future internet

Article

On the Need for a General REST-Security Framework

Luigi Lo Iacono *, Hoai Viet Nguyen and Peter Leo Gorski

Data and Application Security Group, Cologne University of Applied Sciences, 50679 Cologne, Germany; viet.nguyen@th-koeln.de (H.V.N.); peter.gorski@th-koeln.de (P.L.G.)
* Correspondence: luigi.lo_iacono@th-koeln.de

Received: 19 December 2018; Accepted: 14 February 2019; Published: 27 February 2019

Abstract: Contemporary software is inherently distributed. The principles guiding the design of such software have been mainly manifested by the service-oriented architecture (SOA) concept. In a SOA, applications are orchestrated by software services generally operated by distinct entities. Due to the latter fact, service security has been of importance in such systems ever since. A dominant protocol for implementing SOA-based systems is SOAP, which comes with a well-elaborated security framework. As an alternative to SOAP, the architectural style representational state transfer (REST) is gaining traction as a simple, lightweight and flexible guideline for designing distributed service systems that scale at large. This paper starts by introducing the basic constraints representing REST. Based on these foundations, the focus is afterwards drawn on the security needs of REST-based service systems. The limitations of transport-oriented protection means are emphasized and the demand for specific message-oriented safeguards is assessed. The paper then reviews the current activities in respect to REST-security and finds that the available schemes are mostly HTTP-centered and very heterogeneous. More importantly, all of the analyzed schemes contain vulnerabilities. The paper contributes a methodology on how to establish REST-security as a general security framework for protecting REST-based service systems of any kind by consistent and comprehensive protection means. First adoptions of the introduced approach are presented in relation to REST message authentication with instantiations for REST-ful HTTP (web/cloud services) and REST-ful constraint application protocol (CoAP) (internet of things (IoT) services).

Keywords: SOA; services; security; REST; web services security; HTTP; IoT services security; CoAP; RACS

1. Introduction

Representational state transfer (REST) [1] is an architectural style for designing distributed services systems that scale at large. This is achieved by a set of defined architectural constraints. REST-based systems have to be, e.g., stateless and cacheable in order to ensure the propagated scalability. The uniform interface is another important constraint, which provides simplicity of interfaces and performance of components' interaction. The benefits coming along by adhering to these constraints are amongst the main driving forces for the increasing adoption of service systems based on REST.

Currently only a limited set of technologies exists, which can serve as a foundation for implementing REST-based systems. HTTP [2] is by far the most dominant choice. This fact is the source for many misinterpretations in which REST is often equated with HTTP. Consequences emerging from this reasoning are many-fold. One related to security is the adoption of transport-oriented protection only, as common for conventional web-based applications by means of transport layer security (TLS) [3]. This is by far not sufficient as an exclusive safeguard for REST-based services, since they are constrained to be layered. Hence, these systems consist of intermediaries, which perform functions on the data path between a source host and destination host, most commonly on the open systems interconnection (OSI) application layer [4]. Examples of such intermediate systems include caches, load-balancers, message routers, interceptors and proxies. In order to be able to perform their tasks, intermediate systems need to terminate transport security, which as a result does not reach from end to end. This remains opaque to the user and the obtained security level depends on many more stakeholders than the two endpoints. [5] revealed that many current security interceptors struggle with the implementation of transport-oriented security protocols, as they build intermediate systems that decrease security or even provide implementations that are severely broken. Also, transport-oriented security is not designed to fulfil the security requirements of ultra large scale (ULS) [6] systems and distributed service-oriented applications in general. The various entities involved in chained processing steps require adopting more fine-grained and message-related security means such as partial encryption and signature as, e.g., provided by the web services (WS)-security [7] standard for simple object access protocol (SOAP) based web services [8].

Moreover, different protocols following the REST principles are starting to emerge in domains other than the web or the cloud. For implementing internet of things (IoT) services, for instance, constraint application protocol (CoAP) [9] is taking root as REST-compliant protocol. Datagram transport layer security (DTLS) [10], the user datagram protocol (UDP) based flavor of TLS, is applicable as transport-oriented security measure here likewise. Again, the REST inherent constraint of composing systems out of layers, in many cases prohibits the adoption of transport-oriented security as single line of protection. Especially in the IoT domain, most of the use cases comprise high security demands, asking for more elaborated and pluralistic safeguards.

The limited protection of transport-oriented security in REST-based systems has already been addressed by several research and development approaches as will be discussed thoroughly in Section 5. From these, some REST message security technologies have emerged that can be used in conjunction with transport security. Still, these approaches are available for certain technologies only, mainly HTTP and CoAP until now. As REST defines an abstract concept, its implementations are not restricted to these two particular technologies, though. Since REST has been established as an important paradigm for building large-scale distributed systems, more REST-ful protocols are expected to evolve prospectively. The remote application protocol data unit (APDU) call secure (RACS) [11] protocol is one example. It is an emerging REST-ful protocol for accessing smartcards. Beside transport-oriented security no further protection means have been proposed for the RACS draft standard so far.

From these basic observations, the need for a general REST-security framework becomes apparent. The objective of this paper is to close this gap while providing the following contributions. First, this paper analyzes the actual security demands of REST-based systems thoroughly and emphasizes the specific REST characteristics that necessitate a dedicated REST-security, which needs to be defined at the same abstraction layer as REST itself and independent from any concrete technology in the first place. Then, a comprehensive consolidated review of the current state of the art in respect to REST-security is provided. Finally, the paper contributes a general REST-security framework alongside with a methodology on how to instantiate it for a particular REST-conformance technology stack in order to facilitate the protection of REST-based service systems of any kind by consistent and comprehensive protection means. Available as well as upcoming REST-ful technologies will benefit from the introduced methodology and the proposed general REST-security framework at its core.

For this purpose, the remaining of this paper is organized as follows. The foundations in respect to the architectural style REST are laid in Section 2. The methodology for deriving the envisioned general REST-security framework is laid in Section 3. The subsequent sections follow this methodology accordingly, starting with capturing the demand in terms of required service security technologies in Section 4. Due to the lack of a widespread adoption of REST other than the Web—but without the loss of generality—the security demands and specific requirements are analyzed based on the Web Services security stack. In Section 5 the related work and current practice is presented and assessed. A general security framework that reflects the particular characteristics and properties of REST is introduced in Section 6. Based on this approach, Section 7 proposes an adoption of the framework to two prevalent concrete REST-based protocols, HTTP and CoAP. An experimental evaluation of these schemes against the related work based on prototype test-beds is given in Section 8. The paper concludes in Section 9 and provides a brief discussion on future research and development demands.

2. REST Foundations

Besides the dissertation of Roy Fielding [1], there neither exists a definition nor an unified understanding of the term REST and its underlying principles and concepts. Often enough it is mistaken as being a standard composed of its underlying foundations HTTP and uniform resource identifier (URI) [12]. The source for this diffuse view on REST lies mainly in the fact that the two aforementioned standards have been the only notable technology choice for implementing REST-based service systems ever since. For the purpose of this paper it is henceforth demanding that the term REST is defined unambiguously.

The aim of REST is to provide a guideline for designing distributed systems that possess certain traits including performance, scalability, and simplicity. These architectural properties are realized by applying specific constraints to components, interfaces, and data elements. These constraints are subsequently introduced with the guidance of Figure 1.

REST is constrained to the client-server model in conjunction with the request-response communication flow. A REST client performs some kind of action on a targeted resource by issuing a request. For this, the request must contain a resource identifier and the action to apply to the addressed resource. Depending on the action, the request and response messages may contain additional meta-data elements, which are categorized in resource data, resource meta-data, representation data, representation meta-data, and control data. The set of available actions in conjunction with a unique scheme for identifying resources as well as the additional meta-data is known as the uniform interface since it is consistent for all managed and provided resources.

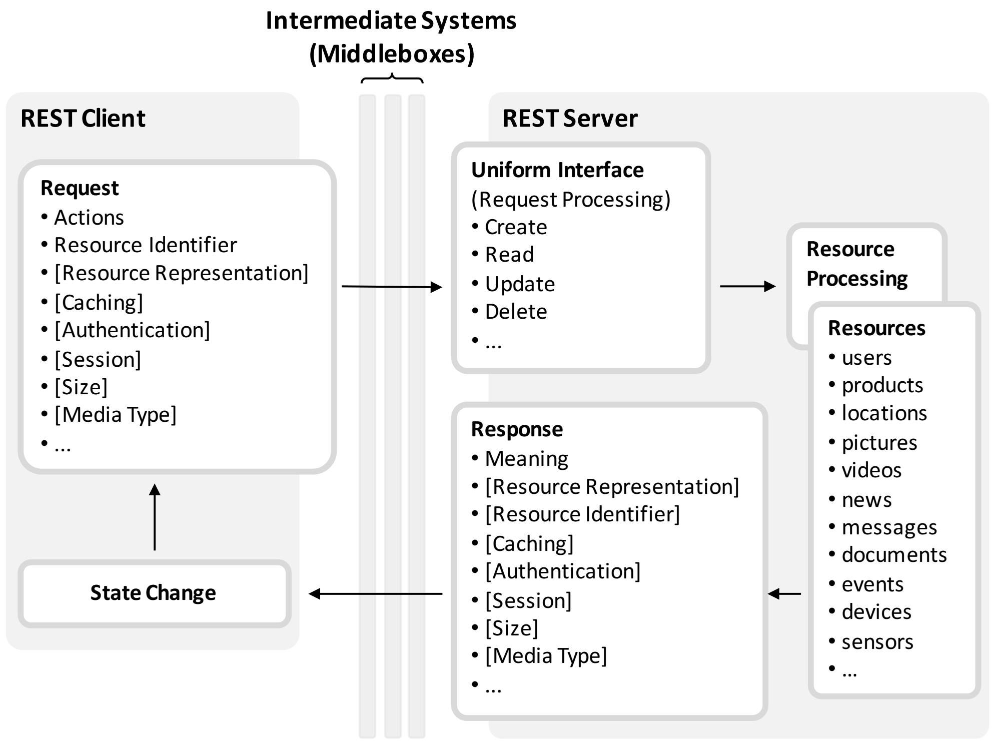

Figure 1. Overview of the representational state transfer (REST) constraints and principles (on the basis of [13]).

Since REST-based systems are constrained to be stateless, messages need to contain all required data elements in order to relieve the server from maintaining state for each client. As REST messages embody all required data elements, which are predefined and standardized by the uniform interface, their semantics are visible and are hence self-descriptive for all intermediaries and endpoints so that all components in a REST architecture can understand the intention of a message without knowing each other in advance. In a request to read access a resource, for instance, the request contains the resource identifier along with representation meta-data to signal in what data format the resource should be delivered from the server to the client. Moreover, a request can include further meta-data required by intermediaries including state and caching information. The according response provides information on its meaning and in case it denotes that the addressed resource is available, it is contained in the message body in the requested representation. Once a response is received, it transfers the receiving client into a new state. In another setting, in which the request triggers the creation of a new resource, the request contains the resource in the request message body in some representation. The response then gives feedback on the resource state and whether it has successfully created or not. Again, further meta-data elements can be included in addition, providing information on the authentication, the session and the freshness of a resource in respect to caching. Moreover, the meta-data elements as well as the resource representation of the response may contain further resource identifiers—i.e., hyperlinks. Based on these resource identifiers and their description, a client is able explore other resources and transfer its state by starting a new request with a distinct resource identifier and meta information. This REST property is known as hypermedia as the engine of application state (HATEOAS).

The principles and constraints representing REST are fairly abstract making it adoptable in any environment that contains technologies suitable for implementing the REST constraints. This coherence is illustrated by Figure 2. HTTP is one protocol that is in conformance with the REST constraints and principles as it is based on the client-server model and the interaction is stateless. Moreover, it specifies a uniform interface, which specifies a set of predefined request actions, i.e., the HTTP methods, and a set of additional meta-data for transferring different resource representation or controlling the cache behavior for example. Additionally, HTTP uses a resource identifier syntax, i.e., the URI [12] standard, for addressing resources. An instantiation on the technological basis of HTTP results in REST-ful HTTP [14], the foundation for building REST-based web, micro or cloud services, which in turn are used to build smart-* and industry 4.0 applications. More specifically, the fifth generation of mobile communication systems (5G), e.g., adopts REST-ful HTTP for implementing a service-based architecture (SBA) providing core network functions as REST-ful services [15]. Another evolving application domain of REST can be found in the IoT [16]. Here, the REST-conformance CoAP [9] is used to implement distributed service systems consisting of a large number of resource-restricted nodes [17]. CoAP adopts most of the HTTP characteristics. It utilizes the same request actions and the URI standard for specifying the uniform interface. Also, CoAP defines similar meta-data for transferring and controlling the cache behavior. The main difference between CoAP and HTTP lies in the fact that CoAP is a binary protocol, whereas HTTP is text-based. Other technical instantiations of REST are equally possible and might appear in the future such as the remote APDU call secure (RACS) [11] protocol, which is still being standardized. This abstraction hierarchy is an important fact to consider carefully when researching on REST or REST-security.

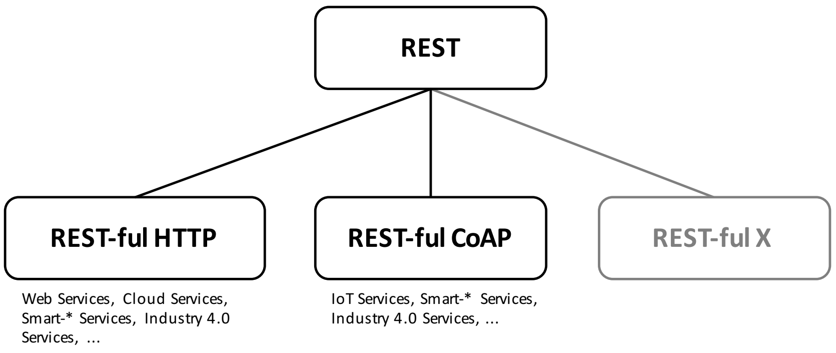
REST
REST-ful HTTP
REST-ful CoAP
REST-ful X
Web Services, Cloud Services, Smart-* Services, Industry 4.0 Services, ...
IoT Services, Smart-* Services, Industry 4.0 Services, ...

Figure 2. Instantiation of the general REST architecture style to specific REST-ful protocols.

3. Methodology

To derive a general framework for REST-security, the methodology depicted in Figure 3 has been applied. In a first phase (see Section 4 for details), the specific needs of REST-based systems in terms of security have been derived by analyzing available standards and academic work in related domains and contrasting them with characteristics of REST-compliant systems. Moreover, a common and realistic threat model is defined and used as a basis for the subsequent phases.

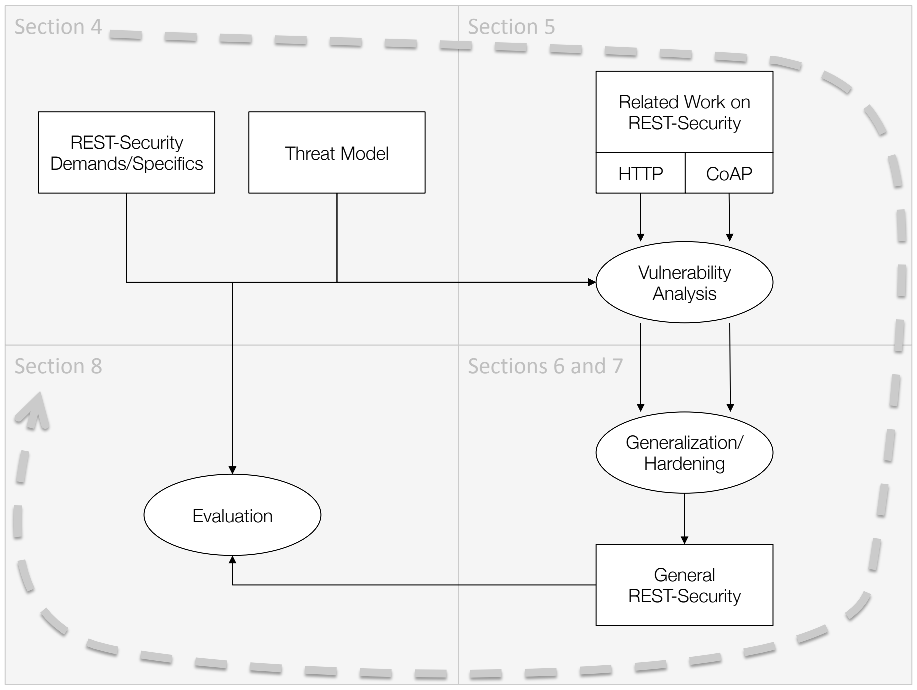

Figure 3. Adopted methodology to derive a general framework for REST-security.

To obtain an in-depth understanding on how REST messages are protected by available means, a comprehensive study of schemes introduced in literature as well as deployed in practice has been executed (see Section 5 for details). Twenty-one approaches have been identified in total and all of them have been evaluated in respect to the specific security demands of REST-based systems and the determined threat model.

As none of the analyzed REST message security schemes fulfil all the necessary requirements and is free of vulnerabilities in the given threat model, a new approach to REST-security has been developed in an adjacent activity (see Sections 6 and 7). Governed by the main outcomes of the previous studies, the generalization as well as hardening of the proposed schemes have been the goal. To be able to get a proof-of-concept, particular entities of the general REST-security framework have been instantiated. As most of the available related work is focusing on REST message authentication, the implemented instantiations of our framework do so as well for HTTP and CoAP.

To evaluate the derived and introduced general REST-security framework and more specifically its particular instantiations have been examined in experimental test-beds using prototypes (see Section 8 for details). For this purpose, implementations of the related schemes—as far as openly accessible—have been integrated in experimental test environments. For comprehensibility reasons the source code of the introduced scheme REST-ful HTTP message authentication (REHMA) and REST-ful CoAP message authentication (RECMA) have been published and made available in the public domain.

4. REST-Security Demands and Specifics

When considering REST for the design of service systems of any kind, the general security demands of service-oriented architecture (SOA) [18] apply. The ability of REST-based systems to also comply with the SOA principles has been analyzed and shown in [19]. As SOAP-based web services have been and still are a dominant technology stack for implementing SOA-based systems, the evolved security stack for SOAP-based web services can serve as reference [13].

4.1. SOAP-Based Web Services Security Stack

The SOAP-based Web Services technology stack includes an extensive set of security standards (see Figure 4) [13].

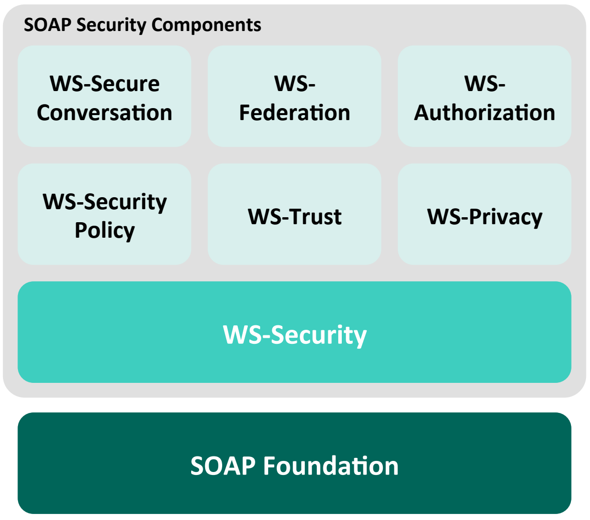

Figure 4. Security stack for SOAP-based web services [13].

SOAP uses XML [20] as a platform-independent and extensible data description language for defining the structure and semantics of the protocol messages. To ensure basic security services for SOAP messages such as confidentiality and integrity, the WS-security [7] specification has been standardized, which is based on XML encryption [21] and XML signature [22]. Upon these foundations, further-reaching security concepts are provided. The fundamental condition for any security systems is trust. WS-trust [23] introduces a standard based upon WS-security for establishing and broking trust relationships between service endpoints. WS-federation [24] extends WS-trust in order to federate heterogeneous security realms. It provides authorization management across organizational and trust boundaries. The authorization management within those realms is described in WS-authorization. Privacy constraints are covered by the WS-privacy specification. It allows handling privacy preferences and policies between client and server. Secure communication, trust, federation, authorization and privacy need a mechanism to negotiate and handle security policies. WS-security policy [25] specifies how constraints and requirements in terms of security are defined for SOAP messages. It is a framework which allows Web Services to express their security demands as a set of so-called policy assertions. WS-secure conversation [26] expands the security mechanisms for a conversation between two communication partners. This organization for the advancement of structured information standards (OASIS) standard defines how a secure exchange of multiple messages has to be established in terms of a session [27].

4.2. REST-Ful Services Security Stack

REST-based services require a comparable set of technologies in order to enable developers to implement message-oriented security mechanisms as required by the surrounding application context. The currently available security stack is, however, rather scarce in comparison to the SOAP-based web services security stack (see Figure 5) [13].

Even the fundamental message security layer is not available completely (visualized by the dashed area) [13,28]. Some standards related to the authorization of service invocations such as OAuth [29] and drafts on identity federation [30] are at hand, but the rest of the higher order security concepts including trust, secure conversation and so forth are lacking entirely. Still, the depicted security stack for REST-based services is a necessity and thus needs to be developed.

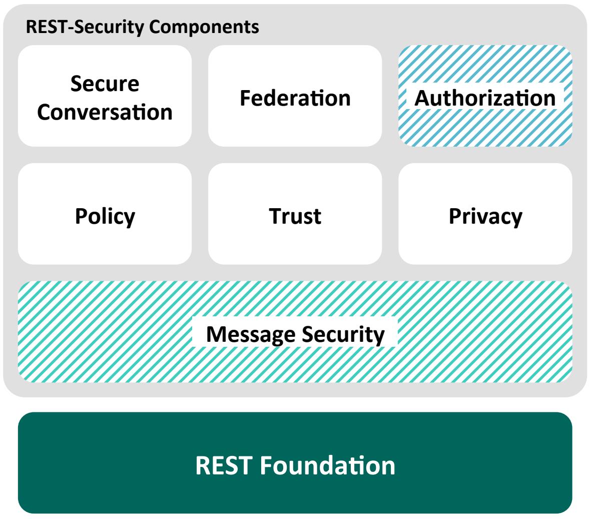

Figure 5. Desired security stack for REST-based web services [13].

4.3. REST-Security Specifics

Although both security stacks have their similarities, the plain adoption of WS-security to REST and instantiations of REST is not feasible in a straightforward manner. The specifics of REST have to be considered carefully in order to obtain a suitable and seamless security for REST-based services.

What needs to be taken into account first is the abstraction layer of REST and its instantiations. REST itself is a very general concept and needs to be handled accordingly. Thus, a simple mapping of the concrete WS-Security technologies to construct REST-security is not feasible, since both reside on different abstraction layers. Since REST represents an abstract model, security components for this architectural style need to be considered and defined on the same abstraction layer as well. Consequently, REST-security needs to be a general framework composed of definitions, structures and rules on how to protect REST-based systems. The term general in this context has to be understood as generic in the sense that the schemes contained in the REST-security framework are not bound to a specific REST-based technology or protocol only, but are applicable to any REST-ful technology. Such a general REST-security framework would then support a guided adoption and implementation to any concrete REST-ful protocol (see Figure 6).

Another REST specific issue is that there is no self-contained REST message, but the relevant data items are scattered around the service protocol and the service payload (see Figure 7). SOAP messages, in contrast, are a self-contained XML structure. Both the meta-data as well as the payload in form of a service operation or its corresponding result are enclosed in one XML document. Thus, with the application of security mechanisms based on the technologies shown in Figure 4, both message parts can be covered. This is, however, not the case for REST messages. Referring again to REST-ful HTTP as an example, the meta-data is included in the HTTP header, whereas the resource representation is inside the HTTP body. Since both parts are disjoint for many reasons, distinct security mechanisms need to be applied in a balanced manner. If this is not being recognized, novel vulnerabilities might be exploitable in the future.

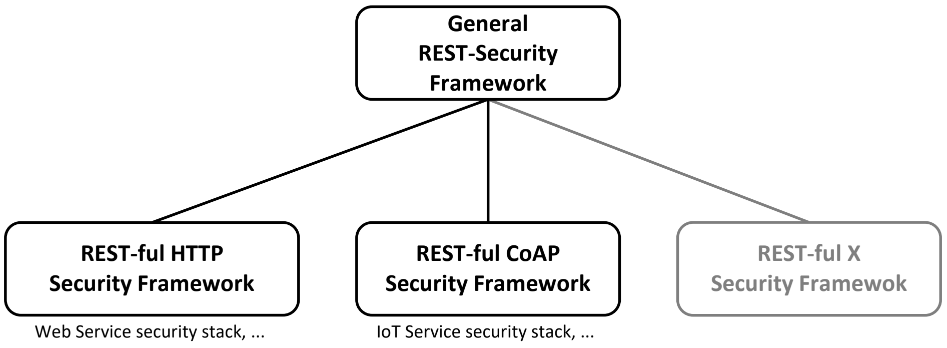
General REST-Security Framework
REST-ful HTTP Security Framework
REST-ful CoAP Security Framework
REST-ful X Security Framewok
Web Service security stack, ...
IoT Service security stack, ...

Figure 6. Instantiation of the general REST-security framework to specific REST-ful protocols.

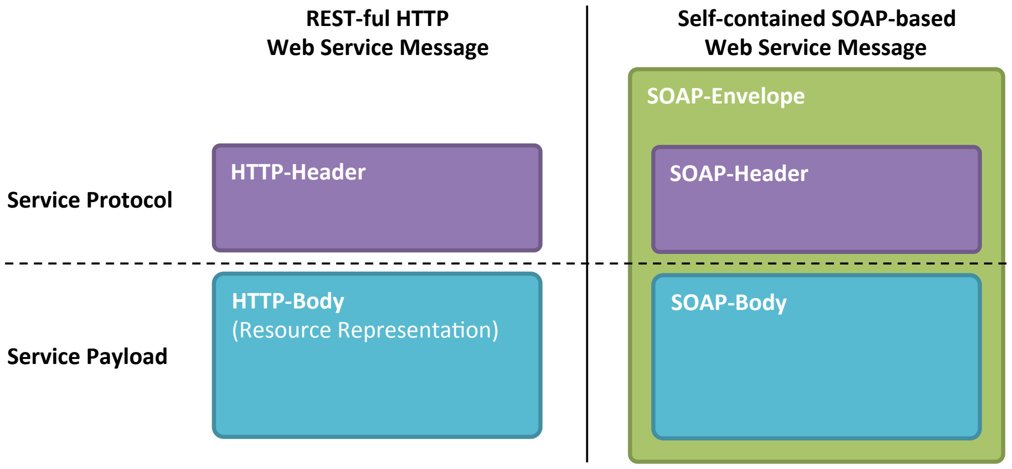
REST-ful HTTP Web Service Message
Self-contained SOAP-based Web Service Message
SOAP-Envelope
Service Protocol
HTTP-Header
SOAP-Header
Service Payload
HTTP-Body (Resource Representation)
SOAP-Body

Figure 7. Comparison of the SOAP-based service message structure with a REST service message structure exemplified by a REST-ful HTTP instantiation.

Table 1 shows a set of possible attack vectors, which can be applied to REST-ful HTTP messages that do carry a protected body only. The assumed attacker model is a common man-in-the-middle (MITM) attack, in which an intruder is able to tamper the whole HTTP request and response messages due to exploited transport security vulnerabilities or a compromised intermediate system.

Attack #1 is based on a GET request that does not contain a resource representation. Thus, the whole HTTP message remains unprotected providing the surface for a malicious twist of the GET method by, e.g., the DELETE method. The second attack tampers the resource path and the host header with the aim of redirecting a client to the attacker's resource. The attack in row three emphasizes that even if a message includes a protected body, an attacker is still able to spoof the unsecured header. In the provided case, the attacker can manipulate the HTTP method and the content-length header in order to construct a valid DELETE request. Moreover, a malicious replacement of a resource representation is also feasible as shown in row four. Here, an adversary can substitute a resource representation with its own resource representation. Similar attacks are also possible on responses. Row five depicts an example, where the location header is changed in order to forward a client to a malicious resource. The attack in row 6 presents a deception of the cache behavior. This manipulation misleads the client or proxy to save the response for two hours. As a consequence, any further requests in the next two hours to this resource will be replied by the cache and not by the origin server so that the client can no longer notice a change of the actual resource state. These possible attack vectors can be transferred analogously to REST-ful CoAP messages that are described in [31]. Note, that Table 1 lists a set of potential attacks vectors, which the authors identified and considered critical. This is not a exhaustive list yet. Future work may uncover additional attack vectors that will provide more arguments for appropriate message-level safeguards.

Table 1. Possible attack vectors on unauthenticated REST-ful HTTP messages.

#	Original REST-Ful HTTP Message	Tampered REST-Ful HTTP Message

1	`GET /resources HTTP/1.1`
	`Host: example.org`
	`Accept: application/json`
	`Content-Length: 0`
	`Connection: keep-alive`
		`DELETE` `/resources HTTP/1.1`
		`Host: example.org`
		`Accept: application/json`
		`Content-Length: 0`
		`Connection: keep-alive`

2	`GET /resources HTTP/1.1`
	`Host: example.org`
	`Accept: application/json`
	`Content-Length: 0`
	`Connection: keep-alive`
		`GET` **`/evilresources`** `HTTP/1.1`
		`Host:` **`attacker.org`**
		`Accept: application/json`
		`Content-Length: 0`
		`Connection: keep-alive`

3	`PUT /resources/3 HTTP/1.1`
	`Host: example.org`
	`Content-Length: 100`
	`Content-Type: application/jose+json`
	`Connection: keep-alive`
	
	`<protected body>`
		`DELETE` `/resources/3 HTTP/1.1`
		`Host: example.org`
		`Content-Length:` **`0`**
		`Content-Type: application/jose+json`
		`Connection: keep-alive`
		
		`<protected body>`

4	`POST /resources HTTP/1.1`
	`Host: example.org`
	`Content-Length: 100`
	`Content-Type: application/jose+json`
	`Connection: keep-alive`
	
	`<protected body>`
		`POST /resources HTTP/1.1`
		`Host: example.org`
		`Content-Length:` **`120`**
		`Content-Type: application/jose+json`
		`Connection: keep-alive`
		
		`<replaced malicious protected body>`

5	`HTTP/1.1 201 Created`
	`Content-Length: 0`
	`Connection: keep-alive`
	`Location: http://example.org/resources/4`
		`HTTP/1.1 201 Created`
		`Content-Length: 0`
		`Connection: keep-alive`
		`Location:` **`http://attacker.org/resources/4`**

6	`HTTP/1.1 200 OK`
	`Connection: keep-alive`
	`Content-Length: 100`
	`Content-Type: application/xml`
	
	`<protected body>`
		`HTTP/1.1 200 OK`
		`Connection: keep-alive`
		`Content-Length: 100`
		`Content-Type: application/xml`
		`Cache-Control: max-age=7200`
		
		`<protected body>`

5. Related Work Analysis

The argued need for a general REST-security framework is further examined by an analysis of the current practice and the available research. The analysis captures the correct security mechanisms and evaluates them according to the specific of REST-based systems and the attacker model given in the previous section. The related work has evolved so far in a relevant manner on REST-ful HTTP and REST-ful CoAP only. Moreover, most of the available work has been conducted in relation to basic service message security with a focus on authentication and authorization. Thus, the subsequent analyzes are driven by these prerequisites [28].

Note, that in comparison to Prokhorenko et al. [32] the related work analysis focused on approaches protecting the specifics of the uniform interface of REST-based systems in general. Security techniques targeting a specific application domain are not considered. That is, protection means referring to vulnerabilities of conventional web applications such as cross-site scripting (XSS), cross-site request forgery (CSRF) or SQL injection are therefore out of scope.

5.1. HTTP Basic and HTTP Digest Authentication

HTTP basic [33] and HTTP digest authentication [34] have been the two first standards for authenticating HTTP requests. Both schemes require a username and a password for the authentication process. If a client tries to access a resource, which is protected by one of these mechanisms, the server returns an error response message including the WWW-authenticate header containing the name of the mechanism, i.e., basic or digest, and a realm which is a description of the secured resource. In case of basic authentication, the client must authenticate itself by sending the former request with an Authorization header, which includes the base64-encoded username and the password. The, i.e., plain-text transfer of username and password transfer is the main downside of this approach. To protect this sensitive information in transit, TLS must be applied additionally. If transport-oriented security is not available, the hole message remains unprotected and password eavesdropping as well as any kind of request manipulations including the ones in Table 1 are feasible.

HTTP Digest Authentication provides a slightly improved approach, as it does not transfer the credentials in clear-text. Here, the username and the password are hashed. Besides the username and password the hash computation includes the URL path, the HTTP method, a client- as well as a server-generated nonce, a sequence number and optionally a quality of protection description. Since this scheme considers the HTTP method and the URL path in the hash calculation, a manipulation of these request message elements is not feasible. However, an attacker can still perform malicious changes of other message entities such as distinct headers and the body.

The other main drawback of both authentication mechanisms is that the request can be authenticated only. Servers are not able to authenticate their responses opening the door for MITM attacks.

5.2. API-Key

Application programming interface (API) keys are randomly generated strings, which are negotiated out-of-band between client and server. An API-key is added to the URL or header of every request. According to an analysis of the web API directory ProgrammableWeb, API-keys are currently the most used authentication mechanism in REST-based web services [35].

API-keys share the same drawbacks as HTTP basic authentication. The API-key is transferred to the server in plain-text. Thus, the credentials are only protected during transit if transport-oriented security means such as TLS are being used.

5.3. HOBA

The experimental request for comments (RFC) HTTP origin-bound authentication (HOBA) [36] is a challenge response HTTP authentication method based on digital signatures. If a distinct resource is protected by HOBA and accessed without authentication, the server returns an error response including the WWW-authenticate header. This header contains a challenge string, an expiration date and an optional realm. To access this protected resource, the client needs to create a signature covering a client-side generated nonce, the base URL of the request, the signature algorithm name, the optional realm, the key identifier and the challenge string. The resulting signature value must then be included in the authorization header together with the key identifier, the challenge string and the nonce.

HOBA does not ensure the integrity of HTTP requests. To do so, each data transfer in HOBA must be protected by TLS. If transport-oriented security is not present, any malicious change of the request can be performed. As with HTTP basic, digest and API-keys, authentication, HOBA considers the authentication of requests only and does not provide the option to protect responses.

5.4. HTTP SCRAM

Salted challenge response HTTP authentication mechanism (HTTP SCRAM) [37] is another experimental RFC. The authentication process is structured in two steps. In the first one, the client sends a request containing an hash-based message authentication code (HMAC) signature algorithm name, the username and a self-generated nonce. The server replies with the client-generated nonce concatenated with a server-generated nonce, a salt and a sequence number. Based on this information, the client performs the second authentication step. It computes an HMAC signature composed of the password, the salt and the sequence number. To access the HTTP SCRAM protected resource, the calculated signature value is embedded in the authorization header including the client-generated nonce concatenated with the server-generated nonce and an HTTP SCRAM-specific description. Once the server receives this request, it verifies the signature and the concatenated nonces. If both values pass the verification process, the server returns the desired response to the client. The response contains a signature value as well which is created by means of the client's password, the salt and the sequence number, so that the client can proof the authenticity of the responding server.

Unlike HOBA, API-keys as well as HTTP basic and digest, HTTP SCRAM provides the option to authenticate requests as well as responses. However, this approach does not guarantee the integrity of the whole message, as the signature does not cover the body and most of the headers.

5.5. Mutual Authentication Protocol for HTTP

The experimental RFC mutual authentication protocol for HTTP [38] is an approach for authenticating requests and responses without sending the user's password in plain-text. To transfer the password in a confidential manner, the client as well as the server each generate a key exchange value. The generated exchange value of the client is sent via a request to server and the generated exchange value of the server is returned to client by a response.

Based on these client- and server-generated key exchange values, a session secret is calculated. The client as well as the server use this session secret to create a verification value. Included within the authorization request header of the request or the authentication-Info response header, the verification value serves as a parameter for the server and the client respectively for validating the authenticity of the communication partner's received messages.

The procedure for computing the verification value, the key exchange value and the session secret is not specified in [38]. A description of algorithms for computing the credentials is provided in a separate specification [39]. Here, the key exchange values are randomly generated. The session secret is a SHA-256 or SHA-512 hash calculated from the key exchange values of the client and the server as well as the user's password. Alternatively, the session secret can be calculated via elliptic curve digital signatures, which integrates the key exchange values and the password in the computation process as well. Both verification values are hashes or digital signatures based on the key exchange values and the session secret.

As the name implies, this approach provides a mutual authentication protocol for clients and servers to verify the authenticity of requests and responses. However, only authenticity can be ensured by this specification. Similar to HOBA and HTTP SCRAM, neither the client nor the server can validate whether other headers or the message body has been manipulated.

5.6. De Backere et al.

De Backere et al. [40] present security mechanisms for REST-based web services focusing on mobile clients. Their protection scheme requires the client to authenticate with the username and the password before retrieving any resources. If the authentication process is successful, the server returns a symmetric key as well as a token representing the key identifier and a timestamp for avoiding replay attacks. Based on these three credentials, a client can send authenticated requests. This is realized by embedding the token and timestamp within the request. The next step signs the request body with the symmetric key. Optionally, the same symmetric key can also be used for encrypting the request body. To protect the response, the server can utilize the generated symmetric key for authenticating and encrypting the body of the responses. Alternatively, the approach of De Backere et al. provides the option to sign the response body with the server's private key.

The advantage of this approach is the consideration of authenticity, integrity and confidentiality of HTTP requests and responses. However, only the message body is protected by this scheme. The header is left unprotected. Another drawback is a missing description defining whether the token, the timestamp and the computed signature value must be included in the header or body.

5.7. Peng et al.

Peng et al. [41] present an academic approach which is based on HTTP basic and HTTP digest authentication (see Section 5.1). This scheme requires the client to compute two hashes, which are then added to the HTTP header. The first hash is calculated on the basis of a server-generated nonce, a timestamp and a password. The second one is a hash of the username, the realm, the server-generated as well as client-generated nonce, the sequence number, the corresponding HTTP method and the URL path. Both computed hashes including the nonces, the timestamp, the sequence number and the realm are stored in new defined headers before sending the message to the server.

The authentication mechanism of Peng et al. only considers the HTTP method and the URL path in the hash calculations. Other header entries and the body are not secured. Moreover, the approach offers neither an authentication nor an integrity protection of the response.

5.8. FOAF + SSL/WebID

The friend-of-a-friend project (FOAF) [42] project aims to define a specification for linking people and information on the web. In FOAF, people, agents, groups and their relations can be described in a machine-readable manner. FOAF + SSL [43], also known as WebID [44], extends FOAF by authentication. The trust model of FOAF + SSL is based on the web of trust (WOT) [45] where each entity acts as a trusted third party. Each WebID certificate contains a link to a corresponding FOAF description, in which a entity and its relations to other entities are defined. Based on this description and references a WOT can be built. As the name FOAF + SSL implies, the WebID certificate is used to establish a TLS connection likewise. Doing so, authenticity, confidentiality and integrity can be ensured in the transport layer.

FOAF + SSL does not provide any safeguards for the application layer. The security is based on TLS and WOT only. Thus, systems using FOAF + SSL for authentication are still vulnerable for the attacks described in Table 1.

5.9. Google, Hewlett Packard and Microsoft

The cloud storage services of Google [46], Hewlett Packard (HP) [47] and Microsoft [48] utilize an enhanced API-keys mechanism that prevents eavesdropping the key in transit. Instead of simply including the API-key directly to the URL or HTTP header, clients signs the request. Conceptually, the core signing process of all three operating cloud storage services is equal. A string to be signed is constructed by concatenating the HTTP method with the resource path including the query (unless HP, which makes use of the resource path only) and a fixed set of headers. Independent of the exact composition of these sets, only the timestamp entry is mandatory. All other specified headers—including for instance the content-type or content-MD5 entries—are optional. The concatenated string is signed by the API-key. The signature value is enriched with further signature-related meta-data such as the signature algorithm name and a key identifier. This generated authentication information is finally inserted into the Authorization header. Google supports an alternative option, which allows incorporating the authentication information inside the query part of the URL.

The defined sets of headers to be considered by each of these provider-proprietary approaches do not consider all security-relevant message elements (see Table 1). Missing entries include, for instance, the host and the connection header. These omissions enable an adversary, e.g., to redirect the message to another system or to manipulate the connection management. Moreover, the providers do not stringently require considering a hash of the body in signature computation. Clients may create the content-MD5 header to integrate a hash of the body in the signature, but they do not have to.

Integrating a hash value covering the body's resource representation into the string to be signed is a vital requirement in order to provide the integrity of the whole REST message. Ignoring this opens the door for spoofing the resource representation. The last but not least observed issue is the lack of mutual authentication, due to leaving the response out of the protection sphere. Thus, a client cannot proof the authenticity of a response providing the surface for MITM attacks.

5.10. Amazon

Another provider-proprietary approach deployed by the Amazon simple storage service (S3) requires service invocations over HTTP by to be signed [49]. As with the other three commercial cloud storage services, S3 concatenates the HTTP method, the URL's resource path including the query and a set of headers to a string that is to be signed. The authentication approach of Amazon offers, however, more flexibility as it allows protecting application-specific headers. This is realized by a list that specifies the headers required to be appended before signing or verifying the HTTP message. When this list is used, the request must contain at least the Host header, a header containing a timestamp and the x-amz-sha256 header, which stores a SHA-256 hash of the body. The list is then stored together with the signature value and the remaining authentication information either within the authorization header or in the URL. Based on this list, the S3 service checks what headers are covered by the signature. If one of the required headers is not contained in the list, the service rejects the request.

The benefit of Amazon's approach is the required hash of the body in the signature generation. Amazon sets, however, the host header, the timestamp and the x-amz-sha256 header as mandatory only. Consequently, further important meta information such as the content-length, the content-type and the connection header are not considered. Thus, an attacker is able to manipulate the resource representation and the connection, if these headers have not been signed. With the aid of the list, an adversary can extract what has been signed and what not. If the content-length and content-type header are not in the list, a replacement of the resource representation with another resource representation with the same hash value is feasible. Taking the two aforementioned headers into account is crucial to mitigate such attacks. By this, the attacker has to find a resource representation that has the same hash value, size and media type as the actual body. Also Amazon's HTTP authentication scheme suffers from not taking the response into account.

5.11. Signing HTTP Messages

A standard dealing with the authentication of HTTP messages is the Signing HTTP messages draft of the internet engineering task force (IETF) [50]. Similar to the discussed proprietary approaches, a signer has to concatenate the HTTP method, the resource path including the query and a set of headers to a string to be signed. The concatenation order of the headers is determined by the signer, which creates a corresponding list. This list is embedded in the authorization or the newly defined signature header together with the signature algorithm name, the key identifier and signature value. Using this list, however, is not required. An absent list results in considering the Date header in the signature generation only. Consequently, a present list must contain at least a date entry.

Besides this header, the proposal does not consider additional meta-data relevant to ensure HTTP message authentication. The client can optionally add more header entries to the signature string if required and aware of the consequences of a too narrow protection sphere. Furthermore, the draft does not require incorporating a hash of the body in the signature computation. Moreover, it does not make clear, how a server needs to authenticate a response. Signing the response is mentioned at the beginning of the draft, but in the rest of the specification it is not elaborated any further.

5.12. OAuth

OAuth [29,51] is an authorization framework for granting access to end users' resources for third party applications. Currently, two versions of OAuth have been published.

The OAuth v1 specification of the IETF [51] has an inherent support for protecting a request by a signature. The signature string is the concatenation of the HTTP method, the resource path including the query, the host header and a set of OAuth v1 specific parameters. The latter parameters consist of a realm, a key identifier that is called consumer key, an OAuth token, a timestamp and a nonce. OAuth v1 does not enable to add any other parameters or headers in the signature. The authentication information is stored in the authorization header. Like the other approaches discussed so far, the authenticity of the request is considered solely. No means for signing a response have been defined.

In contrast to the first version, OAuth v2 does not include any security means on its own [29]. Instead, the security is merely based on TLS. If a message-oriented protection is yet required, OAuth v2 can be augmented by either the OAuth MAC tokens [52] specification or by the extension a method for signing an HTTP requests for OAuth [53]. The OAuth MAC token draft demands to sign the HTTP method, the resource path including the query and at least the host header. Further meta-data can be considered by defining a list similar to some of the previously discussed approaches. The resulting signature value has to be included into the authorization header.

The second OAuth v2 extension a method for signing an HTTP requests for OAuth uses a JSON web signature (JWS) [54] to guarantee the authenticity of HTTP messages. The JWS object used in this specification owns a set of members, which contains the method, the host including the port, the resource path, the query, the headers, an HMAC authenticator of the body and a timestamp. Using JWS as the pillar can be a stable groundwork, since it is a well advanced IETF draft for signing JSON objects [55] that is already used in many applications. However, the main drawback of this specification lies in the fact that all mentioned JSON members are optional. Even though most of these elements are vital to guarantee the authenticity and integrity of an HTTP message, none of them is set as mandatory for the signature. Also, this draft does not state any information whether the JWS object is stored in a header or in the body.

The common problem of both OAuth versions is the tight coupling to the actual application domain of these authorization frameworks. As a result, adopting these standards to other contexts is not feasible in a straightforward manner. As with the other approaches, the major disadvantage of the OAuth protocols is that they do not specify a protection of the response.

5.13. Serme et al.

Serme et al. [56] introduce the first approach addressing the protection of HTTP responses by proposing a REST-ful HTTP message authentication protocol, which protects the request as well as the response. Their approach introduces new headers containing the certificate ID, the hash algorithm and the signature algorithm name. The input to the signature algorithm is a concatenation of the body, the URL, the hash algorithm name, the signature algorithm name, the certificate ID and a set of headers forming a string to be signed. The generated hash and signature values are stored in separated, newly defined headers each. Moreover, Serme et al. propose an encryption and decryption scheme for HTTP messages.

One drawback of [56] is the missing reference implementation. This paper provides two pseudo code notations of the signature generation and verification schemes as well as another two of the encryption and decryption schemes. These algorithms do not clearly state whether a timestamp or the HTTP method are considered in the processing. Moreover, they do not specify any order of the concatenation or some form of policy, which retains the order. Likewise, the approach does not define what headers need to be obligatory protected. That is, it is not clear whether all headers or a subset of them must be signed/encrypted.

5.14. Lee et al.

Lee et al. [57,58] define a method for signing and encrypting HTTP messages. The key pairs for performing the encryption, decryption and digital signatures are generated by a third party entity.

Before a client and a server are starting to communicate with each other, both parties must request two public points (p1, p2), representing the master public key, and a private point (sQ) from a private key generator (PKG). The PKG is the trusted third party service. Its only task is to send the master public key and the private point.

Based on this master public key and the other endpoint's URL, the client and the server can compute the public key of their communication partner. The private point sQ is then used by both parties to calculate their own private key respectively. Once the key pairs and the public key of the counterpart is present, the client and the server can use the communication partner's public key to encrypt the HTTP message. Before starting the data transfer, the encrypted message is signed by the private key of the corresponding endpoint.

The approach of Lee at al. ensures the authenticity, integrity and confidentiality of the whole HTTP message. Moreover, requests as well as responses are protected by this scheme. However, the encryption of the whole message with the aim that only the endpoints are able to decrypt and interpret, violates the self-descriptive constraint of REST messages [1]. That is, only the client and the server can understand the intention of the message. Intermediate systems are not able to process the fully encrypted and signed message, as they possess neither the corresponding private key of the client nor the server. If an intermediary is not able to understand and process a traversing message, it may reject forwarding the message or cancel the communication. Hence, ensuring the confidentiality of REST messages requires to cope with special challenges in order to be in conformance with the REST principles. Requirements for defining a confidentiality scheme in REST are discussed in Section 6.3. Another shortcoming of this approach is a missing time variant parameter in the signature process, which makes the scheme vulnerable to replay attacks.

5.15. OSCORE

Object security for CoAP (OSCORE) [59] is a draft standard providing encryption, integrity and replay attack protection for CoAP messages. The CoAP message payload and a set of security-relevant headers are protected by OSCORE. Still, some other security-critical header entries including the token length, message ID, token and max-age option as well as the meta information for the body length are left unprotected. The reason why leaving out the first four meta-data lies in the fact that these entries may be changed by intermediate nodes. However, not considering these meta-data elements opens the door for man-in-the-middle attacks. Possible attack vectors can be spoofing the message ID and token in order to provoke a mismatch between requests and responses. Also, sending the max-age option without any protection is critical, as it has a similar functionality like the cache-control header in HTTP. When it gets tampered, the freshness of the response is corrupted analogously to attack six in Table 1. On the contrary, signing these header entries to avoid the aforementioned attacks prevents middleboxes from changing these elements. Thus, leaving out these header entries from the protection sphere and protecting these meta-data elements induce issues on both sides. To resolve this problem, an enhanced approach has to ensure the integrity of these header entries and it must allow intermediate systems to modify the meta-data elements simultaneously. Moreover, OSCORE does not consider protecting the integrity of the body length meta information. The reason behind this might be that no header for the body length is specified in the CoAP standard, as this information must be extracted from the UDP packet. This omission enables the manipulation of the body as manifested by attack four in Table 1.

According to the specification, messages protected by OSCORE are not intended for being cached. Each response is strictly bound to its corresponding request. Not supporting the option to cache messages may lead to a low scalability and violates the cacheability principle, which is one of the most vital REST constraints. Also, the OSCORE specification does not provide a protection for acknowledgment and reset messages. As both messages are utilized to confirm or reject a CoAP request or response, they must be secured as well. This prevents attacker from replacing an acknowledgment message by a reset message or vice versa. Another issue of OSCORE is a missing description that lists additional or application-specific header entries to be signed and/or encrypted.

5.16. Granjal et al.

Granjal et al. [60] propose a scheme that signs and/or encrypts CoAP messages. This approach offers the options to encrypt a message, sign a message or sign as well as encrypt a message. However, the authors do not provide any policy, which specifies a list on the to be protected header entries. The proposed approach computes a signature and an encryption over the entire CoAP message including the payload and all present header entries. The resulting cipher-text and signature value are then stored in newly defined security headers which contain information on the security context such as the key type, whether the message is encrypted, signed or both and the destination. The latter information can refer to endpoints, i.e., client and server, or intermediate systems. A CoAP message may include one or multiple instances of these security header entries. Thus, the approach of Granjal et al. [60] allows to compute signatures and cipher text for multiple endpoints and intermediaries which enables an intermediate system to verify and decrypt traversing messages. However, the paper does not describe whether an intermediary is able sign and decrypt a message itself. This is an important property of a REST-ful security scheme in order to comply with the layered system constraint. This principle enables intermediate systems to interpret and transform the content of a message. The ability of intermediaries to sign and encrypt messages by themselves allows them to transform messages or parts of them and inform the endpoint that a distinct intermediary has processed certain message elements. Moreover, encrypting and signing the entire message without obeying a policy, which defines what headers are protected, violates the self-descriptive constraint. This prevents certain intermediate nodes from accessing and modifying a message. That is, the signature of the message is invalid if an intermediary changes the message, as no policy for describing the modification exists. The other drawback is that a completely encrypted message is not accessible by intermediate systems not possessing the required decryption key. Both scenarios may occur, as a lot of intermediaries are either transparent or reside outside organizational boundaries of the client and the server.

5.17. Consolidated Review of Analysis Results

The obtained insights from the conducted analyzes of the available related work are summarized in Tables 2 and 3. Note, that only approaches are listed, which ensure authenticity and integrity of distinct message elements. The schemes proposed in [33,34,36–38,40,43,44,61] and API-keys are omitted for readability reasons, since they do not provide any integrity protection for headers.

The related work analysis reveal that a lot of REST-based HTTP and Coap message authentication attempts have been evolved so far. However, none of the examined approaches targets the same abstraction layer as REST. Also, the evaluated mechanisms contain many vulnerabilities or are not in conformance with the REST constraints.

The concrete adoptions to web, cloud and IoT services are very diverse, emphasizing the need for a more methodical approach to REST message authentication and to REST-security in general. Moreover, due to the lack of a general REST-security framework, the same situation can be expected to take place in any other appearing implementation domain, in which REST gets adopted. All this emphasizes the need for a more advanced and elaborated security for REST-based service systems.

Table 2. Analysis of related work in HTTP message authenticity and integrity.

Message Elements to Be Signed	**Amazon [49]**								**Google [46]**							
	Request				**Response**				**Request**				**Response**			
	C	**R**	**U**	**D**	**C**	**R**	**U**	**D**	**C**	**R**	**U**	**D**	**C**	**R**	**U**	**D**
URI	●	●	●	●	-	-	-	-	●	●	●	●	-	-	-	-
Version number	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Method	●	●	●	●	-	-	-	-	●	●	●	●	-	-	-	-
Status code	-	-	-	-	○	○	○	○	-	-	-	-	○	○	○	○
Connection	◐	◐	◐	◐	○	○	○	○	○	○	○	○	○	○	○	○
Cache-control	◐	◐	◐	◐	○	○	○	○	○	○	○	○	○	○	○	○
Location	-	-	-	-	○	-	-	-	-	-	-	-	○	-	-	-
Accept	-	◐	-	-	-	-	-	-	-	○	-	-	-	-	-	-
Content-type	◐	-	◐	-	-	○	-	-	◐	-	◐	-	-	○	-	-
Content-length	◐	-	◐	-	-	○	-	-	○	-	○	-	-	○	-	-
Transfer-encoding	◐	-	◐	-	-	○	-	-	○	-	○	-	-	○	-	-
Host	●	●	●	●	-	-	-	-	○	○	○	○	-	-	-	-
Hash of body	●	●	●	●	○	○	○	○	◐	◐	◐	◐	○	○	○	○
Time variant parameter	●	●	●	●	○	○	○	○	●	●	●	●	○	○	○	○

Message Elements to Be Signed	**Microsoft [48]**								**HP [47]**							
	Request				**Response**				**Request**				**Response**			
	C	**R**	**U**	**D**	**C**	**R**	**U**	**D**	**C**	**R**	**U**	**D**	**C**	**R**	**U**	**D**
URI	●	●	●	●	-	-	-	-	●	●	●	●	-	-	-	-
Version number	-	-	-	-	-	-	-	-	○	○	○	○	○	○	○	○
Method	●	●	●	●	-	-	-	-	●	●	●	●	-	-	-	-
Status code	-	-	-	-	○	○	○	○	-	-	-	-	○	○	○	○
Connection	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
Cache-control	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
Location	-	-	-	-	○	-	-	-	-	-	-	-	○	-	-	-
Accept	-	○	-	-	-	-	-	-	-	○	-	-	-	-	-	-
Content-type	◐	-	◐	-	-	○	-	-	◐	-	◐	-	-	○	-	-
Content-length	◐	-	◐	-	-	○	-	-	○	-	○	-	-	○	-	-
Transfer-encoding	○	-	○	-	-	○	-	-	○	-	○	-	-	○	-	-
Host	○	○	○	○	-	-	-	-	○	○	○	○	-	-	-	-
Hash of body	◐	◐	◐	◐	○	○	○	○	◐	◐	◐	◐	○	○	○	○
Time variant parameter	●	●	●	●	○	○	○	○	●	●	●	●	○	○	○	○

Message Elements to Be Signed	**OAuth v2 MAC Tokens [52]**								**Signing an HTTP Request ... [53]**							
	Request				**Response**				**Request**				**Response**			
	C	**R**	**U**	**D**	**C**	**R**	**U**	**D**	**C**	**R**	**U**	**D**	**C**	**R**	**U**	**D**
URI	●	●	●	●	-	-	-	-	◐	◐	◐	◐	-	-	-	-
Version number	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
Method	●	●	●	●	-	-	-	-	◐	◐	◐	◐	-	-	-	-
Status code	-	-	-	-	○	○	○	○	-	-	-	-	○	○	○	○
Connection	◐	◐	◐	◐	○	○	○	○	◐	◐	◐	◐	○	○	○	○
Cache-control	◐	◐	◐	◐	○	○	○	○	◐	◐	◐	◐	○	○	○	○
Location	-	-	-	-	○	-	-	-	-	-	-	-	○		-	-
Accept	-	◐	-	-	-	-	-	-	-	◐	-	-	-		-	-
Content-type	◐	-	◐	-	-	○	-	-	◐	-	◐	-	-	○	-	-
Content-length	◐	-	◐	-	-	○	-	-	◐	-	◐	-	-	○	-	-
Transfer-encoding	◐	-	◐		-	○	-	-	◐	-	◐		-	○	-	-
Host	●	●	●	●	-	-	-	-	◐	◐	◐	◐	-	-	-	-
Hash of body	◐	◐	◐	◐	○	○	○	○	◐	◐	◐	◐	○	○	○	○
time variant parameter	●	●	●	●	○	○	○	○	◐	◐	◐	◐	○	○	○	○

Table 2. *Cont.*

Message Elements to Be Signed	**Signing HTTP Messages** [50]								**OAuth v1** [51]							
	Request				**Response**				**Request**				**Response**			
	C	R	U	D	C	R	U	D	C	R	U	D	C	R	U	D
URI	●	●	●	●	-	-	-	-	●	●	●	●	-	-	-	-
Version number	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
Method	●	●	●	●	-	-	-	-	●	●	●	●	-	-	-	-
Status code	-	-	-	-	⦸	⦸	⦸	⦸	-	-	-	-	○	○	○	○
Connection	◐	◐	◐	◐	⦸	⦸	⦸	⦸	○	○	○	○	○	○	○	○
Cache-control	◐	◐	◐	◐	⦸	⦸	⦸	⦸	○	○	○	○	○	○	○	○
Location	-	-	-	-	⦸	-	-	-	-	-	-	-	○	-	-	-
Accept	-	◐	-	-	-	-	-	-	-	○	-	-	-	-	-	-
Content-type	◐	-	◐	-	-	⦸	-	-	○	-	○	-	-	○	-	-
Content-length	◐	-	◐	-	-	⦸	-	-	○	-	○	-	-	○	-	-
Transfer-encoding	◐	-	◐	-	-	⦸	-	-	○	-	○	-	-	○	-	-
Host	◐	◐	◐	◐	-	-	-	-	●	●	●	●	-	-	-	-
Hash of body	◐	◐	◐	◐	⦸	⦸	⦸	⦸	○	○	○	○	○	○	○	○
Time variant parameter	●	●	●	●	⦸	⦸	⦸	⦸	●	●	●	●	○	○	○	○

Message Elements to Be Signed	**Serme et al.** [56]								**HTTP Digest Authentication** [34]							
	Request				**Response**				**Request**				**Response**			
	C	R	U	D	C	R	U	D	C	R	U	D	C	R	U	D
URI	●	●	●	●	-	-	-	-	●	●	●	●	-	-	-	-
Version number	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
Method	⦸	⦸	⦸	⦸	-	-	-	-	●	●	●	●	-	-	-	-
Status code	-	-	-	-	⦸	⦸	⦸	⦸	-	-	-	-	○	○	○	○
Connection	⦸	⦸	⦸	⦸	⦸	⦸	⦸	⦸	○	○	○	○	○	○	○	○
Cache-control	⦸	⦸	⦸	⦸	⦸	⦸	⦸	⦸	○	○	○	○	○	○	○	○
Location	-	-	-	-	⦸	-	-	-	-	-	-	-	○	-	-	-
Accept	-	⦸	-	-	-	-	-	-	-	○	-	-	-	-	-	-
Content-type	⦸	-	⦸	-	-	⦸	-	-	○	-	○	-	-	○	-	-
Content-length	⦸	-	⦸	-	-	⦸	-	-	○	-	○	-	-	○	-	-
Transfer-encoding	⦸	-	⦸	-	-	⦸	-	-	○	-	○	-	-	○	-	-
Host	⦸	⦸	⦸	⦸	-	-	-	-	○	○	○	○	-	-	-	-
Hash of body	●	●	●	●	●	●	●	●	○	○	○	○	○	○	○	○
Time variant parameter	⦸	⦸	⦸	⦸	⦸	⦸	⦸	⦸	○	○	○	○	○	○	○	○

Message Elements to Be Signed	**Peng et al.** [41]								**Lee et al.** [57,58]							
	Request				**Response**				**Request**				**Response**			
	C	R	U	D	C	R	U	D	C	R	U	D	C	R	U	D
URI	●	●	●	●	-	-	-	-	⦸	⦸	⦸	⦸	-	-	-	-
Version number	○	○	○	○	○	○	○	○	⦸	⦸	⦸	⦸	⦸	⦸	⦸	⦸
Method	●	●	●	●	-	-	-	-	⦸	⦸	⦸	⦸	-	-	-	-
Status code	-	-	-	-	○	○	○	○	-	-	-	-	⦸	⦸	⦸	⦸
Connection	○	○	○	○	○	○	○	○	⦸	⦸	⦸	⦸	⦸	⦸	⦸	⦸
Cache-control	○	○	○	○	○	○	○	○	⦸	⦸	⦸	⦸	⦸	⦸	⦸	⦸
Location	-	-	-	-	○	-	-	-	-	-	-	-	⦸	-	-	-
Accept	-	○	-	-	-	-	-	-	-	⦸	-	-	-	-	-	-
Content-type	○	-	○	-	-	○	-	-	⦸	-	⦸	-	-	⦸	-	-
Content-length	○	-	○	-	-	○	-	-	⦸	-	⦸	-	-	⦸	-	-
Transfer-encoding	○	-	○	-	-	○	-	-	⦸	-	⦸	-	-	⦸	-	-
Host	○	○	○	○	-	-	-	-	⦸	⦸	⦸	⦸	-	-	-	-
Hash of body	○	○	○	○	○	○	○	○	⦸	⦸	⦸	⦸	⦸	⦸	⦸	⦸
Time variant parameter	●	●	●	●	○	○	○	○	⦸	⦸	⦸	⦸	⦸	⦸	⦸	⦸

Legend: ● mandatory signed, ○ not signed, ◐ optionally signed, ⦸ not specified, - not required.

Table 3. Analysis of related work in constraint application protocol (CoAP) message authenticity and integrity.

CoAP Message Elements to Be Signed	**OSCORE [59]**								**Granjal et al. [60]**							
	Request				**Response**				**Request**				**Response**			
	C	R	U	D	C	R	U	D	C	R	U	D	C	R	U	D
Version number	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
Type	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
Token Length	○	○	○	○	○	○	○	○	●	●	●	●	●	●	●	●
Code (method code, response code)	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
Message ID	○	○	○	○	○	○	○	○	●	●	●	●	●	●	●	●
Token	○	○	○	○	○	○	○	○	●	●	●	●	●	●	●	●
Uri-host, Uri-port, Uri-path	●	●	●	●	-	-	-	-	●	●	●	●	-	-	-	-
Max-age	○	○	○	○	○	○	○	○	●	●	●	●	●	●	●	●
Location-path, location-query	-	-	-	-	-	-	-	-	-	-	-	-	●	●	●	●
Accept	●	●	●	●	-	-	-	-	●	●	●	●	-	-	-	-
Content-format	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
Body length (payload-length)	○	○	○	○	○	○	○	○	⦸	⦸	⦸	⦸	⦸	⦸	⦸	⦸
Body	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
Time variant parameter	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●

Legend: ● mandatory signed, ○ not signed, ◐ optionally signed, ⦸ not specified, - not required.

6. Towards a General REST-Security Framework

The previous sections motivated and highlighted the need for a general REST-security framework. The available approaches provide security solutions for REST-ful HTTP and REST-ful CoAP only and do not offer any concepts residing on the same abstraction layer as REST itself. Moreover, the introduced and discussed specifics of REST-based services of any kind made apparent that the application of the available standards, technologies and research is neither developed in a manner that suits REST nor evolved enough in maturity for an adoption in security-sensitive or mission-critical environments.

This section, therefore, proposes a methodology for defining general REST-security framework components. It starts by developing a generic authentication scheme for REST messages. This security concept marks an initial step towards a REST message security, which forms the vital foundation for the general REST-security framework. However, before being able to design any security schemes for REST, the specifics and constraints of the architectural style require to be addressed first.

The REST message elements as well as the resource identifier forming the uniform interface can be implemented by different standards and are equally important for the message processing. Hence, a REST-security scheme needs to consider them all in order to avoid otherwise possible vulnerabilities (see Section 4.3). Such security specifications must be defined on the same abstraction layer as REST itself, so that they can be applied to any concrete protocol instantiation in a methodical manner (see Figure 8). To do so, a formal description of REST messages and an identification of security relevant parts in such messages need to be at hand [28].

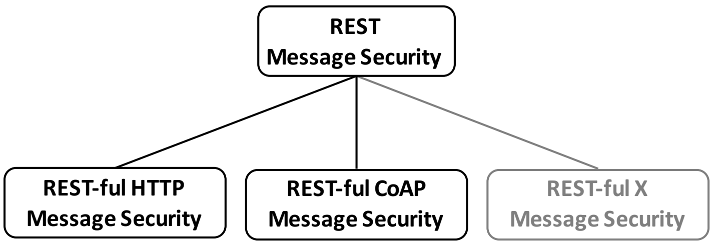

Figure 8. General REST message security and its instantiation to concrete REST-ful protocols.

6.1. Formal Description of REST Messages

Since REST is constrained to the client-server model in conjunction with the request-response model, it is always the client issuing a request message to which the server replies with a corresponding response. The request message space is denoted by R_c and the response message space is referred to as R_s respectively. The whole REST message space R is henceforth

$$R := R_c \cup R_s. \tag{1}$$

The meta-data space M is composed of the set of resource meta-data M_r, the set of resource representation meta-data M_b and the set of control data M_c:

$$M := M_r \cup M_b \cup M_c. \tag{2}$$

The control data set M_c consists of the set of request actions M_{ca}, the set of response meanings M_{cm}, the set of message parameterisation M_{cp} and the set of data to overwrite the default processing of a message M_{co}:

$$M_c := M_{ca} \cup M_{cm} \cup M_{cp} \cup M_{co}. \tag{3}$$

A REST message $r \in R$ consists of two parts: a header h containing meta-data and a body b comprising a resource representation. With H denoting the header and B the body space, the structure of a REST message is defined as

$$r := h||delimiter||b, \{(r,h,b) : r \in R \wedge h \in H \wedge (b \in B \vee b \in \emptyset)\}, \tag{4}$$

where *delimiter* is a set of characters separating the header from the body and $||$ represents the concatenation operation. Note, that the actual embodiment of the delimiter depends on the concrete implementation of the uniform interface, i.e., the service protocol. In case of a binary protocol, the delimiter set might even be empty. For the sake of readability but without the loss of generality, the following explanations will focus on text-based protocols only, since these protocols include additional challenges in terms of the ordering, normalization and separation of headers. To obtain an according description for binary protocols, these aspects can simply be omitted.

A header h holds a subset $\dot{M} \subset M$ of the meta-data entities:

$$h := \begin{cases} (\dot{M}, i), & \text{if } r \in R_c, \\ (\dot{M}), & \text{if } r \in R_s. \end{cases} \tag{5}$$

If h is part of a request message, it additionally includes a resource identifier $i \in I$, where I defines the set of resource identifiers. The constitution of h can further be characterized by the following policy:

- A message $r \in R$ comprising a resource representation must include at least the two resource representation meta-data entities $m_{bl} \in M_b$ and $m_{bt} \in M_b$ describing the length and the media type of the contained resource representation respectively.
- A request $r \in R_c$ must contain at least one control data element $m_{ca} \in M_{ca}$ and one resource identifier i describing the action and the target of the action respectively.
- A response $r \in R_s$ must contain one control data element $m_{cm} \in M_{cm}$ expressing the meaning of the response.

On the basis of this formal description, the following subsections introduce two generic schemes for ensuring the authenticity, integrity and non-repudiation of REST messages.

6.2. REST Message Authentication (REMA)

Following the introduced methodology and the results obtained from the related work analysis, the general REST message authentication (REMA) can be instantiated to REST-ful protocols of any kind (see Figure 9).

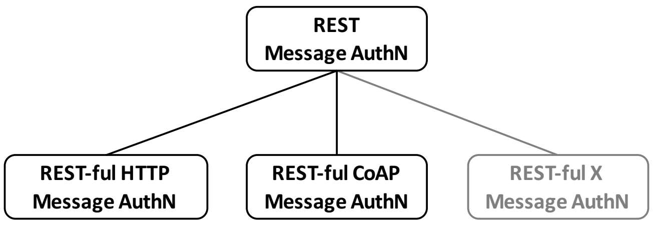

Figure 9. General REST message authentication and its instantiation to concrete REST-ful protocols

In order to illustrate the methodology, a REST-ful HTTP message authentication (REHMA, see Section 7.1) and a REST-ful CoAP message authentication (RECMA, see Section 7.2) are derived from the general framework subsequently.

6.2.1. Message Parts to Be Authenticated

The listed headers in the policy of Section 6.1 are crucial for the intended message processing and therefore need to be protected. In the following, the set of header entries containing the security-relevant and to be protected headers is denoted as $\tilde{h}$. Note, that $\tilde{h}$ varies depending on whether it is part of a request or response, the action of the request, the meaning of the response and whether the message contains a resource representation or not. The variability of $\tilde{h}$ can be especially substantiated by the request actions. Depending on the objective of the action, $\tilde{h}$ requires a different set of meta-data.

The following rules extend the policy of Section 6.1 and define additional security-relevant and mandatory headers to be authenticated and integrity protected for service protocols supporting CRUD actions. The combined rules are henceforth denoted as the REMA policy.

- A read request must contain at least one resource representation meta-data element $m_{br} \in M_b$ describing the desired media type being requested. Moreover, this request must not include a resource representation.
- A creation request must contain a resource representation.
- An update request must contain a complete or partial resource representation.
- A delete request does not require any additional prerequisite headers until further requirements. Moreover, this request must not include a resource representation.

Further extension of the REMA policy in terms of additional security-relevant header entries contained in $\tilde{h}$ are a matter of the technical instantiation of REST and the application domain. Based on these abstract notations, a general signature generation and verification scheme for REST messages can be defined.

6.2.2. REST Message Signature Generation

Algorithm 1 defines a general method for ensuring the authenticity and integrity of REST messages by generating a digital signature over the body and security-vital header entries as defined above. Note, that error conditions are not made explicit for readability reasons. Each error will cancel the signature generation process with an according error message.

Algorithm 1 Representational state transfer (REST) message signature generation [28].

```
Input: REST message $r$, description $desc$ of the application-specific header entries to be signed,
    signature generation key $k$
Output: Signature value $sv$, time-variant parameter $tvp$
 1: $b \leftarrow getBody(r)$
 2: $h \leftarrow getHeader(r)$
 3: $\tilde{h} \leftarrow getTbsHeaders(h)$
 4: $\tilde{h} \leftarrow \tilde{h} \| getTbsHeaders(h, desc)$
 5: $tvp \leftarrow generateTimeVariantParameter()$
 6: $tbs \leftarrow tvp$
 7: $i \leftarrow 0$
 8: while $i < |\tilde{h}|$ do
 9:     $tbs \leftarrow tbs \| delimiter \| normalize(\tilde{h}_i)$
10:     $i \leftarrow i + 1$
11: end while
12: $tbs \leftarrow tbs \| delimiter \| hash(b)$
13: $sv \leftarrow sign(k, tbs)$
```

As input, the signature generation algorithm requires a REST message r, a signature generation key k and a description $desc$. The latter parameter contains application-specific headers, which are to be appended to $\tilde{h}$. After obtaining the body b and the header h from the message r, the function in line 3 checks by means of the REMA policy that all required header entries are included in h and if so, constructs $\tilde{h}$ out of them. Then eventually specified additional headers in $desc$ are appended to $\tilde{h}$. In order to avoid replay attacks, the signature generation algorithm creates of a fresh time-variant parameter tvp. This parameter is the first element to be assigned to the tbs variable, which is gradually filled with the data to be signed. These two steps must not be omitted even when a concrete instantiation of this scheme already includes a time-variant parameter in $\tilde{h}$, since between message generation and signature generation might exist a considerable time spread. All headers contained in $\tilde{h}$ are normalized and concatenated to tbs. In order to tie the resource representation b to $\tilde{h}$ inducing the integrity of the conjunction of security-relevant header entries and body, it needs to be appended to tbs as well. The resource representation b is therefore hashed by a cryptographic hash function and the resulting hash value is attached to tbs. Note, that in case a message does have an empty resource representation, a hash of an empty body is computed and added to tbs. The next statement signs the crafted tbs with a signature generation key k. Algorithm 1 outputs the generated signature value sv and the time-variant parameter tvp.

With these two outputs, an authentication control data element $m_{cpa} \in M_{cp}$ can be generated, containing the signature algorithm name sig, the hash algorithm name $hash$, a key identifier kid, the time-variant parameter tvp, the signature value sv and the presence of additional header entries given by $desc$ in the specified order. This control data element m_{cpa} needs ultimately to be embedded into the respective message r. Since resource representations can vary, m_{cpa} must be integrated into the header h of the message r in order to remain data format independent.

6.2.3. REST Message Signature Verification

Algorithm 2 specifies the signature verification procedure for REST messages signed by Algorithm 1. The signature verification algorithm requires a signed REST message r as input and it returns a boolean value expressing the signature validation result. From the signed message r the required parts are extracted, including the message body b and the message header h. From h the authentication control data header m_{cpa} is obtained next containing the concatenated values sig, $hash$, kid, tvp, sv and $desc$. After building $\tilde{h}$ in line 5, the next statement appends the additional header entries defined in *desc* to $\tilde{h}$ in order of appearance. Then the headers in $\tilde{h}$ are iterated in the same manner—and especially the same order—as during the signature generation process to build *tbs*. With *tbs* and the signature verification key identifier *kid*, the verification of the signature value *sv* can be performed. The boolean verification result is assigned to the variable *valid*, which represents the output of the signature verification procedure.

Algorithm 2 REST message signature verification [28].

```
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:     $tbs \leftarrow tbs \| delimiter \| normalize(\tilde{h}_i)$
11:     $i \leftarrow i + 1$
12: end while
13: $tbs \leftarrow tbs \| delimiter \| hash(b)$
14: $verify \leftarrow getVerificationAlgorithm(sig)$
15: $valid \leftarrow verify(kid, tbs, sv)$
```

6.3. REST Message Confidentiality (REMC)

In layered systems such as those constrained by REST, confidentiality is of specific importance, since intermediate systems otherwise have plain-text access to traversing messages and those systems most commonly reside outside organizational boundaries of service providers and consumers. To prevent intermediaries from accessing sensitive message parts, the encryption of REST messages is a required foundational REST message security building block.

REMA ensures the authenticity, integrity and—when using asymmetric digital signatures in conjunction with a suitable PKI—non-repudiation of REST-ful protocol messages. In order to approach a comprehensive REST message security, the confidentiality must be taken into account as well. Following the introduced methodology of this paper, a REST message confidentiality scheme has to define a general policy and algorithms for protecting REST messages from unauthorized data disclosure. Such a scheme then serves as a guideline for adapting and implementing confidentiality services for concrete REST-ful technologies including HTTP, CoAP and prospective ones (see Figure 10).

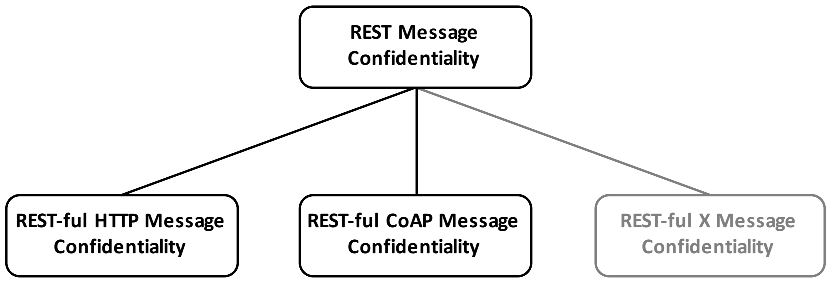
REST Message Confidentiality
REST-ful HTTP Message Confidentiality
REST-ful CoAP Message Confidentiality
REST-ful X Message Confidentiality

Figure 10. General REST message confidentiality and its instantiation to concrete REST-ful protocols.

In contrast to REMA, we do not specify the complete REMC framework, as this is not required to prove the proposed concept. REMA is sufficient and more suitable for this purpose, since there is much more related work available that can be used for evaluation. Still, we want to briefly discuss the requirements and challenges REST message confidentiality framework needs to tackle.

Encrypting the whole REST message—so that only the endpoints can read and interpret the intention of it—does not conform with the self-descriptive messages and layered systems constraint, though. As mentioned before, the both principles require that the semantics of REST messages have to be visible to intermediaries for enabling intermediate processing [1]. This means that the key challenge of a REST message confidentiality scheme is to shield REST message from unauthorized data disclosure while retaining the self-descriptiveness for endpoints and authorized intermediate systems. Hence, distinct message elements, which are required for a particular intermediary in order to render the message self-describing, must remain accessible for this respective intermediary. Consequently, a general policy for encrypting REST messages need to consider and specify what message parts are required to be accessible for which class of intermediaries.

This is especially true for caches, which must be able to read required message elements in order to store responses. As cacheability represents one of the core REST constraints for ensuring scalability, encrypted REST messages must therefore still provide the option to be cacheable. This aspect is, e.g., neglected by OSCORE [59] and Lee et al. [57,58]. The first approach does not consider cacheability of messages protected by OSCORE. The latter mechanism encrypts a REST message as a whole inducing so that an intermediate cache system is not able to interpret and store the message.

The body of a REST message is special in this context. In some cases it may contain a resource representation in others it does not. As XML, JSON and CBOR [62] are prevalent data formats for the resource representation in REST-based service systems, such a resource representation might already been encrypted by an according data encryption technology, such as XML encryption [21], JSON web encryption [63] and CBOR encryption [64,65] respectively. Independent of an existing application-controlled resource representation encryption, the REST layer needs to incorporate own mechanisms for ensuring the confidentiality of the body. This is especially important for resource representations which do not include an encryption scheme such as HTML [66], YAML [67] or CSV [68]. As discussed before, the access to the body can then be granted to classes of intermediaries requiring it.

All these aspects will be elaborated in future work in order to develop a REST message confidentiality scheme that is in conformance with the architectural principles and constraints of REST. Combining such a REST-ful message confidentiality with the introduced REST-ful message authentication provides the fundamental layer of the REST-Security stack depicted in Figure 5.

7. Implementation of REST Message Authentication

To proof the proposed conceptual approach, we introduce two distinct instantiations of the general REST message authentication framework REMA, one for HTTP (see Section 7.1) and one for CoAP (see Section 7.2). Both concrete REST-ful message authentication schemes are intended to evaluate the coherent security construction in both distinct protocols and the robustness against the observed vulnerabilities contained in the current state of the art.

7.1. REST-Ful HTTP Message Authentication (REHMA)

This section introduces the REST-ful HTTP instantiation of REMA denoted as REST-ful HTTP message authentication (REHMA). The following table emphasizes the instantiation of the generic REST message signature generation algorithm of Lo Iacono and Nguyen [28] as presented in Section 6.2.2 for HTTP requests and responses. This implementation uses string concatenation to build the string to be signed (*tbs*), which consists of a time-variant parameter (*tvp*), security-relevant header entries, and the hash of the body. Note that for the specific instantiation to HTTP, the delimiter becomes the newline character '\n' and the binary hash value needs to be text-encoded by making use of a Base64URL transformation [69].

tbs **string template for HTTP request**	*tbs* **string template for HTTP response**
tvp + "\n" +	tvp + "\n" +
UpperCase(Method) + "\n" +	UpperCase(Version) + "\n" +
RequestTarget + "\n" +	StatusCode + "\n" +
UpperCase(Version) + "\n" +	LowerCase(Header0) + "\n" +
LowerCase(Header0) + "\n" +	LowerCase(Header1) + "\n" +
...	...
LowerCase(HeaderN) + "\n" +	LowerCase(HeaderN) + "\n" +
Base64URL(hash(Body))	Base64URL(hash(Body))

Assume, that the following example request and response messages require to be authenticated.

Example HTTP request message	**Example HTTP response message**
GET /courses HTTP/1.1	HTTP/1.1 200 OK
Host: example.org	Content-length: 19
Accept: application/json	Content-type: application/json
Connection: keep-alive	Server: Apache
Cache-control: max-age=3600	Connection: keep-alive
	Cache-control: max-age=3600
	Transfer-encoding: gzip
	{"REST":"Security"}

Based on the definitions, rules and policies specified in Lo Iacono and Nguyen [28] and the templates shown in the previous table, the *tbs* strings are constructed for the request message as shown in the left column of the following table respectively for the response message as shown on the right:

tbs **string of example HTTP request message**	*tbs* **string of example HTTP response message**
2014-11-21T15:26:43.483Z	2014-11-21T15:26:45.351Z
GET	HTTP/1.1
/courses	200
HTTP/1.1	max-age=3600
application/json	keep-alive
max-age=3600	19
keep-alive	application/json
example.org	max-age=3600
47DEQpj8HBSa...km5NMpJWZG3hSuFU	gzip
	mIxp6LC6E2cl...zHQQBHU_PI9zWBG8

The elements of the HTTP start line of the request and response respectively are added to *tbs* according to their predefined positions. The security-relevant header values are concatenated in alphabetical order of the header names. Note, that the construction of *tbs* does not include the Server header, since this meta-data is—from an authenticity viewpoint—not a crucial information for the message processing.

The next step encodes the constructed *tbs* to UTF8 and signs the string with a key *k*. Since header entries in HTTP must be text, a transformation of the binary signature value to a text-based equivalent is required. This implementation uses a URL-safe Base64 transformation.

$$sv = \text{Base64URL}(sign(k, \text{UTF8}(tbs)))$$

The final step integrates the resulting text-encoded signature value *sv* along with the corresponding signature meta-data to the newly defined signature header.

Authenticated example HTTP request	**Authenticated example HTTP response**
GET /courses HTTP/1.1	HTTP/1.1 200 OK
Host: example.org	Content-length: 19
Accept: application/json	Content-Type: application/json
Connection: keep-alive	Server: Apache
Cache-control: max-age=3600	Connection: keep-alive
Signature: sig=RSA/SHA256,	Cache-control: max-age=3600
↪hash=SHA256,	**Signature: sig=RSA/SHA256,**
↪kid=https://myid.org/cert,	**↪hash=SHA256,**
↪tvp=2014-11-21T15:26:43.483Z,	**↪kid=https://example.org/crt,**
↪addHeaders=null,	**↪tvp=2014-11-21T15:26:45.351Z,**
↪sigValue=<*sv*>	**↪addHeaders=null,**
	↪sigValue=<*sv*>
	{"REST":"Security"}

Since both messages do not consider additional as well as application-specific headers to be protected by the signature, the addHeaders parameter within the Signature header, contains the value null. If further headers to be signed are required, a list containing the header names separated by a semicolon must be included. REHMA protects all HTTP messages against the attack vectors of Table 1. A Java implementation of REHMA is available at: https://das.th-koeln.de/developments/jrehma.

7.2. REST-Ful CoAP Message Authentication (RECMA)

This section introduces the REST-ful CoAP instantiation of REMA denoted as REST-ful CoAP Message Authentication (RECMA). The following table shows the adoption of the REST-ful message signature algorithm defined in Section 6.2.2 for CoAP. It contains two templates, each constructing a byte concatenation (symbolized by $||$) of a sequence of all security-relevant message elements including a time-variant parameter (tvp). The resulting concatenation is transformed into a byte array that is the tbs variable for CoAP. The implementation considers signing request and response messages as well as acknowledgment ($T = 0 \times 02$) and reset ($T = 0 \times 03$) messages. The template contained in the left column describes the construction rules of tbs for requests as well as responses and the template contained in the right column defines the construction of tbs for reset and acknowledgment messages.

tbs **constructing template for CoAP request and response**	*tbs* **constructing template for CoAP ACK and RST**

`tvp`
`‖Version`
`‖Type`
`‖TokenLength`
`‖Code`
`‖MessageID`
`‖Token`
`‖Options0`
`...`
`‖OptionsN`
`‖hash(Body)`
	`tvp`
	`‖Version`
	`‖Type`
	`‖TokenLength`
	`‖Code`
	`‖MessageID`

Assume that the following two example messages require to be authenticated. The message on the right is a POST request represented by the code value 0×02 ($C = 0 \times 02$), which requires a confirmation by the server, whether the message has successfully been received. Confirmable messages are denoted by the message type value 0×00 ($T = 0 \times 00$). Moreover, the example request contains the protocol version number 0×01 ($V = 0 \times 01$), the message identifier 0×01 ($MID = 0 \times 01$) and the token value $0 \times 0A$ that has a length of one byte ($TKL = 0 \times 01$). The left example message is an acknowledgement message for the POST request specified on the left. It confirms the reception ($T = 0 \times 02$) of a request identified by $MID = 0 \times 01$. The delimiter to separate the header form the body in all CoAP messages is $0 \times FF$.

Example CoAP request message	**Example CoAP ACK message**

`V=0x01,T=0x00,TKL=0x01,C=0x02,MID=0x01`
`Token:  0x0A`
`Uri-path:  0x6974656D73 # items`
`Content-format:  0x32`
`Payload-length:  0xF`
`0xFF`
`{"item":"pork"}`
	`V=0x01,T=0x02,TKL=0x00,C=0x00,MID=0x01`
	`0xFF`

According to the previous table and the requirements defined by Nguyen and Lo Iacono [31] and Nguyen and Lo Iacono [70], the tbs for both messages are constructed as follows:

tbs of example CoAP request message	*tbs* of example CoAP ACK message
`0x14D14486B51` *#tvp*	`0x14D14486B57` *#tvp*
`‖0x01` *#Version*	`‖0x01` *#Version*
`‖0x00` *#Type*	`‖0x02` *#Type*
`‖0x01` *#TokenLength*	`‖0x00` *#TokenLength*
`‖0x02` *#Code*	`‖0x00` *#Code*
`‖0x01` *#Message-ID*	`‖0x01` *#Message-ID*
`‖0x0A` *#Token*	
`‖0x00` *#Uri-Host(3)*	
`‖0x00` *#Uri-Port(7)*	
`‖hash(0x6974656D73)` *#Uri-Path(11)*	
`‖0x32` *#Content-Format(12)*	
`‖0x00` *#Max-Age(14)*	
`‖0x00` *#Uri-Query(15)*	
`‖0x0F` *#Payload-Length (65001)*	
`‖hash(UTF8({“item”:“pork”}))` *#Body*	

The concatenation order of the CoAP start header items and the token follows the order of the predefined positions stated in the CoAP specification. The CoAP options are added in numerical order of the option numbers. Once the construction of *tbs* for the two messages is finished, it is signed with the signature key *k*.

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

The last step incorporates the resulting signature value *sv* and the corresponding signature meta-data to newly introduced CoAP options, which are signature-value, signature-algorithm, hash-algorithm, TVP and a key-ID.

Signed CoAP request message	Signed CoAP ACK message
`V=0x01,T=0x00,TKL=0x01,C=0x02,MID=0x01`	`V=0x01,T=0x02,TKL=0x00,C=0x00,MID=0x01`
`Token: 0x0A`	**`Signature-algorithm: 0x01`**
`Uri-path: “items”`	**`Hash-algorithm: 0x01`**
`Content-format: 0x32`	**`TVP: 0x14D14486B57`**
`Payload-length: 0x0F`	**`Signature-value:`** <*sv*>
`Signature-algorithm: 0x01`	**`Key-Id:`** <*kid*>
`Hash-algorithm: 0x01`	`0xFF`
`TVP: 0x14D14486B51`	
`Signature-value:` <*sv*>	
`Key-ID:` <*kid*>	
`0xFF`	
`{“item”:“pork”}`	

Both messages use numbers to declare the signature and hash algorithm name. Here, the number 0×01 within the signature-algorithm option represents an HMAC-SHA256 signature and the same number defines a SHA256 hash for the hash-algorithm option. A description on the additional and application-specific header entries is not present in both messages, since an acknowledgment must not contain further options besides the options for the signature description and the request does not intend to include additional header entries.

As the CoAP standard does not define a meta-data element for defining the length of the body, RECMA uses the payload-length option to declare the size of the body [71]. Even though this option is not a standardized header, i.e., it is only a proposed draft specification, it is still considered in the authentication process of RECMA to detect attacks that try to forge the body similar to attack four in Table 1. Moreover, RECMA utilizes this option to comply with the self-descriptive messages constraint that requires to be transport independent [1]. RECMA foils all attack vectors presented by Nguyen and Lo Iacono [31]. A Java implementation of RECMA is available at: https://das.th-koeln.de/developments/jrecma.

8. Evaluation and Discussion

The proposed REST message authentication scheme and the requirements defined for implementing REST message confidentiality are the first steps towards a general REST-security framework. The REMA-policy defines mandatory message elements for requests and responses, which need to be available in order to render a message self-descriptive. As these message entities are mandatory, they are also security-critical. Hence message elements defined in the REMA-policy must be signed in order to be protected against malicious modifications. REHMA [28] as well as RECMA [31] apply the REMA-policy as a security-baseline for identifying corresponding mandatory and security-critical HTTP and CoAP message elements respectively. Moreover, the policy on the to be signed message elements in REHMA and RECMA are extended by protocol-specific mandatory header entries which are required for the self-descriptiveness of HTTP and CoAP messages.

The concrete adoption of the REST message authentication scheme in HTTP and CoAP are therefore not vulnerable to the attacks defined in Table 1 as well as the threats detected by the related work analysis, since all essential and security-critical message elements are signed (see Table 4). Note that the to be signed message elements defined in REMA, REHMA and RECMA protect against attacks which are generally-valid for all REST-based applications implemented with HTTP, CoAP other REST-based protocols. If another REST-based protocol or a distinct application domain utilize additional security-critical header entries, these elements must be added either to the policy of the concrete adoption or to the list of application-specific header entries included in *desc* in order to thwart protocol- or application-specific man-in-the-middle attacks.

Table 4 illustrates the message elements signed by REHMA and RECMA. As the REMA-Policy as well as the corresponding adoptions in REHMA and RECMA cover all the to be signed header in order to avoid the documented vulnerabilities (see Section 4, Tables 2 and 3), it can also be utilized as an analytical framework for the evaluation and enhancement of related work in HTTP/CoAP signature schemes. For instance, REHMA may serve as a guideline for adding the missing to be signed message elements of the HTTP signature schemes required by the cloud storage services of Amazon [49], Microsoft [48], Google [46] and HP [47]. The signature schemes of these cloud storage providers will benefit from the security specification of REHMA, as it will increase the level of security. This is especially important as many companies use the cloud storage services of Amazon, Microsoft, Google and HP in production.

This paper shows that a general REST-security scheme builds the basis for generally-valid policies and requirements. By means of this common foundation, REST-ful security technologies can be implemented based on the same security-baseline. This methodology has been conducted for the REST message authentication. The implementation of REMA in HTTP and CoAP shows an increasement of the level of security, as the documented vulnerabilities can be avoided. Other or future REST-based protocols such as RACS [11] can use the same methodology for defining security schemes. An adoption of REHMA in RACS is proposed in [70]. The reader is henceforth referred to this paper for further details.

Table 4. To be signed message elements by REST-ful HTTP message authentication (REHMA) and REST-ful CoAP message authentication (RECMA).

Message Elements to Be Signed	**REHMA [28]**							
	Request				**Response**			
	C	R	U	D	C	R	U	D
URI	●	●	●	●	-	-	-	-
Version number	●	●	●	●	●	●	●	●
Method	●	●	●	●	-	-	-	-
Status code	-	-	-	-	●	●	●	●
Connection	●	●	●	●	●	●	●	●
Cache-control	●	●	●	●	●	●	●	●
Location	-	-	-	-	●	-	-	-
Accept	-	●	-	-	-	●	-	-
Content-type	●	-	●	-	●	-	●	-
Content-length	●	-	●	-	●	-	●	-
Transfer-encoding	●	-	●	-	●	-	●	-
Host	●	-	●	-	-	-	-	-
Hash of body	●	●	●	●	●	●	●	●
Time variant parameter	●	●	●	●	●	●	●	●
Message Elements to Be Signed	**RECMA [31]**							
	Request				**Response**			
	C	R	U	D	C	R	U	D
Version number	●	●	●	●	●	●	●	●
Type	●	●	●	●	●	●	●	●
Token length	●	●	●	●	●	●	●	●
Code (method code, response code)	●	●	●	●	●	●	●	●
Message ID	●	●	●	●	●	●	●	●
Token	●	●	●	●	●	●	●	●
Uri-host, Uri-port, Uri-path	●	●	●	●	-	-	-	-
Max-age	●	●	●	●	●	●	●	●
Location-path, location-query	-	-	-	-	-	-	-	-
Accept	●	●	●	●	-	-	-	-
Content-format	●	●	●	●	●	●	●	●
Body length (payload-length)	●	●	●	●	●	●	●	●
Hash of body	●	●	●	●	●	●	●	●
Time variant parameter	●	●	●	●	●	●	●	●

Legend: ● mandatory signed, ○ not signed, ◐ optionally signed, ⃠ not specified, - not required.

9. Conclusions and Outlook

REST is an established approach for designing distributed applications and service systems that scale at large. This is especially true for the web while other domains are following likewise. At the same time, the areas of adoption increase in criticality, making the need for appropriate security measures a necessity. The application of transport-oriented security is by far not sufficient and needs to be supplemented by adjacent message-oriented security mechanisms. In the latter respect, REST behaves very specific in comparison to existing approaches such as SOAP in the Web Services domain. This renders a straightforward adoption of available schemes and technologies from this domain infeasible. This is due to REST being an abstract architectural style on the one hand, that can be applied to many distinct technologies and environments. On the other hand, the particularities of REST demand for tailored approaches and schemes in order to not contradict with the REST constraints.

The introduced methodology marks an important step towards a structured and controlled procedure for developing appropriate security means for REST-based service systems and applications.

The practical applicability of the introduced methodology has been proven by an adoption of it to authentication. The introduced generic REST message authentication scheme has then been instantiated to the REST-ful protocols HTTP and CoAP. A comparison with the current state of the art revealed that the available technologies are inhomogeneous and contain many vulnerabilities or do not comply with the REST constraints. This further emphasizes the need for a general and methodical approach towards REST-security as has been proposed by this paper. Finally, an initial attempt towards REST message confidentiality is introduced discussing requirements and specifics to be considered while developing the complete picture of a general REST message security framework.

More research and development efforts in REST-ful message security are required in order to reach the necessary understanding of an adequate REST-security defined at the proper abstraction layer while considering the specifics of REST. This is especially essential, since message security for REST-based service systems builds the foundation of many high-level security components (see Figure 5). Moreover, a stable and robust REST-security cannot only set the scene for a mature service security stack, but it can also enhance available REST-based security technologies including OAuth [29] and OpenID connect [72], which still suffer from many vulnerabilities [73,74].

Future work will focus on elaborating the REST-security framework in the light of aspects such as performance and scalability. This includes the cacheablity of protected REST-ful messages. Moreover, concepts that enable intermediate systems transforming signed and encrypted REST messages will be studied as well. This is an important feature in a REST-ful architecture, since transforming the content of a message is an essential property of the layered system constraint.

Author Contributions: Conceptualization, L.L.; methodology, L.L. and H.V.N.; software, H.V.N.; formal Analysis, H.V.N.; investigation, H.V.N.; data curation, H.V.N.; writing—original draft preparation, L.L. and H.V.N. and P.L.G.; writing—review and editing, L.L. and P.L.G.; visualization, H.V.N. and P.L.G.; supervision, L.L.; project administration, L.L.; funding acquisition, L.L.

Funding: This work has been funded by the German Federal Ministry of Education and Research within the funding program "Forschung an Fachhochschulen" (contract no. 13FH016IX6).

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Fielding, R. Architectural Styles and the Design of Network-Based Software Architectures. Ph.D. Thesis, University of California, Irvine, CA, USA, 2000.
2. Fielding, R.; Gettys, J.; Mogul, J.; Frystyk, H.; Masinter, L.; Leach, P.; Berners-Lee, T. *Hypertext Transfer Protocol—HTTP/1.1*; RFC 2616; IETF: Fremont, CA, USA, 1999.
3. Dierks, T.; Rescorla, E. *The Transport Layer Security (TLS) Protocol Version 1.2*; RFC 5246; IETF: Fremont, CA, USA, 2008.
4. Carpenter, B.; Brim, S. *Middleboxes: Taxonomy and Issues*; RFC 3234; IETF: Fremont, CA, USA, 2002.
5. Durumeric, Z.; Ma, Z.; Springall, D.; Barnes, R.; Sullivan, N.; Bursztein, E.; Bailey, M.; Halderman, J.A.; Paxson, V. The Security Impact of HTTPS Interception. In Proceedings of the 24th Network and Distributed Systems Symposium (NDSS), San Diego, CA, USA, 26 February–1 March 2017.
6. Feiler, P.; Sullivan, K.; Wallnau, K.; Gabriel, R.; Goodenough, J.; Linger, R.; Longstaff, T.; Kazman, R.; Klein, M.; Northrop, L.; et al. *Ultra-Large-Scale Systems: The Software Challenge of the Future*; Software Engineering Institute, Carnegie Mellon University: Pittsburgh, PA, USA, 2006.
7. Nadalin, A.; Kaler, C.; Monzillo, R.; Phillip, H.B. *Web Services Security: SOAP Message Security 1.1*; OASIS Standard: Burlington, MA, USA, 2006.
8. Gudgin, M.; Hadley, M.; Mendelsohn, N.; Moreau, J.J.; Nielsen, H.F.; Karmarkar, A.; Lafon, Y. *SOAP Version 1.2 Part 1: Messaging Framework*, 2nd ed.; Recommendation; W3C: Cambridge, MA, USA, 2007.
9. Shelby, Z.; Hartke, K.; Bormann, C. *The Constrained Application Protocol (CoAP)*; RFC 7252 IETF: Fremont, CA, USA, 2014.
10. Rescorla, E.; Modadugu, N. *Datagram Transport Layer Security Version 1.2*; RFC 6347 IETF: Fremont, CA, USA, 2012.
11. Urien, P. *Remote APDU Call Secure (RACS)*; Internet-Draft; IETF: Fremont, CA, USA, 2018.

12. Berners-Lee, T.; Fielding, R.; Masinter, L. *Uniform Resource Identifier (URI): Generic Syntax*; Request for Comments 3986; IETF: Fremont, CA, USA, 2005.
13. Gorski, P.L.; Lo Iacono, L.; Nguyen, H.V.; Torkian, D.B. Service Security Revisited. In Proceedings of the 11th IEEE International Conference on Services Computing (SCC), Anchorage, AK, USA, 27 June–2 July 2014.
14. Lo Iacono, L.; Nguyen, H.V. Towards Conformance Testing of REST-based Web Services. In Proceedings of the 11th International Conference on Web Information Systems and Technologies (WEBIST), Lisbon, Portugal, 20–22 May 2015.
15. Mao, Y.; Yong, L.; Bo, L.; Depeng, J.; Sheng, C. Service-oriented 5G network architecture: An end-to-end software defining approach. *Int. J. Commun. Syst.* **2016**, *29*, 1645–1657. doi:10.1002/dac.2941. [CrossRef]
16. Atzori, L.; Iera, A.; Morabito, G. The Internet of Things: A survey. *Comput. Netw.* **2010**, *54*, 2787–2805. [CrossRef]
17. Bormann, C.; Castellani, A.; Shelby, Z. CoAP: An Application Protocol for Billions of Tiny Internet Nodes. *IEEE Internet Comput.* **2012**, *16*, 62–67. [CrossRef]
18. Kanneganti, R.; Chodavarapu, P. *Soa Security*; Manning Publications Co.: Greenwich, CT, USA, 2008.
19. Gorski, P.L.; Lo Iacono, L.; Nguyen, H.V.; Torkian, D.B. SOA-Readiness of REST. In Proceedings of the 3rd European Conference on Service-Oriented and Cloud Computing (ESOCC), Como, Italy, 12–14 September 2014.
20. Bray, T.; Paoli, J.; Sperberg-McQueen, C.M.; Maler, E.; Yergeau, F. *Extensible Markup Language (XML) 1.0*, 5th ed.; Recommendation; W3C: Cambridge, MA, USA, 2008.
21. Imamura, T.; Dillaway, B.; Simon, E.; Kelvin, Y.; Nyström, M. *XML Encryption Syntax and Processing Version 1.1*; Recommendation; W3C: Cambridge, MA, USA, 2013.
22. Bartel, M.; Boyer, J.; Fox, B.; LaMacchia, B.; Simon, E. *XML Signature Syntax and Processing*, 2nd ed.; Recommendation; W3C: Cambridge, MA, USA, 2008.
23. Nadalin, A.; Goodner, M.; Gudgin, M.; Barbir, A.; Granqvist, H. *WS-Trust 1.3*; OASIS Standard: Burlington, MA, USA, 2007.
24. Goodner, M.; Nadalin, A. *Web Services Federation Language (WS-Federation) Version 1.2*; OASIS Standard: Burlington, MA, USA, 2009.
25. Nadalin, A.; Goodner, M.; Gudgin, M.; Turner, D.; Barbir, A.; Granqvist, H. *WS-SecurityPolicy 1.3*; OASIS Standard: Burlington, MA, USA, 2012.
26. Nadalin, A.; Goodner, M.; Gudgin, M.; Barbir, A.; Granqvist, H. *WS-SecureConversation 1.4*; OASIS Standard: Burlington, MA, USA, 2009.
27. Rosenberg, J.; Remy, D. *Securing Web Services with WS-Security: Demystifying WS-Security, WS-Policy, SAML, XML Signature, and XML Encryption*; Pearson Higher Education: San Francisco, CA, USA, 2004.
28. Lo Iacono, L.; Nguyen, H.V. Authentication Scheme for REST. In Proceedings of the International Conference on Future Network Systems and Security (FNSS), Paris, France, 11–13 June 2015; Springer International Publishing: New York, NY, USA, 2015.
29. Hardt, D. *The OAuth 2.0 Authorization Framework*; RFC 6749; IETF: Fremont, CA, USA, 2012.
30. Hedberg, R.; Solberg, A.; Gulliksson, S.; Jones, M.; Bradley, J. *OpenID Connect Federation 1.0—Draft 07*; Draft; OpenID: San Ramon, CA, USA, 2019.
31. Nguyen, H.V.; Lo Iacono, L. REST-ful CoAP Message Authentication. In Proceedings of the International Workshop on Secure Internet of Things (SIoT), in Conjunction with the European Symposium on Research in Computer Security (ESORICS), Vienna, Austria, 21–25 September 2015.
32. Prokhorenko, V.; Choo, K.K.R.; Ashman, H. Web application protection techniques: A taxonomy. *J. Netw. Comput. Appl.* **2016**, *60*, 95–112. [CrossRef]
33. Reschke, J. *The 'Basic' HTTP Authentication Scheme*; RFC 7617; IETF: Fremont, CA, USA, 2015.
34. Shekh-Yusef, R.; Ahrens, D.; Bremer, S. *HTTP Digest Access Authentication*; RFC 7616; IETF: Fremont, CA, USA, 2015.
35. Nguyen, H.V.; Tolsdorf, J.; Lo Iacono, L. On the Security Expressiveness of REST-based API Definition Languages. In Proceedings of the 14th International Conference On Trust, Privacy and Security In Digital Business (TrustBus), Lyon, France, 30–31 August 2017.
36. Farrell, S.; Hoffman, P.; Thomas, M. *HTTP Origin-Bound Authentication (HOBA)*; Experimental RFC 7486; IETF: Fremont, CA, USA, 2015.

37. Melnikov, A. *Salted Challenge Response HTTP Authentication Mechanism*; Experimental RFC 7804; IETF: Fremont, CA, USA, 2016.
38. Oiwa, Y.; Takagi, H.; Maeda, K.; Hayashi, T.; Ioku, Y. *Mutual Authentication Protocol for HTTP*; Experimental RFC 8120; IETF: Fremont, CA, USA, 2017.
39. Oiwa, Y.; Takagi, H.; Maeda, K.; Hayashi, T.; Ioku, Y. *Mutual Authentication Protocol for HTTP: Cryptographic Algorithms Based on the Key Agreement Mechanism 3 (KAM3)*; Experimental RFC 8121; IETF: Fremont, CA, USA, 2017.
40. De Backere, F.; Hanssens, B.; Heynssens, R.; Houthooft, R.; Zuliani, A.; Verstichel, S.; Dhoedt, B.; De Turck, F. Design of a security mechanism for RESTful Web Service communication through mobile clients. In Proceedings of the IEEE Network Operations and Management Symposium (NOMS), Krakow, Poland, 5–8 May 2014; pp. 1–6.
41. Peng, D.; Li, C.; Huo, H. An extended UsernameToken-based approach for REST-style Web Service Security Authentication. In Proceedings of the 2nd IEEE International Conference on Computer Science and Information Technology, Windsor, ON, Canada, 7–9 June 2009. doi:10.1109/ICCSIT.2009.5234805.
42. Brickely, D.; Miller, L. *FOAF Vocabulary Specification 0.99*; Technical Report; Namespace: London, UK, 2014.
43. Story, H.; Harbulot, B.; Jacobi, I.; Jones, M. FOAF+SSL: RESTful Authentication for the Social Web. In Proceedings of the 6th European Semantic Web Conference, Heraklion, Crete, Greece, 31 May–4 June 2009.
44. Story, H.; Hausenblas, M. *WebID Specifications*; W3C Editor's Draft; W3C: Cambridge, MA, USA, 2013.
45. Khare, R.; Rifkin, A. Weaving a Web of Trust. *World Wide Web J.* **1997**, *2*, 77–112.
46. Google. *Migrating from Amazon S3 to Google Cloud Storage*; Google Inc.: Mountain View, CA, USA, 2017.
47. Hewlett Packard. *HP Helion Public Cloud Object Storage API Specification*; Hewlett Packard: Waltham, MA, USA, 2014.
48. Microsoft. *Authentication for the Azure Storage Services*; Microsoft Research: Redmond, WA, USA, 2017.
49. Amazon. *Signing AWS Requests By Using Signature Version 4*; Amazon Web Service: Seattle, WA, USA, 2017.
50. Cavage, M.; Sporny, M. *Signing HTTP Messages*; Internet-Draft, IETF: Fremont, CA, USA, 2014.
51. Hammer-Lahav, E. *The OAuth 1.0 Protocol*; RFC 5849; IETF: Fremont, CA, USA, 2010.
52. Richer, J.; Mills, W.; Tschofenig, H. *OAuth 2.0 Message Authentication Code (MAC) Tokens*; Internet-Draft; IETF: Fremont, CA, USA, 2014.
53. Richer, J.; Bradley, J.; Tschofenig, H. *A Method for Signing an HTTP Requests for OAuth*; Internet-Draft; IETF: Fremont, CA, USA, 2014.
54. Jones, M.; Bradley, J.; Sakimura, N. *JSON Web Signature (JWS)*; RFC 7515; IETF: Fremont, CA, USA, 2015.
55. Crockford, D. *The Application/Json Media Type for JavaScript Object Notation (JSON)*; RFC 4627; IETF: Fremont, CA, USA, 2006.
56. Serme, G.; De Oliveira, A.S.; Massiera, J.; Roudier, Y. Enabling message security for RESTful services. In Proceedings of the 19th IEEE International Conference on Web Services (ICWS), Honolulu, HI, USA, 24–29 June 2012.
57. Lee, S.; Jo, J.Y.; Kim, Y. A Method for Secure RESTful Web Service. In Proceedings of the IEEE/ACIS 14th International Conference on Computer and Information Science (ICIS), Las Vegas, NV, USA, 28 June–1 July 2015.
58. Lee, S.; Jo, J.Y.; Kim, Y. Authentication system for stateless RESTful Web service. *J. Comput. Methods Sci. Eng.* **2017**. *17*, 21–34. [CrossRef]
59. Selander, G.; Mattson, J.; Palombini, F.; Seitz, L. *Object Security for Constrained RESTful Environments (OSCORE)*; Internet-Draft; IETF: Fremont, CA, USA, 2018.
60. Granjal, J.; Monteiro, E.; Silva, J.S. Application-Layer Security for the WoT: Extending CoAP to Support End-to-End Message Security for Internet-Integrated Sensing Applications. In Proceedings of the 11th International Conference on Wired and Wireless Internet Communications, St. Petersburg, Russia, 5–7 June 2013.
61. Graf, S.; Zholudev, V.; Lewandowski, L.; Waldvogel, M. Hecate, Managing Authorization with RESTful XML. In Proceedings of the 2nd International Workshop on RESTful Design (WS-REST), Hyderabad, India, 28 March 2011; doi:10.1145/1967428.1967442. [CrossRef]
62. Bormann, C.; Hoffman, P. *Concise Binary Object Representation (CBOR)*; RFC 7049; IETF: Fremont, CA, USA, 2013.

63. Jones, M.; Rescorla, E.; Hildebrand, J. *JSON Web Encryption (JWE)*; Internet-draft, IETF: Fremont, CA, USA, 2014.
64. Bormann, C. *Constrained Object Signing and Encryption (COSE)*; Internet-Draft; IETF: Fremont, CA, USA, 2014.
65. Schaad, J. *CBOR Encoded Message Syntax*; Internet-Draft; IETF: Fremont, CA, USA, 2015.
66. Hickson, I.; Berjon, R.; Faulkner, S.; Leithead, T.; Navara, E.D.; Pfeiffer, S. HTML5—A Vocabulary and Associated APIs for HTML and XHTML. 2014. Available online: http://www.w3.org/TR/html5/ (accessed on 19 December 2018).
67. Ben-Kiki, O.; Evans, C.; dot Net, I. YAML Ain't Markup Language Version 1.2. Technical Report. 2009. Available online: http://www.yaml.org/spec/1.2/spec.html (accessed on 19 December 2018).
68. Shafranovich, T. *Common Format and MIME Type for Comma-Separated Values (CSV) Files*; RFC 4180; IETF: Fremont, CA, USA, 2005.
69. Josefsson, S. *The Base16, Base32, and Base64 Data Encodings*; RFC 4648; IETF: Fremont, CA, USA, 2006.
70. Nguyen, H.V.; Lo Iacono, L. RESTful IoT Authentication Protocols. In *Mobile Security and Privacy—Advances, Challenges and Future Research Directions*, 1st ed.; Au, M.H., Choo, K.K.R., Eds.; Advanced Topics in Information Security; Elsevier/Syngress: Boston, MA, USA, 2016; pp. 217–234.
71. Li, K.; Sun, R. *CoAP Payload-Length Option Extension*; Internet-Draft; IETF: Fremont, CA, USA, 2014.
72. Sakimura, N.; Bradley, J.; Jones, M.; de Medeiros, B.; Mortimore, C. *OpenID Connect Core 1.0*; Specification; OpenID Foundation: San Ramon, CA, USA, 2014.
73. Yang, F.; Manoharan, S. A security analysis of the OAuth protocol. In Proceedings of the IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM), Victoria, BC, Canada, 27–29 August 2013.
74. Sun, S.T.; Beznosov, K. The devil is in the (implementation) details: an empirical analysis of OAuth SSO systems. In Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS), Raleigh, NC, USA, 16–18 October 2012.

© 2019 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (http://creativecommons.org/licenses/by/4.0/).