Microservices for Web Based Applications and Security

Matthew Knill
u6052043

A Report submitted as the assessment for an Independent Studies 6 units (COMP3740) in
The Research School of Computer Science
Australian National University
25/10/19
Acknowledgements

A great deal of time and energy was spent on this project and I would like to thank those who have helped and provided contributions to the project.

I would like to express my gratitude towards Dr Henry Gartner for his ongoing support over the course of this project. He has provided valuable advice and been highly encouraging throughout.

Finally, I would also like to express my gratitude towards my parents who have also provided me with encouragement and support.
Abstract
A microservice is a modern term used to describe an individually deployable component that is part of larger software applications [1].

This report discusses some of the architectural concerns of using microservices for large web applications. In particular, by comparing monolithic and microservice based approaches to web development, it will explore how both Clean Architecture and Domain Driven Design patterns and methodologies can be used to drive microservice development.

As a challenging case study, this report will consider a software system that uses microservices as an approach to secure authentication. It will provide detail on the potential challenges, advantages and disadvantages of using microservices from a security perspective for such a system. It will also discuss whether designing a secure system will result in a substandard microservice design.

The key findings of the report are:

- Microservices are relatively immature when it comes to security; however, the report gives evidence to some useful principles and design patterns that would enhance microservice security.
- In order to preserve the existing advantages of good software architecture, more concrete microservice design principles and patterns are needed for their further adoption into large systems.
# Table of Contents

Acknowledgements ........................................................................................................... 2

Abstract............................................................................................................................... 3

1 Introduction....................................................................................................................... 6
   1.1 Monolithic Approach to Web Development ............................................................ 6
      1.1.1 Load Balancing a Monolithic Application ....................................................... 7
   1.2 Microservice Approach to Web Development ....................................................... 7

2 Architecture.................................................................................................................... 8
   2.1 Clean Architecture .................................................................................................... 9
   2.2 Domain Driven Design ............................................................................................ 9
      2.2.1 Ubiquitous Language ....................................................................................... 10
      2.2.2 Model ............................................................................................................ 10
      2.2.3 Context ......................................................................................................... 10
      2.2.4 Bounded Context .......................................................................................... 10

3 Microservice Design and Implementation ..................................................................... 11
   3.1 Domain Driven Design and Clean Architecture .................................................... 11
   3.2 Microservice Design .............................................................................................. 11
      3.2.1 Microservice Communication ....................................................................... 13
   3.3 Size of Microservice .............................................................................................. 14
   3.4 Difference between Microservices and Code Separation .................................... 15

4 Secure Authentication System using Microservices ....................................................... 17
   4.1 Case Study ............................................................................................................. 17
   4.2 Analysis .................................................................................................................. 17
      4.2.1 Difference between Authentication and Authorization ...................................... 19
   4.3 Token Based Security .............................................................................................. 20
      4.3.1 Authenticate in Microservice Independently .................................................... 20
      4.3.2 Authenticate through Authentication MS ......................................................... 22
      4.3.3 Authenticate to Authentication MS from each microservice ......................... 23
   4.4 Certificate Based Security ...................................................................................... 25
   4.5 Security as a Service .............................................................................................. 26

5 Critique........................................................................................................................... 26
   5.1 Exposed Business Logic .......................................................................................... 27
      5.1.1 Business Logic exposed to Security Bounded Context .................................... 27
5.1.2 Business Logic exposed to other Bounded Contexts ...................................................... 28
5.1.3 Custom Web Tokens ........................................................................................................... 28
5.2 Authorization using Observer design pattern ......................................................................... 29
5.3 Authenticate through an Adapter ............................................................................................ 31
6 Conclusion .................................................................................................................................. 33
7 Further Research ......................................................................................................................... 33
Bibliography .................................................................................................................................. 34
Appendix ........................................................................................................................................ 36
    A. Compromised Microservice ................................................................................................. 36
    B. Man-in-the-Middle Attack ..................................................................................................... 36
    C. Mitigate Attacks .................................................................................................................... 37
    D. Implementations in Existing Research ................................................................................. 38
1 Introduction
Microservices are increasingly being used to build web applications that are highly flexible, maintainable and scalable. While many large organizations such as Amazon, Netflix and Twitter have successfully used microservices to their advantage, they are still in their early stages when it comes to building web applications in wider software engineering.

1.1 Monolithic Approach to Web Development
Monolithic web applications, while often requiring different components, modules and services, essentially behave as a single component [2]. A typical monolithic web application has a web server (such as Apache) with a backend language (such as PHP) and a database (such as MySQL). The web server and the database are used to serve the web application to the user along with the frontend logic.

As shown in Figure 1, the web server and the database are separate services and can even be running on different machines depending on the needs of the web application. Conceptually, since they are both required to effectively serve the web application to the end user, this is a monolithic application.
1.1.1 Load Balancing a Monolithic Application

An example of a web application that has been deployed to handle higher loads is shown in Figure 2. In this setup, there are cache servers, application servers and replica servers all to ensure performance and reliability for the web application. However, once again all these servers work in conjunction with one another to serve the same monolithic application [2].

1.2 Microservice Approach to Web Development

Microservices are a way of separating an application into individually serviced components. What differentiates this from a monolithic approach is that each microservice can behave independently from one another. In fact, each microservice could be using an entirely different architecture which better serves its predefined purpose. Microservices can then communicate via an API or some other form of message passing.
An example of a very simple web store built using microservices is shown in Figure 3. While there could be some middleware between the microservices and the end user, each microservice behaves within its own context (context is discussed further in section 2.2). For instance, it is possible that the end user in this scenario would not have to use the Payments MS to create an order with requests almost exclusively to and from the Order MS.

Within each microservice, components could exist that are similar to the monolithic web application as shown on the right of Figure 3. However, not all microservices require the same components. For instance, the Payments MS may not require a database but require some infrastructure to facilitate payments with banks or services such as Paypal.

It is not a requirement for each microservices to be hosted on separate physical hardware. Microservices could be built in virtual machines or on containers (isolated application components [4]).

As will be discussed, implementing microservices can be highly complex and is often unnecessary for smaller web applications.

2 Architecture
This report will consider two particular software architectures and methodologies for the development and maintenance of large software systems.
2.1 Clean Architecture

Clean Architecture, coined by Robert C. Martin, is a collection of design principles and patterns that result in applications whose “domain logic” (including business rules) is maintainable and testable independently from other parts of the software such as the UI, database and external agencies. This essentially means that all “high-level” business rules are entirely separate from the implementation details. The following is a common representation of Clean Architecture:

![Clean Architecture Diagram](sourced from [5])

As shown, all dependencies point inward, and the enterprise and application business rules do not depend on any external layers [6].

Clean Architecture uses the Dependency Inversion Principle (DIP) among the other SOLID principles. As quoted by Martin in his book on Clean Architecture in relation to DIP (p. 78):

“The code that implements high-level policy should not depend on the code that implements low-level details. Rather, details should depend on policies.” [7]

The precursor to this report, titled *Refactoring with Clean Architecture*, looks at Clean Architecture in considerably more detail and thus will only be touched on in this report [6].

2.2 Domain Driven Design

Domain Driven Design (DDD) is a software development methodology invented by Eric Evans [8]. It is a method of application development which strongly connects implementation details to an evolving model of the business logic. This evolving model is constructed through constant collaboration with domain experts () and a *Ubiquitous Language* that all team members understand [9]. Evans in his book on DDD uses the following terms:
2.2.1 Ubiquitous Language
Often, domain experts have little understanding of technical terms. It can therefore be quite difficult for developers and domain experts to effectively communicate. Evans suggests using a language that is built around the domain model to empower all forms of communication of the products design. Establishing the Ubiquitous Language early means that developers do not need to explain unnecessary technical details to domain experts as the project progresses [8].

The following table shows some examples of the Ubiquitous Language for the online store example used above. Over time, these terms should evolve with more detail to better fit the domain model.

<table>
<thead>
<tr>
<th>Ubiquitous Language</th>
<th>Equivalent Pseudo Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Customer creates user</td>
<td>createUser(email, name, ...)</td>
</tr>
<tr>
<td>Customer adds an item to their cart</td>
<td>customer.addToCart(item)</td>
</tr>
<tr>
<td>Customer creates an order</td>
<td>buildOrder(customer.getCart())</td>
</tr>
<tr>
<td>Customer pays for order</td>
<td>payments.payOrder(order)</td>
</tr>
</tbody>
</table>

Note: the above table is an extremely basic and high-level example. Both the model and Ubiquitous Language would have to be far more extensive for a project like this.

2.2.2 Model
Evans suggests that the model is an important part of a software project and should be the ‘foundation of the design’. This model is not just an analytical model or design model but rather a combination of both. It leaves out technical issues and relies on a conceptual understanding of the project accepted by both the domain experts and development team. From this model, new models can be derived which more closely relate to the domain experts or development team [8].

2.2.3 Context
Evans uses a context to define the meaning of a statement or word in the model and Ubiquitous Language. For instance, in the web store example above, cost would likely have a different meaning in the context of an order as opposed to product. This is because an order can be composed of one or more products [9].

2.2.4 Bounded Context
The bounded context is a hypothetical boundary that separates contexts in a large project. Large projects can have multiple contexts that coexist but when combined cause errors in both the model and implementation. Bounded contexts are used as a means of uncoupling contexts from one another so that they can be modified independently. They allow small teams to work on their own context without much input from the outside world.

The following could be the bounded contexts used for the online store example:
As shown, the bounded contexts used in this example strongly relate to how the microservices were arranged above. As will be discussed more, bounded contexts can be used as a way of determining how to split up a large application into microservices.

3 Microservice Design and Implementation

3.1 Domain Driven Design and Clean Architecture

Both DDD and Clean Architecture have numerous similarities as they both favour placing all the business logic in the centre of an onion type design model. As discussed, this results in software whose business logic does not depend on low level details.

However, there are some key differences and a combination of both approaches might be used to adequately build a microservice based application.

DDD, as a very broad overview, is about conveying information to all stakeholders of the project. It is about working with domain experts to ensure that the application fits their needs even when they have little technical knowledge.

Clean Architecture, as the name suggests, is an architecture that is more focussed on implementation specifics. This can be differentiated from DDD by the fact that only the developers really need a grasp of Clean Architecture for it to be used effectively and the ubiquitous language is immaterial. Clean Architecture, as mentioned, strongly uses dependency inversion to ensure that the business logic is uncoupled from the outer layers. Clean Architecture, however, can be used without other stakeholders having an understanding of what it is.

3.2 Microservice Design

Some authors claim that DDD can be a useful approach for driving microservice design [10]. One of the primary issues with microservices is how they should be divided. Each microservice should be self-
contained, have a single responsibility and be maintainable by a small team [11]. Therefore, a good way to split up microservices is by ensuring that the service boundary (i.e. the boundary that a single microservice operates on) of each match that of a single DDD bounded context (refer to Figure 5).

From the perspective of Clean Architecture, according to Martin, it is possible for microservices to traverse layers as the Clean Architectural pattern “doesn’t care” how it’s deployed [5]. This, however, means that each microservice is no longer independently maintainable by a small team as outer layer microservices likely require inner layer microservices to function. Another issue is that this would often require high impact communications between microservices which may be unfavourable.

As such, a potential design option is that each microservice has its own implementation of Clean Architecture that best relates to its own context.

The follow diagram shows how Clean Architecture based microservices can be connected to each other:

*Figure 6 Clean Architecture and Domain Driven Design*
As presented in Figure 6, each microservice is separated into their own bounded context and decoupled by using the External Interfaces layer of the Clean Architecture model. Nevertheless, it is possible that Clean Architecture is not a requirement for certain microservices.

The following is a possible method with which to go about microservice design and implementation.

1. Identify Domain – build domain model by working with domain experts
2. Define bounded contexts – separate the model into logical independent components
3. Identify Microservices – design microservices to correlate with the bounded contexts defined previously
4. Design Microservice Specific Architecture – this is where the team managing the microservice within the bounded context can build an architecture that best suits the microservices responsibility. This could be Clean Architecture or another type of architecture
5. Implement – implement and test the architecture
6. Deploy – deploy in collaboration with other microservices

Note: these steps could be repeated or returned to at any point in the project’s development.

As shown, the architectural design takes place after the bounded contexts have been defined and the microservices identified. This means that the structure of the microservice specific design can be entirely dependent on its respective task [12].

3.2.1 Microservice Communication
It is likely that microservices will need to communicate with each other and not just the user. It is therefore necessary to ensure that microservices can communicate without coupling themselves to one another. In Figure 6, each microservice is connected via the external interfaces layer of the Clean Architecture model and thus each remains decoupled from one another.

However, rather than requiring a separate external interface for each microservice (as shown in Figure 6), often a common set of rules govern how microservices communicate. Microservices can use
something called a Representational State Transfer (REST) API. This is essentially an API that uses specially formatted HTTP requests to communicate.

Figure 8 shows how each microservice could implement a common set of rules to communicate with each other via a REST API.

3.3 Size of Microservice

It was mentioned that the service boundary of the microservice should be the same size as a bounded context. More rigorously, the service boundary of a microservice should be no larger than a bounded context. This means that each small team could be responsible for multiple microservices all within the same bounded context.

Essentially, a microservice could range from a simple “aggregate” (a DDD term describing a collection of classes with a single entry point used to model a concept in the domain) to one that handles the entire bounded context as it does with the online store example above. It can therefore be difficult finding the balance between using many smaller microservices or a few larger ones.

While heavily dependent on the project, the service boundaries should start off being the same size as the bounded context and then potentially reduce as the model evolves [13]. This is because it is very easy to fall into the trap of using too many microservices to solve a particular problem. Each new service boundary adds a level of complexity to the application making it potentially far more difficult to manage.
As shown in Figure 9, the Payments bounded context is later split up into multiple microservices to potentially better suit the problem. In this case, however, it is possible that having both the Bank Payment MS and the Cash Payments MS may be unnecessary and should rather be just a single microservice.

### 3.4 Difference between Microservices and Code Separation

The logical boundaries possible with microservices are often also possible with code through the use of classes, packages and libraries etc... depending on the language and implementation used. However, what sets microservices apart is that they have the ability to store state and provide the same implementation to all consumers [14].

For example, say the team behind the Customer Bounded Context required data stored using a relational database such as MySQL while the team behind the Order Bounded Context wanted to store data using document-based database such as MongoDB. In a well-designed monolithic application, both MySQL and MongoDB would be entirely decoupled from the rest of the code potentially through the use of Clean Architectural principles. This means that the bounded contexts are somewhat less defined in the implementation since it now wraps around the outer database dependencies and the inner business logic as shown in Figure 10.
When they are on separate microservices, however, the bounded contexts can be entirely separate, right through to the storage implementation and joined by something such as a REST API. As shown in Figure 11, microservices allow bounded contexts to be more elegantly be separated.
Having these more defined separations can be useful for developer allocation and resource allocation. For instance, say given team requires more resources for their bounded context. In a monolithic application, it may be difficult to isolate more resources to a certain bounded context and not to others.

Nevertheless, this level of separation may only be necessary for large projects with a large team behind it. To reiterate from before, microservices might over-complicate smaller projects that don’t require them [15].

4 Secure Authentication System using Microservices

Security is often a demanding task even for monolithic applications. On top of that, microservices add new security challenges as they require traffic to be sent over a distributed network.

Below is a case study that will help demonstrate some of these challenges.

4.1 Case Study

Following on from the examples describe above, this case study focusses on an online store pre-built using an elegantly structured microservices approach. The developers worked heavily with domain experts to segregate the microservices based on bounded context. Each microservice was build using Clean Architecture, are loosely coupled to each other and maintainable by a single small team. However, a secure authentication system has yet to be implemented as it was assumed that it can be deployed in an entirely separate bounded context post development. It was also assumed that deploying it after the implementation of the other microservices would mean that the authentication system could be better tailored to the domain.

4.2 Analysis

Security is a substantial field in almost all software systems. It is also, in many ways, difficult to get right. As put by Bruce Schneier:

“Security involves making sure things work, not in the presence of random faults, but in the face of an intelligent and malicious adversary trying to ensure that things fail in the worst possible way at the worst possible time...again and again. It truly is programming Satan’s computer.” [16]

In this statement, Schneier is differentiating between a system that is resilient against accidental malformed input and one that is resilient against intelligent malicious malformed input. The former might be user friendly and satisfy the requirements of the system, however it may remain insecure.

Furthermore, microservices bring new security challenges that monolithic applications do not have. Since microservices are a distributed system, sensitive code now must be exposed through message passing or web APIs to other microservices.

The literature outlines that the common issues with microservice security are that of authenticating users over a distributed network and securely communicating between microservices [17]. Coupled with this is the complications of implementing security using DDD [18].

Initially, perimeter defence was considered to be a safe way to secure microservice communication. Through perimeter defence, the microservices are secured by a firewall and other networking equipment to ensure that sensitive code is not exposed to the end user. This theoretically protects the
microservices from malicious users while allowing them to freely communicate amongst each other. The following diagram shows how perimeter defence could be used to secure the online store example:

![Diagram of perimeter defence]

*Figure 12 Perimeter Defence*

This, however, does not satisfy modern security paradigms as it assumes the integrity of all other microservices. For instance, if one microservice was compromised, the attacker would likely also be able to communicate with other microservices and access sensitive information. Essentially, a “trust no one” approach to security is now desirable [19].

The principles outlined in DDD would suggest keeping security in its own bounded context like shown in Figure 12 below:
The issue with this is that often security needs to be a component in the domain logic of other bounded contexts. For example, say a malicious user wanted to access some protected customer data records of another user. If the bounded contexts were strictly enforced, it may be possible that a malicious user could simply bypass the Authentication MS by accessing the Customer MS directly [18] thus making the Authentication MS almost redundant.

In relation to the case study above, the following are some security principles, suggested by the relevant literature, for microservice authentication that satisfies the “trust no one” approach to security.

- Token Based Security
- Certificate Based Security
- Security as a Service

4.2.1 Difference between Authentication and Authorization
Before delving further into these approaches, it is necessary to outline the difference between authentication and authorization.

- **Authentication** – the process of ‘authenticating’ a user to a system
- **Authorization** – the process of ‘authorizing’ a user to a specific task

The importance of both authentication and authorization can be demonstrated in the following code. Assume a genuine user eve is authenticated to the online store though no authorization has not been implemented.
$eveToken = authenticateUser("eve");  // authenticate using Authentication MS

$eveCart = addItemToCart($item1, "eve", $eveToken);  // add to cart using Customer MS

$eveOrder = createOrder($eveOrder, "eve", $eveToken);  // create order using Order MS

payForOrder($eveCart, "alice", $eveToken);  // pay for order using Order MS  
// (but impersonating alice)

As shown in the code above, user eve is able to impersonate user alice when paying for her cart which would mean that item1 is delivered to eve but paid for by alice. This is obviously a security implication that must be resolved via implementation of authorization into the web application.

While authentication and authorization are both highly important for many applications, authorization is often more difficult [20].

### 4.3 Token Based Security

In this approach, security is applied by passing a digitally produced authentication token to the required microservice. These tokens are then analysed to verify the integrity of the sender.

The following are some methods derived from M. Uithol in his paper *Security in Domain Driven Design* [18]. While this paper is somewhat dated and does not directly mention microservices, it is likely to be useful due to microservices' close connection to DDD.

For the following approaches, assume that a user wants to pay for an order they have just produced. This involves communicating with the *Payments MS* to make the necessary payment.

Note: *the function requestProtectedResource(token) was used in place of for instance payForOrder(orderId, amount, token) to reduce clutter and confusion*

#### 4.3.1 Authenticate in Microservice Independently

In this approach, each microservice is secured independently. This is possibly one of the simplest approaches to enable both authentication and authorization to a given microservice.
As shown in Figure 13, initially the user authenticates with the Authentication MS to obtain a token. Upon the user requesting the protected resource, the Payments MS can then both verify the token and authorize the user if they have the required permissions.

**Issues with this approach**

While potentially simple to implement and quite secure, this approach can violate some of the microservice design principles outlined above. This is because each microservice ultimately now has two responsibilities, that of its original responsibility and also that of authenticating and authorizing users based on their token.
For example, if the security team decided to change the format of the token, all other teams managing the other bounded contexts now have to make the necessary changes to support this new token.

4.3.2 Authenticate through Authentication MS

The next method of authentication is by using the Authentication MS as middleware for any communications between the user and a microservice. In other words, the user is required to connect to other microservices through the Authentication MS (could be enforced by network security). In this approach, the user initially requests a token in a similar way to the previous approach. However, the user now sends all requests for a protected resource through the Authentication MS.

Upon successful verification of the token, the Authentication MS, proceeds to request the protected resource from the Payments MS and finally pass that back to the user.

![Diagram](image)

*Figure 15 Authenticate through Authentication MS*

The advantage of this approach is that all authentication and authorization is done entirely by the Authentication MS. All updates done with the Authentication MS can be done independently and managed entirely by the security team.
Issues with this approach

The largest issue with this approach is that the Authentication MS now requires knowledge of all other microservices and their responsibilities. This means that there are essentially no clear boundaries between it and the other microservices. It is possible that even a simple update in one microservice would require that the Authentication MS to be updated.

For instance, say the team behind the Payment MS have implemented a new payment method to a certain group of users. The Authentications MS would now need to know about this payment method and the group of users that it relates to otherwise unauthorised users may have access to it.

4.3.3 Authenticate to Authentication MS from each microservice

This method of authentication involves each microservice authenticating users by forwarding the authentication request on to the Authentication MS. This means that each microservice can be independently secured while most of the authentication logic is done on the Authentication MS.
Like the previous approach, this method of authentication allows the user to negotiate a token with the Authentication MS. However, in this case, after obtaining a token, the user requests the protected resource directly from the Payment MS, which then passes it on to the Authentication MS to verify. This method of authentication means that each microservice can behave almost entirely independently with very few overlapping boundaries with the Security Bounded Context.

As with the other approaches, this approach requires that each microservice has some code that is ultimately separate from its own bounded context. However, this code is small and can operate outside of the domain logic of the microservice as it simply needs to forward the token to the Authentication MS and reply to the result.

An immediate issue with this approach is that it does not support authorization. However, shown below is how this method could also be used for authorization.
As shown in Figure 17, the additional `checkAccess` function is added to authorize a given token that was previously authenticated using the `Authentication MS`.

**Issues with this approach**

The issues with this approach are similar to the issues with the approach described in section 4.3.1 in that the `Payments MS` is tied to the implementation specifics of the token in the `Authentication MS`. However, the `checkAccess` function likely has less code than the `checkToken` function since it can assume that the token is already authentic.

### 4.4 Certificate Based Security

Another approach for microservice authentication involves the use of digital certificates along with asymmetric encryption. A digital certificate is a cryptographically signed electronic document that is used to verify the integrity of the sender. These certificates are only issued by a trusted certificate authority [21].

**Figure 18 Digital Certificates (sourced from [22])**

Public/private keys are a form of asymmetric encryption in which only the private key can decrypt data that was encrypted via the public key [23]. In Figure 20, the public key of *Mario Rossi* is digitally signed by the certificate authority to prove that that public key is owned by *Mario Rossi*. Information can then be securely sent to *Mario Rossi* without the possibility of being read by someone else. The process of using signed public keys is commonly used in protocols such as Secure Sockets Layer (SSL) which allows users to verify the integrity of a website [24].

For secure authentication between microservices, each microservice could have their own private key and share a signed public key with all the other microservices. This could, if implemented correctly, help prevent a Man-in-the-Middle attack as outlined in Appendix B. The difference between this and token-based security is that microservices can talk to other microservices independently without having to pass a token around.
However, while it can prevent a Man-in-the-Middle attack, certificate-based security ultimately assumes the integrity of the other microservices and thus could fail with the issues outlined in Appendix A. If a microservice was compromised such that the attacker was able to obtain the private key, they could then access anything that key permits access to.

More importantly for the purposes of this report though is that certificate-based security is more focused on authentication. It has similar issues to that of token-based security when authorization functionality is added.

4.5 Security as a Service

A recent paper by Nanda and Jaeger suggest segregating the security service (one or more microservice operating in the Security Bounded Context) by forwarding the incoming and outgoing traffic of the relevant microservice to it. The security service then inspects this information and takes the appropriate action when necessary.

This, like some of the other approaches, requires the microservices within the Security Bounded Context know about all possible actions in all other microservices. As shown in Figure 19, the security server (i.e. the Security MS) is receiving the incoming and outgoing traffic of the Payments MS.

There are many implementation specifics in the paper about how this could work, however, it is possible that this could scale poorly for certain applications. For example, busy microservices could put the security service under equal or greater load than the load they exhibit themselves [25].

5 Critique

As shown above, there are numerous ways of implementing authentication and authorization into a microservice based application. Often a combination of all three approaches is necessary with large companies such as Netflix and Amazon having a multifaceted security system that is highly resilient to a malicious adversary. The reason for multifaceted approaches to authentication and authorization is to help mitigate some of the issues described in Appendix A and Appendix B (and many other issues which have not been mentioned).

Nevertheless, while numerous talks have been given [26], the implementation details are closed-source and thus security remains a challenge for companies looking to use microservices. Yarygina and Bagge in their paper on Overcoming Security Challenges in Microservice Architectures have created a framework. However, as discussed in more in Appendix D, this framework does have a few issues.
For this report, however, the focus is more on design and more importantly, the possibility of exposed business logic between bounded contexts.

### 5.1 Exposed Business Logic

The primary design issue, as alluded above, is that the elegant method of separating microservices entirely into their own bounded context may not be possible with regards to security. In each approach, either the business logic of the Security Bounded Context must be exposed to the other bounded contexts or vice versa to implement authorization. The reason for this can best be demonstrated in code.

#### 5.1.1 Business Logic exposed to Security Bounded Context

Assume for example that setup like that of Figure 13 is being implemented for the online store. Each microservice has actions for both admin users ($userLevel = 1$) and other users ($userLevel = 0$). For instance, admin users may have the ability to perform all actions for any user. The following is a possible method of building authorization exclusively within the Security Bounded Context on the relevant microservice:

```java
// Business logic of security bounded context coupled to microservices
// Note: assume different permissible actions for each block
if ($userLevel == 1) {
    if ($ms instanceof PaymentsMS) {
        // Authorize admin actions of Payments MS
    } else if ($ms instanceof OrderMS) {
        // Authorize admin actions of Order MS
    } else if ($ms instanceof CustomerMS) {
        // Authorize admin actions of Customer MS
    }
} else if ($userLevel == 0) {
    if ($ms instanceof PaymentsMS) {
        // Authorize user actions of Payments MS
    } else if ($ms instanceof OrderMS) {
        // Authorize user actions of Order MS
    } else if ($ms instanceof CustomerMS) {
        // Authorize user actions of Customer MS
    }
}
```

This code is ultimately performing the task of authorizing certain actions of a given microservice (defined by the $ms$ variable) for a given user level (defined by the $userLevel$ variable). This code could be abstracted from the business logic via DIP. However, even though it is abstracted, it is still executed within the bounded context and must have knowledge of the business logic available in all other microservices that require authorization for certain actions.
5.1.2 Business Logic exposed to other Bounded Contexts
Assume in this example that each microservice manages their own authorization like in the approach outlined in section 4.3.3. Each microservice requiring authorization could have a small amount of code, such as a Security Adapter, that elegantly handles the authorization. This Security Adapter uses the Adapter design pattern which is used to convert from one interface to another [27]. In this way, the team managing the Security Bounded Context does not need to know each action of every microservice that requires authorization. The following code demonstrates how this could be implemented on the relevant microservices:

Security Adapter

```php
class SecurityAdapter implements Security {
    function getAccessLevel() {
        // Code to get user access level
    }
}
```

Authorize Task

```php
// To secure a task in a given microservice
$security = new Security();
$userLevel = $security->getAccessLevel();

if ($userLevel == 1) {
    // Admin actions
} else if ($userLevel == 0) {
    // User actions
}
```

This code ultimately means that there is little of the business logic from the Security Bounded Context in the other bounded contexts and most of it can also be abstracted away using the Security Adapter.

However, further assume that the customer wanted another user level which is a staff level user who can do all user level actions and some admin level actions. The security team might have no trouble implementing it in the relevant microservice, however, now the other teams need to implement this functionality into the relevant parts of their microservices. Whereas in the approach discussed in section 5.1.1, the security team could ultimately operate entirely on their own.

5.1.3 Custom Web Tokens
One solution to the issues discussed in section 5.1.2 is that redundant user levels are added in case more are needed later on. A better option though, would involve the use of a token that can hold the permissions of a user. Rather than a simple number, this data is an array of variables that can each be used by one or more microservices to authorize users to specific actions.

The following are some example of tokens that could enable this type of authorization. These tokens have a similar syntax to a very simplified Json Web Token (JWT) which has this functionality through the use of custom claims [28]:
Admin User
{
    "userName": "AdminUser",
    "userId": "LP1L07S",
    "permissions": ["readAnyUserInfo", "updateAnyUserInfo",
                    "deleteAnyUserInfo", "authorizePaymentFromAnyone"]
}

Basic User
{
    "userName": "BasicUser",
    "userId": "2MRV84S",
    "permissions": ["readSelfInfo", "updateSelfInfo",
                    "deleteSelfInfo", "authorizePaymentFromSelf"]
}

As shown, the admin user can act on behalf on anyone in the system while the basic user can only act on behalf of themselves. Permissions such as authorizePaymentFromSelf may only be relevant to certain microservices such as the Payments MS. Whereas, other permissions such as readSelfInfo could be used by multiple microservices.

Given the above permission, a staff user could then be added with little effort:

Staff User
{
    "userName": "StaffUser",
    "userId": "BZBL94D",
    "permissions": ["readAnyUserInfo", "readSelfInfo",
                    "updateSelfInfo", "deleteSelfInfo"]
}

Since the permissions are already variable, a staff user could be just another user with different permissions.

If a microservice requires an additional permission added to the token, it’s possible that some business logic could be exposed to the Security Bounded Context. However, this exposed business logic is likely minute and is only necessary if the additional permission cannot be inferred by the existing permissions (which could possibly be done in the Security Adapter if authorization is done on each microservice).

5.2 Authorization using Observer design pattern

Assume for this approach that custom web tokens are used along with an Adapter, similar to Security Adapter above, which decouples microservices from the implementation specifics of these tokens. Furthermore, assume that the Security Adapter is hosted on a separate microservice.

This approach uses the Observer design pattern which allows objects to ‘subscribe’ to other objects and get relevant updates [27]. In this case, rather than objects, however, microservices will be subscribing to
other microservices. More specifically, the Security Adapter microservice for each bounded context could subscribe to the Authentication MS as shown in Figure 20.

When a user authenticates, the Authentication MS publishes the created resulting token to all the subscribed Security Adapter microservices. These Security Adapters then convert the token into a specific format used among the microservices within the bounded context.

Following that, the Security Adapter microservices then store the minimum amount of data required to map an incoming token from the user to their respective permissions. For example:

<table>
<thead>
<tr>
<th>userId</th>
<th>permissions</th>
</tr>
</thead>
<tbody>
<tr>
<td>2MRV84S</td>
<td>[&quot;readSelfInfo&quot;,&quot;updateSelfInfo&quot;,&quot;deleteSelfInfo&quot;,&quot;authorizePaymentFromSelf&quot;]</td>
</tr>
<tr>
<td>BZBL94D</td>
<td>[&quot;readAnyUserInfo&quot;,&quot;readSelfInfo&quot;,&quot;updateSelfInfo&quot;,&quot;deleteSelfInfo&quot;]</td>
</tr>
</tbody>
</table>

Thus, an incoming token from the user is then sent through to the Security Adapter to unpack the userId and map it to its respective permissions.

This means that the verification business logic within the Authentication MS does not need to be exposed in order for the authorization to occur entirely within the relevant bounded context. In other words, microservices do not need to communicate with the Authentication MS for each user request to check the token (the only other approach that does this is outlined in section 4.3.1).

Note: the reason the Security Adapters run on a separate microservice is so that the Authentication MS does not need to publish to all other microservices but rather publish once for each bounded context (in
some cases it may be better to have the Security Adapter on the relevant microservice such as when it’s the only microservice in a given bounded context).

Issues with this approach

Like the approach discussed in section 4.5, it would also potentially generate a large amount of network traffic. Even if there was only one Security Adapter microservice for each bounded context, it would need to be updated for every user that authenticates with the Authentication MS. Thus, many of the published tokens are redundant since they are only relevant to the microservices that the user is requesting from. There is also the possibility of increased traffic between microservices within the bounded context (i.e. between the Security Adapter microservice and the other microservices). This, however, could be outweighed by the decrease in traffic to the Authentication MS for each user request.

Furthermore, it is also possible that the authenticated tokens could get out of sync with the Authentication MS (i.e. the published token failed to get to the Security Adapters microservice) which would mean genuine users are rejected. Finally, it is possible that the entire token will have to be stored rather than just the userId for reasons discussed in Appendix C.

5.3 Authenticate through an Adapter

The final approach is almost identical to what was described in section 4.3.3. However, this approach also makes use of customer web tokens and an Adapter to uncouple from the implementation specifics of these tokens.
As shown in Figure 21, the user authenticates with the Authentication MS in the same way as before. It also requests the protected resource, with the token, to the Payments MS which forwards it to the Authentication MS for verification. However, in this case, the token is changed to a ‘verified’ token and then passed through the Payments Adapter to be useable by the Payments MS for the authorization.

**Issues**

Like the approach in section 4.3.3, this approach also suffers from some exposed business logic from the Security Bounded Context. However, this exposed logic is mostly abstracted away using an Adapter while the custom web tokens enable variable permissions.

Furthermore, like many of the other approaches, the Authentication MS must verify the token for every request. This, however, may be a small task depending on the implementation.
6 Conclusion

The outcome of this report suggests that microservice development is difficult and few concrete design principles exist. DDD was suggested by most of the relevant literature to drive microservice design. However, DDD can only really be used superficially as it is a methodology rather than a framework or standard. For example, it was mentioned that a microservice should be no larger than a bounded context. A bounded context, though, can be quite large in size and how to split up microservices within a bounded context can be highly difficult even before it’s decided how they are to communicate.

Nevertheless, the previously mentioned steps that are summarised in the following could prove useful in building a microservice based web application:

1. Identify Domain
2. Define bounded contexts
3. Identify Microservices
4. Design Microservice Specific Architecture
5. Implement
6. Deploy

With regards to security though, it was also found that some of the tight boundaries between bounded contexts may need to be slightly relaxed in order to accommodate both authentication and authorization. The small amount of exposed business logic, however, is outweighed by the possibility of a cyber-attack which can cost companies millions [29]. The approach drawn from the literature, as discussed in section 5.3, seems to minimize the exposed business logic while enabling both authentication and authorization. Nevertheless, other approaches may be more suitable for certain projects even if they result in more exposed business logic.

7 Further Research

Some of the approaches above may function hypothetically but break down when applied to functioning microservices. Thus, further research would involve implementing these approaches and stress testing them each from both a security and performance perspective. The result of these tests could then be used to further develop a viable solution. Each microservice could be, for instance, built using a different language, database and architecture to ensure that the solutions accommodate the necessary diversity.
Bibliography


Appendix

A. Compromised Microservice

With monolithic applications, security is often applied to all facets of the application similarly. This is because, if one component is compromised, then the other components are likely also compromised since they are required by each other to function.

However, for a microservice based application, as mentioned, there is a possibility of single microservice being compromised and the others not. The method described in section 4.2.4 does not enable the attacker to directly access other microservices as they are secured using a token. Nevertheless, if a malicious user Eve has compromised a microservice, she could simply wait for a genuine user Alice to send her token to it. Eve could then act on behalf of Alice by accessing the other microservices using her token.

![Figure 22 Compromised Microservice](image)

In Figure 18, Eve has compromised the Order MS and then manages to intercept Alice’s token. Upon doing so, Eve now has access to the Payments MS (and likely other microservices).

B. Man-in-the-Middle Attack

Furthermore, assume in another scenario that the microservices are more secure though Eve manages to compromise a network device (such as a router). It may not be possible to snoop the token directly on the router, however, Eve can disguise herself as the Order MS and then trick Alice into using that instead of the genuine Order MS.
As shown in Figure 19, the Alice’s Token, is now accessible by Eve via the disguised Order MS.

C. Mitigate Attacks

The attacks described in Appendix A and Appendix B can be avoided in several ways which often make use of a multifaceted approach. For example, tokens could also be blacklisted if suspicious activity is found on the network (via the Security-as-a-Service discussed in section 4.5). Also, token-based security could make use of token timeouts that do not give an attacker enough time to replay the token and obtain access to the protected resource.

Furthermore, certificate-based security could be used along with the tokens to verify the integrity of the user who sends it. This prevents an attacker from simply building their own token and gaining access to protected resources. The following diagram shows how this could work for a JWT web token:
As shown in Figure 24, the JWT token is signed by the private key of the server (which would be a microservice in the case of this report) and then sent to the client. This token is then sent back to the server which validates it using the public key preventing a malicious user from building their own token [30].

To implement this in the approach discussed in section 4.3.1, each microservice would also need to public key of the Authentication MS for the checkUser function.

For the approach discussed in section 5.2, rather than the userId used to uniquely a given token, it’s likely that the entire token will have to be stored (i.e. entire encrypted text of the respective token). This is because each Security Adapter microservice would otherwise need to know the public key of the Authentication MS which exposes even more business logic.

D. Implementations in Existing Research

Yarygina and Bagge conduct a case study tasked with building a secure microservice framework in Python using containers in Docker. This framework addresses problem of building trust between microservices [31]. It follows DDD design principles and does not require a specific set of API rules to function. This framework was created as a generic open-source (can be found Github here: https://github.com/yarygina/MiSSFire) implementation that responds to closed-source implementations such as that of Netflix [19].

However, through analysis, not only can this framework really only be used as a guide, it does not seem to be actively maintained since the initial publication of the paper. Furthermore, according to the paper, this framework seems to be more focussed on efficiently executing security mechanisms in
microservices and somewhat less on the structural details of applying security in the perspective of DDD. Nevertheless, this framework could prove to be very useful for those looking to implement microservice security.