CHAPTER 10

RESTful IoT Authentication Protocols

H.V. Nguyen, L. Lo Iacono
Cologne University of Applied Sciences, Cologne, Germany

1 INTRODUCTION

The Internet of Things (IoT) has evolved into an important building block for future IT visions such as Smart City, Smart Building, Smart Home, Smart Mobility and Industry 4.0. As those systems cover a large number of sensors and nodes that communicate frequently with each other, design concepts for developing IoT systems must be highly scalable (Atzori et al., 2010). One approach to fulfilling this requirement is the architectural style of the web known as representational state transfer (REST) (Fielding, 2000). REST is a guideline for designing large-scale distributed systems. Due to its strength in terms of scalability, interoperability, and efficiency, the application of REST has been adopted in areas such as service-oriented architecture (SOA) (Erl et al., 2012; Gorski et al., 2014a,b) and cloud computing (Lo Iacono and Nguyen, 2015a). Consequently, REST is gaining traction as an approved concept for implementing large-scale IoT systems (Shelby et al., 2014; Urien, 2015). Since IoT services and applications share sensible information, security is another prerequisite of paramount importance (Atzori et al., 2010). Hence, REST-Security for IoT-based environments is becoming relevant.

This chapter introduces an authentication concept for RESTful IoT protocols, which consider scalability and resource-restrictiveness constraints stemming from the architectural style REST and IoT networks and devices. The chapter is organized as follows: The foundations of REST are introduced in Section 2. Based on this background, Section 3 briefly introduces Constrained Application Protocol (CoAP) (Shelby et al., 2014) and Remote APDU Call Secure (RACS) (Urien, 2015), two RESTful protocols for the IoT domain. On the basis of the REST foundations and their technical instantiations, Section 4 proposes a methodology for developing security schemes for REST-based (IoT) systems of any kind. Following this methodology, Section 5 extends REST by an integrated authentication (Lo Iacono and Nguyen, 2015b) while remaining on the same abstraction level as REST itself. Section 6 utilizes the proposed security scheme as a guideline for implementing concrete authentication protocols to CoAP.
(Nguyen and Lo Iacono, 2015) and RACS. Based on these results, Section 7 concludes this chapter with a summary, as well as a future research and development challenges.

2 REST FOUNDATIONS

Roy Fielding introduced the architectural style REST (Fielding, 2000) in his doctoral dissertation. The basic idea behind this concept is to provide a guideline proposing architectural constraints for designing highly scalable distributed software systems. These constraints are illustrated in Fig. 1.

The communication in REST is based on the client-server and request-response model. Therefore it is always the client who initiates the communication by issuing a request addressing a resource from a server. In the context of REST, a resource is an abstract definition of information intended for human interpretation or machine processing. Thus, a resource can have multiple representations. Moreover, a resource must be addressable by a unique resource identifier. Hence, each request must include a resource identifier. In conjunction with the requested action, both data elements define the intention and destination of a request. The resource identifier syntax and the request actions must be standardized and predefined by the uniform interface so that all components in a REST architecture can understand the purpose of a

![Fig. 1: REST constraints and principles (Gorski et al., 2014a).](image-url)
request. Fielding does not specify any concrete actions for REST-based systems; the definition of a fixed set of actions is rather a matter of the implementation of the uniform interface. REST-based systems mostly use actions to create, read, update and delete a resource. Depending on the action, a request can comprise a resource representation such as that for creating or updating. In addition to the resource identifier syntax and request action, the uniform interface also defines a fixed set of further metadata elements describing, for example, the size and the media type of a resource representation. Since REST messages are constrained to be stateless and cacheable, metadata can also define state information such as authentication or session data and caching information. As requests in REST contain all required data elements including the action, the resource identifier, state and cache information, and further metadata, their semantics are self-descriptive for each server. This means that every server can understand the intention of a request without maintaining any particular state and without knowing the client in advance, since all requests are self-descriptive and all data elements are standardized.

The stateless and self-descriptive nature of REST messages makes them well suited for intermediate processing. Thus in many cases, the communication flow in REST-based systems is layered by multiple intermediate systems to ensure efficiency and scalability. For instance, intermediaries are utilized to cache messages, saving a server from replicated processing with the aim to reduce communication latency. A load balancer is another prevalent intermediate component to distribute workloads across multiple servers in order to provide scalability. Further intermediaries can be, for example, security gateways performing authentication, as well as access control or cross-protocol proxies encapsulating legacy or other related service systems.

Once a request receives a server, the endpoint returns a response including a response meaning informing about the result of a request. As with requests, a response can contain further metadata, such as authentication or caching information, and a resource representation accompanied by resource representation metadata. Moreover, the metadata and resource representation of a REST response may contain hypermedia elements defining application control information; that is, description of actions to be applied to resource identifiers, which are embedded in the metadata and resource representation.

The metadata and resource representation of the returned response triggers a state change inside the client. Based on hypermedia information within the response, a client can choose the next desired request, or state change, to repeat the described cycle. This kind of application control concept is called hypermedia as the engine of application state, one of the key interface constraints of REST.

All of these aforementioned constraints and principles describe a RESTful architecture that promotes scalability, generality of interfaces, and independent deployment of components, as well as reduces latency, enforces security, and encapsulates legacy and related systems.

Hypertext Transfer Protocol (HTTP) (Fielding and Reschke, 2014) is one protocol that is in conformance with the REST constraints and principles, as it is based on the client-server and request-response model. Moreover, it specifies a set of request actions (i.e., HTTP methods) and a set of further metadata such as header fields or status codes. Resources in HTTP can be addressed by a standardized resource identifier syntax, namely the URI (Berners-Lee et al., 2005) syntax. Also, HTTP messages can include a resource representation such as JSON (Crockford, 2006), HTML (Hickson et al., 2014), or XML (Bray et al., 2008). The metadata and resource representation may contain description on hypermedia relationships (i.e., links or resource identifiers) to describe the next possible state changes or requests for the client.
Additionally, HTTP messages are stateless and cachable, so they can be processed in intermediate systems, such as proxies, cache servers, or load balancers, without saving any contextual information. HTTP was originally invented as the technical foundation for the web, the world’s largest distributed system.

### 3 RESTful IoT PROTOCOLS

Due to the success of the web and the strengths of REST, SOA, cloud computing, and the IoT domain have adopted principles and constraints of this architectural style to implement highly scalable service systems. For IoT, which aims at realizing a worldwide distributed and interconnected system, CoAP and RACS have been proposed as two RESTful protocols with a specified focus on constrained devices and networks.

#### 3.1 RESTful CoAP

CoAP (Shelby et al., 2014) is binary application protocol based on HTTP. As for HTTP messages, CoAP message are divided in two parts: a header comprising metadata and a body (payload) containing a resource representation. Each CoAP header begins with a start header containing a version number (V), a message type (T), a token length (TKL), a code (C), and a message ID (MID). In contrast to HTTP, CoAP defaults to User Datagram Protocol (UDP) (Postel, 1980), an unreliable means of transport. To ensure transport reliability, CoAP messages can be confirmed. Such messages comprise the message type 0 (T=0). The reception of confirmable messages must be approved by an acknowledgement message (ACK), which is represented by the message type 2 (T=2). A receiver can also reject a confirmable message by sending a reset message (RST). These messages contain the message number 3 (T=3). Nonconfirmable CoAP messages contain the message number 1 (T=1). The token length describes the size of the token, which is used to match a response to its corresponding request. In case of a request, the code defines the method (i.e., the request action). CoAP provides four methods: GET (C=0.01), POST (C=0.02), PUT (C=0.03), and DELETE (C=0.04). These methods have the same functionality and properties as the methods in HTTP. CoAP responses use the code to represent the status code, such as Content (C=2.05), declaring that the response contains a resource representation, while Bad Request (C=4.00) or Internal Server Error (C=5.00) informs the client about a client-side or server-side error, respectively. The message ID concludes the CoAP start header. This ID is an identifier for linking a reset or an acknowledgement message to its respective confirmable message. Further metadata can be described by CoAP options. Important options for defining the resource identifier in a CoAP request are Uri-Path and Uri-Query. The Accept option is another crucial metadata element for declaring the desired media type being requested. Another mandatory option for defining the media type of a resource representation is content-format. The delimiter to separate the CoAP header from the body is 255 (111111112).

Fig. 2 depicts a CoAP request retrieving a resource representation in JSON (denoted by the number 50, which represents application/json) from a CoAP server. This request is given a so-called piggybacked response, which represents an acknowledgement as well as a response message. Such a response contains the same token value and message ID as the corresponding request.
If a server is not able to return a piggybacked response immediately, it can send a plain acknowledgement message instead to inform the client about the successful request reception (Fig. 3). This acknowledgement message contains an empty token value and same the message ID as the respective confirmable request. Once a server is able to return a response, it sends a confirmable response with the same token value as the corresponding request to the client. This so-called separate response contains a new message ID, as it represents a to-be confirmed message by itself. A client receiving a separate response must approve the reception of the message, with the new acknowledgement message containing the message ID of the separate response.

Since CoAP is based on HTTP, it contains all the characteristics for being a RESTful protocol. The communication in CoAP is stateless and follows the client-server and request-response models. CoAP requests always include a URI for identifying the requested resource. Moreover, CoAP specifies a set of request actions (i.e., CoAP methods) and further metadata describing caching information or the media type of the resource representation. In conjunction with the provided set of standardized metadata and stateless nature of CoAP, messages are self-descriptive so that they are optimized to be processed in layered systems.
3.2 RESTful RACS

RACS (Urien, 2015) is an application layer protocol designed according to REST. The goal of RACS is to remote control Secure Elements (SE) within a Grid of Secure Elements (GoSEs); that is, a RACS server. A SE is a tamper-resistant microcontroller, which provides secure storage and cryptographic operations in smart cards. As the name RACS implies, this protocol is also intended for transporting remote Application Protocol Data Unit (APDU) (The International Organization for Standardization (ISO), 1987) messages. An APDU is an independent protocol that specifies and manages the execution of application-oriented operations on SEs. RACS is a text-based protocol using the Transmission Control Protocol (TCP) (Postel, 1981) as means of transport secured by Transport Layer Security (TLS) (Dierks and Rescorla, 2008), whereas APDU is a binary protocol protected by its own standardized security specification. As with HTTP and CoAP, RACS uses the URI syntax for identifying resources such as GoSEs or SEs. A GoSE can be addressed by an IP address and a TCP port. Each SE within a GoSE has a unique identifier Secure Element Identifier (SEID), thus an SE is accessible by invoking a request with an URI composed of the IP address, TCP port, and SEID.

Each message in RACS is composed of a set of command lines, which are separated by a carriage return and line feed. A command may contain further parameters, which are separated by a space character. The first command of each RACS message must be BEGIN and the last must be END. The BEGIN command may contain a request identifier as a parameter, which can be any kind of string. The request identifier must be echoed by the response in its BEGIN command. The RACS protocol defines the following set of request action commands: GET-VERSION, SET-VERSION, LIST, RESET, SHUTDOWN, POWERON, ECHO, and APDU. A request can comprise multiple request action commands, each of which is in its own command line. In such a case, the corresponding response must only return the status line of the last request action command. To force a server into returning a status line for a distinct request action command, APPEND must be added as the last parameter of the request action command line.

Each request action command is responded by a status header, which indicates the status line. Thus, a RACS response may contain multiple status lines, each returning the process result of its corresponding request action command. In case of a successful request processing, the status line starts with a plus (+). If a request contains an error, then the status line begins with a dash (-). The second parameter of the status line is an integer defining the status code. This parameter is followed by a number indicating the command line of the request action command being processed. A status line may contain further parameters such as a status phrase explaining the status code in a human-readable form.

Table 1 depicts some example request-response communications in RACS. The first row shows a RACS request comprising one GET-VERSION request action command. This request action command is replied by a RACS response containing a successful status code, the command line of the corresponding request action command, and the requested version number. The second example illustrates a request including a request ID and LIST request action command, and the requested version number. This request is given a response including a request ID and LIST request action command. This request is given a response containing the corresponding request ID and status line expressing the meaning and requested list of SEIDs of the GoSE. In Row 4, a request performing one RESET request action command is shown. This request action command includes the parameter WARM, which triggers a warm reboot of the SE. The last example shows a request, which executes two request action commands. Here, two APDU requests
are sent. To inform the server on returning a status line for each request action command, the APPEND parameter is added at the end of each line.

RACS also supports an HTTP interface. To perform a RACS request via an HTTP interface, a URI with the following syntax must be created:

```plaintext
https://<GoSEAddr:port>/<path>?cmd0=param0,...,<paramN>&cmdN=paramN0...,paramNM
```

The following example shows a request including one RESET request action command, which performs a warm reboot.

```plaintext
https://GoSE.org/RACS?BEGIN=myRequestID&RESET=SmartCard1,WARM&END=
```

The returned HTTP response includes a resource representation, which describes the RACS response in an XML document.

```xml
<RACS-Response>
  <begin>myRequestID</begin>
  <Cmd-Response>
    <status>+005</status>
    <line>001</line>
    <parameters>
      <parameter>SmartCard1</parameter>
      <parameter>Reset</parameter>
      <parameter>Done</parameter>
    </parameters>
  </Cmd-Response>
  <end>
</RACS-Response>
```

The current stage of the RACS draft specification does not specify which HTTP method must be used to perform a RACS request on the HTTP interface. Also, it does not specify the to-be returned HTTP status code of the HTTP response, which includes the RACS response. Moreover, the current version does not contain all required characteristics for being a RESTful

---

**TABLE 1** Example RACS Requests and Responses

<table>
<thead>
<tr>
<th>#</th>
<th>RACS Request</th>
<th>RACS Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>BEGIN GET-VERSION END</td>
<td>BEGIN +002 001 1.0 END</td>
</tr>
<tr>
<td>2</td>
<td>BEGIN myRequestID LIST END</td>
<td>BEGIN myRequestID +004 001 &lt;SEID1&gt; &lt;SEID2&gt; END</td>
</tr>
<tr>
<td>3</td>
<td>BEGIN RESET &lt;SEID&gt; WARM END</td>
<td>BEGIN +005 001 &lt;SEID&gt; Reset Done END</td>
</tr>
<tr>
<td>4</td>
<td>BEGIN APDU &lt;SEID&gt; &lt;APDU Request1&gt; APPEND APDU &lt;SEID&gt; &lt;APDU Request2&gt; APPEND END</td>
<td>BEGIN +006 001 &lt;APDU Response1&gt; +006 002 &lt;APDU Response2&gt; END</td>
</tr>
</tbody>
</table>
protocol. The RACS protocol is based on a stateless client-server in addition to a request-response model and defines a set of predefined request actions, as well as standardized further metadata. Additionally, RACS servers and SEs within RACS servers can be addressed by an URI. However, RACS does not specify any metadata for caching or transferring resource representation. RACS requests can only include APDU calls. The aforementioned missing RESTful message properties might not be required for the application domain of RACS, or they may be defined in future versions.

4 SECURITY FOR RESTful IoT PROTOCOLS

Security is one fundamental requirement when it comes to design systems in IoT. In REST-based IoT systems transport-oriented security has been established as the de facto mean to secure the message exchange. CoAP uses Datagram TLS (DTLS) (Rescorla and Modadugu, 2006) for ensuring the confidentiality and integrity of the transport layer. Also, several works have been published to optimize DTLS for the deployment in constrained networks and environments (Park and Kang, 2014; Kang et al., 2015). The RACS protocol requires TLS as the mandatory security layer over TCP.

In layered systems, such as that by the adoption of REST, transport security with DTLS or TLS alone is, however, not sufficient to cope with requirements in large-scale distributed systems (Gorski et al., 2014a). Transport-oriented security protocols can only ensure the integrity, authenticity, and confidentiality of information during transit. If a message resides in an intermediate system, the data is unprotected, leaving the surface vulnerable to man-in-the-middle attacks (Lo Iacono and Nguyen, 2015b; Nguyen and Lo Iacono, 2015).

For this purpose, a more comprehensive set of security means needs to be developed in order to support software engineers in implementing message-oriented protection mechanisms that will supplement transport security. Therefore Gorski et al. (2014a) propose a required REST-Security stack comprising security components for REST-based service systems (see Fig. 4). This stack is adopted from the mature Simple Object Access Protocol (SOAP) security domain.

Besides the required security building blocks for developing a holistic protection of REST-based systems, this stack also shows the missing and fragmented work in REST-Security. Here, only approaches for authorization and message security are available so far. Solutions for secure conversation, federation, policy, trust, and privacy are still missing. Moreover, the schemes for authorization and message security include much vulnerability, as revealed in Lo Iacono and Nguyen (2015b), Nguyen and Lo Iacono (2015), Yang and Manoharan (2013), and Sun and Beznosov, 2012, showing that despite using these technologies, man-in-the-middle attacks are still possible. This indicates that none of them are mature enough to be applied in mission- and business-critical environments. Additionally, the available approaches are only designed for HTTP or CoAP, meaning that no message security technologies exist for RACS. HTTP, CoAP, and RACS merely represent three possible protocols for implementing REST-based systems. With the increased adoption of REST in, for example, SOA, microservices, cloud computing, and IoT, more RESTful protocols are expected to evolve prospectively. This requires the development of a general REST-Security framework, which provides safeguards for all REST-based systems, including current and prospective RESTful technologies.
Based on this finding, this section proposes a methodology for defining REST-Security components, which relies on the same idea as REST itself: REST is an abstract model for designing large-scale distributed systems. This model can be adopted with suitable technologies of any kind, such as HTTP, CoAP, or RACS, to build highly scalable service systems such as the web, IoT, SOA, or cloud applications (see Fig. 5).

Following this concept, REST-Security schemes should rely on the same abstraction level as REST itself. These schemes then form a REST-Security abstract model building a set of guidelines for implementing security technologies for RESTful protocols of any kind (see Fig. 6), as REST is a guideline for designing high scalable distributed systems with RESTful technologies such as HTTP, CoAP, and RACS (see Fig. 5).
The next section proposes a REST Message Authentication (REMA) scheme, which follows this methodology. REMA marks the initial steps towards a REST message security, which build the foundation for further REST-Security components of the REST-Security stack (see Fig. 4).

5 REST MESSAGE AUTHENTICATION

A REMA must ensure the authenticity and integrity of the whole REST messages, making them immune to all kinds of man-in-the-middle-attacks. Following the introduced methodology, this section introduces an approach that augments REST by authentication scheme while remaining on the same abstraction level as REST itself. This generic scheme serves as a message authentication guideline for adopting a RESTful message authentication to distinct RESTful protocols including HTTP, CoAP, RACS, and other prospective protocols (see Fig. 7).

The key idea of REMA is to thwart the man-in-the-middle attacks revealed in Lo Iacono and Nguyen (2015b) and Nguyen and Lo Iacono (2015) by protecting the whole message at the application layer. To do so, a digital signature over all security-relevant message elements is computed. Therefore, a message signature algorithm and a message verification algorithm need to be defined. Before being able to sign and verify REST messages, a general policy defining what REST message elements to be authenticated is required. Note that this policy and this section use the abstract notation for the REST messages of Lo Iacono and Nguyen...
(2015b) and Nguyen and Lo Iacono (2015). For further details on these notations, the reader is referred to Lo Iacono and Nguyen (2015b) and Nguyen and Lo Iacono (2015).

1. A message $r \in R$ comprising a resource representation must include at least the two resource representation metadata entities, $m_{bl} \in M_{b}$ and $m_{bt} \in M_{p}$, describing the length and the media type of the contained resource representation, respectively.

2. A request $r \in R_{r}$ must contain at least one control data element, $m_{ca} \in M_{ca}$, and one resource identifier, $i \in I$, describing the action and the target of the action.

3. A response $r \in R_{s}$ must contain at least one control data element, $m_{cm} \in M_{cm}$, expressing the meaning of the response.

4. A read request must contain at least one resource representation metadata element, $m_{br} \in M_{b}$, describing the desired media type being requested. Moreover, this request must not include a resource representation.

5. A creation request must contain a resource representation.

6. An update request must contain a complete or partial resource representation.

7. A delete request does not require any additional prerequisite header element until further requirements. Moreover, this request must not include a resource representation.

Based on these definitions, this section proposes a generic signature and verification algorithm for REST messages. Further requirements are the matter of technical instantiation of the uniform interface and application domain.

5.1 REST Message Signature

Algorithm 1 defines the procedure for signing a REST message. Note that error conditions are not made because of readability reasons. This algorithm requires a REST message $r$, a description on application-specific, to-be signed header elements $desc$ and a signature generation key $k$ as input.

**Algorithm 1** REST MESSAGE SIGNATURE (LO IACONO AND NGUYEN, 2015B)

**Input:** REST message $r$, description desc of the application-specific, to-be signed header entries, signature generation key $k$

**Output:** Signature value $sv$, time-variant parameter $tvp$

1: $b \leftarrow \text{getBody}(r)$
2: $h \leftarrow \text{getHeader}(r)$
3: $\overset{\sim}{h} \leftarrow \text{getTbsHeaders}(h)$
4: $\overset{\sim}{h} \leftarrow \overset{\sim}{h} \ || \ \text{getTbsHeaders}(h, \text{desc})$
5: $tvp \leftarrow \text{generateTimeVariantParameter}()$
6: $tbs \leftarrow tvp$
7: $i \leftarrow 0$
8: while $i < | \overset{\sim}{h} |$ do
9: $tbs \leftarrow tbs \ || \ \text{delimiter} \ || \ \text{normalize}(\overset{\sim}{h} )$
10: $i \leftarrow i + 1$
11: end while
12: $tbs \leftarrow tbs \ || \ \text{delimiter} \ || \ \text{hash}(b)$
13: $sv \leftarrow \text{sign}(k, tbs)$
The first two statements extract the body $b$ and the header $h$ from the message $r$. The next step gathers security-relevant header entries from $h$ and store them to $\tilde{h}$, which represents the header containing the security-critical header elements defined by the policy. After that, the application-specific, security-relevant header entities defined by $desc$ are attached to $\tilde{h}$. In order to thwart replay attacks, the following step generates a time-variant parameter $tvp$ and assigns it to the variable $tbs$. These two steps must not be omitted even when another time-variant is already available in $\tilde{h}$ because a considerable time span might exist between a message generation and signature generation. In the next step, all security-relevant header entries in $\tilde{h}$ are normalized and concatenated to $tbs$. Next, a cryptographic hash of the body $b$ is computed and appended to $tbs$. The final step signs $tbs$ with the signature generation key $k$. The algorithm then returns the signature value $sv$ and time-variant parameter $tvp$ as output.

To guide the receiver in verifying a signed REST message, an authentication control data $m_{cpa} \in M_{cp}$ must be created, which contains the signature algorithm name $sig$, the hash algorithm $hash$, a key id $kid$, the time-variant parameter $tvp$, the signature value $sv$, and the description on application-specific header elements $desc$. After the creation of $m_{cpa}$, this metadata is included to the header $h$.

## 5.2 REST Message Verification

Algorithm 2 describes the process for verifying REST messages, which are signed by Algorithm 1. Note that error conditions are not made due to readability reasons; this algorithm requires a signed REST message $r$ as input only. As with Algorithm 1, the first two steps of Algorithm 2 extract the body $b$ and header $h$ from the message $r$. After that $m_{cpa}$, which contains all the information guiding the receiver in verifying the message, can be obtained from $h$. The statement in line 5 constructs $\tilde{h}$ according to the policy. The next step appends

### ALGORITHM 2  REST MESSAGE SIGNATURE VERIFICATION

**Input:** Signed REST message $r$

**Output:** Boolean signature verification result $valid$

1: $b \leftarrow$ getBody($r$)
2: $h \leftarrow$ getHeader($r$)
3: $m_{cpa} \leftarrow$ getAuthenticationControlData($h$)
4: $(sig, hash, kid, tvp, sv, desc) \leftarrow$ split($m_{cpa}$)
5: $\tilde{h} \leftarrow$ getTbsHeaders($h$)
6: $\tilde{h} \leftarrow \tilde{h} \ || \ getTbsHeaders(h, desc)$
7: $tbs \leftarrow tvp$
8: $i \leftarrow 0$
9: **while** $i < | \tilde{h} |$ **do**
10: \[ tbs \leftarrow tbs \ || \ delimiter \ || \ normalize( \tilde{h} ) \]
11: \[ i \leftarrow i + 1 \]
12: **end while**
13: $tbs \leftarrow tbs \ || \ delimiter \ || \ hash(b)$
14: $verify \leftarrow$ getVerificationAlgorithm($sig$)
15: $valid \leftarrow$ verify($kid, tbs, sv$)
the application-specific header entries specified by desc to \( h \). After building \( h \), tvp, all elements in \( h \) and a cryptographic of \( b \) are concatenated in the same order and manner as in Algorithm 1 to \( tbs \). With kid, \( tbs \), and sv, the signed message \( r \) can be verified. The result of this verification process is then returned as output.

## 6 RESTful IoT MESSAGE AUTHENTICATION

This section adopts the proposed authentication scheme to CoAP and RACS to show how it is implemented in concrete RESTful IoT technologies. Further instantiation to prospective RESTful IoT protocols can likewise be conducted.

### 6.1 RESTful CoAP Message Authentication (RECMA)

The two templates of the following table show the instantiation of the concatenation process of Algorithm 1 for building \( tbs \). The left template describes construction rules for CoAP request and responses, where a time-variant parameter (tvp), all security-relevant header elements (\( h \)), and the body (\( b \)) are concatenated to byte array. The right one defines the concatenation process for acknowledgment (ACK) and reset messages (RST). Here, only tvp and \( h \) are concatenated to a byte array as both message types do not contain a message body. The adopted and extended policy for constructing \( h \) in CoAP is described in Nguyen and Lo Iacono (2015); the reader is referred to this paper for further reference.

<table>
<thead>
<tr>
<th>tbs Constructing Template for CoAP request and responses</th>
<th>tbs Constructing Template for CoAP ACK and RST</th>
</tr>
</thead>
</table>
| Tvp  
  ||Version  
  ||Type  
  ||TokenLength  
  ||Code  
  ||MessageID  
  ||Token  
  ||Options0  
  …  
  ||OptionsN  
  ||hash(body) | tvp  
  ||Version  
  ||Type  
  ||TokenLength  
  ||Code  
  ||MessageID |

Assuming that the following request and acknowledgement message require to be signed:

<table>
<thead>
<tr>
<th>Example CoAP request</th>
<th>Example CoAP ACK</th>
</tr>
</thead>
</table>
| POST (V=1,T=0,TKL=1,C=0.02,MID=1)  
  Token: 10  
  Uri-Path: “items”  
  Content-Format: 60  
  Payload-Length: 15  
  11111111  
  {“items””:“pork”} | ACK (V=0,T=2,TKL=0,C=0.00,MID=1)  
  11111111 |
According to the policy of Nguyen and Lo Iacono (2015), the tbs of both messages is constructed as follows:

<table>
<thead>
<tr>
<th>tbs of Example CoAP request</th>
<th>tbs Constructing Template for CoAP ACK and RST</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x14D14486B51 #tvp</td>
<td></td>
</tr>
<tr>
<td>0x01 #Version</td>
<td></td>
</tr>
<tr>
<td>0x00 #Type</td>
<td></td>
</tr>
<tr>
<td>0x01 #TokenLength</td>
<td></td>
</tr>
<tr>
<td>0x02 #Code</td>
<td></td>
</tr>
<tr>
<td>0x01 #Message-ID</td>
<td></td>
</tr>
<tr>
<td>0x0A #Token</td>
<td></td>
</tr>
<tr>
<td>0x00 #Uri-Host(3)</td>
<td></td>
</tr>
<tr>
<td>0x00 #Uri-Port(7)</td>
<td></td>
</tr>
<tr>
<td>hash(UTF8(“items”)) #Uri-Path(11)</td>
<td></td>
</tr>
<tr>
<td>0x60 #Content-Format(12)</td>
<td></td>
</tr>
<tr>
<td>0x00 #Max-Age(14)</td>
<td></td>
</tr>
<tr>
<td>0x00 #Uri-Query(15)</td>
<td></td>
</tr>
<tr>
<td>0x0F0 #Payload-Length (65001)</td>
<td></td>
</tr>
<tr>
<td>hash(UTF8(“item”::“pork”)) #Body</td>
<td></td>
</tr>
</tbody>
</table>

The concatenation order of the items of the CoAP start header follows the order of the predefined positions of these header entries. The CoAP are appended according the order of option numbers. After constructing both tbs, these two variables are signed by signature generation key $k$.

$$sv = sign(k, tbs)$$

The last step assigns the computed signature value $sv$ and corresponding authentication metadata to newly introduced CoAP options: Signature-Value ($sv$), Signature-Algorithm ($sig$), Hash-Algorithm ($hash$), TVP ($tvp$), and Key-ID ($kid$). These options represent $m_{cpu}$ and are included with the header $h$.

<table>
<thead>
<tr>
<th>Signed CoAP request</th>
<th>Signed CoAP ACK</th>
</tr>
</thead>
<tbody>
<tr>
<td>POST (V=1,T=0,TKL=1,C=0.02,MID=1)</td>
<td>ACK (V=0,T=2,TKL=0,C=0.00,MID=1)</td>
</tr>
<tr>
<td>Token: 10</td>
<td>11111111</td>
</tr>
<tr>
<td>Uri-Path: “items”</td>
<td>Signature-Algorithm: 1</td>
</tr>
<tr>
<td>Content-Format: 60</td>
<td>Hash-Algorithm: 1</td>
</tr>
<tr>
<td>Payload-Length: 15</td>
<td>TVP:</td>
</tr>
<tr>
<td>Signature-Algorithnm: 1</td>
<td>Signature-Value: &lt;sv&gt;</td>
</tr>
<tr>
<td>Hash-Algorithnm: 1</td>
<td>Key-ID: &lt;kid&gt;</td>
</tr>
<tr>
<td>TVP:</td>
<td></td>
</tr>
<tr>
<td>Signature-Value: &lt;sv&gt;</td>
<td></td>
</tr>
<tr>
<td>Key-ID: &lt;kid&gt;</td>
<td></td>
</tr>
<tr>
<td>11111111</td>
<td></td>
</tr>
<tr>
<td>(&quot;items&quot;:&quot;pork&quot;)</td>
<td></td>
</tr>
</tbody>
</table>
This CoAP implementation of REMA uses the numbers for declaring the signature and hash algorithm name. The number 1 within the Signature-Algorithm options stands for an HMAC-SHA256. The same number in the Hash-Algorithm option represents an SHA256. The description on application-specific header entries desc is omitted in these examples, because ACK and RST messages must not contain CoAP options and the example request does intend to include application-specific options to $h$. Moreover, this RECMA utilizes the Payload-Length option to define the size of the body. This option is not standardized metadata element; rather, it is draft specification, which has been expired yet. Still, RECMA uses this option and declares this metadata entry as an element of $h$ in order to both avoid man-in-the-middle attacks manipulating the body and to comply with transport independence constraint (Fielding, 2000; Nguyen and Lo Iacono, 2015).

### 6.2 RESTful RACS Message Authentication (RERMA)

The following table shows two templates for authenticating RACS messages. In contrast to CoAP, the RERMA utilizes a string concatenation instead of a byte concatenation, as it is a text-based protocol.

<table>
<thead>
<tr>
<th>$tbs$ Constructing Template for RACS request</th>
<th>$tbs$ Constructing Template for RACS request</th>
</tr>
</thead>
<tbody>
<tr>
<td>$tvp + &quot;\n&quot; +$</td>
<td>$tvp + &quot;\n&quot; +$</td>
</tr>
<tr>
<td>$\text{rid} + &quot;\n&quot; +$</td>
<td>$\text{rid} + &quot;\n&quot; +$</td>
</tr>
<tr>
<td>$a0 + &quot; + p0 + &quot; + \ldots + pN + &quot;\n&quot; +$</td>
<td>$s0 + &quot; + cl0 + &quot; + p0 + &quot; + \ldots + pN + &quot;\n&quot; +$</td>
</tr>
<tr>
<td>$\ldots$</td>
<td>$\ldots$</td>
</tr>
<tr>
<td>$aM + &quot; + pM0 + &quot; + \ldots + pMN + &quot;\n&quot; +$</td>
<td>$sM + &quot; + clM + &quot; + p0 + &quot; + \ldots + pMN + &quot;\n&quot; +$</td>
</tr>
</tbody>
</table>

The left template describes the concatenation process for requests. According to **Algorithm 1**, the first parameter assigned to $tbs$ is the time-variant parameter ($tvp$), followed by line break (\n). Next, the request ID (rid) with a line break is appended. If the request does not include a request ID, an empty string must be added instead. After that, each command line containing a request action command separated by a line break is appended to $tbs$. For each command line, the request action command (a) must be added first, then the parameters of the corresponding request action command (p) are appended. Each parameter must be separated by a whitespace. The right template denotes the concatenation process for RACS response. $tvp$ and the request ID are added in same manner as in the left template. The status code (sc) followed by the processed command line of corresponding request action command (cl) and parameters are appended next. As the current stage of RACS does not define how to transfer and declare resource representation, this part is omitted in the current stage of RERMA. Further work will extend RERMA by this missing property, if transferring and declaring resource representation is defined in RACS.
Assuming that the following two RACS messages require authentication:

<table>
<thead>
<tr>
<th>Example RACS request</th>
<th>Example RACS response</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN</td>
<td>BEGIN</td>
</tr>
<tr>
<td>APDU SmartCard1 &lt;APDU Request1&gt; APPEND</td>
<td>+006 001 &lt;APDU Response1&gt;</td>
</tr>
<tr>
<td>APDU SmartCard1 &lt;APDU Request2&gt; APPEND</td>
<td>+006 002 &lt;APDU Response2&gt;</td>
</tr>
<tr>
<td>END</td>
<td>END</td>
</tr>
</tbody>
</table>

Based on Algorithm 1 and the templates in the previous table, the \( tbs \) of both messages is constructed as follows:

<table>
<thead>
<tr>
<th>( tbs ) of Example RACS request</th>
<th>( tbs ) of Example RACS response</th>
</tr>
</thead>
<tbody>
<tr>
<td>1455190341456</td>
<td>1455190341556</td>
</tr>
<tr>
<td>APDU SmartCard1 Base64(&lt;APDU Request1&gt;) APPEND</td>
<td>+006 001 Base64(&lt;APDU Response1&gt;)</td>
</tr>
<tr>
<td>APDU SmartCard1 Base64(&lt;APDU Request2&gt;) APPEND</td>
<td>+006 002 Base64(&lt;APDU Response2&gt;)</td>
</tr>
</tbody>
</table>

Then both strings are encoded to UTF8 and signed with a key \( k \). As the RACS is a text-based protocol, the resulting binary signature value must be converted by a Base64 transformation to a string:

\[
sv = \text{Base64} \left( \text{sign} \left( k, \text{UTF8} \left( tbs \right) \right) \right)
\]

Finally, \( sv \), along with the corresponding authentication metadata \( m_{cap} \) is added to the request as a new command line, starting with the newly defined command action SIGNATURE. Note that this command action is considered experimental, as it does not exist in the current RACS specification. Therefore the representation of \( m_{cap} \) may change in the future in order to be compliant with forthcoming versions of the RACS draft specification. In the RACS response, \( m_{cap} \) is included into the status line, which is in conformance with the RACS specification, as a status header can contain additional response parameters.

<table>
<thead>
<tr>
<th>Signed Example RACS request</th>
<th>Signed Example RACS response</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN</td>
<td>BEGIN</td>
</tr>
<tr>
<td>APDU SmartCard1 &lt;APDU Request1&gt; APPEND</td>
<td>+006 001 &lt;APDU Response1&gt;</td>
</tr>
<tr>
<td>APDU SmartCard1 &lt;APDU Request2&gt; APPEND</td>
<td>+006 002 &lt;APDU Response2&gt;</td>
</tr>
<tr>
<td>SIGNATURE RSA/SHA256 &lt;kid&gt; 1455190341456 null &lt;sv&gt;</td>
<td>+006 003 SIGNATURE RSA/SHA256 &lt;kid&gt; ↓</td>
</tr>
<tr>
<td>END</td>
<td>1455190341556 null &lt;sv&gt;</td>
</tr>
<tr>
<td>END</td>
<td>END</td>
</tr>
</tbody>
</table>

Both messages do not include application-specific metadata elements to be signed. Therefore, the fifth parameter of the SIGNATURE command line is defined as null. If additional header elements need to be signed, a list containing the position number of the parameter separated by a comma must be included instead.
If the HTTP interface is utilized to perform a RACS request, the RESTful HTTP Message Authentication (REHMA) (Lo Iacono and Nguyen, 2015b) must be used to authenticate the HTTP messages.

### 7 CONCLUSION AND OUTLOOK

REST has established itself as an important architectural style for developing large-scale hypermedia distributed systems. In IoT environments, the principles and constraints of REST have been adopted by several application domains including CoAP- and RACS-based systems. Other IoT areas with prospective RESTful protocols will eventually arise likewise. The increasing implementation of the REST concept in various technologies, as well as application domains, and the insufficient protection of transport-oriented protection demand generic security approaches augmenting REST on the same abstraction layer.

This chapter therefore proposes an approach that extends REST by an authentication scheme while remaining on the same abstraction layer of REST itself. This security scheme then serves as a guideline for implementing message authentication for RESTful (IoT) protocols. Based on this guideline, this chapter introduces a REST message authentication scheme for two RESTful protocols for the IoT domain, CoAP and RACS, respectively.

REMA, RECMA, and RERMA provide integrity and authenticity, as well as non-repudiation for REST messages and RESTful protocol when using asymmetric signature algorithms in conjunction with an appropriate public key infrastructure. Still, in order to approach a comprehensive message security, confidentiality must be considered as well. In layered systems, this security service is of specific importance, as many intermediate systems, such as cache servers, load balancers or content delivery networks are operated by third-party services. If REST messages are not encrypted, those intermediate services have plain-text access to traversing messages. This is especially critical for IoT environments, as sensitive information is transferred from node to node. Therefore a REST message confidentiality scheme needs to be developed. This scheme must follow the introduced methodology by defining a guideline for adopting and implementing confidentiality services for RESTful technologies including HTTP, CoAP, RACS, and prospective RESTful protocols (see Fig. 8).

![Diagram](image-url)  
FIG. 8 General RESTful message confidentiality and its instantiation to concrete RESTful protocols.
Following this methodology, further security components for the REST-Security stack in Fig. 4 can be developed. All these steps will be elaborated in further work in order to build a generic and robust security framework for mission-critical REST-based (IoT) systems.

References


