### **Securing Microservices**

**Encryption**

Encryption plays an important role in protecting our data. We want to ensure that the data is encrypted across the network, so that no intermediate party is able to see what we are communicating. For HTTP communications, encryption can be achieved by configuring TLS (Transport Layer Security). So, all requests are made using HTTPs.

You may also require that you data should be encrypted at rest/storage. Means data should also be encrypted whenever stored on disk.

**Authentication**

Each incoming request needs a way to tell us who the caller is, so we can make decision as to whether they are authorized or not. With HTTP protocol, the most common approach is to include an authorization header in each HTTP request. Like,

**Basic Authentication**: Where we pass the username and password in the request Header.

**API Key:** With this approach we give each valid client of your microservice their own API key.

**Client Certificates:** By using public-key cryptography, a certificate gives us a very secure way to allow a caller to prove their identity.

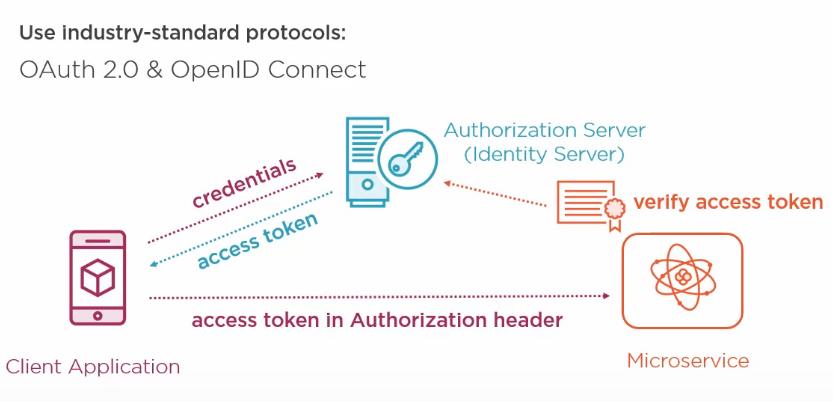
**OAuth 2.0 & OpenID Connect:** This introduces the concept of an Authorization Server. The way it works is:

i- The clients of our microservices authenticate themselves with the identity server by sending credential of some sort.

ii- And then the identity server return an access token which has got a limited lifetime.

iii- That access token can then be passed in an Authorization HTTP request header as part of the call to the microservice.

And these tokens are signed using pulbic key cryptography, meaning that it’s possible to verify that the token really was issued by the identity/authorization server.



OAuth has lots of benefits:

1. Only one service has the job of authentication users and managing their credential securely. And because the identity server is implementing industry standard protocol, means you don’t need to write the component by yourself. You can make use of a third party solution. Which is great, because writing your own identity server is a lot of work.

**JWT (JSON Web Token):** This is another way of securing microservices. We provide username and password, and the the server gives us a token in json form. And that token is being used, which will be present in the request header, whenever we make request to the microservice.

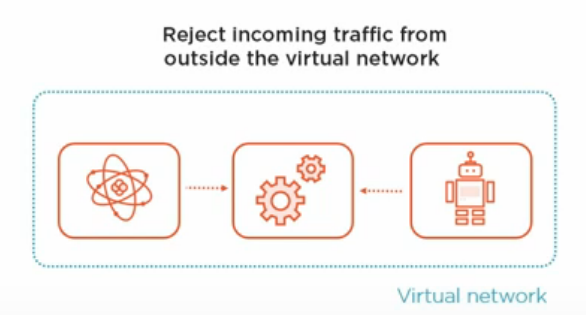
**Authorization**

When a user is authenticated, that simply means that we know who they are. But we also need to check that they are **authorized** to perform the action that they’ve requested. For example, If I log in to my e-shopping account, then I’m authorized to view my own order history. But I shouldn’t be authorized to view your order history.

We need to provide mechanism that helps in performing authorization by checking various things about the user, such as whether they’ve got a particular role or not.

**Securing the Network**

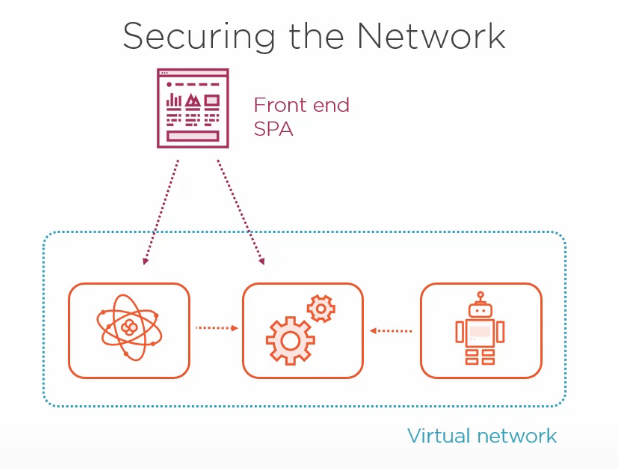
To secure a network, firewall, IP White listing and virtual network is use.



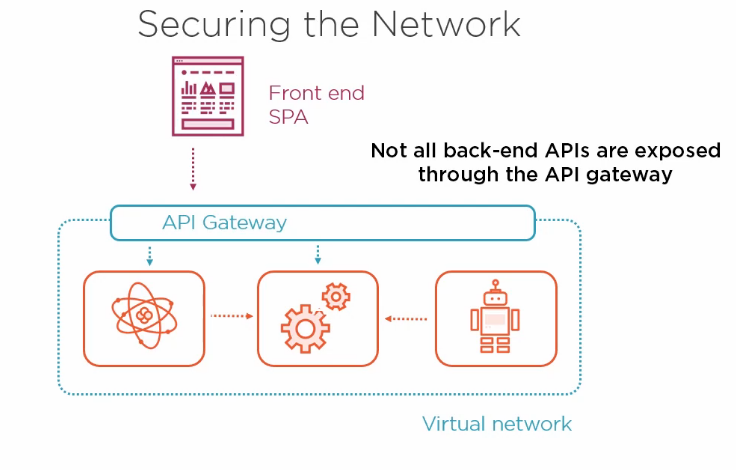
In the above image we can see 3 microservices communication with each other. And if we can place them inside a virtual network, then we have the ability to reject any incoming traffic from outside the virtual network and only allow them to communicate with each other. It a good approach to protect the back-end services.

But what about the situation, where you do need to accept traffic from the internet?

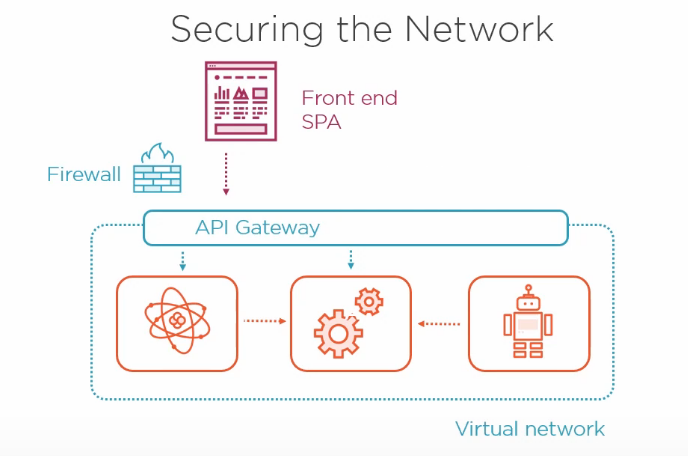
For example, if you’ve got a website, implemented as a single-page application.



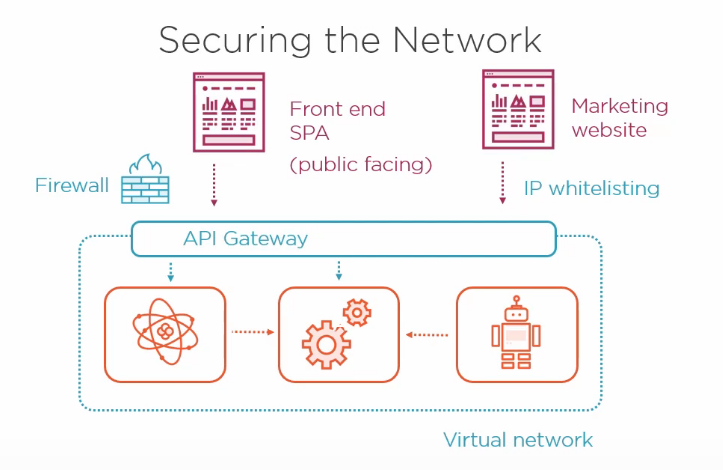
Then it’ll want to make request to your microservices. So does that mean you have to forgo any kind of network restrictions and just open everything up? Well this is actually an example of where the pattern that we discussed earlier, the API gateway or back-end-for-front-end (BFF) can be really helpful. We can have an API gateway that accepts incoming requests from the public internet but is also connected to the virtual network and so is able to pass on incoming requests to the backend microservices. And this means we can be very selective about exactly which back-end API are able to be called from the outside.



It also means that we’ve got a single point of entry. Some cloud API gateways can also be configured with a firewal or other types of attack protection such as guarding against distributed denial-of-service attacks Or SQL Injection attacks.



Often in a microservices application, there’s a public-facing website. But there may also be a additional website intended to use by a smaller targeted group of users, such as system administrators. The only people who are authorized to accesses such website only the admins. So we might decide to apply IP address white-listing, so that this website rejects any traffic that doesn’t come from certain known IP addresses.

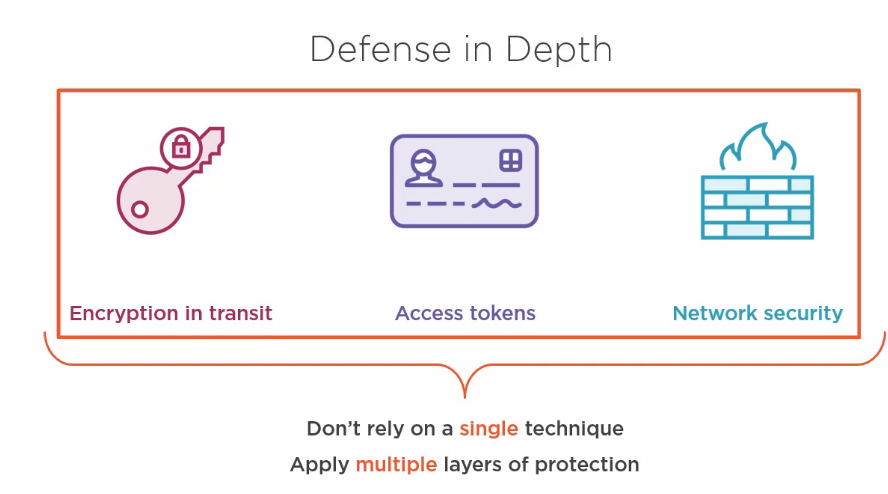


So far in this topic, we have learned about different technique for securing our microservices, and you might be wondering whether you need to use all of them or whether you can just pick one or two. And this is where the **Defense in Depth** principle comes in. This principle encourages us to avoid relying solely on a single layer of protection and to use multiple layer of security wherever possible.

**Defense in Depth**

We’ve discussed a number of techniques to secure our microservices. By using TLS (Transport Layer Security) as encryption in HTTPs. By Using Access Tokens issued by an identity server to authenticate ourselves in the requests. And by using virtual network and IP white-listing, we can reject any calls to our microservices that come from unauthorized network location.

The Defense in Depth principle states that you shouldn’t rely entirely on a single technique to secure your application, and that’s because if that one defense is breached, then everything is lost, and your attacker has gained free access to everything. So by combining several layers of security, we can significantly reduce the possibility of a data breach. And the most sensitive the data that you’re dealing with, the more important it is that you apply several layers of protection.



There are some additional defensive measures that we can take. For one thing, modern hackers have variety of very sophisticated tools and techniques at their disposal. And so often software developers aren’t aware of all of these attack techniques and don’t have the knowledge of how best to mitigate against them. And for that reason, it’s very good idea to arrange for penetration testing to be performed by a team of experts who can test the defenses of your application against state of the art hacking techniques and can also give you some expert advise on how to increase the security of your applications. Also, it is recommended to create automated tests that verify that your security is working as expected.

It may also be possible to detect when an attack is in progress. Port scanning, repeated login attempts, malicious HTTP requests, Phishing attacks, SQL injection attack attempts, all of these things can be detected in real time.