USENIX Security '24 Artifact Appendix: Stateful Least Privilege Authorization for the Cloud

Leo Cao* UC San Diego
Luoxi Meng* UC San Diego
Deian Stefan UC San Diego
Earlence Fernandes UC San Diego

A Artifact Appendix

A.1 Abstract

Widely-used authorization protocols like OAuth create overprivileged credentials because they do not provide developers of client apps and servers the tools to request and enforce minimal access. In the status quo, these overprivileged credentials are vulnerable to abuse when stolen or leaked. We introduce an authorization framework StatefulAuth that enables creating and using bearer tokens that are least privileged.

This artifact evaluation aims to reproduce the key results presented in our paper. We will guide reviewers through the setup of stateful authorization using the provided example code. In this evaluation, the reviewers will observe that this artifact introduces only a modest performance overhead compared to standard OAuth 2.0 and the previous stateless authorization framework.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

We provide access to live AWS instances for the evaluation. The evaluators do not need to build or execute the code on their own machines. This helps to enhance the security and privacy of evaluators' machines.

A.2.2 How to access

The artifact is available on our GitHub repository (`https://github.com/earlence-security/stateful-auth/tree/a48dabad03e18cb70e225e12b2a9bc228dd87203`). A Markdown version of the instructions in A.3 and A.4 is available on (`https://github.com/earlence-security/stateful-auth/blob/a48dabad03e18cb70e225e12b2a9bc228dd87203/docs/artifact_evaluation.md`).

A.2.3 Hardware dependencies

We implement a client-server framework to measure end-to-end latency. We host our server on a c5d.2xlarge instance with 8 vCPUs and 16 GiB memory on Amazon Web Services (AWS) in the US East (Ohio) region, and our client app on another instance with the same configuration in the US West (North California) region. We provide a private SSH key for the evaluators to access the client and server instances.

*Equal Contribution.

A.2.4 Software dependencies

We build our stateful authorization library based on Authlib [3], a Python library of generic OAuth implementations. We require Python>=3.10. `requirements.txt` in our repo lists all the Python packages required to evaluate the artifact. All these packages are already installed in the AWS instances.

A.2.5 Benchmarks

To simulate the overhead of a real system, we mock up an implementation of Google Calendar APIs. Specifically, we mirror the Events API [2]. We use an in-memory SQLite database [4] for all the DB operations.

The client generates a sequence of requests. Each request is randomly selected from the set of Events API endpoints. Please see Section 4.3 **Experiment Setup** for more details. We provide scripts that automatically send requests from the client to the server, simplifying the evaluation process.

A.3 Set-up

A.3.1 Installation

The reviewers will connect to our AWS instances with the private SSH key.

1. Open a terminal and SSH to the **server**:

```
$ ssh -i <PATH_TO_PRIVATE_KEY> ubuntu@ec2-3-18-8-25.
    us-east-2.compute.amazonaws.com
```

2. Open another terminal and SSH to the **client**:

```
$ ssh -i <PATH_TO_PRIVATE_KEY> ubuntu@ec2-52-8-5-83.
    us-west-1.compute.amazonaws.com
```

3. Download the policy and state updater WebAssembly (WASM) on your local machine. These are the client-supplied programs that attenuate the token's authority and define the state, respectively (Section 2.2).

We have the source code available and all the required Python packages installed on our AWS instances. Please note that we mark the following steps with **[Server-side]** and **[Client-side]** to indicate which terminal each command should be executed on.

A.3.2 Basic Test

In this test, we will first register the client app on the server (Section 3.1.1). Then, the client app will request a token from the server and proceed to make a simple request.

1. **[Server-side]** Start the server with:

```
$ cd stateful-auth/server
$ AUTHLIB_INSECURE_TRANSPORT=1 gunicorn -b
     0.0.0.0:5000 app:app --limit-request-line=0 --
     limit-request-fields=0 --limit-request-
     field_size=0
```

2. **[Server-side]** Open a browser on your local machine and go to `http://3.18.8.25` and input any username you like to sign up (for example, `artifact-eval`).
3. **[Server-side]** Create a client by clicking the "Create Client" button, and filling out the form as follows:

```
Client Name: client_00
Client URI: http://52.8.5.83
Allowed Scope: profile
Upload Policy WASM: Upload 77a6a23
     a3c4d424dba1e7efae21cef16d01f156c6d0194e406c98
     cf46c22bf37.wasm (the policy WASM you have
     downloaded)
Upload State Updater WASM: Upload update_program.
     wasm (the state updater WASM you have
     downloaded)
Redirect URIs: http://52.8.5.83/auth
Allowed Grant Types: authorization_code
Allowed Response Types: code
Token Endpoint Auth Method: client_secret_basic
```

   Note: we leave `Policy Program Hashes` and `Policy Program Endpoint` as blank - this is an alternative way to upload policy program WASM.
4. **[Server-side]** Click "Submit". You will be directed to the page listing all the registered clients.
5. **[Client-side]** Start the client with:

```
$ cd stateful-auth/client
$ CLIENT_ID=<CLIENT_ID> CLIENT_SECRET=<CLIENT_SECRET
     > ACCESS_TOKEN_URL='http://3.18.8.25/oauth/
     token' AUTHORIZE_URL='http://3.18.8.25/oauth/
     authorize' REDIRECT_URI='http://52.8.5.83/auth'
      gunicorn -b 0.0.0.0:8080 app:app
```

   Replace `<CLIENT_ID>` and `<CLIENT_SECRET>` with `client_id` and `client_id` you find in the server-side page after the client's registration from the previous step.
6. **[Client-side]** Go to `http://52.8.5.83`, and click "Request a Token from Auth Server". Select a policy the client has registered. For this test, let's select the one starting with `77a6a`. The string is a hash of the policy WASM we downloaded. Click "Submit".
7. **[Client-side]** You will see a page saying that the client application is requesting a scope with a policy hash. Click "Consent?" box and click "Submit".
8. **[Client-side]** You will see your access token, which will be used when the client makes a request to the server. Click "send requests with this token".
9. **[Client-side]** You will be directed to `http://52.8.5.83/make_request`. The two selection boxes are default to `GET` and `api/me`. Click "Make API call". The expected results should be:

```
{
 "id": 1,
 "message": "Hello World!",
 "username": <YOUR_USERNAME>
}
E2E Latency: ??? milliseconds
```

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): StatefulAuth introduces modest end-to-end latency overhead, compared with vanilla OAuth. This is proven by E1 described in Section 4.3 **End-to-End Latency**, whose results are illustrated in Figure 6(a).

(C2): The server-side latency of StatefulAuth increases proportionally with the number of objects. State validation is the predominant contributor to server-side latency. This is proven by the server-side log of E1 described in Section 4.3 **Server-Side Latency Breakdown**, whose results are presented in Figure 6(b).

(C3): The end-to-end latency overhead of StatefulAuth is modest when executing stateless policies, compared with Macaroons [1]. This is proven by E2 described in Section 4.3 **Comparison of Stateless Policies with Macaroons**, whose results are presented in Figure 7.

A.4.2 Experiments

(E1): [Latency Overhead on Stateful Policies Compared with vanilla OAuth] [15 human-minutes + 5 compute-minutes + 1GB disk]: (1) Measure and compare the worst-case end-to-end latency of vanilla OAuth and StatefulAuth by sending requests with different numbers of objects; (2) Breakdown the server-side latency of StatefulAuth.

Preparation: Please complete Step 1 – 8 in Section A.3.2. Then, on the client side, press Ctrl+C to shut down the client web app. Keep the server running.

Execution:

1. **[Client-side]** Send requests with a provided script:

```
$ cd ../scripts
$ python3 ./eval_latency.py --token <
     ACCESS_TOKEN> --base-url http://3.18.8.25
     --n-iters 30 --mode stateful
```

   Replace `<ACCESS_TOKEN>` with the access token you find on `http://52.8.5.83`. The script will print the end-to-end latency table of StatefulAuth (HMAC).
2. **[Server-side]** Press Ctrl+C to shut down the server. You can find the generated log file `server/logs/logs_<TIME>.txt`
3. **[Server-side]** Restart the server *without* StatefulAuth enabled and turn off the server-side logging:

```
$ cd stateful-auth/server
$ AUTHLIB_INSECURE_TRANSPORT=1
     ENABLE_STATEFUL_AUTH=False ENABLE_LOGGING=
     False gunicorn -b 0.0.0.0:5000 app:app --
     limit-request-line=0 --limit-request-fields
     =0 --limit-request-field_size=0
```

4. **[Client- and server-side]** Repeat Step 2 – 8 in Section A.3.2, except (i) in Step 3, leave policy and state updater WASM as blank, (ii) in Step 6, click "Pop the token" and select the policy `null`.
5. **[Client-side]** Run the command in E1 Step 1, but replace `--mode stateful` with `--mode baseline`, and `<ACCESS_TOKEN>` with the token in Step 4. The script will print the baseline end-to-end latency table.

Results: The end-to-end latency tables generated in Step 1 and 5 correspond to StatefulAuth (HMAC) and OAuth end-to-end latency (height of the entire bar) in Figure 6(a), respectively. To get the server-side latency breakdown in Figure 6(b), on server side:

```
$ python3 ../scripts/server_log_summary.py --file
    logs/logs_<TIME>.txt
```

(E2): [Latency Comparison of Stateless Policies with Macaroons] [15 human-minutes + 5 compute-minutes + 1GB disk]: Measure and compare the end-to-end latency of StatefulAuth for stateless policies against the same policy implemented with Macaroons

Preparation for StatefulAuth Stateless Latency: Please complete Step 1 – 8 in Section A.3.2 unless for the special cases mentioned below. For Step 3, upload Policy WASM program with hash starting with `e265cb` instead. Policy `e265cb` performs multiple stateless checks on the request. Do not upload any History Updater, since the policy is stateless. Then, on the client side, keep note of the new token requested and shut down the client web app. Keep the server running.

Execution for StatefulAuth Stateless Latency:

1. **[Client-side]** Measure latency:

```
$ cd ../scripts
$ python3 ./eval_macaroon.py --token <
    ACCESS_TOKEN> --base-url http://3.18.8.25
    --n-iters 30 --accept=True
$ python3 ./eval_macaroon.py --token <
    ACCESS_TOKEN> --base-url http://3.18.8.25
    --n-iters 30 --accept=False
```

Replace `<ACCESS_TOKEN>` with the access token you find on http://52.8.5.83. The first run of script will print out the end-to-end latency of policy accept for StatefulAuth, while the second run should print out the latency for policy deny.

Preparation for Macaroon Latency: Please complete Step 1 – 8 in Section A.3.2 unless for the special cases mentioned below.

1. For Step 1, add `MACAROON=True` to the environment

```
$ cd stateful-auth/server
$ MACAROON=True ... gunicorn ...
```

2. For Step 3, do not upload any Policy WASM and State Updater.
3. For Step 5, add `MACAROON=True` to the environment

```
$ MACAROON=True CLIENT_ID=<CLIENT_ID> ...
    gunicorn
```

4. For Step 6, choose the policy with hash `macaroon`. This is a dummy policy to make macaroons work inside our infrastructure.

Execution for Macaroon Latency:

1. **[Client-side]** Send requests with a provided script:

```
$ cd ../scripts
$ python3 ./eval_macaroon.py --token <
    ACCESS_TOKEN> --base-url http://3.18.8.25
    --n-iters 30 --accept=True
$ python3 ./eval_macaroon.py --token <
    ACCESS_TOKEN> --base-url http://3.18.8.25
    --n-iters 30 --accept=False
```

Replace `<ACCESS_TOKEN>` with the access token you find on http://52.8.5.83. The first run of the script will print out the end-to-end latency of policy accept for Macaroon, while the second run should print out the latency for policy deny.

Results: The 4 measured end-to-end latency numbers should align with the numbers documented in Figure 7 of the paper.

A.5 Notes on Reusability

This artifact is designed as a "drop-in" solution - an authorization library for a variety of servers and clients. As mentioned Section 2.3, our framework provides client developers flexibility and extensibility, allowing them to customize the attenuation policy and state updater to best suit their needs. Besides, although we integrate with Authlib [3] (Section 4), our design can be integrated with any authorization library to enable client-defined permissions and stateful authorization.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.

References

[1] Arnar Birgisson, Joe Gibbs Politz, Úlfar Erlingsson, Ankur Taly, Michael Vrable, and Mark Lentczner. Macaroons: Cookies with Contextual Caveats for Decentralized Authorization in the Cloud. In *Network and Distributed System Security Symposium*, 2014.

[2] Google. Events | Google Calendar. https://developers.google.com/calendar/api/v3/reference/events, 2023.

[3] Hsiaoming Ltd. Authlib. https://authlib.org, 2017.

[4] SQLite. In-Memory Databases. https://www.sqlite.org/inmemorydb.