Usable Security for RIOT and the IoT

RIOT-Summit 2018, Amsterdam, NL

Olaf Bergmann, Stefanie Gerdes

2018-09-13

Communication in Constrained Environments

- Constrained Application Protocol (CoAP, RFC 7252)
  - designed for special requirements of constrained environments
  - Similar to HTTP (RESTful architecture style)
    - server has items of interest
    - client requests representation of current state
- Datagram Transport Layer Security (DTLS) binding
- How can users keep the control over their data and devices? $\rightarrow$ Authorization

Building Blocks

RIOT already has the all tools you need:

- CoAP implementations
- Data representation libraries
- Crypto tools
- DTLS implementations

Building Blocks

RIOT already has the all tools you need:

- CoAP implementations
- Data representation libraries
- Crypto tools
- DTLS implementations

How to use these for securing your IoT application?

Option 1: sock_secure + tlsman (Raul Fuentes)

PRs #7397 and #7649

- basic idea: provide API based on existing socket primitives
  - `secure_sock_connect()`, `secure_sock_send()`,...
- (D)TLS implementation agnostic API
  - `tlsman_create_channel()`, `tlsman_send_data_app()`, `tlsman_close_channel()`,...
- can work with nanocoap and gcoap

Example: sock_secure server

```
sock_secure_session_t secure_sess = { .flag=0, .cb=NULL };
secure_sess.flag = TLSMAN_FLAG_STACK_DTLS | TLSMAN_FLAG_SIDE_SERVER;
uint16_t ciphers[] = SECURE_CIPHER_LIST;
sock_secure_load_stack(&secure_sess, ciphers, sizeof(ciphers));

sock_udp_t sock;
sock_udp_ep_t local = ...;
sock_udp_ep_t remote = ...;

sock_udp_create(&sock, &local, NULL, 0);
ssize_t res = sock_secure_initialized(&secure_sess, cb, (void *)&sock,
                                      (sock_secure_ep_t*)&local,
                                      (sock_secure_ep_t *)&remote);

while(sock_secure_read(&secure_sess)) { ... }

sock_secure_release(&secure_sess);
sock_udp_close(&sock);
```

Option 2: gcoap + sock_tdsec (Ken Bannister)

https://github.com/kb2ma/RIOT/tree/sock/tdsec

- basic idea: simplified API for secure sockets with tinydtls
  - `tdsec_create()`, `tdsec_connect()`, `tdsec_read()`, `tdsec_send()`
- hidden from application developer

```
size_t gcoap_req_send2(...)
{
   ...
#ifdef MODULE_SOCK_TDSEC
    ssize_t res = tdsec_connect(&_tdsec, remote);
    if (res >= 0) {
        res = tdsec_send(&_tdsec, buf, len, remote);
    }
```

Current Limitations

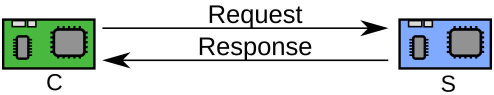
- credentials defined at build-time (`tdsec_params.h`, `dtls_keys.h`)

```
tdsec_psk_params_t tdsec_psk_params[] = {
    { .client_id = "homer", .key = "secretPSK", },
    { .client_id = "marge", .key = "anotherPSK", }
};
```

- need to know every potential communication peer in advance
- no multiplexing of security associations, applications are not aware of underlying transport session
- no dynamic *authorization* (cleartext vs. protected resources)

Our Goal

- A Client (C) wants to access an item of interest, a web resource (R), on a Server (S).
- A priori, C and S do not know each other, have no security association. They might belong to different owners.
- C and / or S are located on a constrained node.



Authorization Protocol Design

- Secure exchange of authorization information
- Establish secure channel between constrained nodes (e.g., DTLS but could be “object security” as well)
- Use only symmetric key cryptography on constrained nodes
- RESTful architectural style
- **Relieve constrained nodes from managing authentication and authorization**

Authenticated Authorization

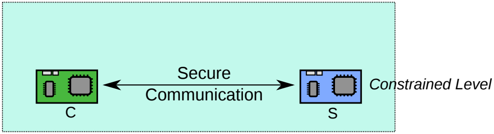
- Determine if the owner of an item of interest allows an entity to access this item as requested.
- **Authentication**: Verify that an entity has certain attributes (cf. RFC4949).
- **Authorization**: Grant permission to an entity to access an item of interest.
- **Authenticated Authorization**: Use the verified attributes to determine if an entity is authorized.

Tasks for Authenticated Authorization

- Beforehand: Provide information for Authenticated Authorization
  - Make attribute-verifier-binding verifiable: Validate that an entity actually has the attributes it claims to have (e.g. that it belongs to a certain user) and bind the attributes to a verifier (e.g. a key) using the endorsement info.
  - Define access policies (entity with attribute x has this set of permissions).
- At the time of the request: Check access request against the provided information
  - Check the verifier a received access request is bound to.
  - Check the verifier-attribute binding.
  - Determine the authorization using the attributes.
  - Enforce the authorization.

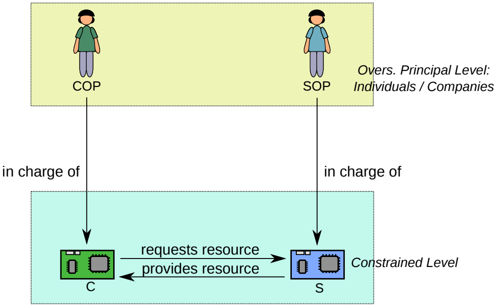
Constrained Level Actors

- C and S are constrained level actors: able to operate on a constrained node.
- C attempts to access a resource.
- S hosts one or more resources.
- Tasks:
  - Determine if sender is authorized to access as requested.
  - Enforce the authorization



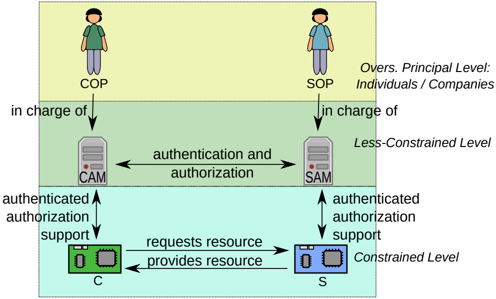
Principal Level Actors

- C and S are under control of principals in the physical world.
- COP is in charge of C: specifies security policies, e.g. with whom S is allowed to communicate.
- SOP is in charge of S: specifies security policies, e.g. authorization policies.



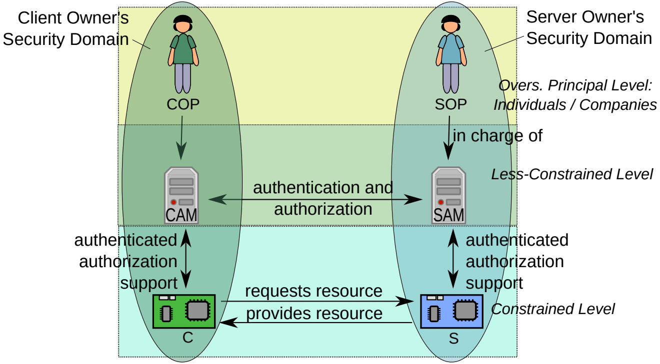
Less-Constrained Level

- CAM and SAM act in behalf of their respective owner.
- Tasks:
  - Obtain the security objectives from their owner.
  - Authenticate the other party.
  - Provide simplified authorization rules and means for authentication to their constrained devices.

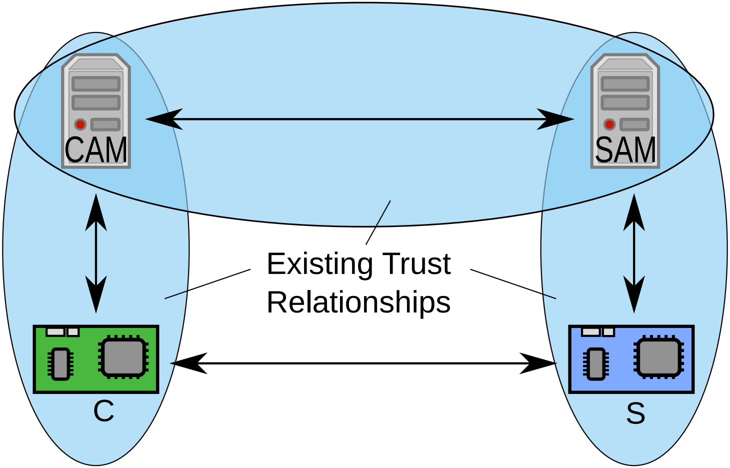


Security Domains

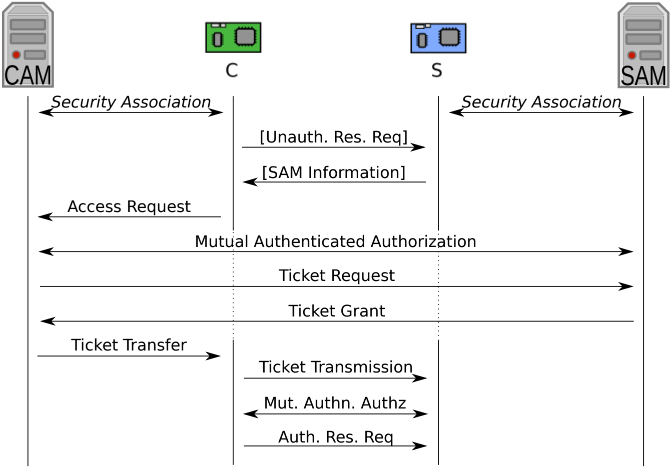
- A priori, C and S do not know each other, might belong to different security domains



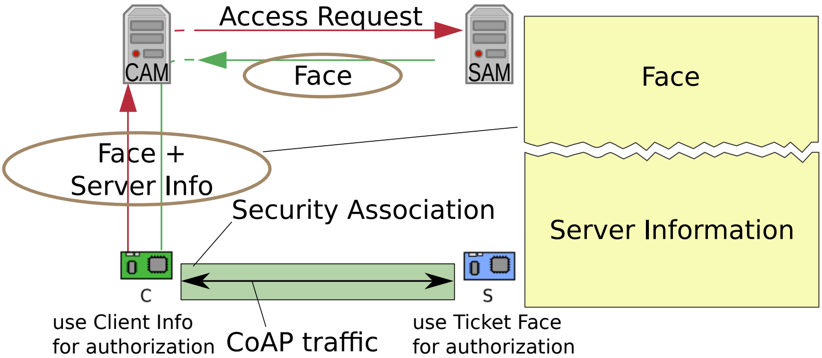
Initial Trust Relationships



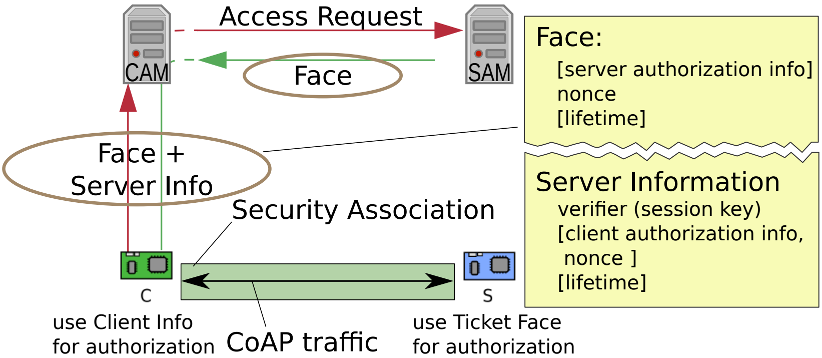
Protocol Overview



Access Ticket



Access Ticket



Summary: The DCAF Protocol

- Less-contrained nodes do the hard work (possibly even public-key crypto)
- Can utilize DTLS to transmit authorization info
- Authenticate origin client by its access ticket:
  - S and SAM share at least one session key
  - SAM creates Ticket Face + Verifier, tells CAM, C
  - C initiates DTLS handshake with S
  - S derives PSK from Ticket Face
- Knowledge of Verifier authenticates C to S!
- Knowledge of PSK authenticates S to C!
- Authorization information valid for the entire session
- Verifier ensures Face's integrity

Example Implementation Using libcoap 1/2

Initialization

```
dcaf_config_t config = { .am_uri = "coaps://am.dcaf.science:7744" };
dcaf_context_t *dcaf = dcaf_new_context(&config);
coap_startup();

/* set credentials for talking to our authorization manager */
coap_context_set_psk(dcaf_get_coap_context(dcaf)), 0),
                     "s.constrained.space", key, key_length);

while (true) { coap_run_once(...); }
```

Example Implementation Using libcoap 2/2

Request Handler

```
void handle_request(...)
{
  ...
  if (!dcaf_is_authorized(session, request)) {
    dcaf_result_t res;
    res = dcaf_set_sam_information(session, DCAF_MEDIATYPE_DCAF_CBOR,
                                   response);
    return;
  }

  ... handle authorized request ...
}
```

Note: Ideally, this would happen in the {nano,micro,g,lib}coap core implementation.

Conclusion

- Observations
  - Usable security requires simple but effective APIs
  - Internet of Things demands multi-domain authorization
  - complex authentication and authorization tasks can be delegated
  - Real-world applications often need to send subsequent messages over the same session
- RIOT topics
  - Finish DTLS/Sock/CoAP integration
  - Add DCAF for key distribution