Design of a Security Mechanism for RESTful Web Service Communication through Mobile Clients

Femke De Backere, Brecht Hanssens, Ruben Heynssens, Rein Houthooft, Alexander Zuliani,
Stijn Verstichel, Bart Dhoedt and Filip De Turck
Information Technology Department (INTEC), Ghent University - iMinds,
Gaston Crommenlaan 8, bus 201, 9050 Ghent, Belgium
Email: Femke.DeBackere@intec.UGent.be

***Abstract*—Security is not taken into account by default in the Representational State Transfer (REST) architecture, but its layered architecture provides many opportunities for implementing it. In this paper, a security mechanism for Web service communication through mobile clients devices is proposed, that conforms to the REST architecture as much as possible. Results indicate that the custom security mechanism outperforms the Transport Layered Security (TLS) based system. Because of the genericness of REST, the proposed security mechanism can be adopted by a wide variety of other RESTful Web services.**

I. Introduction

Representational State Transfer (REST) [1] is an architectural pattern, specifically tailored to building applications and Web Services that are distributed over the public Internet. Resources within the REST architecture can be manipulated through a set of unique Uniform Resource Identifiers (URIs). By sending universal Hypertext Transfer Protocol (HTTP) methods to these URIs, certain actions on the resources they represent can be performed. HTTP status codes are then used to send feedback to the client. These actions are already provided by HTTP, accelerating the adoption of REST [2]. The use of these URIs to transfer important data may result in privacy breaches as data is not anonymized.

As Web Services conforming to the REST constraints are based on HTTP, they suffer from the same inconveniences as many other standard web applications. The following malicious activities: Man-in-the-Middle attack (MITM), replay attack, spoofing and message altering, need to be taken into account when designing a good security mechanism. RESTful Web services are stateless. This means that every request is only dependent on itself, one simply has to examine the request to gather all the details concerning it. Stateless also means that the service is more reliable, because there are less steps where something can go wrong. In distributed services, the lack of state means that there is no overhead to keep the different servers consistent. A downside, however, is that data tends to be sent in a rather repetitive way, because no history is saved, possibly resulting in a larger overhead than typical stateful alternatives. The notion of state in larger applications is often very present, making communication in a stateless way a non-trivial task.

REST is often treated as an alternative to Simple Object Access Protocol (SOAP), although both cannot be directly compared. REST is an architecture for building (web) applications, whereas SOAP is a protocol for exchanging structured information between services and applications. SOAP is commonly used in today's Web Services, and uses Web Services Security (WSS) [3] to cope with the aspect of security.

This paper introduces a security mechanism, using only a bare minimum of non-RESTful elements, keeping the lightweight character of mobile clients in mind, which means preserving battery power and limiting data transfer and message overhead. The suggested implementation is then compared to a fully TLS-based solution. The remainder of this paper is organized as follows. Related work of existing security mechanisms can be found in Section II. Section III will explain the custom security mechanism, followed by evaluation results in Section IV. Finally, conclusions can be found in Section V.

II. Related Work

In Table I, a comparison is made between known security mechanisms, keeping in mind their suitability in a RESTful architecture. Nowadays, the capabilities and restrictions of smartphones need to be taken into account. Security mechanisms thus need to require low processing power to support the battery lifetime and the ability to function in varying network circumstances with respect to mobile interaction.

As RESTful Web services are stateless, they do not usually have any kind of session, in which to perform a challenge-response mechanism. Popular known mechanisms such as Open Authorization (OAuth) [4] and OpenID [5] therefore violate the strict principles of a RESTful architecture, simply because they are stateful.

Transport Layer Security/Secure Sockets Layer (TLS/SSL) provides secure peer-to-peer authentication, but this mechanism is inadequate when requests for authentication are based on delegation, allowing sites to authenticate on behalf of their users [6]. HTTP Secure (HTTPS) is widely used for confidentiality but it only provides hop-to-hop security.

A good solution, obeying the RESTful principles, is a token-based approach [7]. This approach lets the service generate a token on the first request, when users enter their user name and password. The token is then sent to the client, who will add it to every further request to access a certain REST resource. This token should not be bound to any data, as it is merely a substitution for a user name and a password. Once a token has

TABLE I
COMPARISON OF KNOWN SECURITY MECHANISMS

Open Authorization (OAuth) [4]	
Pros: Purpose is to grant other systems access to one's account without sharing user name/password. Done by sharing authentication tokens. Tokens can be revoked at any time.	**Cons**: Relies on sessions in which client information must be stored at the server-side in order to authorize them, not stateless.
OpenID [5]	
Pros: General authentication service for multiple web applications, e.g. Google. Use of authentication tokens for authentication.	**Cons**: Third party service trusted for all authentication purposes. Authentication by URI, users needs to remember this URI, hence not stateless.
Transport Layer Security (TLS/SSL) [6]	
Pros: Can cope with message-sniffing & -altering and MITM- & replay-attacks. Data is fully encrypted. Client and server authenticate using a digital certificate. Use of new session keys with every connection makes it resistant to offline key cracking.	**Cons**: Use of certificates is hard to manage and binds users to a specific device. Requires connection to stay alive. Huge overhead with each connection setup due to handshaking process. REST requires new connection and session keys with each client/server interaction.
Token-based authentication (sessions) [7]	
Pros: Standard authentication principle used by many Web services, applications and sites. Sessions are easy to manage.	**Cons**: Login details only have to be sent once, whereupon a session is set up. Malicious client can take over communication when the token is obtained.
HTTP Digest Authentication Scheme (DAS) [8]	
Pros: Extension of HTTP BAS. Relatively low processing needed. User name/password sent with nonce after hashing, can only be retrieved by inverse transformation. Use of nonce to withstand chosen plaintext and replay attacks.	**Cons**: Server chooses reduced security if client does not support latest technology. Can be avoided by disallowing server to reduce its security level. Servers do not authenticate to clients, making it vulnerable to MITM-attacks. MD5, used as hash function is outdated.

been obtained the user can offer this token to the remote site, which guarantees them access to a specific REST resource for a certain amount of time. When the token is obtained by a malicious client, it can take over communication with the RESTful Web Service.

III. PROPOSED SECURITY MECHANISM

In most existing mechanisms, a lot of the security elements used are stateful, and thus not conform to the RESTful principles. In the presented system, only the most essential non-RESTful elements are added. Whenever a non-RESTful element is added, a motivation is given concerning the reason why. The constructed system makes use of different cryptographic functions.

A. Security requirements

In many applications, four different types of client authentication for REST resource access can be distinguished: i) public, ii) proof of previous login is required, iii) direct login details are necessary and iv) an offline challenge-response system is recommended.

Since the emphasis in this paper is on a lightweight solution for mobile clients, where using battery power and data transfer is kept to a bare minimum, a distinction has to be made concerning the need for message confidentiality. Not every message should be encrypted, e.g., public GET requests can be sent over in plain text. However, more critical messages can require an encrypted connection to thwart packet sniffing. It can be important that a message is not-modifiable, e.g., an adversary should not be able to change the date of a newly created entry by altering the POST message contents. Moreover, it can be important to avoid replayed messages. For example, a replayed delete request should be noticed. However, not every message should be protected against replay attacks, as for example, a regular GET request does not alter a REST resource.

B. Login mechanism

The login mechanism used in the designed system is shown in Algorithm 1. Users authorize themselves by the combination of a user name (`un`) and a password (`pw`). The back-end server will store this `un` and a hashed version of the `pw` combined with a salt: `H(H(pw)+salt)`. By only sending and storing a hashed version of the `pw`, the server can never leak the passwords of users, even in case of a hack. Adding a `salt` prevents an adversary (in case of a cracked database) from performing an offline *brute force* or *dictionary* attack on different passwords simultaneously. This approach also greatly decreases the effectiveness of *rainbow tables* to reverse hashes. The server should thus store at least the following data: `un`, `H(H(pw)+salt)`, `salt`. Although storing user-bound data is not conform to the REST principles, it is essential for a strong security system.

The server authorizes to the user by using a digital certificate. When downloading the application, the certificate is downloaded as well. This certificate is signed by a certificate authority (CA), ensuring its correctness. It can be checked by verifying the certificate chain up to the root CA. When the client connects to the server, he/she will provide his/her `un` and `pw`. This combination cannot be sent in plain text for security reasons and the server has to be trusted first. That is why during the login phase a TLS connection will be used. The client sends `un/H(pw)`, this way his/her password never leaves the client system in plain text. By using a TLS connection, replay and offline guessing attacks can be avoided.

Upon receiving the combination of `(un, H(pw))`, the server will calculate `H(H(pw)+salt)` and compare it to the value stored in the database. After this, the server will calculate an authentication token `AT` and a symmetric key `SK`. It also binds an expiry date `expDate` to this token. The `AT` can be seen as RESTful, it is simply a surrogate for a `un` and `pw`. The `SK` and the `expDate` however are not. Either way, the `SK` is essential for providing data integrity and the

Algorithm 1: User login

input : user fills in details and presses login button
output : user is logged in

```
1  C ↔ S : setupTLSConnection(serverCertificate)
2  C     : TS_c ← getCurrentTime()
3  C     : hash_c ← H (pw)
4  C → S : sendLoginDetails(un, hash_c, TS_c)
5      S : hash_salt_s, salt ← lookupInDB(un)
6      S : verify(H(hash_c + salt), hash_salt_s)
7      S : prevTS_c ← TS_c
8      S : AT ← calcAuthToken()
9      S : SK ← calcSecretKey()
10     S : TS_s ← getCurrentTime()
11     S : saveExpDateInDB(TS_s + T)
12 C ← S : send(AT, SK, TS_s)
13 C     : prevTS_s ← TS_s
```

expDate is needed to close the session when a user forgets to logout or his/her AT/SK gets stolen. Now, the database holds the following data: un, H(H(pw)+salt), salt, SK, AT, expDate. When the expDate is reached, the AT/SK is destroyed. This prevents other people to keep using this AT/SK combination when using a public device where a previous user forgot to logout. After this the AT/SK is sent to the client over the TLS connection, after which the connection is shut down.

C. REST resource access

The user-resource interaction mechanism used in the system is shown in Algorithm 2. When a client wishes to access a REST resource, he/she will add a timestamp TS and his/her AT to his/her request. The total message will be signed by a HMAC using the SK. A Hash-based Message Authentication Code (HMAC) provides message integrity and authentication. Because the user never has to sent his/her un again, this provides an extra layer of security. The AT will be thrown away after a time expDate, thus it is not as critical as a permanent user name. By making use of an AT, the un and pw do not have to be stored on the (possibly public) client system.

On receiving this message, the server will look up the correct database entry by using the AT. Then, he/she will calculate the HMAC using the SK stored. He/she will also check if the expDate has not been reached and that the TS of the latest message is higher than the previous one. Because of the use of a TS, replay attacks are impossible. It thus ensures message freshness. However, using a user-bound timestamp is not RESTful. An adversary can never forge a request as he/she does not possess the SK to calculate the HMAC. This provides protection to offline guessing attacks. A request can also never be modified as a hash function is being used.

The server will add a TS to its own response to the client to ensure message freshness. The server can either encrypt the data by using the SK with symmetric encryption (or make use of TLS), or simply not encrypt the data at all. Either way, the server should authenticate its message by signing it with his/her own private key (the one used to sign the digital certificate). Alternatively, it can use the same HMAC as the client to authenticate itself. Upon acquisition of this response the client will check the TS to the last saved TS. If it is higher, the message passes the first check. Then the digital signature is checked using the certificate stored in the application. If this passes the test, the message is considered safe and thus accepted. Alternatively, instead of timestamps, nonces can be used. However, this requires extra message overhead as these nonces have to be sent back and forth at every request/response. Nevertheless, it does eliminate the requirement of clock synchronization between client and server, which is essential for loose coupling. When logging out, the user does not send a TS, only the SK is needed. When logged out the same thing happens as if the expDate was reached. The AT/SK are destroyed at server-side. If a user logs in on a different system, the old AT/SK combination is

Algorithm 2: User-resource interaction

input : user is logged in, possesses SK and AT
output : user receives resource representation

```
1  C     : HTTPReq ← createHTTPReq()
2  C     : TS_c ← getCurrentTime()
3  C     : HTTPReq.add(AT, TS_c)
4  C     : HMAC ← HMAC(HTTPReq.body, SK)
5  C     : HTTPReq.add(HMAC)
6  C     : symEncrypt(HTTPReq.body, SK)
7  C → S : send(HTTPReq)
8      S : AT ← HTTPReq.get(AT)
9      S : expDate, SK, prevTS_c ← lookupInDB(AT)
10     S : symDecrypt(HTTPReq.body, SK)
11     S : verify(HTTPReq.TS_c > prevTS_c)
12     S : verify(currentTime < expDate)
13     S : HMAC_c ← remove(HTTPReq.HMAC)
14     S : HMAC_s ← HMAC(HTTPReq.body, SK)
15     S : verify(HMAC_s = HMAC_c)
16     S : prevTS_c ← HTTPReq.TS_c
17     S : HTTPResp ← createHTTPResp()
18     S : resource ← retrieveResource()
19     S : TS_s ← getCurrentTime()
20     S : HTTPResp.add(resource, TS_s)
21     S : sign ← digSign(HTTPResp.body)
22     S : HTTPResp.add(sign)
23     S : symEncrypt(HTTPReq.body, SK)
24 C ← S : send(HTTPResp)
25 C     : symDecrypt(HTTPReq.body, SK)
26 C     : verify(HTTPResp.TS_s > prevTS_s)
27 C     : sign ← remove (HTTPResp.sign)
28 C     : verify(HTTPResp.body, sign, PK_s)
29 C     : prevTS_s ← HTTPResp.TS_s
```

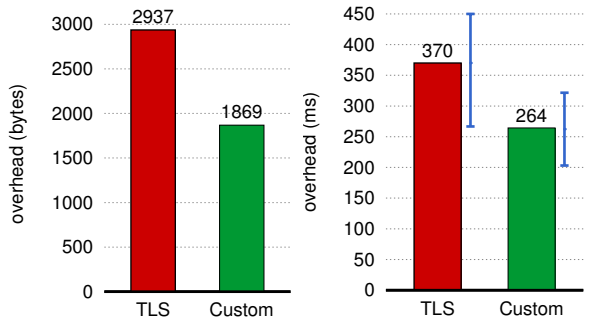

Fig. 1. Messaging overhead (left) and processing overhead (right).

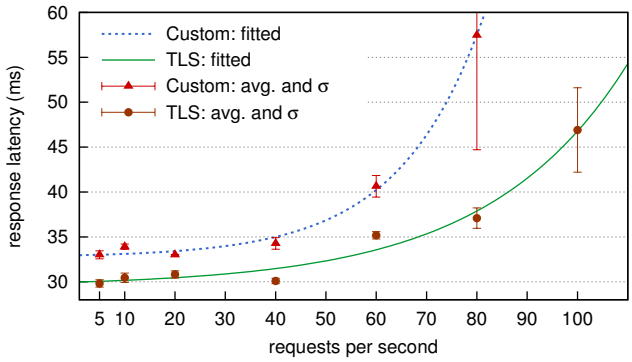

Fig. 2. Scalability: response latency as a function of request rate

also destroyed. This message may be replayed as a logout is only possible once. If a user forgets to log out, the next user of the (possibly public) device can only capture the `AT/SK`. The `un/pw` are not stored in the application. This `AT/SK` are only usable for a limited period of time. However, much damage can be done in this short period of time. This is why critical functions require the use of the `un/H(pw)` of the user.

The symmetric encryption process on lines 6, 10, 23 and 25 in Algorithm 2 is optional and should only be used when data confidentiality is required. It can be switched on and off dynamically based on the communication context.

IV. Experimental Setup and Results

To evaluate this security mechanism, it is applied to the login procedure of a prototype of a location-aware social network. Then, it is compared with a system that is fully TLS-based. This TLS-based system simply requires a user to login after which an authentication token is returned, which is added to every message. After this, every REST resource access is tunneled through a TLS link, which is a strong way to secure RESTful Web services.

The two mechanisms are compared based on their message and execution overhead in Figure 1. The client used for these measurements was a 2.66 GHz Intel Core 2 Duo with 4 GB RAM laptop. The test application was deployed on Google App Engine (GAE). For the messaging tests Wireshark has been used to compare the size of the IP-packets. The results indicate that the custom security system performs 36.4% better in terms of messaging overhead and 28.6% better in terms of processing overhead. Also the ratio of the standard deviation to the average value is lower for the custom built system (22.6%) compared to the TLS-based system (27.9%).

The scalability of the proposed mechanism was also investigated. The Virtual Wall at iMinds was used in combination with the Spirent Avalanche framework to simulate a large number of users simultaneously. For the acquisition of this data, the request rate was steadily increased until the desired level and waited until the response time, averaged over a four second interval, was stabilized. The actual response time average and standard deviation were calculated over a 40-second interval. Figure 2 shows a large increase in deviation as the load increases, this is caused by the development server provided by Google, which is not meant for production use and as such it does not fairly schedule a large amount of simultaneous requests. The custom system started failing (dropping a high number of requests) around 100 requests per second. The TLS-based system lasted until around 140 requests per second. On GAE, this would not happen because of the automatic scaling provided for both the application and the database. These results remain relevant because it gives an idea of how both mechanisms compare under stress in terms of computational overhead. It is obvious the custom built system contains less unnecessary overhead data. The system is also faster than the TLS system, because TLS has to perform a handshake and generate keys for every request.

V. Conclusions

The aspect of security remains a big issue for RESTful Web services. Many of the current security mechanisms violate the RESTful principles and are, because of their stateful nature, not able to cope with the scalability advantages that REST provides. Basic RESTful security standards are outdated and omit vital security solutions. TLS seems to be a usable standard, nonetheless, the overhead introduced is too large for non-continuous connections, as with mobile interaction. Therefore, a custom security mechanism is proposed, using only a bare minimum of non-RESTful elements. Comparing this implementation with a fully TLS-based solution shows that this method outperforms the latter, both on the aspect of messaging as processing overhead. Because of the genericness of REST, our proposed security mechanism can be adopted by a wide variety of other RESTful Web services.

References

[1] R. T. Fielding, "Architectural Styles and the Design of Network-based Software Architectures," Ph.D. dissertation, Uni. of Cal., Irvine, 2000.
[2] G. Serme *et al.*, "Enabling Message Security for RESTful Services," in *Proc. 19th IEEE ICWS*, Jun. 2012, pp. 114–121.
[3] M. Naedele, "Standards for XML and web services security," *Computer*, vol. 36, no. 4, pp. 96–98, Apr. 2003.
[4] H. Eran, "The OAuth 1.0 Protocol," Internet Requests for Comment, RFC Editor, Fremont, CA, USA, Tech. Rep. 5849, Apr. 2010.
[5] D. Recordon *et al.*, "OpenID 2.0: A Platform for User-Centric Identity Management," in *Proc. 2nd ACM DIM*, 2006, pp. 11–16.
[6] R. Malisetti, "Securing RESTful Services with Token-Based Authentication," *CA Technology Exchange*, vol. 1, pp. 43–48, Apr. 2011.
[7] A. Renner, "RESTful Web Services," *Fort Lewis College*, 2008.
[8] D. Peng *et al.*, "An extended UsernameToken-based approach for REST-style Web Service Security Authentication," in *Proc. 2nd IEEE ICCSIT*, Aug. 2009, pp. 582–586.