SCUMA - Smart Contract based User Managed Access

Table of Contents

This concept defines a smart contract which allows the owner of a resource to grant access to resources he owns to particular users.

Definitions

EOA - external owned account

In general, there are two types of accounts: externally owned accounts, controlled by private keys, and contract accounts, controlled by their contract code. See Ethereum Accounts.

Resource - web resource

A resource is available via the internet and can be localized and access by an Uniform Resource Locator (URL).

Resource Provider

The resource provider holds resources and provides interfaces to access them.

ProtectionAuthorizationId

The ProtectionAuthorizationId is the 20 byte address of an EOA controlled by the provider. The owner authorizes the controller of this Id to use a smart contract to protect his resources.

User

The user accesses resources by using the interfaces provided by the resource provider.

UserId

The UserId uniquely identifies the user. It is the 20 byte address of an EOA controlled by the user.

Owner

The owner is a special user which owns resources. The ownership of resources is defined by the content of resources. The resources owned by the owner contain the UserId of the owner.

Smart contract

Smart contracts digitize agreements by turning the terms of an agreement into computer code that automatically executes when the contract terms are met. They are stored on a blockchain, addressed by a contract account, behave exactly as programmed and cannot be changed. See Smart Contracts.

ContractId

The ContractId is the 20 byte address of a contract account. See Ethereum Accounts.

Use Cases

Protect resource

Resources are protected by a default deny policy. In order to enable the resource owner to define access policies the resource needs to be registered at the smart contract by the resource provider.

Beforehand, the resource provider needs the entitlement of the resource owner to protect the resource on his behalf.

Figure 1. sequenz diagram protect resource

Access resource

The user access the resource via the interfaces provided by the resource provider using his EOA. The provider maps the resource request onto the protected resources and asks the smart contract for permissions. The smart contract returns the permissions which then are enforced by the provider.

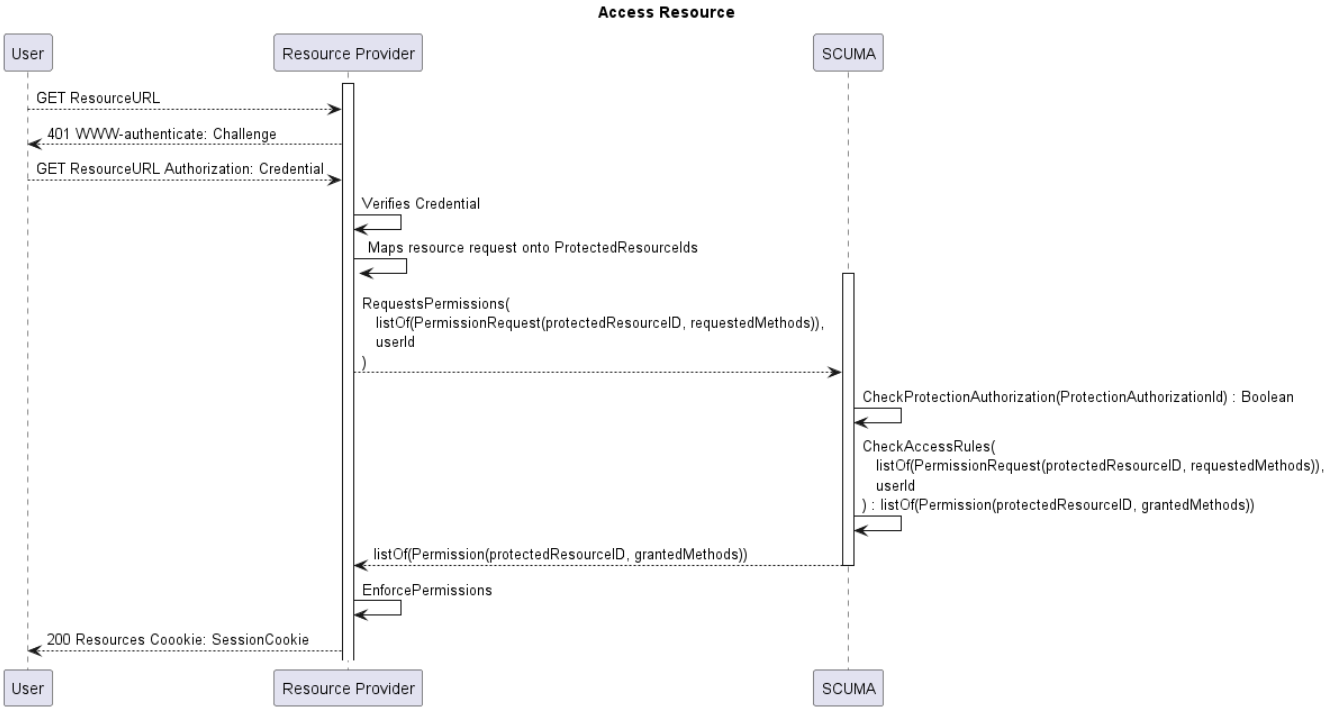

Figure 2. sequenz diagram access resource

Usage

There are four API supported by the scuma library:

- Management API between resource owner and resource provider
- Access API between user and resource provider
- Control API between resource owner and SCUMA
- Protection API between resource provider and SCUMA

The Management API and the Access API should be implemented according to the API used to access the resoures. The scuma library only provides support functions for EOA based authentication. The actual implementations of the Management API and the Access API are out of scope of this specification.

The Control and Protection API uses the Ethereum JSON RPC protocol to interact with Ethereum clients and the smart contracts hosted on the Ethereum block chain. The scuma library hides the complexity of the Ethereum JSON RPC protocol and allows to interact with the scuma contract by an interface entirely defined in kotlin.

User APIs (Access API and Management API)

The User APIs are the interfaces between the users (resource owner and resource user) and the resource provider. They are mainly defined by the capabilities of the resource provider. In case of FHIR the resource server provides a REST interface HL7 - FHIR to provide access to medical resources. Further, the resource owner uses the Management API to request protection of accessed resources and to get information about protected resources (e.g. the protected resource id). In order to make sure that these interfaces are only used by the authorized users, the users need to be

authenticated by the resource server.

HTTP Origin-Bound Authentication (HOBA)

HTTP Origin-Bound Authentication (HOBA) RFC7486 is a digital-signature-based design for an HTTP authentication method. HOBA is specified to use RSA signature but allows registration of other signature schemes. HOBA was extented to use SECP2561K1 signatures. This extension allows the user to authenticate against the server provides by using his external owned address and the corresponding private key.

1. The user connects to the resource server and makes a FHIR request:

```
GET /resource HTTP/1.1
HOST: resourceserver.com
```

2. The server rejects the request with status code 401 and includes a challenge in the WWW-Authenticate header:

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: HOBA challenge="MPl_cQSW5Aa40kGGo6haUsm4Kkzs7pQ8t4are0mzD9s="
max-age=10 realm="scuma"
```

   - `challenge` is a base64url-encoded challenge value that the server chose to send to the client. The challenge is chosen so that it is infeasible to guess and is derived from a random byte string of 32 bytes (256 bits).
   - `max-age` specifies the number of seconds from the time the HTTP response is emitted for which responses to this challenge can be accepted; for example, "max-age: 10" would indicate ten seconds. If max-age is set to zero, then that means that only one signature will be accepted for this challenge.
   - `realm` indicates the scope of protection in the manner described in RFC7235. The `realm` attribute MUST NOT appear more than once.

3. The user signs the challenge RFC7486-section2 and repeats the original request with an Authorization header containing the signed challenge RFC7486-section3 :

```
GET /resource HTTP/1.1
HOST: resourceserver.com
Authorization: HOBA result="0xfe3b557e8fb62b89f4916b721be55ceb828db
d73.eNIv3lOevIwjzuyrxGkliYnyAXUUuNC_oQZqplhO6rwp555smaglDfbHCroJdNG
K9eqFgcVy4dL89nKC18hPk=.OncuAM2XAKi97RdjL7JgImdZ4a2FmCZSWgULpXF0q_B
YAyALY35DlJGSiZjMb-2oDvvIcuh7teYJ4j2xXFikPAA="
```

   - `result` is a dot-separated string that includes kid, challenge, nonce and signature: `kid + '.' + challenge + '.' + nonce +'.' + sig`
     - `kid` key identifier. EOA of the resource owner

- `challenge` challenge as received in the WWW-Authentication header
- `nonce` a random value chosen by the resource owner derived from a random byte string of 32 byte length

4. The resource server verifies the credential received in the Authorization header by verifying the signature using the received parameters and additional context information RFC7486-section2. Further it checks that the response was received within the specified `max-age`.
5. After successful authentication the server returns the requested resource. The response shall include a session cookie that allows the user client to indicate its authentication state in future requests - HTTP State Management Mechanism - RFC6265 .

Server side usage of library

The server uses the class ``HobaAuthenticationChallenge to create the Hoba challenge and sends the challenge in the WWW-Authententication header in a 401 response:

```
// challenge is 256bit random
val challenge = Random.nextBytes(32)
// realm defines the context
// max-age requests the client to answer within the next 10s
val hobaAuthenticationChallenge = HobaAuthenticationChallenge(
    maxAge = 10,
    realm = "scuma",
    challenge = challenge
)
val wwwAuthenticationContent = hobaAuthenticationChallenge.toString()
```

The user client signs the challenge, repeats the original request with an authorization header which contains the signed challenge. The server receives the request and verifies the authorization header:

```
val hobaAuthorizationCedential = HobaAuthorizationCedential.fromString
(authorizationHeaderContent)
// In order to verify the signature the server has to hand over its orgin as specified
in its server certificate and the predefined realm used in the challenge.
assert(hobaAuthorizationCedential.verify(
        origin = origin,
        realm = "scuma"
    )
)
```

Client side usage of library

The client receives the challenge from the server in the WWW-Authententication header of a 401 response and repeats the rejected request with an authorization header which contains the signed challenge as credential:

```
// nonce is 256bit random
val nonce = Random.nextBytes(32)
val hobaAuthorizationChallenge = HobaAuthenticationChallenge.fromString
(wwwAuthenticationHeaderContent)
// In order to create the credential the client takes the challenge from the received
hoba authentication challenge and an randomly choosen nonce.
val hobaAuthorizationCedential = hobaAuthorizationCedential(
    challenge = hobaAuthorizationChallenge.challenge,
    nonce = nonce
).apply{
// The client signs the challenge using his private key and the origin of the server
taken from the server certificate.
    sign(privateKey, origin)
}
val authorizationHeaderContent = hobaAuthorizationCedential.toString()
```

The scuma library has extended the HOBA RFC by a new signature algorithm. Instead of the defined RSA algorithms the scuma library uses SECP256K1. That is the crypto algorithm used by the Ethereum block chain. So the cryptographic keys bound to the EOAs of the users can be used for authentication.

In case high level of assurance is required it is recommended to use hardware backed keys (e.g. Android StrongBox). Ethereum uses SECP256K1 by default which unfortunately is not supported by secure elements (e.g. „Titan™ M“- secure chip). However, when running nodes in a private network, it is possible to configure an alternative elliptic curve. E.g. Hyperledger Besu allows to configure the elliptic curve in the network section of the genesis file (see Hyperledger Besu: Using alternative elliptic curves).

Management of protected resources

A detailed description of the protocol for the management of protected resources is out of scope of this specification. The actual implementation of the protocol messages should be choosen in a way that fits best to the access protocol of the managed resources. E.g. in case of FHIR a REST-API is defined to request and cancel protection and to get information about protected resources.

Control API

Protection API

Open issues

- DID instead of EOA
- attribute/group based access control (e.g. by using 1-of-N tree signatures)
- privacy aware policies (e.g. by using zkay: A Language for Private Smart Contracts on Ethereum)

Appendix

Smart Contract

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract ScumaContract {

    address Owner;

    address[] protectionAuthorizationIds;
    mapping(address => uint) protectionAuthorizationIdIndices;
    Policy[] policies;
    Rule[][] ruleLists;
    mapping(uint256 => uint) policyIndices; // protectedResourceId to policy index

    struct Policy {
        uint256 what;  // protected resource id
        Rule[] ruleList;
    }

    struct Rule {
        address who;  // userId
        uint256 how;  // bit set representing the defined access methods; methods
defined by application
    }

    struct Permission {
        uint256 protectedResourceId;
        uint256 grantedMethods; // bit set representing the granted access methods;
methods defined by application
    }

    struct PermissionRequest {
        uint256 protectedResourceId;
        uint256 requestedMethods; // bit set representing the requested access
methods; methods defined by application
    }

    constructor () {
        Owner = msg.sender;
        protectionAuthorizationIds.push();
        // index 0 reserved to indicate no id
        policies.push();
        // index 0 reserved to indicate no id
    }

    modifier onlyOwner(){
```

```
        require(msg.sender == Owner, 'Not owner');
        _;
    }

    modifier onlyAuthorizedProviders(){
        require(protectionAuthorizationIdIndices[msg.sender] > 0, 'Provider not
authorized');
        _;
    }

    function registerProvider(address protectionAuthorizationId) public onlyOwner {
        uint protectionAuthorizationIdIndex = protectionAuthorizationIdIndices
[protectionAuthorizationId];
        require(protectionAuthorizationIdIndex == 0, 'rejected - protection
authorization id is already registered');
        protectionAuthorizationIds.push(protectionAuthorizationId);
        protectionAuthorizationIdIndices[protectionAuthorizationId] =
protectionAuthorizationIds.length - 1;
    }

    function unregisterProvider(address protectionAuthorizationId) public onlyOwner {
        uint index = protectionAuthorizationIdIndices[protectionAuthorizationId];
        require(index > 0, 'rejected - protection authorization id does not exist');
        protectionAuthorizationIds[index] = protectionAuthorizationIds
[protectionAuthorizationIds.length - 1];
        protectionAuthorizationIds.pop();
        delete protectionAuthorizationIdIndices[protectionAuthorizationId];
    }

    function getProviderCount() public onlyOwner view returns (uint256){
        return protectionAuthorizationIds.length - 1;
    }

    function getProviders() public onlyOwner view returns (address[] memory){
        address[] memory providerIds = new address[](protectionAuthorizationIds.length
- 1);
        for (uint i = 1; i < protectionAuthorizationIds.length; i++) {
            providerIds[i - 1] = protectionAuthorizationIds[i];
        }
        return providerIds;
    }

    function unregisterAllProviders() public onlyOwner {
        for (uint i = 1; i < protectionAuthorizationIds.length; i++) {
            delete protectionAuthorizationIdIndices[protectionAuthorizationIds[i]];
        }
        delete protectionAuthorizationIds;
        protectionAuthorizationIds.push();
    }

    function registerResource(uint256 protectedResourceId) public
```

```
onlyAuthorizedProviders {
        uint policyIndex = policyIndices[protectedResourceId];
        require(policyIndex == 0, 'rejected - protected resource id is already
registered');
        ruleLists.push();
        policies.push();
        policies[policies.length - 1].what = protectedResourceId;
        policies[policies.length - 1].ruleList = ruleLists[ruleLists.length - 1];
        policyIndices[protectedResourceId] = policies.length - 1;
    }

    function unregisterResource(uint256 protectedResourceId) public
onlyAuthorizedProviders {
        uint policyIndex = policyIndices[protectedResourceId];
        require(policyIndex > 0, 'rejected - protected resource id does not exist');
        policies[policyIndex] = policies[policies.length - 1];
        policies.pop();
        delete policyIndices[protectedResourceId];
    }

    function getResourceCount() public onlyAuthorizedProviders view returns (uint256){
        return policies.length - 1;
    }

    function getResourceIds() public onlyAuthorizedProviders view returns (uint256[]
memory){
        uint256[] memory resourceIds = new uint256[](policies.length - 1);
        for (uint i = 1; i < policies.length; i++) {
            resourceIds[i - 1] = policies[i].what;
        }
        return resourceIds;
    }

    function unregisterAllResources() public onlyAuthorizedProviders {
        for (uint i = 0; i < policies.length; i++) {
            delete policyIndices[policies[i].what];
        }
        delete policies;
        policies.push();
        delete ruleLists;
    }

    function setRule(uint256 protectedResourceId, address userId, uint256 methods)
public onlyOwner {
        uint policyIndex = policyIndices[protectedResourceId];
        require(policyIndex > 0, 'protected resource does not exist');
        policies[policyIndex].ruleList.push(Rule(userId, methods));
    }

    function getPolicy(uint256 protectedResourceId) public onlyOwner view returns
(Rule[] memory){
```

```
        uint policyIndex = policyIndices[protectedResourceId];
        require(policyIndex > 0, 'protected resource does not exist');
        return policies[policyIndex].ruleList;
    }

    function deleteRule(uint256 protectedResourceId, uint index) public onlyOwner {
        uint policyIndex = policyIndices[protectedResourceId];
        require(policyIndex > 0, 'protected resource does not exist');
        Policy storage policy = policies[policyIndex];
        require(index < policy.ruleList.length, 'rule does not exist');
        policy.ruleList[index] = policy.ruleList[policy.ruleList.length - 1];
        policy.ruleList.pop();
    }

    function requestPermissions(address userId, PermissionRequest[] calldata
permissionRequests) public onlyAuthorizedProviders view returns (Permission[] memory){
        // first count permissions
        uint count = 0;
        for (uint i = 0; i < permissionRequests.length; i++) {
            PermissionRequest memory permissionRequest = permissionRequests[i];
            uint policyIndex = policyIndices[permissionRequest.protectedResourceId];
            if (policyIndex > 0) {
                Policy storage policy = policies[policyIndex];
                for (uint j = 0; j < policy.ruleList.length; j++) {
                    Rule memory rule = policy.ruleList[j];
                    if (rule.who == userId && (rule.how & permissionRequest
.requestedMethods != 0)) {
                        count++;
                    }
                }
            }
        }
        // allocate array and fill
        Permission[] memory permissions = new Permission[](count);
        count = 0;
        for (uint i = 0; i < permissionRequests.length; i++) {
            PermissionRequest memory permissionRequest = permissionRequests[i];
            uint policyIndex = policyIndices[permissionRequest.protectedResourceId];
            if (policyIndex > 0) {
                Policy storage policy = policies[policyIndex];
                for (uint j = 0; j < policy.ruleList.length; j++) {
                    Rule memory rule = policy.ruleList[j];
                    if (rule.who == userId && (rule.how & permissionRequest
.requestedMethods != 0)) {
                        permissions[count++] = Permission(permissionRequest
.protectedResourceId, rule.how & permissionRequest.requestedMethods);
                    }
                }
            }
        }
        return permissions;
```

```
    }
}
```