# Security in Service Oriented Architectures – Volatility-Based Hedging Support System

**Cristina-Ana TODEROIU**
*IT&C Security Master*
*Department of Economic Informatics and Cybernetics*
*The Bucharest University of Economic Studies*
*ROMANIA*
*cristina.toderoiu@gmail.com*

***Abstract***: The nature of today's communication pattern has reverberated on all technical aspects, making them distributed in order to satisfy the need for immediate consumption of quality content. This is especially visible in the enterprise world, where latency reflects directly into financial results and, potentially, the survival of specific entities that are in tight competition. As people has discovered that by intertwining their thoughts through the means of social media is beneficial, enterprises have evolved towards a similar direction, of communicating with each other using services. Securing such Service Oriented Architectures presents a series of concerns, which are the main topic of the following paper.

***Key-Words:*** Architecture, Service Orientation, Restful APIs, Certificates, Database Security, Cross-Site Request Forgery, Cross-Origin Resource Sharing

## 1. Introduction

A robust Service Oriented Architecture can offer a myriad of advantages to the organizations which manage to implement them correctly, the most important of them in the current time and age being the ability to reduce downtime, by implementing secure failover mechanisms. Offering constant services to the clients of the application is vital especially, but not limited to the financial field, where every second of instability or latency can lead to detrimental losses. What is more, Software Oriented Architectures enable the reuse of existing interconnected services in order to obtain new functionality, which leads to reduces application development times, a general improvement in general efficiency of the development team and further flexibility. In theory, it sounds perfect, but what really makes the difference between perfect and disastrous is the way in which their corresponding security mechanisms were implemented. The main benefits of implementing such an architecture, the loose coupling between the components and the exposure of services also represent the main vulnerability concerns. Exposed applications and services become vulnerable to attacks, and the greater the number of integrations and endpoints, the greater the number of potential points of attack. What is more, with communication between services and consumers, ensuring secure operations over trust boundaries is crucial. Without a Service Oriented Architecture security model in place, the entire business ecosystem is at risk [1].
The current paper will address the full-stack concerns of designing such a solution, starting with some network considerations, database-level security and continuing up to the top of the stack with web security concerns related to consuming RESTful services in a web application.

## 2. Security Concerns

Although network topology for a system and the corresponding security aspects fall into the responsibilities of infrastructure architects, a small discussion can be made on top of it, to address the main concerns. The majority of arguments appear as part of the intercommunication of the services and the web aspects. When designing a service oriented architecture, non-functional requirements such as the following also need to be considered:

ISSN 2067 – 4074

create a system that is fault tolerant, be able to track cascading failures in the system easily, plan for the capacity that will be used and prepare disaster recovery plans accordingly.

## 2.1 Network considerations

The main aspect that needs to be taken in consideration here is that one needs to be absolutely certain that only the machines exposing the public services need to be exposed to the outside network. The internal, private applications which serve the main one should be made inaccessible at least through means of a firewall with a tight policy setup [2]. As such, virtualization is the safest bet in this case, thus making it hard for an attacker to understand the topology of the system. Of course, having the additional security of an Intrusion Detection or, better yet, an Intrusion Protection system will ensure a level of stability.

## 2.2 Database considerations

Database security represents a critical aspect when it is part of any type of system. It is true that not any Service Oriented Architecture needs a database. Based on the actual functionality of the system, it may result that in-memory caches or even storing information to files are better for that particular solution. However, many of them do require one. The security mechanism for it, even if the database is relational or not, must be multi-tiered, controlling access to the system, the system resources and the system data. As such: access to the system needs to be controlled, authorized users must be able to access (insert, modify, retrieve or delete) data that they are authorized to access, authorized users must be restricted to the data and resources that they are duly authorized to access and nothing more and, what is more, unauthorized users must have absolutely no access [3]. What is more, as far as cryptographic storage is concerned, one must make sure that the data is encrypted everywhere it is stored long term, particularly in backups, using a strong standard encryption algorithm and a strong key.

## 2.3 Back-end considerations

In the Web 2.0 world, it is the back-end web services that become the target of attack. This is sometimes referred to as the "large attack surface" of Web 2.0. An attacker can try to attack an application through its client interface or they can simply bypass the interface and go straight after the underlying services instead.

### 2.3.1 Authentication

As far as authentication is concerned, RESTful web services should make use of session-based authentication, taking good care to establish a session token or using an API key through secure means, as a POST body argument or a cookie, which should not appear in any way in the web server logs. These mechanisms pave the way for great vulnerability, because many web services are written to be as stateless as possible, which leads to having a session blob sent as part of the transaction [4]. In order to avoid replay attacks, one could consider using a time limited encryption key, keyed against the session token or API key, date and time and incoming IP address. What is more, additional protection of local client storage of the authentication token can mitigate such attacks. This is an important concern, since in applications in which the SSL/TLS communication is implemented, the details of the handshake are essential, else an attacker can simply replay the whole communication.

### 2.3.2 Protection against farming

In securing web services, one must offer special attention to mitigation in case of farming. Using mutually assured client side TLS certificated might be a method of limiting access to trusted organizations, but this is by no means certain, particularly if certificates are posted deliberately or by accident to the internet [4].

### 2.3.3 Protecting protocol methods

RESTful APIs often use the methods of the HTTP protocol, provided that's the protocol they use: GET (read), POST (create), PUT (update) and DELETE (delete a record). Not all of these are valid choices for every single resource collection, user of action [5]. It all comes down to authorization for each of these operations based on the role of the subject requesting them. This is vital, as administrative web services should not in any way be misused. The session token or API key should be sent along as a cookie or body parameter to ensure that privileged collections or actions are properly protected from unauthorized use. Also, when only one of the methods should be available for a resource, the others should return a proper response code, such as 403 Forbidden.

### 2.3.4 Protection against CSRF

For resources exposed by RESTful web services, it's important to make sure any method request is protected from Cross Side Request Forgery. CSRF is a type of attack that occurs when a malicious web site or program causes a user's web browser to perform an unwanted action on a trusted site for which the user is authenticated. A transparent solution is the Synchronizer Token pattern, which requires the generating of random "challenge" tokens that are associated with the user's current session.

### 2.3.5 Input validation

Input validation is not a necessity restricted to front-end, but is extremely vital when discussing about entering information into a system through the means of a web service. All input validation failures need to be logged and rate limiting the API to a certain number of requests per hour or day to prevent abuse is also a good idea.

All incoming messages need to be parsed using a secure parser and converted to strong typed entities. Also, content-types should also be verified and, in case it doesn't match the expected one, the service must reject the request with a 406 Not Acceptable error.

### 2.3.6 Securing data transport

The usage of TLS should be mandated, unless the public information is completely read-only, especially where credentials or any value transactions are performed [4].

## 2.4 Front-end considerations

Cross-site scripting is a type of attack where the attacker injects code into the remote server. There are two types of cross-site scripting: persistent and no persistent. The persistent one occurs when the code injected by the attacker gets stored in a secondary stage, such as a database. The no persistent XSS requires an unsuspecting user to visit a crafted link made by the attacker to visit said link, executing the malicious code.

Another front-end concern is related to phishing, which is an attempt to acquire sensitive information, by masquerading as a trustworthy entity in electronic communication [6].

# 3. Secure SOA Implementation

The following subsections present the practical study of the application.

## 3.1 Premise of the application

The associated application focuses on portfolio management and revolves around the financial notion of volatility. The option trader, like a trader in the underlying instrument is interested in the direction of the market. But unlike a trader in the underlying, an option trader is also sensitive to the speed of the market. If the market for an underlying contract fails to move at a sufficient speed, options on that contract will have less value because of the reduced likelihood of the market going through an option's exercise price. In a sense, volatility is a measure of the speed of the market. If one knows whether a market will be relatively volatile or relatively quiet and can convey this information to a theoretical pricing model, any evaluation of options on that market will be more

ISSN 2067 – 4074

accurate than if we simply ignore volatility [7].
The purpose of the application is to make use of the provided implied volatility and calculate mathematical values – the Greeks, which can be further used for hedging.

## 3.2 Application architecture

Before going deeper into the actual architecture of the application, Figure 1 illustrates the actual Web Graphical Interface that is provided to the end-user in this example.

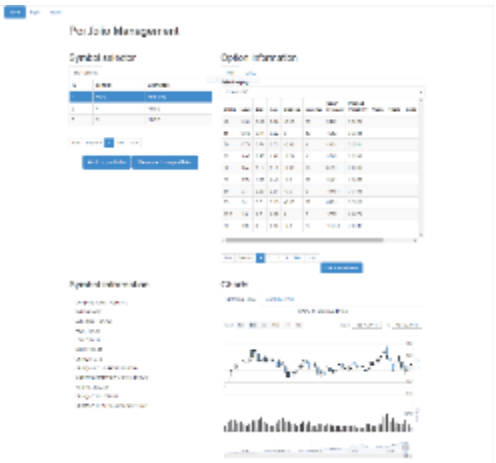

*Figure 1. The Graphical User Interface of the Application*

Taken into consideration the destination of the application, it is bound to be part of a complex enterprise risk and profit and losses platform, the best decision would be to create it based on services. In that way, a graphical user interface can be used in order to access and calculate the hedging information and the underlying services can be exposed, will it be necessary to other systems in the platform.
Figure 2 depicts the language and implementation agnostic architecture of the system. Needless to say, all services communicate through HTTPS over TLS. As main technologies used throughout the entire system the Spring stack takes an important role – core, boot, security, cloud.

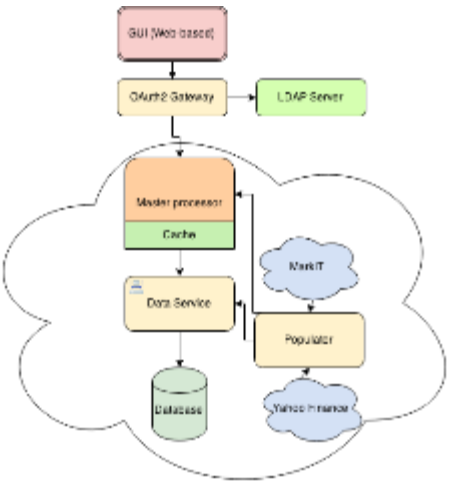


*Figure 2. System architecture*

### 3.2.1 Populator service

A populator service is required to obtain market data from outside data sources. MarkIT was chosen because it offers free REST calls to its API, provided there are no more than 10 requests in the span of 30 seconds. It offers symbol information and quotes. Since there are no free API alternatives for obtaining option data anymore, that information is parsed directly from the HTML of the Yahoo Finance website. The input from both sources is then passed through REST services to the Data Service, which covers the role of a persistence gateway – all requests to the database go through it and no other server has the permission of accessing the database directly.

### 3.2.2 Database and Data service

At the database level, the chosen server implementation was PostgreSQL. The reason behind this choice is the fact that database security is addressed at several levels: database file protection – all files stored within the database are protected from reading by any account other than the Postgres superuser account, connections from a client to the database are, by default, allowed only via a local Unix socket, not via TCP/IP sockets and the fact that PostgreSQL has great configurable host-based access control. Host-based access control is the name for the controls PostgreSQL exercises on what clients are allowed to access a database and how the users on those client must authenticate themselves. Each

database system contains a file named pg_hba.conf, in its PGDATA directory, which controls who can connect to each database. Every client accessing a database must be covered by one of the entries in the pg_hba.conf. Otherwise, all attempted connections from that client will be rejected. This configuration for the current database is:

```
hostssl all +sslcertusers
192.168.220.128/32 trust
clientcert=1
hostssl all all 192.168.220.1/32
trust clientcert=1
```

The Ubuntu VM on which the database is deployed was configured to use a static IP - 192.168.220.128, thus the translation of the two lines above is to allow connections only from the 2 IPs: the host on which the database is deployed and the IP of the machine on which the Data Server is deployed, but granting access only to those providing a certificate granted by the same Certification Authority which granted the server-side certificate.
Data service then connects through the use of JPA with a Hibernate underlying implementation to the database and Spring Data facilitates the access to the services exposed by the Data service, which are basically Create, Read, Update, Delete operations.
As depicted by Figure 1, replication would be needed at this level, because, given its nature, it might become a bottleneck or, worse, single point of failure.

### 3.2.3 The Master processor service

The Master processor represents the calculation engine, the one that transforms the market data from the data service and serves some of it to the GUI and efectuates the actual processing of the Black-Scholes Greeks values, using a Java-based implementation of the aforementioned mathematical model. The Master processor queries the database through the Data Service REST API and gets live market data using the Processor REST web services and offers the processed results to the UI server through REST API, as well.

### 3.2.4 The API Gateway

The API Gateway is a single point of entry and control for front end clients. The client only has to know the URL of one server and the back-end can be refactored at will with no change, which a significant advantage. There are other benefits in terms of centralization of control: rate limiting, authentication, auditing and logging. Spring Cloud allows for easy configuration of a reverse proxy, the main being represented by adding the following lines to the application.yml configuration file of the UI server:

```
zuul:
  routes:
    resource:
      path: /resource/**
      url: http://localhost:9000
```

Thus, all requests to the UI server which have the /resource/ prefix will be redirected transparently to the server hosted at *http://localhost:9000* location.

### 3.2.5 The UI server

Thanks to the server-side configuration with Spring Cloud, the client will be able to resolve all the paths to external systems as relative ones, so there is no need for cross-origin resource sharing mitigation. As client-side technology, AngularJS was chosen, primarily due to the fact that it offers built in support for CSRF, based on cookies. In order to obtain a working model, the server needs a custom filter that will send the cookie and that is achievable through the means of Spring Security. AngularJS also encodes all untrusted data by default, which is the main defense against XSS [8].

## 4. Conclusions

Service Oriented Architectures represent the future in enterprise software, especially in the financial industry, but architects and developers must take great care in order to avoid the security pitfalls ensured by such architectures. The current frameworks, however, such as the Spring stack, in the case of Java, and AngularJS can offer great leverage,

ISSN 2067 – 4074

provided their features are used diligently and the developer respects the global security standards.

## Acknowledgement

Parts of this paper were presented at The 8th International Conference on Security for Information Technology and Communications (SECITC 2015), Bucharest, Romania, 11-12 June 2015.

## References

[1] Mulesoft, "Service Oriented Architecture - Security Matters," Mulesoft, [Online]. Available: www.mulesoft.com. [Accessed 3 May 2015]

[2] C. P. Pfleeger, S. L. Pfleeger and J. Margulies, Security in Computing, Fifth Edition, Upper Saddle River: Prentice Hall, 2015

[3] E. C. Foster and S. V. Godbole, Database Administration, New York City: Apress, 2014

[4] OWASP, "OWASP Top 10," OWASP, 16 December 2014. [Online]. Available: https://www.owasp.org/. [Accessed 3 May 2015]

[5] R. Enríquez and A. Salazar, RESTful Java Web Services Security, Birmingham: Packt Publishing, 2014

[6] Y. E. Liang, JavaScript Security, Birmingham: Packt Publishing, 2014.

[7] S. Natenberg, Option Volatility and Pricing: Advanced Trading Strategies and Techniques, 2nd Edition, New York City: McGraw-Hill, 2014.

[8] Google, "Mustache Security," Google, 26 September 2014. [Online]. Available: https://code.google.com/p/mustache-security/wiki/AngularJS. [Accessed 3 May 2015]