| | |
|---|---|
| Authors: | Arie Kurniawan |
| State | Tested |
| Version | 1 |
| Create date | June 8th, 2023 |

# Simulate race conditions in gRPC

Here's a step-by-step guide that includes code examples to simulate race conditions in gRPC

**1. Create a Protobuf Code for Race Conditions**

Start by defining the protobuf service and message definitions. Let's create a simple example with a Counter service that increments and retrieves the current value of a counter.
Create a file called counter.proto with the following content:

code:

```python
syntax = "proto3";

service Counter {
    rpc Increment (IncrementRequest) returns (IncrementResponse);
    rpc GetValue (GetValueRequest) returns (GetValueResponse);
}

message IncrementRequest {}
message IncrementResponse {}

message GetValueRequest {}
message GetValueResponse {
    int32 value = 1;
}
```

in order to use the protobuf file (counter.proto) in your Python code, you need to compile it using the Protocol Buffers compiler (protoc). The compilation step generates the necessary code that allows your Python application to interact with gRPC services defined in the protobuf file.

To compile the counter.proto file, follow these steps:

1. Install Protocol Buffers:
    - If you haven't already installed Protocol Buffers, visit https://developers.google.com/protocol-buffers/docs/downloads and download/install it based on your operating system instructions.
2. Compile Proto File:
    - Open a terminal/command prompt.
    - Navigate to the directory where counter.proto is located.
    - Run the following command to generate Python classes from .proto files:
    - shell
    - Copy code

```
Unset
```

    - protoc --python_out=. --grpc_python_out=. counter.proto

3. Verify Generated Files:
   After executing the above command, check if two new files are generated: counter_pb2.py and counter_pb2_grpc.py.

Now you can import these generated modules (counter_pb2, counter_pb2_grpc) into your server and client scripts for interacting with gRPC services defined by counter.proto.

Notes: Remember to repeat this process whenever you make changes to your .proto file so that any modifications are reflected in the compiled code used by your application.

**2. Create Vulnerable Server and Client for Testing Race Conditions using Python**

Next, we'll create both the vulnerable server and client implementations in Python.

**Vulnerable Server (vulnerable_server.py)**

code:

```python
import time
from concurrent import futures
import grpc

import counter_pb2
import counter_pb2_grpc

class CounterService(counter_pb2_grpc.CounterServicer):
    def __init__(self):
        self.value = 0

    def Increment(self, request, context):
        # Simulate race condition vulnerability by introducing
delay between read-modify-write operations.
        time.sleep(0.5)

        self.value += 1

        return counter_pb2.IncrementResponse()

    def GetValue(self, request, context):
        return counter_pb2.GetValueResponse(value=self.value)

def serve():
    server =
grpc.server(futures.ThreadPoolExecutor(max_workers=10))

    # Register our servicer class instance to handle gRPC
requests.
```

```
counter_pb2_grpc.add_CounterServicer_to_server(CounterService(),
server)

    server.add_insecure_port('[::]:50051')

    print("Starting vulnerable gRPC server...")

    server.start()
    server.wait_for_termination()

if __name__ == '__main__':
    serve()
```

**Client (client.py)**

code

```
Python
import grpc

import counter_pb2
import counter_pb2_grpc

def run():
    channel = grpc.insecure_channel('localhost:50051')

    stub = counter_pb2_grpc.CounterStub(channel)

    # Simulate multiple concurrent client requests to trigger
race condition vulnerability.
    for _ in range(10):
        stub.Increment(counter_pb2.IncrementRequest())
```

```
        response =
stub.GetValue(counter_pb2.GetValueRequest())
        print(f"Current value: {response.value}")

if __name__ == '__main__':
    run()
```

**3. Create a Server with Fixed Race Conditions**

Now, let's create an updated version of the server that fixes the race conditions.

**Fixed Server (fixed_server.py)**

code

Python

```
from concurrent import futures
import grpc

import threading

import counter_pb2
import counter_pb2_grpc


class CounterService(counter_pb2_grpc.CounterServicer):
    def __init__(self):
        self.value = 0

        # Use a lock to synchronize access to shared data.
        self.lock = threading.Lock()

    def Increment(self, request, context):
        with self.lock:
```

```
            self.value += 1

        return counter_pb2.IncrementResponse()

    def GetValue(self, request, context):
        return
counter_pb2.GetValueResponse(value=self.value)


def serve():
    server =
grpc.server(futures.ThreadPoolExecutor(max_workers=10))

     # Register our servicer class instance to handle gRPC
requests.

counter_pb2_grpc.add_CounterServicer_to_server(CounterServi
ce(), server)

     server.add_insecure_port('[::]:50051')

     print("Starting fixed gRPC server...")

     server.start()
     server.wait_for_termination()


if __name__ == '__main__':
    serve()
```

**4. How to run simulation**

1. Compile Protobuf Files:
    - Make sure you have already compiled the counter.proto file using Buf or protoc as mentioned earlier in this docs.
    - Ensure that you have counter_pb2.py and counter_pb2_grpc.py generated from the compilation process.
2. Start the Vulnerable Server:
    - Open a terminal or command prompt.
    - Navigate to the directory containing vulnerable_server.py.
    - Run the following command to start the vulnerable server on you CLI:

```
Python

python3 vulnerable_server.py
```

3. Run Multiple Instances of Vulnerable Client:
    - Open multiple new terminals or command prompts (one for each client instance).
    - In each terminal, navigate to the directory containing vulnerable_client.py.
    - Run the following command to start the client server on you CLI

```
Python

python3 client.py
```

4. Observe Race Condition Behavior: As multiple instances of clients are concurrently making requests, observe how they interfere with each other's updates due to race conditions caused by lack of synchronization.
5. Stop Vulnerable Server and Clients: Press Ctrl+C in both server and client terminals/command prompts to stop their execution.
6. Start Fixed Server: Now let's run a fixed version of our server that addresses race condition vulnerabilities the same way as you run vulnerable server.
7. Start Client(s): Repeat Step 3 by running multiple instances of clients against this fixed server.
8. Observe Synchronized Behavior: Notice how with proper synchronization mechanisms implemented in the fixed server, there are no unexpected behaviors caused by race conditions, ensuring consistent results across concurrent client requests.

**5. Explain the Difference Between Results of Having Race Conditions and the Fixed Version**
When running the vulnerable server and client (vulnerable_server.py and vulnerable_client.py) together, you may observe inconsistent results due to race conditions. Multiple concurrent client requests can interfere with each other, leading to incorrect counter values being returned.

On the other hand, when using the fixed server (fixed_server.py) along with the same client code, you should observe consistent results without any unexpected behavior caused by race conditions. The use of a lock ensures that only one thread can access or modify shared data at a time, preventing race condition vulnerabilities.

By fixing race conditions through proper synchronization mechanisms like locks or mutexes, we ensure that critical sections of code are executed atomically without conflicts between multiple threads or clients accessing shared resources simultaneously.

Remember to run each Python script in separate terminal windows for testing purposes.
I hope this guide helps you understand how to create protobuf code for gRPC services, implement vulnerable/fixed servers and clients for testing race conditions in Python, and compare their behaviors.

**Example observation result:**

- Vulnerable servers will return responses continuously one by one, while fixed servers will lock data and respond to requests all at once in one response.
- In the previous code provided, the vulnerable part is in the implementation of the Increment method within the CounterService class of the vulnerable_server.py file. Here's an excerpt from that code:

```python
class CounterService(counter_pb2_grpc.CounterServicer):
    def __init__(self):
        self.value = 0

    def Increment(self, request, context):
        # Simulate race condition vulnerability by introducing
delay between read-modify-write operations.
        time.sleep(0.5)
        self.value += 1
        return counter_pb2.IncrementResponse()
```

The vulnerability arises due to a lack of synchronization when multiple concurrent client requests try to increment and modify the shared self.value. The call to time.sleep(0.5) introduces an intentional delay to increase the likelihood of triggering a race condition.

When multiple clients concurrently invoke this method (Increment) on the server, they may interfere with each other's updates, leading to incorrect results or unexpected behavior due to inconsistent state management.

To fix this vulnerability and ensure thread safety, proper synchronization mechanisms like locks or mutexes should be used when accessing and modifying shared data. This issue is addressed in the fixed version of the server (fixed_server.py) by using a lock (implemented as a threading.Lock) inside critical sections where shared resources are accessed or modified.