УДК 004

Обеспечение безопасности запуска, отправленного пользователем Python кода на стороне сервера

Фатеенков Данила Витальевич
Приамурский государственный университет имени Шолом-Алейхема
Студент

Аннотация
В статье рассмотрены способы обеспечения безопасности процесса запуска исходного кода, который отправляет пользователь на сервер. Безопасность данного процесса необходима для предотвращения сбоев при компиляции программ на стороне сервера. Рассматриваются встроенные методы ограничения использования ресурсов компилятора, а также способы компиляции кода на стороне сервера.
Ключевые слова: Python, PyPy, Docker, информационная безопасность, компиляция кода, виртуализация, изолирование процессов

Securing the server-side launch of Python code sent by the user

Fateenkov Danila Vitalievich
Sholom-Aleichem Priamursky State University
Student

Abstract
The article discusses how to secure the process of launching the source code that the user sends to the server. The security of this process is necessary to prevent failures when compiling programs on the server side. We consider the built-in methods of limiting the use of compiler resources, as well as ways to compile the code on the server side.
Keywords: Python, PyPy, Docker, information security, code compilation, virtualization, process isolation

1. Введение

1.1 Актуальность

В настоящее время существует большое количество компиляторов, работающих в онлайн среде. Процесс компиляции кода работает намного сложнее, чем может показаться на первый взгляд. Важно учитывать, что пользователь может отправлять вредоносный код на сайт, который может помешать работе сервера (а в худших сценариях и привести к повреждению оборудования).

Код, написанный пользователями, может содержать ошибки или уязвимости, которые могут быть использованы для получения несанкционированного доступа или сбоев системы. Обеспечение

безопасности при запуске и компиляции пользовательского кода позволяет проверить его на наличие потенциальных уязвимостей и предотвратить возможные проблемы.

1.2 Обзор исследований

И.В. Буянова и И.С. Замулин рассмотрели доступные технологии изоляции недоверенного кода от системы, в которой выполняется его тестирование [1]. В статье рассмотрены преимущества виртуализации с использованием ПО Docker и другие методы обеспечения безопасности при проверке кода.

Д.А. Дорофеев рассмотрел способы безопасного запуска программного кода, написанного третьими лицами [2]. В статье рассмотрены способы изоляции вычислительных ресурсов для обеспечения безопасности.

Д.И. Валиуллина и И.А. Зиганшин описали применение PyTest для проведения тестирования Python кода [3]. В статье рассмотрены преимущества и способы применения библиотеки PyTest.

1.3 Цель исследования

Цель – рассмотреть способы обеспечения безопасности при запуске и компиляции кода на стороне сервера.

2. Материалы и методы

Для реализации поставленной цели используется язык программирования Python и встроенные в него библиотеки.

3. Результаты и обсуждения

Запуск отправленного третьими лицами кода, написанного на языке программирования Python, является опасным процессом, если не предпринять меры по обеспечению безопасности. Отсутствие предварительной проверки и запуска кода в виртуальных средах может привести к большому количеству проблем:

1. Злоумышленники могут использовать сервер как платформу для запуска вредоносного кода, который может привести нарушению функциональности системы.
2. Неконтролируемое выполнение пользовательского кода может привести к использованию больших объемов ресурсов, таких как процессорное время, память или сетевая пропускная способность. Это может вызывать проблемы производительности и отказы в обслуживании других пользователей.
3. При отсутствии изоляции запускаемого кода, пользователь может получить доступ к данным системы или взаимодействовать с другими приложениями и элементами сервера, что может привести к утечке информации или нарушению работы сервера.

Важно предусмотреть такие моменты, чтобы избежать серьёзных проблем в будущем. Процесс компиляции и запуска кода можно рассмотреть на примере работы IDE в онлайн пространстве.

В настоящее время существует большое количество онлайн IDE и многие работают по следующему принципу:

1. Пользователь вводит в специальную форму код на интересующем его языке программирования и нажимает кнопку «Запустить».

2. Код компилируется на стороне сервера и проверяется на ошибки.

3. В случае удачной компиляции пользователю предлагается ввести входные данные в программу.

4. После ввода данных, код запускается и на вход ему поступают те данные, которые ввёл пользователь. После этого сервер возвращает результат работы кода на заданных значениях.

Как правило, на стороне сервера уже предустановлены компиляторы различных ЯП и виртуальные среды, на которых производится запуск кода. Один из самых популярных языков программирования Python не является исключением и также используется в онлайн компиляторах. Его достоинство в обеспечении безопасности заключается в том, что в нём присутствует большое количество методов, библиотек и функций для ограничения работы тех или иных частей кода.

Один из самых простых способов ограничить возможности пользователя (в рамках обеспечения безопасности) в работе с системой, на которой запускается отправленный им код – это создание списка запрещённых библиотек. Метод подразумевает, что названия библиотек известны и, как правило, не изменяется с последующими обновлениями модуля. Идея работы способа заключается в проверке исходного кода на наличие подключения запрещённых модулей через построчный проход по всему коду. Недостаток способа заключается в том, что стандартные библиотеки зачастую задействованы в компиляции программ и в работе самого интерпретатора. То есть их отключение/запрет на использование может привести к неполадкам работы программы.

Пример реализации данного способа выглядит следующим образом и результатом будет процесс проверки на наличие запрещённых библиотек (см. рис. 1):

```
import sys

DISALLOWED_MODULES = ['os','random','sys']

def compile_code(code):
    for module in DISALLOWED_MODULES:
        if f"import {module}" in code or f"from {module}" in code:
            raise Exception(f"You're not allowed to use {module}
module")
    compiled_code = compile(code)
    return compiled_code
```

```
user_code = '''
import os
print(os.listdir())
'''

try:
    compiled_code = compile_code(user_code)
    exec(compiled_code)
except Exception as e:
    print(f"Compilation error: {e}")
```

```
Python 3.10.0 (tags/v3.10.0:b494f59, Oct  4 2021, 19:00:18) [MSC v.1929 64 bit (AMD64)] on win
32
Type "help", "copyright", "credits" or "license()" for more information.

= RESTART: C:/Users/[illegible]/AppData/Local/Programs/Python/Python310/security1.py
Compilation error: You're not allowed to use os module
```

Рисунок 1. Пример работы проверки наличия запрещённых библиотеки в компилируемом коде

Но стоит отметить, что вышеприведённый способ не является надёжным, так как пользователь может знать о динамическом подключении модулей через “importlib” во время работы программы. “importlib” это стандартный модуль языка программирования Python, используемый для динамического подключения модулей во время работы программы.

Также стоит заметить, что пользователь не сможет подключить библиотеку, название которой вынесено в отдельную переменную, что облегчает работу по обеспечению безопасности (см. рис. 2). В примере ниже представлен тот же код проверки наличия запрещённых библиотек, но в пользовательском коде производится попытка импорта модуля os через переменную MODULE_NAME:

```
import sys

DISALLOWED_MODULES = ['os','random','sys']

def compile_code(code):
    for module in DISALLOWED_MODULES:
        if f"import {module}" in code or f"from {module}" in code:
            raise Exception(f"You're not allowed to use {module}
module")
    compiled_code = compile(code,'<user_code>','exec')
    return compiled_code

user_code = '''
MODULE_NAME = "os"
import MODULE_NAME
print(os.listdir())
'''
```

```
try:
    compiled_code = compile_code(user_code)
    exec(compiled_code)
except Exception as e:
    print(f"Compilation error: {e}")
```

```
Python 3.10.0 (tags/v3.10.0:b494f59, Oct  4 2021, 19:00:18) [MSC v.1929 64 bit (AMD64)] on win
32
Type "help", "copyright", "credits" or "license()" for more information.

= RESTART: C:/Users/[illegible]/AppData/Local/Programs/Python/Python310/security1.py
Compilation error: No module named 'MODULE_NAME'
```

Рисунок 2. Результат попытки компиляции кода с названием модуля в отдельной переменной

Теперь стоит обратить внимание на модуль importlib. Импорт модулей осуществляется через метод import_module и выглядит в базовой реализации следующим образом:

```
import importlib

def import_library(library_name):
    try:
        library = importlib.import_module(library_name)
        print("Added!")
    except ImportError:
        print("Import Error!")

library_name = input()
import_library(library_name)
```

Пользователь, если знает про данный модуль, может попробовать обойти ограничения и выполнить вредоносный код с использованием запрещённых системой библиотек:

```
import sys

DISALLOWED_MODULES = ['os','random','sys']

def compile_code(code):
    for module in DISALLOWED_MODULES:
        if f"import {module}" in code or f"from {module}" in code:
            raise Exception(f"You're not allowed to use {module}
module")
    compiled_code = compile(code,'<user_code>','exec')
    return compiled_code
user_code = '''
import importlib

MODULE_NAME = 'os'
```

```
module = importlib.import_module(MODULE_NAME)
print(module.listdir())
'''

try:
    compiled_code = compile_code(user_code)
    exec(compiled_code)
except Exception as e:
    print(f"Compilation error: {e}")
```

Данный пример наглядно показывает, что обойти алгоритм проверки наличия запрещённых библиотек возможно (см. рис. 3). Можно запретить использование importlib совсем, но тогда стоит учитывать, что этот модуль в дальнейшем использовать никто не сможет. В контексте обучения основам программирования такой подход имеет место быть, но если необходимо использовать Python для более сложных задач (где может быть необходим динамический импорт библиотек), то тогда этот способ не является подходящим решением.

```
Python 3.10.0 (tags/v3.10.0:b494f59, Oct  4 2021, 19:00:18) [MSC v.1929 64 bit (AMD64)] on win
32
Type "help", "copyright", "credits" or "license()" for more information.

= RESTART: C:/Users/[redacted]/AppData/Local/Programs/Python/Python310/security1.py
['208.py', '291.py', '64.py', '670.py', '712.py', '905.py', 'aa.py', 'asdfaf.py', 'bb.py', 'DLLs', 'Doc', 'fjkgh
jkfsd.py', 'H.py', 'include', 'keane.py', 'K_Matrix.py', 'Lib', 'libs', 'LICENSE.txt', 'module_sum.py', 'N
EWS.txt', 'python.exe', 'python3.dll', 'python310.dll', 'pythonw.exe', 'Scripts', 'security1.py', 'share', 't
cl', 'test1.py', 'test1.pyc', 'test2.py', 'Tools', 'vcruntime140.dll', 'vcruntime140_1.dll']
```

Рисунок 3. Пример обхода способа проверки наличия запрещённых модулей через динамический импорт

Если требуется более надёжная реализация, то необходимо прибегнуть к изоляции процесса запуска программы. Для этого существует большое количество различных модулей и способов реализации. Среди стандартных модулей Python можно выделить subprocess – модуль, который предоставляет возможность запускать отдельные процессы из исходной программы. Он может быть полезен для выполнения кода в изолированной среде или для выполнения внешних команд и программ.

Пример использования данного модуля представлен ниже:

```
import subprocess

input_file = open('input.txt', 'r')
process = subprocess.Popen(['python', 'taskCheck.py'],
stdin=input_file, stdout=subprocess.PIPE, stderr=subprocess.PIPE)
output = process.communicate()
print(process.returncode)
print(output.decode())
input_file.close()
```

В данном примере представлен следующий алгоритм: происходит считывание входных данных из файла “input.txt”, а после запускается процесс “taskCheck.py”, в который загружаются входные данные и после происходит взаимодействие с самим процессом. На выходе выводится код, с которым завершилась работа программы, и результат выполнения алгоритма.

Процесс запускается с помощью функции Popen, которая на вход принимает следующие аргументы:

1. Массив аргументов командной строки, который определяет команду или программу, которую нужно выполнить. Он может быть строкой или списком строк.
2. stdin – данные, поступающие на вход программы.
3. stdout – вывод программы.
4. stderr – вывод ошибок работы программы.

В качестве значения для аргументов stdin, stdout, stderr можно использовать PIPE – это объект в Python, который можно использовать для записи и чтения данных. Удобно, если нет необходимости сохранять информацию в отдельных файлах.

Но subprocess не обеспечивает безотказную безопасность работы системы и если пользователь знает, как работает командная строка и модуль subprocess, то он может повлиять на работу компьютера в целом.

Использование subprocess можно совместить с проверкой запрещённых для использования библиотек, чтобы повысить безопасность запуска и компиляции исходного кода. Можно рассмотреть алгоритм на примере работы задачи нахождения НОДа двух чисел:

```
a,b = map(int,input().split())
num1,num2 = a,b
while a * b > 0:
  if a >= b:
    a = a % b
  else:
    b = b % a
c = a + b
print(f"GCD of {num1} and {num2} is {c}",end="")
```

Таким образом, совместив оба способа, которые были описаны ранее, можно получить следующий алгоритм проверки работы исходного кода:

```
import sys
import subprocess
DISALLOWED_MODULES = ['os','random','sys','importlib']

def compile_code(code):
    for module in DISALLOWED_MODULES:
        if f"import {module}" in code or f"from {module}" in code:
            raise Exception(f"You're not allowed to use {module}
module")
```

```
    compiled_code = compile(code,'<user_code>','exec')
    return compiled_code

user_code = open("taskCheck.py","r")

try:
    compiled_code = compile_code(user_code.read())
    input_data = open("input.txt","r")
    process                                                    =
subprocess.Popen(['python','taskCheck.py'],stdin=input_data,stdo
ut=subprocess.PIPE)
    output = process.communicate()
    print(output[0].decode())
except Exception as e:
    print(f"Compilation error: {e}")
```

```
Python 3.10.0 (tags/v3.10.0:b494f59, Oct  4 2021, 19:00:18) [MSC v.1929 64 bit (AMD64)] on win
32
Type "help", "copyright", "credits" or "license()" for more information.

= RESTART: C:/Users/Сергей/AppData/Local/Programs/Python/Python310/sec2/security1.py
GCD of 12 and 42 is 6
```

Рисунок 4. Пример работы алгоритма проверки исходного кода на наличие запрещённых модулей с последующим созданием процесса

Каким бы надёжным решение использовать такие 2 алгоритма вместе не казалось и всё равно достигнуть наиболее безопасного способа запустить сторонний код не получится. Наилучшим решением является виртуализация процесса, то есть создания обособленной среды, в которой производится запуск кода. Этот способ можно считать лучшим по причине того, что таким образом запущенный код не взаимодействует с главным устройством напрямую (а значит и не может повлиять на его работоспособность). Можно выделить следующие подходы к виртуализации в контексте применения Python: PyPy и Docker.

PyPy это реализация языка программирования Python, написанная на языке Python. Она является альтернативной реализацией интерпретатора Python, отличающейся от стандартной CPython реализации. Во многих задачах PyPy работает быстрее стандартного Python. Также для PyPy были написаны отдельные модули, которые не будут работать в Python. Важным в контексте данной статьи является модуль PyPy Sandbox – модуль, предоставляемый PyPy, который обеспечивает безопасное выполнение пользовательского Python-кода в изолированной среде. Он предназначен для ограничения доступа к опасным операциям и ресурсам во время выполнения кода.

PyPy Sandbox обеспечивает безопасность путем ограничения доступа к опасным функциям и модулям языка Python. Также он позволяет устанавливать ограничения на использование ресурсов, таких как время выполнения, использование памяти и размер стека. Это помогает

предотвратить злоупотребление ресурсами и защищает от ошибок или намеренного замедления системы.

Простая реализация «песочницы» через PyPy Sandbox выглядит следующим образом:

```
from pypy_sandbox import pypy_sandbox

sandbox = pypy_sandbox.Sandbox()
result = sandbox.execute("print('Hello, PyPy Sandbox!')")
print(result)
```

PyPy позволяет также задавать ограничения по времени и памяти при запуске программы:

```
from pypy_sandbox import pypy_sandbox

sandbox = pypy_sandbox.Sandbox()

sandbox.set_time_limit(5)
sandbox.set_memory_limit(256)

code = """
import math

def square_root(x):
    return math.sqrt(x)

result = square_root(121)
print(result)
"""

output = sandbox.execute(code)

print(output)
```

Проблема PyPy Sandbox заключается в том, что разработчики перестали поддерживать данный модуль. Ранее поставлялась отдельная версия для PyPy с предустановленным модулем и настроенным окружением под него. Хоть и можно найти модуль в интернете и установить его – использовать его не рекомендуется (если не предполагается выполнение специфичных для PyPy алгоритмов и скриптов). Данный модуль был продемонстрирован для наглядности примера существования отдельных библиотек, которые позволяют создавать изолированные среды для запуска пользовательского кода.

Можно выделить следующие аналоги PyPy Sandbox:

1. RestrictedPython.
2. PyBox.
3. PySafer.

Помимо виртуализации изолированных сред можно также прибегнуть к использованию Docker.

Docker это открытая платформа, которая позволяет упаковывать, развертывать и запускать приложения в контейнерах. Контейнеры это легкие, изолированные среды, которые включают в себя все необходимые зависимости для запуска приложения, включая операционную систему, библиотеки и другие компоненты.

В Python существует отдельный модуль для работы с контейнерами Docker. Пример работы представлен ниже:

```
import docker

client = docker.from_env()

code_file = open("taskCheck.py","r")
code = code_file.read()

container = client.containers.run(
    'python:3',
    command=['python', '-c', code],
    detach=True
)

exit_code = container.wait()['StatusCode']

logs = container.logs().decode('utf-8')

print(f"Exit Code: {exit_code}")
print(f"Output:\n{logs}")

container.remove()
```

В этом примере сначала создаётся клиент через “docker.from_env()”. Затем в переменную code сохраняется пользовательский код и после создаётся контейнер (в скобках указываются его параметры, в том числе образ). После этого скрипт ждёт завершения работы контейнера и выводит содержимое вывода на экран. Контейнер после этого удаляется.

Docker является отличным решением при создании изолированных сред, но требует углубленного понимания в вопросах виртуализации и контейнеризации.

Если должным образом рассмотреть подход к обеспечению безопасности во время запуска и компиляции пользовательского кода, то на стороне сервера остаётся только запустить процесс через командную строку. Сделать это можно через PHP, заранее подготовив шаблон для проверки кода.

В данной статье были рассмотрены способы организации безопасности при работе с пользовательским кодом. Приведены примеры реализации базовых алгоритмов, а также алгоритмов при работе с изолированными средами и Docker.

Библиографический список

1. Буянова И.В., Замулин И.С. Выбор технологии защиты ресурса для онлайн-обучения программированию от недоверенного кода // В сборнике: Инженерные технологии: традиции, инновации, векторы развития. Материалы VIII Всероссийской научно-практической конференции с международным участие. Науч. и отв. редактор Д.Ю. Карандеев. Абакан. 2022. С. 122-124.
2. Дорофеев Д.А. Изоляция среды выполнения программного кода в системах обучения и тестирования // Аллея науки. 2017. Т.1. № 15. С. 774-776.
3. Валиуллина Д.И., Зиганшин И.А. Применение фреймворка PyTest для тестирования программного кода на языке Python // В сборнике: Учёный XXI века. сборник статей Международного научно-исследовательского конкурса. Пенза. 2022. С. 35-37.