Security Considerations in Implementing Robust Stateless APIs

Muhtar Hanif Alhassan
Department of Computer Science
National Open University of Nigeria
Abuja, Nigeria

Abstract:- This paper focuses on the salient security implications of the inherent statelessness that characterises Representational state Transfer (REST) APIs. The ever rising popularity of the REST architecture has resulted in its widespread application in the development and deployment of web services for implementing efficient enterprise solutions. REST techniques enable applications to share resources of a variety of nature seamlessly over the web thus making it possible to provide integration and interoperability. The REST API technique does not require storage of complex states in memory and is also characterised by simple logical structures. It is this inherent simplicity that leads to issues with security that will be covered in this paper.

Keywords:- *Component; Formatting; Style; Styling; Insert.*

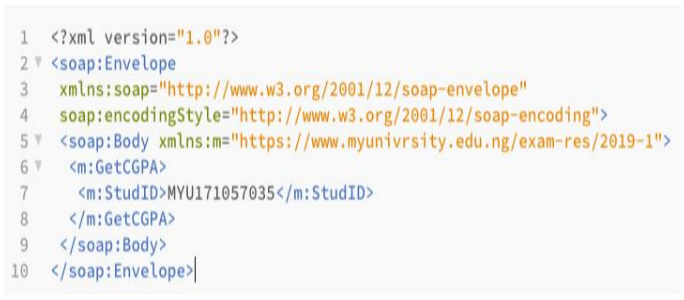
I. INTRODUCTION

Web applications are increasingly dominating the ICT enterprise solution arena for a few reasons. These applications provide platform independence, enhanced interoperability, and high levels of integration through application programming interfaces (API). Thus, there is a growing tendency to replace desktop software with Web based solutions in supporting modern enterprise information systems. As we brace ourselves to embark on inevitable digital transformation in Nigeria there is the need to support resource exchange among stakeholders so as to avoid duplication of efforts and ensure optimum utilization of resources.

The REST technique provides a set of rules that define how resources are exchanged in a distributed system. REST-based APIs enable mobile devices, web browsers, web servers, and other hosts to create, read, update and delete resources according to the REST rules. When conforming to statelessness, no state of the client session is ever stored on the server side. This implies that each client request must contain the total information needed to process it.

Statelessness provides significant advantages some of which are

- *Enhanced Scalability* The APIs can be scaled to millions of concurrent users by deployment on multiple servers. Any server can handle any request because there is no inherent session related dependency.
- *Simple Implementation* REST APIs are less complex because there is no need for server-side state synchronization logic.



```
<?xml version="1.0"?>
<soap:Envelope
 xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
 soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
 <soap:Body xmlns:m="https://www.myunivrsity.edu.ng/exam-res/2019-1">
  <m:GetCGPA>
   <m:StudID>MYU171057035</m:StudID>
  </m:GetCGPA>
 </soap:Body>
</soap:Envelope>
```

Figure 1 SOAP Message Structure

II. WEB SERVICES

Generally, a web service is a piece of software that facilitates communication between two web entities or devices on a network. Thus, a web service must include a service provider, (the server) and a service requester, (the client). Web services are language independent making it possible to implement services in Java, PHP, C#, or any other language while the client application is written in any language too. In their initial appearance web services had huge specifications and cumbersome formats such as WSDL (Web Service Definition Language) for describing the services and SOAP (Simple Object Access Protocol) f or specifying the message formats. On the other hand, we can describe the REST service in a plain text file and use any message format we want, such as JSON, XML or even plain text again. The simplified approach was applied to the security of REST services as well; no defined standard imposes a particular way to authenticate users.

III. SIMPLE OBJECT ACCESS PROTOCOL (SOAP) WEB SERVICES

A SOAP Web service is implemented as an XML-based protocol and all security related data can be inherently defined in the SOAP header element. The protocol enables the development of interoperable software and is not tied to any specific operating system or programming language.

The SOAP message has an envelope structure as depicted in Figure 1. The message envelope is designed to carry application payload in the body portion and control information in the header portion. The header portion thus can contain any number of SOAP header blocks that include addressing, security, and other message requirements. The SOAP security header is Web Service Security (WS-Security, WSS) which an extension to SOAP for supporting web service security. It is a part of Web service specifications published by the Organization for the Advancement of Structured Information Standards (OASIS).

The WSS extension enables SOAP to specify how integrity and confidentiality can be enforced on messages and allows the communication of various security token formats, such as Security Assertion Markup Language (SAML), Kerberos, and X.509. Its main focus is the use of XML Signature and XML Encryption to provide end-to-end security.

IV. RESTFUL WEB SERVICES

RESTful web services are implemented based on REST Architecture which is centred around the concept of resources. These web services are light weight, simple, scalable and easy to maintain. These attributes make RESTful web services popular for creating APIs for web-based applications. The REST architecture is based on the HTTP protocol and revolves around resources as it views every component as a resource accessible via a common interface using HTTP.

Thus, a typical REST system consists of a REST Server that provides access to resources and a REST client that accesses and / or modifies the resource. Each resource is identified by either a URI (Uniform Resource Identifier) or a global ID. A resource can typically represented as text, JSON or XML. RESTful web services must be secured so as to protect the data provided via RESTful endpoints. Clear access rights must be defined especially for methods that destroy or modify resources. Thus, proper authentication of users allowed access to such methods is a high priority. RESTful web services rely on the inherent security of the API rather than including within the REST architecture. Though RESTful API calls are typically secured with HTTPS protocol, there is usually a need to implement some form of session-based security. Popular solutions that provide this include OAuth 2.0 and JWT which we shall discuss later.

V. SECURITY VULNERABILITIES IN REST-BASED APIS

The current popularity of APIs as the drivers of web applications and their integration to support resource exchanges over the Internet means that organizations consider API security as the single biggest challenge that should be tackled in the years ahead. A 2018 survey by Jitterbit found that 64% of organizations are creating APIs for both internal and external consumption. It was also established that while about 25% of the respondents are not creating APIs at all, 40% leverage APIs for their operations.[3]

The widespread deployment of REST APIs means that sensitive data such as credit card information, health records, financial information, business information, and many other categories gets exchanged over the network at a colossal scale making it necessary for developers to pay attention to the security issues that are inherent in the process. Some of these issues include:

- HTTP requests and HTTP responses are accessible to potential hackers and since REST APIs rely on HTTP to exchange information usually saved and sometimes executed on many servers this could lead to many unseen breaches and leaks.
- The REST API server can be attacked from the client side by the consumer of the REST API just as the attack could be from the REST API server under the control of an attacker who creates a malicious app to be consumed by REST API client.
- Since APIs inherently provide a mechanism for exchange of resources the other side typically has full control of resource presentation, which makes it possible for the injection of malicious payload that could attack resource handling.
- Potential vulnerabilities can occur during controller mapping (from /to the HTTP message with the resource URI).
- There is also a risk of attack when instantiating the object representing the target resource and invoking the requested operation.
- Furthermore, vulnerabilities arise during the generation of state representation for the target resources.
- Another point of vulnerability is during the accessing and or modification of data in the backend system hosting the resource state like when saving into the database storage.

VI. REST API SECURITY THREATS

In this section, we consider the potential threats that must be considered when implementing a robust REST-based API. As organizations increasingly depend on APIs for their business needs, developers must proactively prepare to confront these threats. System and application developers in collaboration with corporations, foundations and volunteers have come together to form the Open Web Application Security Project (OWASP) a nonprofit foundation that works to improve the security of software. [2]

Some of the key (OWASP defined) threats facing REST APIs are listed below:

1. *Injection Attack* This threat is realised by embedding malicious code into an unsecured application to facilitate attacks such as SQL injection and cross-site scripting. The attacker tries to manipulate the system by transferring untrusted data into the API as part of the request. If the interpreter processes the corrupted input this can result in the attacker gaining unauthorised access to information or the ability to modify data.
2. *Denial of Service (DoS) Attacks* The aim of the attacker in this case is to render the RESTful API non-functional

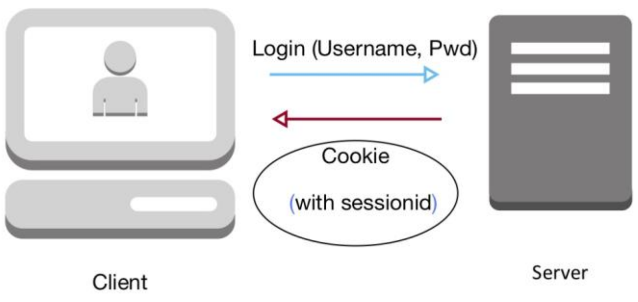
by overwhelming it with enormous messages with requests that contain invalid return addresses.

3. *Broken Authentication* In this case the attacker is able to bypass or take control of the authentication methods implemented in the web programme. Thus web tokens, API keys, passwords , etc can be compromised.
4. *Sensitive Data Exposure* This threat is brought about by poor or missing sensitive data encryption. The encryption should normally be available for sensitive data both in transit and at rest.
5. *Broken Access Control* This threat is characteristic of missing or inadequate access control leading to the attacker gaining control of other users account, and the ability to alter access privileges, change data and carry out other malicious operations.
6. *Parameter Tampering* This is an attack that seeks to manipulate the parameters that are exchanged between the client and the server so as to change application data like user credentials, permissions, sensitive values , and so on. The success of the attack depends on the integrity and logic of the validation mechanisms implanted in the API.
7. *Man-In-The-Middle (MITM) Attack* In this attack the attacker secretly relays and possibly alters the communications between the client and the server even though they believe that they are directly communicating with each other. Thus the attacker intercepts the private and confidential data that is passed between the two parties. MITM attacks occur in two stages: interception and decryption of the sensitive data. A MITM attack can succeed only when the attacker impersonates each endpoint sufficiently well to satisfy their expectations. thereby circumventing mutual authentication.

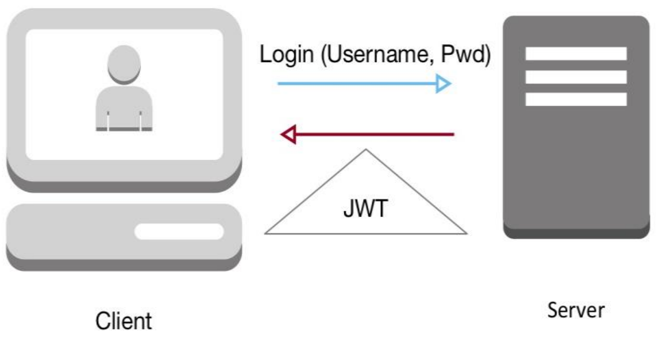
VII. STRATEGIES FOR SECURING REST APIS

We have seen that RESTful Web services rely on HTTP URL Paths to operate. It is therefore clear that to safeguard them we need to take similar steps used in protecting regular websites. Some of these measures include:

- *Input Validation* All inputs to the server must be validated to prevent SQL and NoSQL injection attacks. Though validation is aimed at protecting the server, it makes sense to have a validation layer on the client so as to interactively indicate errors and give advice on how to improve the input. The primary goal of validation is to verify that the value of a data item belongs to a given set of acceptable values. For an API this means verifying that data items coming to the application from external sources have acceptable values. This is achieved by defining data validation rule for every type of data item coming into the system.
- *Authentication* The stateless nature of HTTP and RESTful services necessitates the use of either sessions or tokens to implement authentication. In session based authentication, the server creates a session for the client after login. Thus a session ID is created and stored on the server. The session ID is then embedded in a cookie which is sent to the client where it is stored. While the client remains logged in, the cookie is sent along with every subsequent request and the server compares the session ID embedded in the cookie against the session information stored in memory. Figure 2 illustrates the basic implementation of session-based authentication. Session based authentication was the traditional way of coping with statelessness. Then at every request the server must check the session before sending the response. Thus every time a user is authenticated, the server creates a record and stores it usually in memory resulting on increased overhead when there are many users authenticating. More importantly, server based authentication poses a a big hinderance on scalability because session information is stored in memory. Having vital information in session memory limits the ability to replicate servers and provide scalability. Keeping session information on a user on a particular server would require us to keep sending that user to the same server that they logged in at (called session affinity) thereby hindering replication and load balancing.



Currently a more popular approach is to use JSON Web Token (JWT) instead of sessions for providing authentication. Figure 3 shows how this is done. Here the server generates JWT with a secret and sends the JWT to the client. The client keeps the JWT in local storage and subsequently includes the token in the header of every request. To provide the appropriate response the server then validates the JWT with every request from the client. The validation is carried out by checking if the JWT signatures match. Thus the user's state is not stored on the server but resides in the token stored on the client side. The server simply keeps the secret signature of the JWT. Token authentication is thus completely stateless and supports scalability. Load balancers are able to pass a use along to any available server since there is no need for state or session information anywhere.



- *Keeping Out Sensitive Data from the URL* The URL should never contain username, password, session token and other sensitive data. Such important values should only be passed to the Web service using POST methods.
- *Place Restriction on Verbs* The use of methods like GET, POST, and DELETE should be restricted. The GET method which tends to encourage placing parameters in the URL should never be used delete data. It is also important to ensure that any PUT, POST, and DELETE requests are protected from Cross Site Request Forgery. XML serialiser so as to avoid XML injection. Similarly, for JSON encoders it is important to use a proper JSON serializer for encoding user-supplied data to avoid arbitrary Javascript remote code execution.

VIII. THIRD PARTY AUTHORISATION

There are numerous situations in which the resource owner might want to provide authorisation to a third party for limited access to some restricted resources. Traditionally this would involve the resource owner having to share credentials with the third party. Such an arrangement pauses issues that include

1. the need to store resource owner's credentials on the third party applications for subsequent use
2. the inherent need to use password-type authentication with its characteristic security weakness
3. the resource owner is unable to restrict duration and scope of access once the credentials have been transferred to the third party application
4. there is very little support for revoking access to an individual third party without revoking access to all other third parties.
5. a security breach on any third party application would result in compromising the end-user's password and all of the data protected by the password.

From the above it soon became clear that a solution is required that does not rely on the use of passwords and their subsequent exchange in supporting third party access to selected resources spread across the globe.

Developers building web, desktop, or mobile applications need a system that will support access to protected data from API servers for use by these applications without requiring the users to login with passwords. On the other hand API developers need to allow application developers secure access to users' data without sharing their passwords.

Clearly the issues at hand can be broadly categorized in to *authentication* and *authorization* requirements that must be ratified to provide security and data integrity.

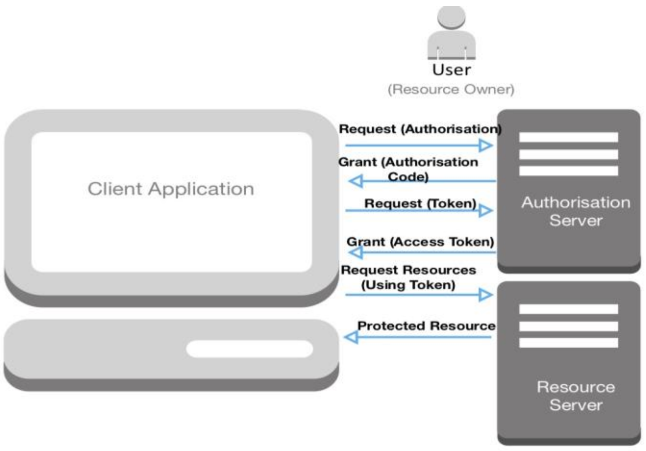
In what follows we look at some solutions that have been developed to address the above issues.

IX. OAUTH

OAuth is short for Open Authorisation which is an open standard authorisation framework built around token-based authorisation on the Web. The framework enables third party services to use an end user's account information without exposing the user's password. OAuth acts as an intermediary on behalf of the end user by providing the third party with an access token that authorises the scope of information to be shared. For this arrangement to work the client application must have registered with the API (Resource and Authorisation Servers) in the first place. Registration involves providing the services with the client application's name, website, callback URL as a minimum. Then the API sends back to the client application a Client ID and Client Secret code that will be required every time the client makes a request.[5]

Figure 4 shows how OAuth2 facilitates authentication. It supports four types of grants to third parties. These are:

1. *Authorisation Code Grant* used for applications running on web servers
2. *Password Grant*is used to exchange a user's credentials for an access token. In this case the client application has to collect the user's password and send it to the authorization server. This type of grant it is not recommended because of the need for password exchange and is only in use by banking systems that are yet to implement best practice in OAuth2.0
3. *Implicit Grant* previously recommended for native apps and JavaScript apps where the access token is returned immediately without an extra authorization code exchange step. This is now considered too risky and many servers do not support it.
4. *Client Credentials Grant* is used by clients to obtain an access token outside of the context of a user. This is typically used by clients to access resources about themselves rather than to access a user's resources.



X. OPENID CONNECT

This is an identity layer on top of the OAuth 2.0 protocol which allows Clients to verify the identity of the End-User based on the authentication performed by an Authorization Server, as well as to obtain basic profile information about the End-User in an interoperable and REST-like manner. OpenID Connect is thus required to provide *authentication* so as to augment *authorisation* that is the focus of OAuth. OpenID Connect provides the following additional features: [6]

- ID token
- Userinfo endpoint for additional user information
- Standard set of scopes
- Standardized implementation of authentication

The ID token is normally provided in JWT format and has three main parts, namely the header, the payload and the signature all encrypted. Thus integration of OAuth2.0 and OpenID connect ensures that the authorisation server is also capable of providing authentication.

XI. CONCLUSION

RESTful web services are based on HTTP and therefore exhibit statelessness which makes their implementation simple. These services have become ubiquitous as they facilitate communication among ever increasing numbers of web entities and devices on the Internet. Thus through these APIs sensitive data, organisational or personal, gets exchanged over the Internet at a colossal scale. It is therefore clear that API security is a big challenge for all organisations. In this paper we have looked at the main vulnerabilities and threats that these services face and identified some of the approaches used in tackling them. Also highlighted are the salient issues of *authentication* and *authorisation* which can be quite tricky when implementing REST APIs. Securing authentication and authorisation requires a combination of front-end and back-end methods in an optimum way to ensure that sensitive data, such as a token, is only exchanged through the more secure back-channel while the less sensitive data like authorisation code, redirect URLs, request scope, etc can be routed via the front-channel.

It is evident that back-channel authentication allows for server-to-server communication and thus removes the need for browser redirections. Many popular and widely used service providers use back-channel authentication to allow access to their services. Furthermore, back-channel protocols provide mutually authenticated end-to-end security via TLS (Transport Layer Security) because the communication is point-to-point. It is hoped that ICT strategists and practitioners in Nigeria will pay special attention to applying these measures to enhance the tendency of resource exchanges among organisations thereby boosting resource utilisation and mitigating against duplication of efforts.

REFERENCES

[1]. 120 HIRSCH, Fredrick, KEMP, John and ILKKA, Jani: ’Mobile Web Services- Architecture and Implementation,’*John Wiley and Sons, 2006.*

[2]. KANG, A., CRUZ, D., MUNZ, A.: ’RESTing on your laurels will get you Pwned!, *RSA Conference, 2014* JITTERBIT: ’2018 State of API Integration’, *Jitterbit Report, 2018*

[3]. HARI, Subramanian: ’REST API Security Vulnerabilities’, Inter Systems Cache: ’OAuth2.0 Introduction’ *1996-2020 InterSystems Corporation, Cambridge, MA*

[4]. OpenID Foundation: https://openid.net/connect/faq/