RESTful Is Not Secure

Tetiana Yarygina(✉)

Department of Informatics, University of Bergen, Bergen, Norway
tetiana.yarygina@uib.no

Abstract. The shift in web service design towards the REST paradigm has spawned a series of security concerns. To date there has been no general agreement on how the REST paradigm addresses security and what web security mechanisms adhere to the REST style. This paper analyzes the REST paradigm from a security perspective and shows significant incompatibilities between the style constraints and typical security mechanisms. We conclude that the REST style was not designed with security properties in mind and does not fit the security requirements of modern web applications.

Keywords: Web services security · REST · Stateless · Token authentication

1 Introduction

Web services enable rapid design, development, and deployment of software solutions. They provide a unified web interface and hide complexity and heterogeneity of the underlying infrastructure, enabling simple integration of diverse clients and external components [1]. Unfortunately, the desirable simplicity does not extend to the security aspects of web services.

Representational State Transfer (REST) is an architectural style for web services that is widely adopted. As an architectural style, REST imposes six general design constraints [2]: *client-server*, *stateless resource*, *cacheable responses*, *uniform interface*, *layered*, and *code-on-demand* (optional constraint). These constraints enforce the original concept of the Web as a scalable distributed hypermedia system with loosely coupled components. Web services that strictly adhere to REST style constraints are commonly referred to as RESTful services, while those with loose adherence are often called REST-like services.

It was long believed [3] that RESTful services should be used for ad hoc integration over the Web, whereas Big Web services (see [1] for naming convention) were preferable in enterprise application integration scenarios with longer lifespans and advanced security requirements. However, today we find that more and more corporate solutions, even the most security demanding ones like financial systems and sensitive data operations, are based on RESTful or REST-like services. In contrast to Big Web services, no formal security framework exists for RESTful services.

L. Batten et al. (Eds.): ATIS 2017, CCIS 719, pp. 141–153, 2017.
DOI: 10.1007/978-981-10-5421-1_12

There are relatively few studies of RESTful services security, herein we mention most of them. A recent study by Gorski et al. [4] compares the security stacks of Big Web services and RESTful services. A paper by Lo Iacono and Nguyen [5] compares RESTful authentication mechanisms with focus on message signing. In particular, the authors propose a signing mechanism not limited to HTTP. Finally, two papers describe approaches to message security for RESTful services [6] and secure communication between mobile clients and RESTful services [7]. Although all of these studies claim to deal with RESTful security, they do not discuss the REST architectural style from security perspective.

A much debated question among practitioners is what security mechanisms are truly RESTful. As an example, discussion threads on RESTful authentication[1] and best practices for securing REST APIs[2] are viewed more than 250,000 times each. The introduction of security components often changes system behavior, which can affect how a system adheres to the REST style constraints. To date there has been no general agreement on how the REST paradigm should address security. Apart from Inoue et al. [8], who argued that a session state is not against the REST architectural style, there is a lack of research in the area.

This paper aims to unravel some of the mysteries surrounding RESTful security. We analyze the REST paradigm from a security perspective and show significant incompatibilities between the style constraints and typical security mechanisms. To our knowledge, we are the first to conduct such a detailed security evaluation of the REST style and prove that RESTful security is impossible.

The rest of the paper is organized as follows. In Sect. 2, an overview of common web security mechanisms and a brief discussion of their security merits are given. Section 3 explores in detail how particular security decisions and especially authentication schemes relate to core principles of the REST style. Section 4 concludes the paper by summarizing the uncovered contradictions, discussing the implications of the findings, and providing insights for future research.

2 Overview of Security Mechanisms for the Modern Web

Adequate security mechanisms are needed to build secure RESTful services. This section focuses on common security mechanisms such as Transport Layer Security (TLS), cryptographic objects in JavaScript Object Notation (JSON), token-based authentication, client side request signing, and delegated authorization and shared authentication. The overview creates a background for a more advanced analysis of how common security mechanisms adhere to the REST style constraints.

TLS was originally designed to be independent of any application protocol and has became a de facto security protocol on the Web. Although the design of

[1] https://stackoverflow.com/questions/319530/restful-authentication.
[2] https://stackoverflow.com/questions/7551/best-practices-for-securing-a-rest-api-web-service.

TLS supports mutual authentication, HTTPS in its current form is largely used to authenticate the gateway, but not the client. Even though the idea of both parties maintaining digital certificates is simple and secure, embedding a unique certificate into each client is a serious implementation obstacle. Therefore, client authentication must be provided on the application (message) level.

To provide higher security, as well as client authentication, TLS can be and often is combined with encryption and signing on the message level. Standards for cryptographic objects in JSON and XML were created to address security needs on the message level and to facilitate interoperability. Cryptographic objects can be seen as containers incorporating secured data and the information necessary for its processing. The JSON Object Signing and Encryption (JOSE) suite of specifications offers powerful and flexible building blocks for message security in web services by providing a general approach to signing and encryption of JSON-formatted messages. The JOSE suite is essential for delegated authorization and shared authentication schemes, such as OAuth 2.0 and OpenID Connect (see Fig. 1).

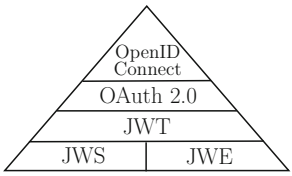

Fig. 1. The hierarchical relation between the JOSE suite, OAuth 2.0, and OpenID Connect. The JOSE suite incorporates JSON Web Signature (JWS), JSON Web Encryption (JWE), JSON Web Token (JWT) [9], and several other specifications.

HTTP is a stateless protocol, which implies that requests are treated independently of each other. Nevertheless, most web applications require sessions. Session management in HTTP is historically performed via HTTP cookies, URL parameters, HTTP body arguments in requests, or custom HTTP headers. A natural extension of session management is client authentication. In modern web applications, there exist two main approaches to authentication: *token-based authentication* and *client side request signing*. The following discussion focuses on the security aspects of these approaches.

Traditionally [10], message authentication methods include Message Authentication Codes (MACs), digital signature schemes, and appending a secret authenticator value before encrypting the whole text. In the context of modern web services, either JWS or XML Signature standards can be used for message authentication depending on the message format. For the sake of simplicity, the term signature is used to refer to both MACs and actual digital signatures.

2.1 Token-Based Authentication

Token-based authentication via HTTP cookies is the most widely adopted authentication mechanism in web applications. The mechanism is based on a notion of security tokens—cryptographic objects containing information relevant for authentication or authorization.

An authentication token is generated by a web service and sent to a client for future use. A service generates a token upon the successful validation of the client's credentials either during the initial user log in or a re-authentication. A token can be seen as a temporary replacement for the client's credentials: every request from a client must include a valid token to be fulfilled. A token-based authentication scheme was first analyzed by Fu et al. in 2001 [11].

Security considerations. Server-created security tokens ensure scalability of the solution and server statelessness by moving the maintenance responsibility for tokens to clients. Additionally, a limited lifetime of security tokens makes them superior to direct use of passwords such as in HTTP Basic/Digest Authentication. A server-side secret used to create tokens is the most important security asset of the server. If the secret is leaked, the damage is not limited to one user: an adversary can impersonate any user of his or her choice.

Hijacking of security tokens is another serious threat. Token-based mechanisms rely on channel confidentiality. If compromised, a security token can be used by an adversary to impersonate the client until the token expires or is revoked. Short expiration time of tokens limits the possible damage, but also reduces usability of a system by requiring frequent user re-authentication.

The severity of security token hijacking is rooted in the static nature of such tokens and their independence of particular requests. Dacosta et al. [12] proposed to switch from static cookies to dynamic ones (request-specific). Channel-binding cookies is another approach to strengthen cookie-based authentication by binding cookies to TLS channels using TLS origin-bound certificates [13]. However, no approach has gained wide adoption mostly due to increased complexity. The evidence presented herein suggests that token-based authentication requires minimal amount of data being stored on the server-side, i.e. contributes to server statelessness, but also has significant security limitations.

2.2 Client Side Request Signing

Many existing RESTful services implement client authentication and in-transit tampering protection by requiring a client to sign each request. Cryptographic keys are established between parties during or after the initial authentication step. Request signing implies signing of an actual message (HTTP payload) and, optionally, HTTP headers.

Request signing involving HTTP headers has been successfully deployed by several major web services such as Amazon Web Services (AWS) [14] and Microsoft Azure [15]. Both are cloud services intended only for programmatic use through REST APIs. An investigation shows that numerous newly developed systems borrow AWS' HMAC-SHA256-based approach to request signing [14].

A comparison of REST message authentication mechanisms based on request signing was performed by Lo Iacono and Nguyen [5]. The paper contributes a detailed HMAC-based scheme for authentication of all types of REST messages, including HTTP messages. A similar, but not as detailed approach to HTTP signing can be found in the IETF draft *Signing HTTP Messages* [16].

Security considerations. Client-signed requests provide stronger authentication than mere token-based schemes. Signing of each client request effectively mitigates session hijacking attacks by limiting damage only to a single request. A signing key never leaves a client which makes stealing the key much more difficult than stealing a token that is not only stored on the client, but also repeatedly sent over the channel. As often happens, higher security comes at a price of lower scalability and higher complexity since a server needs to maintain a separate key for each user.

2.3 Delegated Authorization and Shared Authentication

Delegated authorization and shared authentication have become an integral part of modern web security. The popular security protocols underlying delegated authorization and shared authentication mostly instantiate the token-based authentication introduced earlier. Therefore, they share both advantages and disadvantages of token-based authentication.

Delegated authorization. We consider a scenario where a user, or resource owner, has stored some sensitive information on a server. The desire to separate the login process on the server from the process of granting permissions to a client application on the behalf of the user has stimulated the emergence of OAuth [17]. OAuth is a delegated authorization protocol providing third-party applications (clients) with delegated access to protected resources on behalf of a user (resource owner). Client side request signing in OAuth 1.0 enables client authentication and message integrity, while OAuth 2.0 does not. Developers often fail to implement OAuth correctly due to its ambiguity and complexity [18–20].

Shared authentication. OAuth 2.0 is used as an underlying layer for shared authentication protocols and Single-Sign-On (SSO) systems. Prominent examples are OpenID Connect [21], Facebook Login, and Sign In With Twitter. In such schemes the user authenticates into a third party service (a Relying Party or RP) using a digital identity at an Identity Provider (IdP) of the user's choice. However, additional steps must be taken in order to use OAuth 2.0 for authentication. Security analyses of commercially deployed OAuth-based SSO solutions (i.e. popular social login providers) [20,22] have revealed various security and privacy issues.

3 REST Architectural Style and Security

So far this paper has focused on the security mechanisms commonly used to secure RESTful services. This section elaborates on why none of the systems

using such mechanisms are strictly RESTful by analyzing the REST style and its constrains from a security perspective. It is worth mentioning that the majority of RESTful services actually fail to adhere to REST for reasons unrelated to security. Absence of custom media types support and use of verbs in URIs are common examples of such violations.

The REST architectural style was introduced by Fielding in his influential dissertation [2] and related paper [23] in 2000. The style is widely adopted and many popular web services, such as Twitter[3] and LinkedIn[4], have REST APIs. The dissertation remains the most fundamental source when talking about the core principles of REST.

3.1 Not Designed with Security in Mind

The REST style was proposed as an architectural standard for the Web and introduced only the properties that seemed necessary for the Web at that time. Fielding makes no attempt to address the question of security in REST. The words security, authentication, and authorization are rarely mentioned in Fielding's work. The words encryption and signing do not appear at all.

According to Fielding [2], "REST emphasizes scalability of component interactions, generality of interfaces, independent deployment of components, and intermediary components to reduce interaction latency, *enforce security*, and encapsulate legacy systems." The claim that REST enforces security is neither justified in the dissertation nor explained in any other literature related to REST.

When talking about scalability of the Web, Fielding writes [2, Sect. 4.1.4.1] "since authentication degrades scalability, the architecture's default operation should be limited to actions that do not need trusted data." In modern Web, and especially for REST APIs, the situation is reversed: some form of authentication is always present. TLS is only mentioned as a connector type [2, Sect. 5.2.2], no encryption on the message level is considered.

The REST architectural style does not incorporate security as one of its goals and leaves it up to the developer to decide how security fits the six core principles. The introduction of security components affects system behavior initially shaped by REST constraints. Most of the constraints, such as client-server, uniform interface, and layered system, are high-level and flexible enough to not interfere with adopted security mechanisms. At the same time, the stateless, cacheable, and code-on-demand constraints have several practical security implications. The security implications of the relevant REST constraints are discussed in the following sections.

3.2 Stateless Constraint

Revisiting the definition. The stateless resource constraint is particularly problematic from a security perspective. The constraint is often misunderstood

[3] https://dev.twitter.com/rest/public.
[4] https://developer.linkedin.com/docs/rest-api.

by practitioners and overlooked in the scientific literature. According to Fielding [2, Sect. 5.1.3], for a resource to be stateless "each request from client to server *must* contain all of the information necessary to understand the request, and cannot take advantage of *any stored context on the server*." Such a definition makes no exceptions and, when followed to the letter, leaves no room for security mechanisms.

Furthermore, [2] specifies that the "session state" (also referred to as "application state") should be stored exclusively on the client side; however, a definition of session state is never given. A commonly used interpretation of the stateless resource constraint introduced in [1] differentiates between *application state* and *resource state*. A resource state is defined as any information about the underlying resource [1].

While the resource state belongs to the server, it still can be changed in response to a client request. If we consider a user as a resource, then the balance of the user's bank account is a resource state that is changing with each performed transaction. Similarly, usernames and passwords are also resource states that change over time.

Security implications. Most security components introduce additional resource states. Stateless security protocols do not exist. It is very hard, if at all possible, to prevent replay attacks without maintaining at least some form of client state on the server side. Nonces (numbers used once), counters, and timestamps are examples of such a resource state. All authentication mechanisms described in Sect. 2 incorporate one or more such components. Thus, web services utilizing these mechanisms are not strictly RESTful.

Differentiating between application state and resource state can be difficult. For example, security tokens are stored by the client, but are issued exclusively by the server. The server must maintain the key(s) used to sign tokens, which introduces more resource states.

The demand of "taking no advantage of any stored context on the server" is impractical. For example, a common security practice of restricting the number of login attempts made per specific account relies on the login history being available.

As pointed out by Fielding [2, Sect. 6.3.4.2], HTTP cookies fail to fulfill the stateless constraint of REST. An example of such a violation is the use of cookies to identify a user's "shopping basket" stored on the server, while the basket can be stored on the client side and presented to the server only when the user checks out. This mismatch between REST and HTTP makes a huge part of the modern Web not RESTful and implicitly deprecates cookie-based authentication for RESTful web services.

When token-based mechanisms, such as JWT, OAuth 2.0, and OpenID Connect are used, a server needs $\mathcal{O}(1)$ resource states to authenticate N users [11]. With client request signing as in OAuth 1.0a and AWS, the server needs to maintain a separate key for each client, thus having $\mathcal{O}(N)$ resource states. Therefore, token-based mechanisms can be considered stateless in a sense that there is no per-user or per-session state when compared to client request signing given

a substantial number of clients. Although token-based authentication fits the REST style better then the client side request signing, the latter is generally more secure as explained in Sect. 2.

Additionally, it is possible to classify application state into two classes, security insensitive and security sensitive, that must be treated differently. The server cannot prevent the client from tampering with the data given to it, nor can the server directly protect data stored on a client from malicious third parties. The latter puts user privacy at risk if the data stored is security sensitive.

Even though the definition of the stateless constraint dictates that a client's request must contain all of the information necessary to understand the request, sensitive information should not be transferred unless absolutely necessary. All security sensitive application states must belong to the server and be resource states.

Advantages and disadvantages. To evaluate immediate importance of stateless resource constraint for modern security-aware applications, the advantages and disadvantages of the constraint must be revisited. According to Fielding [2], stateless resource constraint induces the properties of visibility, reliability, and scalability.

The original argument for improved visibility [2] was that the server should process a client request without looking beyond this request. The argument is valid until security is involved. Let us consider an online store. If some items are added to the shopping basket, the only allowed step should be a payment step, and not goods delivery. To ensure this restriction, the user must have state within the system.

Additionally, intrusion detection systems (IDS), anti-denial-of-service, and anomaly detection mechanisms are more likely to mitigate attacks when they have knowledge of the state and the history of requests. If we consider security sensitive data such as authentication tokens, the server unavoidably needs to validate the token, which requires retrieval of the cryptographic key used to generate the token. The step of token verification can also be seen as one that decreases visibility. The aforementioned suggests that improvement of visibility can only be seen for security insensitive data.

The common belief is that maintaining client states on the server side can potentially create a high load of session management and degrade system performance. However, storing clients states on the server side does not cause significant performance problems for existing high load systems and Cloud services; a study of REST session state [8] showed that the impact of the stateless resource constraint on scalability and reliability of REST in the modern Web is insignificant.

Moreover, maintaining client states on the server side is a desired property in many cases, for example personalized services, targeted advertisement, smart suggestion systems, and IDS benefit from it. An alternative solution to scalability and reliability issues is adoption of special software architecture styles, such as microservices [24].

The stateless constraint puts significant limitations on handling session synchronization. In the example with the shopping basket, the problems occur when

the user has initialized a session on a mobile device and wants to continue the session using the browser on a laptop. Storing session state exclusively on the client side and not on the server makes it impossible to keep persistent state in such situations. Hence, current demand for client state synchronization negates the stateless resource constraint of REST.

3.3 Other Constraints Affecting Security

Cache constraint. The cachebility constraint is affecting security much less than the stateless criteria, but the effect is still noteworthy. The definition of the constraint [2] states that the server responses must be explicitly marked as cacheable or noncacheable. Of course, only actual caching of responses improves scalability and network efficiency by eliminating identical repeating interactions. Caching of server responses can be performed by intermediates, i.e. proxies and gateways, or clients themselves.

Caching by intermediates has less value on the modern Web due to an increasing amount of encrypted traffic such as HTTPS traffic. As of February 2017, 52.8% of the most popular websites implemented HTTPS [25]. When encrypted either by TLS or on the message level, server responses are not cacheable by intermediate proxies. Encrypted content cannot be cached unless the intermediates are allowed to decrypt the traffic, which defeats the purpose of encryption in the first place.

Although caching by clients is not affected by encryption, it loses its importance due to different reasons. Modern websites include large amounts of dynamic personalized content that cannot and should not be cached. In case of online banking or online shopping the content (the bank account balance or availability of specific items in the shopping basket) is dynamic and gets outdated fast. Such content is not suitable for caching due to reliability reasons. Similarly, sensitive content should never be cached for security reasons.

Taken together, encryption and personalized content dramatically reduce the benefits of traditional web caching in general, and the importance of cache constraint of the REST style in particular. While the content marked as noncacheable does not contradict the definition of the cache constraint (since the constraint only requires proper labeling), it brings no actual benefit in terms of scalability or network efficiency.

Code-on-demand constraint. In the code-on-demand paradigm the code for a specific task is requested by the client, provided by a server, and executed in the client's context. As argued in [2], the code-on-demand constraint of REST improves system extensibility, but also reduces visibility. Therefore, it is only an optional constraint.

It should be noted that the code-on-demand constraint is relevant primarily within the browser environment. In semantic web with machine-to-machine communication and native clients consuming REST APIs, execution of external JavaScript code in the native applications is currently uncommon.

An important security implication of the code-on-demand paradigm is an increased attack surface on a client. Among the major security concerns are authenticity of the received code and the client's ability to limit the behavior of the code. These problems have been studied for a long time and mitigation techniques, such as sandboxing, Address Space Layout Randomisation (ASLR), and Data Execution Prevention (DEP), are implemented in modern browsers. However, the problems still persist.

4 The Way Forward

4.1 Security Failure of REST

The main goal of this paper was to asses how the REST style addresses security and whether security mechanisms adhere to the style constraints. The study has shown that the REST style fails to take security into account, or to explain security implications of the constraints. To fill the gap, we provided the missing security interpretation of the relevant style constraints and made the following observations:

- *Stateless resource constraint.* The more security critical a system is, the more resource states it is likely to have. Among authentication approaches, token-based authentication most closely fits the stateless resource constraint. However, it is not entirely stateless.
- *Cache constraint.* Although formally the cache constraint (labeling of responses) is not directly affected by security mechanisms, the constraint loses its meaning for security critical systems. Encrypted, dynamic, and personalized content is not suitable for caching.
- *Code-on-demand constraint.* The optional code-on-demand constraint reduces security of the system by increasing the attack surface on the client side.

To be strictly RESTful and follow all the constraints as they were originally defined, a system should neither deploy authentication nor store session identifiers in HTTP cookies or headers. Since only the absence of security mechanisms allows an entity to provide truly RESTful APIs, a bank claiming to have RESTful APIs either has serious security problems or the APIs do not satisfy all the RESTful requirements.

An important finding is that the concept of RESTful security is impossible. We conclude that the strict REST style on one side and security mechanisms and security best practices on the other side are incompatible. We suggest that secure applications trying to adhere to the REST style should never be called RESTful, but REST-like, i.e. partially adhering to the REST style constraints. Although the term REST-like does appear in some security specifications, such as OpenID Connect [21], it has never been justified from a security perspective.

4.2 What to Do

The right security approach is system-specific and heavily dependent on the context. In particular, the frameworks OAuth 2.0 and OpenID Connect rely on TLS for confidentiality, integrity, and server authentication. These frameworks prioritize scalability over security because they use server signed tokens for client authentication. The overall conclusion from the analysis is that systems with high security requirements should deploy client signatures, even though it comes with the cost of reduced performance when compared to token-based approaches. Social login solutions are both easy to support and convenient for users, but should be avoided if privacy is a serious concern. OAuth should not be relied on for authentication and needs to be combined with a component for authentication. Figure 2 contains a flow chart showing how to choose the correct security architecture.

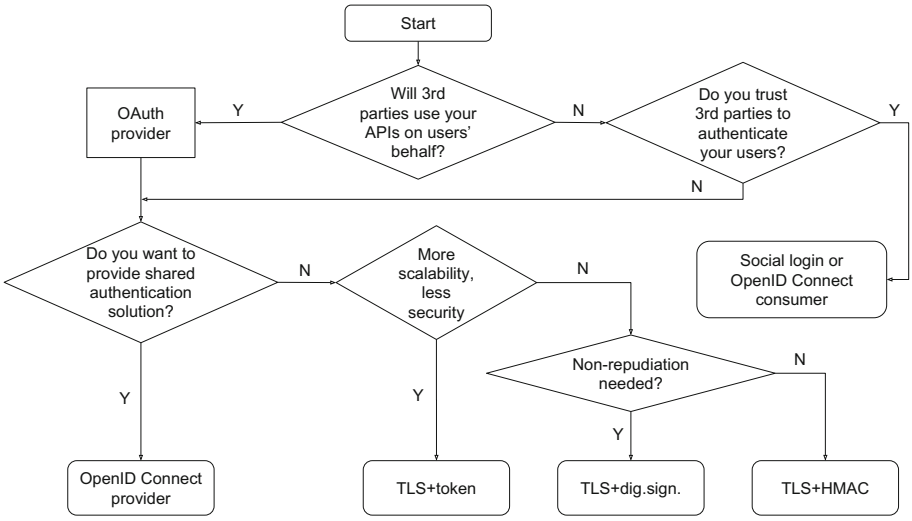

Fig. 2. Making the right security decision

4.3 Future Research

Inoue et al. [8] introduced an architectural style called RESTUS, which incorporates session state at the server-side as an additional constraint. RESTUS partially addresses the security issues of the stateless resource constraint, but not the issues related to the cache and code-on-demand constraints. Similarly to REST, it does not accommodate security. Future research should therefore concentrate on resolving the existing conflicts. A natural progression of this work is to propose an architectural style that incorporates basic security principles.

References

1. Richardson, L., Ruby, S.: RESTful Web Services. O'Reilly Media, Sebastopol (2007)
2. Fielding, R.T.: Architectural Styles and the Design of Network-based Software Architectures. Ph.D. thesis, University of California, Irvine (2000)
3. Pautasso, C., Zimmermann, O., Leymann, F.: RESTful web services vs. big web services: making the right architectural decision. In: 17th International World Wide Web Conference (WWW 2008), Beijing, China, pp. 805–814 (2008)
4. Gorski, P., Lo Iacono, L., Nguyen, H., Torkian, D.: Service security revisited. In: IEEE International Conference on Services Computing, pp. 464–471. IEEE Computer Society, Washington, DC (2014)
5. Lo Iacono, L., Nguyen, H.: Authentication scheme for REST. In: International Conference on Future Network Systems and Security, pp. 113–128 (2015)
6. Serme, G., de Oliveira, A., Massiera, J., Roudier, Y.: Enabling message security for RESTful services. In: IEEE 19th International Conference on Web Services, pp. 114–121. IEEE Computer Society, Washington, DC (2012)
7. 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: IEEE Network Operations and Management Symposium, pp. 1–6. IEEE, Krakow (2014)
8. Inoue, T., Asakura, H., Sato, H., Takahashi, N.: Key roles of session state: not against REST architectural style. In: IEEE 34th Computer Software and Applications Conference, pp. 171–178. IEEE (2010)
9. Jones, M., Bradley, J., Sakimura, N.: RFC 7519. JSON Web Token (2015)
10. Menezes, A.J., van Oorschot, P.C., Vanstone, S.A.: Handbook of Applied Cryptography. CRC Press, Boca Raton (1996)
11. Fu, K., Sit, E., Smith, K., Feamster, N.: The dos and don'ts of client authentication on the Web. In: USENIX Security Symposium, pp. 251–268 (2001)
12. Dacosta, I., Chakradeo, S., Ahamad, M., Traynor, P.: One-time cookies: preventing session hijacking attacks with stateless authentication tokens. ACM Trans. Internet Technol. **12**(1), 1:1–1:24 (2012)
13. Dietz, M., Czeskis, A., Balfanz, D., Wallach, D.S.: Origin-bound certificates: a fresh approach to strong client authentication for the web. In: 21st USENIX Security Symposium, pp. 317–331. USENIX, Bellevue, WA (2012)
14. Amazon S3: Authenticating requests (AWS Signature v4). https://docs.aws.amazon.com/AmazonS3/latest/API/sig-v4-authenticating-requests.html
15. Microsoft Azure documentation: Authentication for the Azure Storage Services (2015). https://msdn.microsoft.com/en-us/library/dd179428.aspx
16. Cavage, M., Sporny, M.: IETF draft. Signing HTTP messages (2015)
17. Hammer-Lahav, E.: RFC 5849. The OAuth 1.0 protocol (2010)
18. Chen, E., Pei, Y., Chen, S., Tian, Y., Kotcher, R., Tague, P.: OAuth demystified for mobile application developers. In: ACM SIGSAC Conference on Computer and Communications Security, pp. 892–903. ACM, New York (2014)
19. Wang, R., Zhou, Y., Chen, S., Qadeer, S., Evans, D., Gurevich, Y.: Explicating SDKs: uncovering assumptions underlying secure authentication and authorization. In: 22nd USENIX Security Symposium, pp. 399–314. Washington, DC (2013)
20. Sun, S.T., Beznosov, K.: The devil is in the (implementation) details: an empirical analysis of OAuth SSO systems. In: ACM Conference on Computer and Communications Security, pp. 378–390. ACM, New York (2012)

21. Sakimura, N., Bradley, J., Jones, M., de Medeiros, B., Mortimore, C.: OpenID Connect Core 1.0 (2014)
22. Wang, R., Chen, S., Wang, X.: Signing me onto your accounts through Facebook and Google: a traffic-guided security study of commercially deployed single-sign-on web services. In: IEEE Symposium on Security and Privacy, pp. 365–379. IEEE Computer Society, Washington, DC (2012)
23. Fielding, R.T., Taylor, R.N.: Principled design of the modern web architecture, pp. 407–416, June 2000
24. Fetzer, C.: Building critical applications using microservices. IEEE Secur. Priv. **14**(6), 86–89 (2016)
25. Trustworthy Internet Movement: SSL Pulse (2017). https://www.trustworthyinternet.org/ssl-pulse/