeusten käsittelyssä ohjelma käsittelee virheitä tai odottamattomia tilanteita. Joitakin mahdollisia tapauksia ovat kokonaislukujako nolllalla, yritetään lukea tai kirjoittaa taulukon ulkopuolelle tai yritetään avata olematon tiedosto lukemista varten. Ohjelma, jossa on hyvin harkittu ja hyvin toteutettu poikkeusten käsittely, ei kaadu tällaisissa olosuhteissa. Sen sijaan se poistuu niistä sulavasti ja aiotun käyttäytymisen mukaan joko lakkaa toimimasta tai jatkaa suoritusta normaalisti. Lisäksi ohjelma voi näyttää kuvauksen poikkeuksesta tai tallentaa sen lokiin – paljastamatta kuitenkaan arkaluonteisia tietoja.

Alla on esimerkki Python-ohjelmasta, joka tekee seuraavat:

1. Otaa tiedostonimen ja tekstin syöteargumenteiksi.
2. Yrittää avata tiedoston nimeltä tiedostonimi ja lukee sen sisällön.
3. Liittää tekstin sisältöön.
4. Tallentaa tiedoston tiedostonimenä.

Esimerkkiohjelmassa on poikkeusten käsittely useissa tilanteissa:

- Tiedostoa ei löydy.
- Jokin muu, määrittelemätön virhe ilmenee tiedostoa luettaessa tai kirjoitettaessa.

```
1. import argparse
2.
3. def read_file(filename):
4.     try:
5.         with open(filename, 'r') as file:
6.             file_contents = file.read()
7.             print("File Contents:")
8.             print(file_contents)
9.             return file_contents
10.    except FileNotFoundError:
11.        print(f"File '{filename}' not found.")
12.        return None
13.    except Exception as e:
14.        print(f"An error occurred while reading the file: {e}")
15.        return None
16.
17. def append_text(filename, text):
18.     try:
19.         with open(filename, 'a') as file:
20.             file.write(text)
```



```
21.         print(f"Appended '{text}' to {filename}")
22.     except FileNotFoundError:
23.         print(f"File '{filename}' not found.")
24.     except Exception as e:
25.         print(f"An error occurred while appending to the file: {e}
26.             ")
27. def main():
28.     parser = argparse.ArgumentParser(description="Read and append
29.         text to a file")
30.     parser.add_argument("filename", help="Name of the file")
31.     parser.add_argument("text", help="Text to append to the file")
32.     args = parser.parse_args()
33.     file_contents = read_file(args.filename)
34.     if file_contents is not None:
35.         append_text(args.filename, args.text)
36.
37. if __name__ == "__main__":
38.     main()
```

Harjoitus: Miten poikkeuksia käsitellään yllä olevassa koodissa? Mitä tapahtuu, jos tiedostoa ei löydy lukemisen tai kirjoittamisen aikana?

Muita suojattuja käytäntöjä

Muut turvalliset ohjelmistokehityskäytännöt sisältävät ainakin seuraavat näkökohdat:

- Muistin hallinta
- Tietotyypit
- tuotantopanosten käsittely (kuten tuotantopanosten puhdistus ja esimerkiksi sen varmistaminen, että annetut arvot ovat hyväksyttävällä alueella)
- moniajonäkökohdat kilpailuolosuhteiden välttämiseksi (arvo päivitetään useita kertoja määrittelemättömässä järjestyksessä, mikä johtaa arvaamattomaan käyttäytymiseen ja mahdollisesti vaarallisiin tilanteisiin)

Muistinhallinnassa on tärkeää välttää muistivuotoja vapauttamalla varattua muistia ja hyviä käytäntöjä objektien viittauksen **poistamiseen**, kun niitä ei enää tarvita, ja resurssien, kuten tiedostokahvojen ja verkkoyhteyksien, hallintaan **kontekstinhallintaohjelmien avulla**.

Alemman tason ohjelmointikielissä, kuten C tai C++, ohjelmistokehittäjän on ehkä vapautettava muisti manuaalisesti (ellei tehtävän automatisoivia kirjastoja käytetä). Toisaalta korkeamman tason ohjelmointikielillä, kuten Pythonilla, on sisäänrakennettu roskien keräys käytetyn muistin vapauttamiseksi, kun sitä ei enää tarvita. Viittausten poistaminen ja kontekstinhallinta voivat kuitenkin parantaa muistin käytön tehokkuutta. Näin voit poistaa viittauksen luettelo-objektiin Pythonissa:



```
1. x = [1, 2, 3]
2. y = x
3.
4. del x
5. # The list still exists because y references it.
6.
7. del y
8. # The list has been completely dereferenced because nothing references it.
9. # Calling x or y at this point will cause a NameError.
```

Kontekstinhallintaa voidaan käyttää Pythonissa seuraavasti:

```
1. # Instead of
2. #     f = open("file.txt")
3. #     data = f.read()
4. #     f.close()
5. # use the uncommented code below.
6.
7. with open ("file.txt") as f:
8.     data = f.read()
9.
10. # The code above is a minimal example to demonstrate a context manager;
11. # in software projects, it is recommended to use exception handling
12. # (e.g. for cases where "file.txt" is missing, permissions to read it are
13. # insufficient, etc.) and other measures to make the program more secure.
```

Toinen turvallisen muistinhallinnan näkökohta liittyy matriisin ylivuotoihin. Vaikka matriisien ulkopuolella lukeminen ja kirjoittaminen voi olla todennäköisempää, kun työskentelet alemman tason kielten kanssa, jopa korkeamman tason kielillä kirjoitetut ohjelmat voivat olla alttiita puskurin ylivuotohyökkäyksille. Tämän lieventämiseksi voidaan sisällyttää poikkeusten käsittely muistinhallintakoodiin. Esimerkiksi alla olevassa koodissa on poikkeuskäsittely tapauksissa, joissa muistin varaus epäonnistuu ja tapaus, jossa puskuriin kirjoittaminen epäonnistuu.

```
1. import ctypes
2.
3. def allocate_secure_memory(size):
4.     """
5.     Allocates secure memory using ctypes.
6.
7.     Args:
8.         size (int): Size of the memory buffer to allocate.
9.
```



```
10. Returns:
11.     ctypes.c_buffer: A secure memory buffer.
12.     """
13.     try:
14.         buffer = ctypes.create_string_buffer(size)
15.         return buffer
16.     except (ValueError, TypeError) as e:
17.         print(f"Memory allocation error: {e}")
18.         return None
19.
20. def main():
21.     # Allocate a buffer of 64 bytes
22.     buffer = allocate_secure_memory(64)
23.     if buffer:
24.         try:
25.             # Securely manage memory by ensuring we don't write past
                # the allocated space
26.             buffer.value = b"Safe string that fits within allocated
                buffer"
27.             print(f"Allocated buffer content: {buffer.value}")
28.         except (ValueError, TypeError) as e:
29.             print(f"Memory error: {e}")
30.
31. if __name__ == "__main__":
32.     main()
```

Tietotyyppien osalta **muuttumattomien tyyppien** käyttäminen tai ohjelman asettaminen **hyväksymään vain tietyt tietotyypit** parantavat turvallisuutta estämällä tahattomat muutokset. Lisäksi **tyyppimerkinnät** voivat parantaa koodin luettavuutta ja auttaa havaitsemaan tyyppiin liittyvät virheet. Alla on esimerkki kahden kokonaisluvun turvallisesta lisäämisestä. Huomaa, että tyyppimerkinnän ja hyväksyttävien syöttötyyppien kokonaislukuihin rajoittamisen lisäksi koodi ottaa huomioon myös tämän tietotyypin rajoitukset käsittelemällä yli- ja alivuotoon liittyviä poikkeuksia. Vaikka Pythonin kokonaisluvut eivät yleensä ylitä tai alivuodata, tällaisten tapausten käsittely voi olla välttämätöntä muiden tietotyyppien kanssa tai käytettäessä eri ohjelmointikieltä.

```
1. def add_securely(a: int, b: int) -> int:
2.     # Ensures that the function parameters are of the expected type
    e
3.     if not (isinstance(a, int) and isinstance(b, int)):
4.         raise TypeError("Input values must be integers")
5.
6.     # Check for overflow
7.     if a > 0 and b > 0 and a + b < 0:
8.         raise OverflowError("Integer overflow occurred")
```



```
9.
10.     # Check for underflow
11.     if a < 0 and b < 0 and a + b > 0:
12.         raise OverflowError("Integer underflow occurred")
13.
14.     return a + b
```

Yllä olevaa toimintoa voidaan käyttää kriittisiin toimintoihin, jotka liittyvät integroituihin lisäyksiin, kuten tarvittavan tai allokoituihin käytettävissä olevan muistin laskemiseen.

Syötteiden käsittelyn osalta tekniikat, kuten **validointi**, **puhdistus** ja kyselyparametrisointi, **voivat** tehokkaasti parantaa koodin suojausta (katso myös edellä oleva kohta [Tietokannan käsittely](#)).

Esimerkiksi seuraava funktio poistaa syötteestä kaikki paitsi aakkosnumeeriset merkit:

```
1. def process_input(user_input: str) -> str:
2.     # Validate input against a whitelist of allowed characters
3.     allowlist = set('abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
4.     TUVWXYZ0123456789')
5.     sanitised_input = ''.join(filter(lambda x: x in allowlist, user
6.     r_input))
7.     # Return the sanitised input
8.     return sanitised_input
```

Turvallinen **moniajo** sisältää **samanaikaisuuden hallinnan** ja **kilpailuolosuhteiden välttämisen** kierteityksen, monikäsitteilyn ja globaalien muuttujien turvallisella käytöllä. Alla oleva koodi käyttää 10 säiettä laskemaan välillä 0 - 10000. Sen turvallisuutta parannetaan käyttämällä lukkoa kilpailuolosuhteiden välttämiseksi ja ottamalla käyttöön vain kaksi globaalia muuttujaa - niin vähän kuin on tarpeen.

```
1. import threading
2.
3. counter = 0
4. lock = threading.Lock()
5.
6. def increment_counter(n):
7.     global counter
8.     with lock:
9.         for _ in range(n):
10.             counter += 1
11.
12. def count(thread_count, target_value):
13.     increment_value = target_value // thread_count
14.
15.     threads = []
16.
17.     for _ in range(number_of_threads):
```



```
18.         thread = threading.Thread(target=increment_counter, args=(
            increment_value,))
19.         threads.append(thread)
20.         thread.start()
21.
22.     for thread in threads:
23.         thread.join()
24.
25. if __name__ == '__main__':
26.
27.     number_of_threads = 10
28.     target_value = 10000
29.
30.     count(number_of_threads, target_value)
31.
32.     print(f"Final counter value: {counter}")
```

Yleisten muuttujien määrän minimointi voi myös parantaa ja optimoida muistinhallintaa, koska vähemmän muuttujia on pidettävä ajan tasalla koko ohjelman käytön ajan.

Pohdintatehtävä: Onko sinulla ohjelmistokoodia, jota voisit parantaa noudattamalla tässä materiaalissa lueteltuja käytäntöjä?

Kolmannen osapuolen kirjastot

Kolmannen osapuolen kirjastojen käytössä ohjelmistoprojekteissa on joitain näkökohtia. Vaikka ne voivat säästää aikaa tarjoamalla toteutuksia ja ratkaisuja, ne voivat sisältää virheitä, olla huonosti ylläpidettyjä (jolloin ne ovat alttiita haavoittuvuuksille tai aiheuttaa integrointiongelmia, jotka voivat johtaa koodin merkittävään uudelleenkirjoittamiseen uusissa versioissa) tai niillä voi olla ei-toivottuja ominaisuuksia, kuten arkaluonteisten tietojen lähettäminen kolmannen osapuolen palvelimille käsittelyä varten. Lisäksi kirjastojen lisensointi on tärkeä huomioitava asia. Näin varmistetaan kehitettävään ohjelmistoon liittyvät oikeudet ja rajoitukset.

Valmis!

Onnittelen! Olet täyttänyt materiaalit!

Viittaukset

HighlightCode. 2024. *Online Highlight Code | Paste into Microsoft Word or OneNote etc.* Available at: <https://highlightcode.com/>. Accessed 25 June 2024.

