Authentifizierung auf NGINX Proxy Level

Im nachfolgenden Artikel werden drei unterschiedliche Verfahren zur Authentifizierung auf NGINX Proxy Level vorgestellt und anhand einer Beispielimplementierung anschließend gezeigt, wann dieses Verfahren eingesetzt werden können beziehungsweise wann sie sinnvoll sind.

Die ursprüngliche Fragestellung, die zugrunde liegt war, ob es eine Möglichkeit gibt Webseiten beziehungsweise Web API's mittels eines Reverse Proxies in Verbindung mit einem Identity Provider zu schützen. Hierzu soll keinerlei Veränderung am Quellcode der zu schützenden Ressource vorgenommen werden. Konfiguration und Schutz werden nur auf Seiten des Identity Provider und des Reverse Proxy vorgenommen.

Zur Simulation dieses Szenarios wurde eine Testumgebung eingerichtet, die neben einem NGINX Reverse Proxy, einen NGINX Webserver mit statischem Content, ein IdentityServer4 als Identity Provider (nachfolgend: IdP) und einer Web API besteht. Innerhalb des Webservers sind zwei HTML-Seiten hinterlegt. Eine der beiden Seiten ist öffentlich zugänglich, die andere ist durch das implementierte Authentifizierungsverfahren vor unbefugtem Zugriff geschützt.

Der Quellcode für das Testszenario ist öffentlich zugänglich unter:

https://github.com/JulianHBuecher/Authentication-Reverse-Proxy

Inhaltsverzeichnis

Vorstellung des Verfahrens

Warum ist diese Art von Authentifizierung praktisch? Diese Frage wird im Anschluss unter Betrachtung einzelner Gesichtspunkte in Bezug auf verteilte Systeme genauer beleuchtet.

Es wird die Annahme getroffen, ein IT Unternehmen besitzt eine Vielzahl unterschiedlicher Webanwendungen, die entweder un- oder teilweise durch ältere Authentifizierungsverfahren geschützt werden. In Anbetracht der ständigen Weiterentwicklung der gängigen Sicherheitsprotokolle wie OAuth2 bzw. dessen baldigen Nachfolger OAuth3, wird es für die älteren Anwendungen deutlich schwieriger auf den aktuellen Stand der Technik aufzuschließen.

Meist ist die Quellcode-Basis veraltet oder benutzt Frameworks, die mit neueren Identity Providern nicht mehr kompatibel sind. Im „worst case“ handelt es sich sogar um einzelne statische Webseiten, die mit „Plain HTML“ konstruiert wurden und somit keinerlei Logik für Authentifizierung beinhalten.

Ein weiterer Aspekt ist die übergreifende Authentifizierung, die es dem User erlaubt sich nur einmal zu Beginn bei dem IdP anzumelden und im Anschluss alle vorhandenen Services zu nutzen, für die er die Berechtigung besitzt.

Wie kann man trotz dieser heterogenen Landschaft an Anwendungen einen Sicherheitsstandard etablieren, der sich an den aktuellsten Trends orientiert und trotzdem die bestmögliche User Experience bietet?

Hier kommt die Authentifizierung auf Basis des Reverse Proxy ins Spiel.

Die Wahl fiel auf den NGINX als Reverse Proxy, der aufgrund seiner hohen Performanz im Gegensatz zu Apache 2 eine weitere Verbreitung in der Industrie findet (z.B. als Ingress in Kubernetes Clustern, etc.). Nach Vornehmen der Einstellungen wird dieser Reverse Proxy zu einem „Authentication Proxy“ und wird je nach Architekturmodell entweder als Sidecar für jeden Container oder als Eintrittspunkt vor die Anwendungslandschaft geschalten [1].

Dieser „Authentication Proxy“ kümmert sich anschließend um den Redirect, Refresh und die Validierung der Anfragen bzw. Tokens und gibt anschließend nur autorisierte Anfragen an die zu schützenden Ressourcen weiter. Die Informationen bezüglich der Authentifizierung können, wenn benötigt, als Header-Informationen vom Proxy aus weitergegeben werden, ohne dass die Anwendung den mitgelieferten Token deserialisieren muss.

Frameworks für die Umsetzung

Bezüglich der Umsetzung haben sich nach umfangreicher Recherche drei Frameworks als geeignet erwiesen. Dabei handelt es sich um „Vouch Proxy“ [2], der als zusätzlicher Service in der Anwendungslandschaft zur Verfügung steht und die Authentifizierung der Anfragen durchführt, „OAuth2-Proxy“ [3], der dem gleichen Prinzip folgt wie „Vouch Proxy“, und andererseits „lua-resty-openidc“ [3], das wiederum in den bestehenden NGINX Reverse Proxy mittels Lua Code eingebettet wird. Neben der verwendeten Sprache zur Implementierung unterscheiden sie sich zusätzlich in ihrer grundlegenden Verwendung. Diese wird in den nachfolgenden Abschnitten nochmals genauer erläutert:

Vouch Proxy

Bei „Vouch Proxy“ handelt es sich um einen zusätzlichen Service neben dem Proxy, der in Golang, der hauseigenen Sprache von Google, implementiert ist. Er wird im Zuge des Deployments in z.B. einem Kubernetes Cluster als zusätzliches Bindeglied zwischen NGINX und dem IdP eingebaut. Hierbei besteht die Möglichkeit, dass Vouch wie alle anderen Services auch hinter den NGINX geschalten wird oder als extra Ressource außerhalb des Netzwerkes zur Verfügung steht.

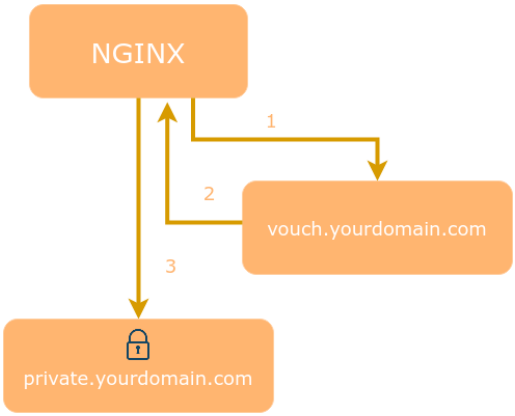

Abbildung 1 Vouch Proxy als Bestandteil der Anwendungslandschaft [2]

Hierbei gilt es jedoch zu beachten, dass zur richtigen Funktionsweise von „Vouch Proxy“ dieser innerhalb einer Subdomain platziert werden muss. Dies ist notwendig, da die Session Cookies, die zwischen Vouch und den zu schützenden Applikationen ausgetauscht werden, domainspezifisch sind. Bedeutet, wenn die zu schützende Domain „*.meinedomain.de*“ lautet, dann sollte Vouch unter „*vouch.meinedomain.de*“ zu erreichen sein. Die Applikationen wären somit unter „*app1.meinedomain.de*“ und „*app2.meinedomain.de*“ erreichbar.

Zur Konfiguration von NGINX und Vouch werden in den config-Dateien von NGINX folgende Werte gesetzt. Beginnend mit dem Validierungsendpunkt, der bei jedem Eingang einer nicht authentifizierten Anfrage aufgerufen wird.

```
location /validate {
        # Endpoint could only be reached from inside the network
        internal;
        # forward the /validate request to Vouch Proxy
        proxy_pass http://vouch/validate;
        # passing the original host header
        proxy_set_header Host $http_host;

        # for CORS preflight requests, just return 200 since a preflight request does not contain a cookie
        # https://stackoverflow.com/questions/41760128/cookies-not-sent-on-options-requests
        if ($request_method = 'OPTIONS') {
            return 200;
        }
        # Vouch Proxy only acts on the request headers
        proxy_pass_request_body off;
        proxy_set_header Content-Length "";

        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;

        # values used by the @error401 call
        auth_request_set $auth_resp_jwt $upstream_http_x_vouch_jwt;
        auth_request_set $auth_resp_err $upstream_http_x_vouch_err;
auth_request_set $auth_resp_failcount $upstream_http_x_vouch_failcount;
    }
```

Abbildung 2 Konfiguration des Validierungsendpunktes

Wenn die Anfrage am Validierungsendpunkt eingeht und dort ein Status-Code 401 erzeugt wird, leitet NGINX den User direkt an Vouch Proxy weiter und dieser wiederum an den IdP. Dort wird anschließend die Authentifizierung des Users vorgenommen. Hierzu wird ein zusätzlicher Endpunkt eingefügt, der nur auf dann reagiert, wenn ein Status Code 401 zurück an den NGINX gesendet wird.

```
{
...
    # if validate returns '401 not authorized' then forward the request to the
    # error401block
    error_page 401 = @error401;

    location @error401 {
        # redirect to Vouch Proxy for Login
        return 302 https://vouch.proxy.localhost/login?url=$scheme://$http_host$request_uri&vouch-failcount=$auth_resp_failcount&X-Vouch-Token=$auth_resp_jwt&error=$auth_resp_err;
    }
...
}
```

Abbildung 3 Konfiguration für die Weiterleitung zum User Login

Im Rahmen dieses Experimentes wurde „Vouch Proxy“ als Docker Container innerhalb eines Docker Bridge Netzwerkes betrieben. Für die korrekte Funktion sind zusätzlich in der *docker-compose.yml* Datei noch einige Umgebungsvariablen zu setzen. Diese beinhalten neben den Informationen zum registrierten Client im IdP noch die notwendigen Endpunkte des IdP und die erhaltenen Berechtigungen (Scopes), die Vouch von Seiten des IdP ausgestellt bekommt.

```
vouch:
    container_name: vouch
    image: voucher/vouch-proxy:latest
    ports:
      - "9090:9090"
    environment:
      - VOUCH_DOMAINS=proxy.localhost
      - OAUTH_PROVIDER=oidc
      - OAUTH_CLIENT_ID=vouch-proxy
      - OAUTH_CLIENT_SECRET=b1612bd0-9bbf-4157-99e3-0e1a65bdad92
      - OAUTH_AUTH_URL=https://reverse.proxy.localhost/identityserver/connect/authorize
      - OAUTH_TOKEN_URL=https://reverse.proxy.localhost/identityserver/connect/token
      - OAUTH_USER_INFO_URL=https://reverse.proxy.localhost/identityserver/connect
/userinfo
      - OAUTH_END_SESSION_ENDPOINT=https://reverse.proxy.localhost/identityserver
/connect/endsession
      - OAUTH_SCOPES=openid,profile,scope1
      - OAUTH_CALLBACK_URLS=https://vouch.proxy.localhost/auth
    networks:
      testnetwork:
        aliases:
          - "vouch.proxy.localhost"
```

Abbildung 4 Umgebungsvariablen für Vouch Proxy

lua-resty-openidc

Im Gegensatz zu „Vouch Proxy“ stellt „lua-resty-openidc“ keinen eigenen Service, sondern eine Erweiterung für den NGINX dar. Sie ist Teil eines offenen Paket-Management-Systems, das via *opm*, dem offiziellen Paket-Manager von OpenResty [4] , oder *LuaRocks*, ein Paket-Manager für das Herunterladen von Lua Modulen [5], abgefragt werden kann.

Somit sind auch die Konfiguration und die Verwendung dieses Moduls eine andere. „lua-resty-openidc“ verwendet das sogenannte „auth_request“ Modul [6], das als Standard-Modul mit NGINX ausgeliefert wird und die grundlegenden Funktionen für die Authentifizierung liefert. Jedoch besitzt das „auth_request“ Modul keinerlei Funktion bezüglich des Erkennens und Cachen von authentifizierten Usern. Hier setzt die oben genannte Erweiterung an.

Mittles Lua Code, kann innerhalb der NGINX-Konfiguration gezielt ein einzelner Endpunkt oder ein ganzer Server mittels Authentifizierung abgesichert werden. Hierzu werden sogenannte Lua-Blöcke definiert, die die gewünschten Variablen setzen.

```
1 access_by_lua_block {
2             local opts = {
3                 -- callback URI have to be covered within the protected ressource
4                 redirect_uri_path = "/secure/callback",
5                 -- Discovery endpoint of the IdP
6                 discovery = "https://reverse.proxy.localhost/identityserver/.well-
known/openid-configuration",
7                 -- if IdP signs token with not known algorithm (deny that)
8                 accept_none_alg = false,
9                 accept_unsupported_alg = false,
10                -- client credentials
11                client_id = "openresty-proxy",
12                client_secret = "438c3e42-5f87-4fad-83a5-a30130943521",
13                scope = "openid profile scope1",
14                -- Refresh the users id_token after 900 seconds without requiring re-
authentication
15                refresh_session_interval = 900,
16                iat_slack = 600,
17                -- Renew the access token automatically
18                renew_access_token_on_expiry = true,
19                -- ... and revoke the token on logout
20                revoke_tokens_on_logout = true,
21                -- do not verify the self-signed certificate
22                ssl_verify = "no",
23                -- setting https as redirect scheme
24                redirect_uri_scheme = "https",
25                -- setting the logout path and its redirect counterpart
26                logout_path = "/logout",
27                redirect_after_logout_uri = "/",
28                redirect_after_logout_with_id_token_hint = false,
29                session_contents = {id_token=true},
30                -- Connection keepalive with the OP can be enabled ("yes")
31                -- or disabled ("no").
32                keepalive = "no",
33                use_pkce = true,
34            }
```

Abbildung 5 Umgebungsvariablen für die Konfiguration mit einem IdentityServer4

Anschließend wird eine Methode definiert, die bei jeder Anfrage eines Users überprüft, ob dieser spezifische User schon authentifiziert ist. Ist dies nicht der Fall wird eine Weiterleitung an den IdP ausgeführt.

```
{
...
1    -- call authenticate for OpenID Connect user authentication
2    local res, err = require("resty.openidc").authenticate(opts)
...
}
```

Abbildung 6 Leitet nicht authentifizierte User an den IdP weiter

OAuth2-Proxy

Als dritte Alternative kann das Framework OAuth2-Proxy für eine Authentifizierung am Reverse Proxy verwendet werden. Diese Anwendung ist wie „Vouch Proxy“ auch mit der Sprache Go implementiert. Die Verwendung von OAuth2-Proxy lässt zwei mögliche Architekturen in der bestehenden Anwendungslandschaft zu:

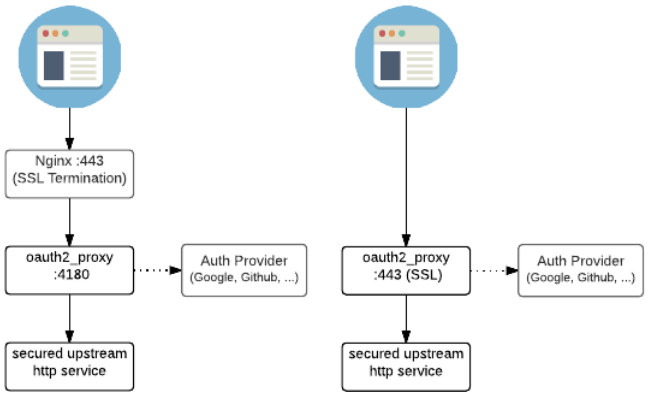

Abbildung 7 Architekturbeispiele für den Einbau von OAuth2-Proxy als separater Service und als eigenständiger Reverse-Proxy [7]

Wie in der Abbildung beschrieben, ist OAuth2-Proxy entweder als selbstständiger Reverse Proxy vor die zu schützenden Anwendungen geschaltet oder als sogenanntes Sidecar neben dem eigentlichen Reverse Proxy. Bei der Benutzung als Sidecar werden alle Anfragen, die über den NGINX Reverse-Proxy ins Backend gelangen zuerst über den OAuth2-Proxy geleitet. Sind die Anfragen entsprechend berechtigt auf die geschützten Ressourcen zuzugreifen, wird der Zugriff gewährt. Ansonsten wird eine Weiterleitung an die Login-Seite des IdP durchgeführt (in unserem Fall IdentityServer4).

Für das funktionierende Zusammenspiel zwischen diesem Framework und IdentityServer4 müssen beiderseits einige Konfigurationen vorgenommen werden.

Beginnend auf der Seite des OAuth2-Proxy. In der simulierten Test-Umgebung wurde der OAuth2-Proxy als vorgefertigtes Docker-Image geladen und via Umgebungsvariablen entsprechend konfiguriert.

Die notwendigen Umgebungsvariablen sind wie folgt:

```
oauth2-proxy:
    container_name: oauth2-proxy
    image: quay.io/oauth2-proxy/oauth2-proxy:latest
    ports:
      - 4180
    environment:
      - OAUTH2_PROXY_PROVIDER=oidc
      - OAUTH2_PROXY_PROVIDER_DISPLAY_NAME="IdentityServer 4"
      - OAUTH2_PROXY_COOKIE_DOMAINS=reverse.proxy.localhost
      - OAUTH2_PROXY_EMAIL_DOMAINS=*
      - OAUTH2_PROXY_COOKIE_SECURE=true
      - OAUTH2_PROXY_COOKIE_SECRET=Jhy3uQq1td7KngPMJsPqow==
      - OAUTH2_PROXY_REDIRECT_URL=
https://reverse.proxy.localhost/oauth2/callback
      - AUTH2_PROXY_OIDC_ISSUER_URL=
https://reverse.proxy.localhost/identityserver
      - OAUTH2_PROXY_CLIENT_ID=oauth2-proxy
      - OAUTH2_PROXY_CLIENT_SECRET=0edc6d7b-633c-42d6-9b60-fbb2f10c49dc
      - OAUTH2_PROXY_STANDARD_LOGGING=true
      - OAUTH2_PROXY_AUTH_LOGGING=true
      - OAUTH2_PROXY_REQUEST_LOGGING=true
      - OAUTH2_PROXY_REVERSE_PROXY=true
      # Because of self-signed certificate
      - OAUTH2_PROXY_SSL_INSECURE_SKIP_VERIFY=true
      - OAUTH2_PROXY_WHITELIST_DOMAINS=reverse.proxy.localhost
      - OAUTH2_PROXY_HTTP_ADDRESS=http://:4180
      - OAUTH2_PROXY_SCOPE=openid profile scope1
      - OAUTH2_PROXY_USER_ID_CLAIM=given_name
    networks:
      - testnetwork
    deploy:
      restart_policy:
        condition: on-failure
```

Abbildung 8 Umgebungsvariablen für OAuth2-Proxy innerhalb des Docker-Compose Files

Die genaue Beschreibung jedes einzelnen Wertes ist der offiziellen Dokumentation (https://oauth2-proxy.github.io/oauth2-proxy/docs/configuration/overview/) zu entnehmen. Jedoch grob beschrieben wird der IdP mit den entsprechenden URLs, der angelegte Client mit dem dazugehörigen Client-Secret und einige zusätzliche Parameter, die für das lokale Testen innerhalb des Docker-Netzwerkes notwendig waren, gesetzt.

Als nächstes wird der NGINX Reverse Proxy über das Vorhandensein des OAuth2-Proxy durch Konfiguration innerhalb des Servers informiert. Hierzu werden zusätzlich zwei Endpunkte eingerichtet, die auf die Proxy-Präfixe des OAuth2-Proxy weiterleiten. Der Standard-Präfix „/oauth2“ kann jedoch bei Bedarf ohne weiteres durch Umgebungsvariablen geändert werden. Eine Liste von Endpunkten, die der OAuth2-Proxy anbietet, sind hier gelistet (https://oauth2-proxy.github.io/oauth2-proxy/docs/features/endpoints).

```
location /oauth2/ {
        proxy_pass http://oauth2-proxy;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Auth-Request-Redirect $request_uri;
        include common_location.conf;
    }

location = /oauth2/auth {
        proxy_pass http://oauth2-proxy;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Auth-Request-Redirect $request_uri;
        proxy_set_header Content-Length "";
        proxy_pass_request_body off;
        include common_location.conf;
    }
```

Abbildung 9 Zusätzliche Endpunkte für die Weiterleitung von nicht autorisierten Anfragen an den Identity Provider

Die gesetzten Werte orientieren sich an der vorhandenen Dokumentation zum Betrieb hinter einem Reverse-Proxy und sind somit auch in der aktuellen Dokumentation zu finden.

Mittels dieser neuen Endpunkte können nun vorhandene Endpunkte folgendermaßen abgesichert werden:

```
location /secure {
        rewrite ^(/secure.*)$ $1.html break;

        auth_request /oauth2/auth;
        error_page 401 = /oauth2/sign_in;

        # pass information via X-User and X-Email headers to backend,
        # requires running with --set-xauthrequest flag
        auth_request_set $user   $upstream_http_x_auth_request_user;
        auth_request_set $email  $upstream_http_x_auth_request_email;
        proxy_set_header X-User  $user;
        proxy_set_header X-Email $email;

        proxy_pass http://website/;
        include common_location.conf;
}
```

Abbildung 10 Einfügen des auth_request Moduls mit dem entsprechenden Endpunkt des OAuth2-Proxy

Somit können nun einzelne Endpunkte vor unbefugtem Zugriff geschützt werden. Eine serverweite Konfiguration wurde anhand dieses Beispiels jedoch nicht getestet. Jedoch dürfte auch diese Möglichkeit meiner Erfahrung nach funktionieren.

In dieser Konfiguration wurde noch eine Ergänzung eingefügt, die es ermöglicht die Email und den User ans Backend weiterzuleiten. Diese muss aber nicht unbedingt verwendet werden.

Zuletzt wird nun der IdentityServer4 von dem neuen Client in Kenntnis gesetzt. Dies geschieht über einen weiteren Eintrag in der `Config.cs`, in der alle In-Memory-Clients definiert werden.

```
new Client
{
    ClientId = "oauth2-proxy",
    ClientName = "OAuth2-Proxy for Authorization of Clients at NGINX",

    AllowedGrantTypes = GrantTypes.CodeAndClientCredentials,
    ClientSecrets = { new Secret("0edc6d7b-633c-42d6-9b60-
fbb2f10c49dc".Sha256()) },

    RedirectUris = { "https://reverse.proxy.localhost/oauth2/callback" },
    FrontChannelLogoutUri = "https://reverse.proxy.localhost/signout-oidc",
    PostLogoutRedirectUris = { "https://reverse.proxy.localhost/" },

    RequireClientSecret = true,
    RequirePkce = false,

    AlwaysIncludeUserClaimsInIdToken = true,
    AllowOfflineAccess = true,

    RefreshTokenUsage = TokenUsage.OneTimeOnly,
    RefreshTokenExpiration = TokenExpiration.Sliding,
    AllowedCorsOrigins =
    {
          "https://reverse.proxy.localhost"
    },

    AllowedScopes = { "openid", "profile", "scope1" }
}
```

Abbildung 11 IdentityServer Client für die Registrierung von OAuth2-Proxy

Aufgrund der fehlenden Funktion der Verwendung von „Proof Key of Code Exchange“, muss diese Standard Authentifizierungsmethode seitens des IdP deaktiviert werden.

Bewertung

Abschließend erfolgt eine Bewertung der beiden Frameworks in Punkto Verwendbarkeit und Konfigurationsaufwand.

Hinsichtlich der Konfiguration ist „Vouch Proxy“ deutlich einfacher zu bedienen. Neben den Umgebungsvariablen für dessen Betrieb in einem Docker Container, war nur noch der Validierungsendpunkt in der NGINX-Konfiguration zu setzen. Selbes gilt für „OAuth2-Proxy“.

Dieser konnte im Anschluss für beliebig viele zu schützende Endpunkte wiederverwendet werden. Auch die Möglichkeit einer Authentifizierung auf Serverebene ist gegeben.

Als Problem stellte sich die Verwendung in der Subdomain dar. Bei vielen Unternehmen ist es üblich, dass sie ihre Cloud-Anwendungen via Subdomains auf Kunden spezifizieren und somit jeder Kunde eine eigene Instanz dieser Services über die Namensgebung erhält. Bei einer Einbindung von „Vouch Proxy“ wäre dieses Konzept nicht mehr ohne weiteres möglich.
Der Proxy könnte im Grunde seine übergeordnete Domain schützen. Jedoch bei Vorliegen unterschiedlicher Kunden mit einzigartigen Rechten, wäre die Möglichkeit einer feingranulareren Rechteverteilung nicht mehr gegeben.

Im Rahmen dieses Testprojektes war es jedoch nicht möglich den „Vouch Proxy“ mit dem IdentityServer4 zu verbinden und somit eine Authentifizierung zu ermöglichen.

Bei „lua-resty-openidc“ funktioniert der Schutz der Subdomains auf der Erweiterung der zu schützenden Endpunkte. Bedeutet, wenn der Endpunkt „/geschuetzt“ durch das Modul mit einer Authentifizierung versehen werden soll, dann wird eine imaginäre Callback-URL unter „/geschuetzt/callback“ eingerichtet, an die der IdP den Authentifizierungs-Code zum Abruf eines Access-Tokens schickt. Somit ist die Konfiguration unabhängig von Domains bzw. Subdomains und kann endpunktspezifisch eingeführt werden.

Einziger Nachteil bei „lua-resty-openidc“ ist sein enormer Anteil an Quellcode für die Konfiguration. Zwar orientieren sich die Namen der einzelnen Parameter an deren Funktion, jedoch sind es zur Absicherung eines Endpunktes deutlich mehr, wie zum Beispiel bei „Vouch Proxy“. Dies resultiert somit in deutlich größeren Konfigurationsdateien für den NGINX und schränkt zusätzlich deren Übersichtlichkeit ein. Jedoch besteht bestimmt die Möglichkeit diese lokalen Variablen in eine Art Umgebungsvariable zu kapseln und somit serverweit einzusetzen.

Bezüglich der gebotenen Sicherheit ist das Lua Modul deutlich dem noch jungen Projekt „Vouch Proxy“ überlegen. Es bietet neben dem Authorization Code Flow + PKCE noch weitere Möglichkeiten der Authentifizierung nach den Standards in OAuth2 an. Aus diesem Grund wurde es mit dem „OpenID Certified“ versehen.

„Vouch Proxy“ und „OAuth2-Proxy“ bieten hingegen noch keinerlei Sicherheitsmechanismen wie PKCE. Neben der Client-Authentifizierung mit „Shared-Secret“ stehen zurzeit nur wenig Möglichkeiten zur Verfügung, die alltagstauglich sind. Jedoch ist der Pull-Request Historie auf GitHub zu entnehmen, dass die beteiligte Community am Wachstum interessiert ist und selbstständig neue Features nachreicht.

Verweise

[1] Jochen Christ, „INNOQ.com OpenID Connect Auth-Proxy," INNOQ, 30 07 2018. [Online]. Available: https://www.innoq.com/de/blog/auth-proxy/. [Zugriff am 30 11 2020].

[2] Benjamin Foote, „GitHub.com Vouch Proxy Repository," [Online]. Available: https://github.com/vouch/vouch-proxy. [Zugriff am 30 11 2020].

[3] Hans Zandbelt, „GitHub.com lua-resty-openidc Repository," [Online]. Available: https://github.com/zmartzone/lua-resty-openidc. [Zugriff am 30 11 2020].

[4] OpenResty, „GitHub OPM OpenResty Repository," [Online]. Available: https://github.com/openresty/opm. [Zugriff am 01 12 2020].

[5] LuaRocks, „GitHub LuaRocks Repository," [Online]. Available: https://github.com/luarocks/luarocks. [Zugriff am 01 12 2020].

[6] L. Crilly, „Nginx.com Blog Validating OAuth 2.0 Access Tokens with NGINX and NGINX Plus," NGINX, 13 05 2019. [Online]. Available: https://www.nginx.com/blog/validating-oauth-2-0-access-tokens-nginx/. [Zugriff am 30 11 2020].

[7] J. Speed, N. Meves, H. Jenkins und D. Bond, „OAuth2 Proxy," Open Source, [Online]. Available: https://oauth2-proxy.github.io/oauth2-proxy/. [Zugriff am 15 12 2020].

Abbildungsverzeichnis