Securing The API Stronghold

The Ultimate Guide to API Security

AUTHORS

Travis Spencer
Kristopher Sandoval
Bill Doerrfeld
Andreas Krohn
Jacob Ideskog

NORDIC APIS
nordicapis.com

Securing The API Stronghold

The Ultimate Guide to API Security

Nordic APIs

©2015 Nordic APIs AB

Contents

Preface

As the world becomes more and more connected, digital security becomes an increasing concern. Especially in the Internet of Things (IoT), Application Programming Interface (API), and microservice spaces, the proper **access management** needs to be seriously addressed to ensure web assets are securely distributed.

During the Nordic APIs World Tour - a five day international conference we held in May 2015 - our speakers consistently reiterated the importance of **API security**. So, to help providers secure their systems, we at Nordic APIs have collated our most helpful advice on API security into this eBook; a single tomb that introduces important terms, outlines proven API security stacks, and describes workflows using modern technologies such as OAuth and OpenID Connect.

Founded on insights from identity experts and security specialists, this knowledge is crucial for most web service platforms that needs to properly authenticate, control access, delegate authority, and federate credentials across a system.

Following an overview of basic concepts, we'll dive into specific considerations such as:

- Vulnerabilities and what whitehackers look for
- How to implement a secure versioning strategy
- The three distinct approaches to API licensing and availability

- Performing delegation of user identity across microservices and IoT devices
- Using the Neo-Security stack to handle identity and access control with OAuth 2.0 and OpenID workflows
- Differentiating Authentication, Authorization, Federation, and Delegation, and the importance of each
- Using OpenID Connect for Native Single Sign On (SSO) and Mobile Identity Management (MIM)
- Ways to introduce a culture of security into your organization
- Securing your data stream at the point-to-point level
- And more...

Please read on, share, and enjoy the 5th free eBook from the Nordic APIs team!

– Bill Doerrfeld, Editor in Chief, Nordic APIs

Connect with Nordic APIs:

Facebook | Twitter | Linkedin | Google+ | YouTube

Blog | Home | Newsletter | Contact

1. Introducing API Security Concepts

> "Knowing *who* has the right to do *what* with your API is key to success" - Andreas Krohn, Dopter

> "Design all API security with public access in mind" - Phillipp Schöne, Axway

Application Programming Interfaces or APIs are not only an extension of the social web, but continue to seriously disrupt entire industries, change how Business-to-business (B2B) communication is throttled, spark innovation, and even inspire social change. Simply put by TechCrunch, "APIs fuel the software that's eating the world."

Within this vibrant and quickly expanding economy, an increasing amount of data is being funneled through systems not designed with the scale of protection that is necessary. The risk of cyber threat is now the highest it has ever been, and it won't stop anytime soon. To combat this threat we must take the smart precautions to arm our systems. We build with the assumption that even private APIs will sooner or later become exposed to the public, and embrace proper security implementation as a top concern.

1.1 Identity is at the Forefront of API Security

API security isn't just about the API itself, but also about the security of entire organizations and mobile products when they intersect with APIs.

When developing an API, the security of the **mobile device** matters just as much as the security of the API. Does it have anti-virus software installed? Is it enrolled in a mobile device management solution (MDM)? Does it have mobile application management software (MAM) installed? You also need to worry about **enterprise security**. Are the servers secure? Do your machines have intrusion detection?

At this junction of APIs, business, and mobile, lies the individual. Only when you know *who* is at this core will you know *what* they should be accessing and *how* they should be accessing it.

1.2 Neo-Security Stack

When we start to expose high-value information and resources, we need to have high-level assurance of who is accessing them. API security is comprised of a number of protocols, which Twobo Technologies refers to as the **Neo-Security stack**. This standards-based cloud security suite is usually comprised of these protocols and technologies:

- **OAuth 2**: The open standard for secure, delegated access
- **OpenID Connect**: For federation which allows for the secure exchange of user authentication data

- **JSON Identity Suite**: The collection of JSON-based protocols for representing the identity of users
- **SCIM**: System for Cross-domain Identity Management for user account provisioning and deprovisioning
- **U2F**: Universal 2-factor authentication for asymmetrically identifying users with a high degree of confidence that they really are who they say they are
- **ALFA**: For defining fine-grained authorization rules in a JSON-like policy language (which compiles down into XACML)

While the Neo-Security stack creates a comprehensive security solution for mobility, it is a great challenge for API developers to manage a myriad of specifications themselves.

1.3 OAuth Basics

As the risk associated with an individual's online identity increases, we need to ask permission before exposing identity and any vulnerable resources with an API. OAuth is a framework used to build API security solutions - a framework or meta-protocol under which we create other protocols to define how tokens are handled. Despite its name, OAuth is not for authentication, federation, or even authorization; it helps *delegate* access, ie. giving an app access to your data or service. A benefit of using OAuth is that somebody else authenticates users.

These following factors make up OAuth2 Protocols:

- **Client:** The web or mobile application involved

- **Authorization Server (AS):** The security token service, which issues credentials and tokens that represent the resource owner
- **Resource Owner (RO):** Authorizes or delegates access to the RS
- **Resource Server (RS):** Often the API itself, a collection of libraries and web applications

How these four OAuth2 actors work together varies with each integration. We'll dive into the processes behind common OAuth server flows in future chapters, but to summarize:

OAuth Web Server Flow

A **"three-legged OAuth"** process occurs when an end user specifies that he or she wants to delegate access to a third-party application for use within the client application. The client application then redirects this request to the AS, which requires authentication for identification. The AS then authorizes that client and the RO is redirected back to the web app with a single-use access code.

The single-use access code is sent back to the AS, which then converts it into an **access token** that the end user may use to access the server. At the same time, the AS may also send back a **refresh token** which will allow the end user to use the same OAuth to access more than once.

Essentially, the access token allows a user to call the API. In return, the API gains access to information about both the client and the resource owner and what path they took, what client they are using, and who is the end user. With this information, you are able to create much more

complete web access control decisions that improve API security. Everything is built right into OAuth2, limiting human-designed security errors.

1.4 OpenID Connect

OpenID Connect is a standard that complements OAuth to add user identity to an API security solution. OpenID Connect gives you a standardized identity layer with standardized, researched risk mitigation and continuous examination of new threats. OpenID Connect is often used to connect internal applications and share user information.

OpenID Connect and its predecessor SAML focus on authentication and federation, identifying a user before information is pushed to a third-party. Using OpenID Connect, you could give strategic partners similar access to make API calls.

There are even third-party organizations responsible for determining identity, often called an "identity provider" or IdP. Developers are able to use an IdP and push that access to a third-party app. This allows the IdP to act as an identity provider as well as an API provider, allowing the IdP, acting as an API, to exchange data with the app.

OpenID Connect builds on top of OAuth2 to define a federation protocol. Optimized around the user consent flows, OpenID adds identity-based information on top of OAuth into the inputs and outputs.

SCIM Identifies User and Group Schemas for Developers

SCIM defines the RESTful API protocol to manage and

specify user and group schemas and the properties on them, including defining the markup for representing things like first/last name, email address, actual address, and phone number. It enables you to identify users within groups so you can easily add or remove them, without having to reinvent the API.

Different layers of the Neo-Security stack can be combined. For example, OAuth is used to secure SCIM API calls for instances like delegated access for creating and updating users. SCIM and SAML or OpenID Connect can be bound to provide just-in-time provisioning (JIT), which allows you to create, update, and delete users without tying it to an authentication event.

1.5 JSON Identity Suite

The identity-based information provided by OpenID Connect is marked up in something called Jason Web Tokens or JWTs (pronounced “Jots”.) JWTs are part of the JSON Identity Suite that the Internet Engineering Task Force (IETF) has defined. JWTs are designed to be light-weight tokens that can be easily passed in that HTTP header. They are like SAML tokens, but are less expressive, more compact, with less security options, all encoded in JSON instead of XML.

1.6 Neo-Security Stack Protocols Increase API Security

There’s no doubt that security risk increases as more and more APIs are opened up to connect and share information. The technologies and procedures described in this

article aim to decrease vulnerability. Though the IETF has outlined a set of protocols to guide API development and exposition, API providers must educate themselves on the language of security in order to avoid human error and prevent API attacks.

1.7 The Myth of API Keys

There are many misunderstandings surrounding API security. For one, API keys are not API security. API keys are inherently insecure - essentially being a password that is hardcoded into applications and often distributed all over the place, giving anyone with the key access to the API. OAuth is not API security either - simply adding OAuth to an API does not make it secure. What an API needs is a *holistic approach* to security, imbued with enterprise and mobile security.

1.8 Access Management

An important API security case is user-to-user delegation. This is when you give another person rights to access your data instead of giving an app the rights to access your data. This is made possible by using OAuth and OpenID Connect with **Delegated Tokens**. A Delegated Token is a form of OAuth Access Token that also contains user information.

1.9 IoT Security

Securing APIs is important, but we need a holistic approach to security. This is especially a challenge throughout the IoT, since very important assets such as the lock on our house or car will be made available online.

> "The dangers lie in the seams between the devices and the Internet and that is what needs to be carefully handled" - Brian Mulloy, Apigee

According to Brian Mulloy, web developers know how to handle security on the Internet using OAuth and OpenID Connect. At the same time, device makers have experience in securing the physical devices themselves. It is important to think about security end-to-end, from physical device all the way up to an app interacting with that device via an API. One weak link can break the whole chain.

1.10 Using Proven Standards

Good API security is an enabler. Unless end users and developers can trust your API, none of the business goals you have defined will be fulfilled. Some reccomend that you do not build your API security solution yourself, but rather base your security on known standards and battle tested products. Keep reading for actionable insights into each section covered in this summary chapter.

2. The 4 Defenses of The API Stronghold

At one point or another, your secure resources will be attacked. This is the unfortunate reality of the modern era, where the skills necessary to invasively crack open a system, network, or API are more commonplace than ever. Millions in resources and potential revenue can be lost in a matter of hours due to poor planning or implementation of a security protocol.

Private information, trade secrets, and even personal data can be exposed to the skilled network penetrator and used against you in ways that are as extreme as they are varied. This chapter aims to bolster your defenses by defining the four foundations of API security: **Authentication**, **Authorization**, **Federation**, and **Delegation**.

The Importance of Comprehension

One of the most common failures of understanding in the development of **API security** is the idea that security is a "one size fits all" solution. The idea that channels of security function in a singular manner with variations only in the final end-user experience is as wrong as it is dangerous; this concept places the operator in a position of fewer tools at their disposal than those trying to "break the system", in turn exposing your data to an extreme level of unnecessary risk.

This issue is made worse by a common misunderstanding regarding the differences between **Authorization, Authentication, Federation,** and **Delegation** — perpetuating a gulf of misinformation and misapplication. This chasm has caused many security woes in the API space.

Given that **these four terms are what the entire concept of API security rides on**, it is imperative that you understand them, their uses, and the implications of their adoption. Lacking a basic understanding of security protocols, methodologies, and concepts can mean the difference between a prosperous, growth-oriented business and a stagnant, deprecating business.

2.1 Balancing Access and Permissions

Having an API that allows for full access to the entirety of your systems and resources is an absolute nightmare — it's akin to being the lord of a castle, and leaving the doors to your vault wide open — it entices theft by its very nature, and unnecessarily opens your materials to the public space. As lord of your keep, why give marauders an avenue for attack?

What then is the proper response? Do we leave systems open for the betterment of functionality and assume users have positive intentions? Or do we follow a complete opposite route, closing down every bit of functionality, designing only proprietary systems?

The solution lies in combination of both approaches. Assume your foe is out to get you at all times — but don't let this affect your need to do business. Constant vigilance allows you to design your API in an intelligent way, opening only that which needs to be opened, and making sure those openings don't tie into vital systems that could be damaged. Functionally, this means assigning elements of authority to API consumers based on the minimal amount of access they need to do the functions they are required to do. By assigning different roles and levels of responsibilities to clients, we can create a **tiered environment** that keeps our data safe.

Understanding the implications of this system is of prime importance — specifically the differences between the types of rights and authorities within the system as a whole. In our definitions, we'll hark back to our castle defense analogy. The brave and powerful knight Lancelot is trying to return to the Arthurian court after a long month of fighting bloodthirsty marauders — let's see what defenses he will have to pass...

2.2 Authentication: Identity

"'Tis I Lancelot cometh, bearing the password 'Guinevere.'" Authentication is two-fold, establishing trust with identity and a secret code.

Authentication is a base security layer that deals specifically with the **identity** of the requesting party. Think of authentication as an agreement based on trust. When Lancelot reaches your drawbridge, he shouts his name along with a secret code to a guard stationed on the castle wall above. These two forms of identification will ensure that Lancelot is *identified* only as Lancelot, and that Lancelot is *allowed* to enter the castle.

In the real world, this level of security can take multiple forms. When a user chooses to request data or resources, they can face a number of protocols, log-in systems, and

verification services. For instance, a user could log in using an authentication service that only requires a username and password. For greater levels of assurance, they may then be asked to provide a single-use token generated on a mobile device or keyfob.

2.3 Authorization: Access

"Says here you're Round Table status." Authorization considers what access level the user has with the system. [Image: CC-BY-SA-3.0 via Wikimedia Commons]

Authorization is a completely separate level of security, though it's often confused with authentication; while authentication verifies that the requester is who they say they are, authorization determines the **access level** they should be granted.

As Lancelot waits for the drawbridge to come down and allow him in, the guard steps back and checks his "Ye Olde Book of Permissiones" to make sure Lancelot has the right to enter the castle. This is different from Authentication. Whereas Lancelot proved he was who he said he was, Authorization makes sure he has the *right* to enter the

castle, and if he indeed has that right, *what levels* he can access within the castle. These permissions are granted to Lancelot by King Arthur, allowing Lancelot access to The Round Table, among other resources that peasants can't access.

Authorization is extremely important, but is often overlooked. It is very easy for API developers to assume that, because they need access to the API for their systems, setting the user default permissions to "SysOp" or equivalent full access is acceptable. This is a faulty line of thinking. While the developer inherently requires a higher level of access than the typical user, **a consumer may only need a small portion of calls or functions at any one time**. In that situation, leaving excess operations open makes the system more vulnerable to hacks and unauthorized access.

2.4 Federation: Reusing Credentials & Spreading Resources

Federated security is a multi-purpose system:

- for **users**, federated security systems allow for the use of a small set of credentials with multiple systems, services, applications, or websites.
- for **administrators**, federated security allows for the separation between the resources requested by the user and the systems used to authenticate and grant authority to the user.
- For **organizations**, it allows them to centrally manage the trust relationships they have with one another and ensure, cryptographically, that that trust is enforceable.

Same User Credentials Across Multiple Services

With federation, the user is granted the ability to use the same set of credentials across multiple services. By having the authentication take place in one single domain, other security realms that trust this primary domain can reuse the authentication and trust the authenticity of the identity established. This results in what is to what is called a **federation**.

Any system in this federation can accept the credentials of the authentication domain. The primary domain is what we call an **Identity Provider** (IdP) or **Asserting Party** (AP); the other security domains that trust the IdP to authenticate users are referred to as **Relying Parties** (RP) or **Service Providers** (SP). Authentication and identity data are passed between these parties using tokens. These Tokens are minted by a system called a **Security Token Service** (STS) or a **Federation Service** (OAuth Authorization Server and an OpenID Connect Provider are examples of an STS and a Federation Service, respectively.)

The end result is that a STS hands a token to the user after they first log into that authentication service. When the user then requests access to another domain, the domain registers that the user already has a token, and grants it access without requesting another log-in.

Introducing Realms

A federated backend stores resources across various systems, allowing users to access multiple services with the same credentials

The King knows that knights like Lancelot need to enter his castle; he also knows that his castle is situated in a very bad location, prone to raids. Because of this, the King has set up another castle some miles away in a more defensible position to house other precious resources. He does this to ensure security between the main castle and the more fortified castle that contains other valuable treasures. This adds an entirely separate layer of security, acting as a "buffer." Federation allows for Single Sign-on (SSO) across these different "security domains" or "realms."

In traditional systems that do not use Federation, the user

would log into a server that is in a particular security domain/realm. This domain would include not only this authentication system but also the actual resources the user is attempting to access. While this works fine in some cases, if the authentication and authorization protocols were to be broken or violated, then the barrier between the resources and the user would also be broken. This could allow an attacker to take complete control of the domain.

When A Breach Occurs

With federated security, if a breach occurs in the Identity Provider, the Relying Parties can revoke the trust it had previously placed in that party — **not all systems are compromised**. Entry to all federated authentication requests made by any user to the resource server are refused. Imagine the King and his knights have retreated to the fortified castle after the main castle was overrun by raiders. They no longer allow entrance from people requesting access to the fortified castle for fear that they are actually raiders using stolen code words.

2.5 Delegation: The Signet of (Limited) Power

Delegation can be compared to a signet, carrying the seal and permissions granted by an API provider

Delegation is another process by which **access** and **rights** can be given to authorized users while maintaining a relatively **limited** amount of access. Whereas federation works by giving the user a token to use on multiple domains, delegation works by authorizing a user to function partially as if they were **another user**.

King Arthur, seeing the plight of Lancelot the Knight, has decided to make future access to his Kingdom easier. He creates a ring with his kingly seal set in pearl, and gives it to Lancelot. He then instructs his subordinates to follow Lancelot's orders, given that they fall within his rights as a Knight and do not exceed the authority previously expressly given to him by the King. In this way, when

Lancelot presents his signet, he is seen to be acting under the orders of the King, and does not need to further verify himself or be authenticated.

For web users, this is done on a site-by-site basis. For example, let's assume the user is trying to log in to "http://mail.corporate.or Under the federation system, they would first have to login on a designated portal page to access any additional resources in the federation using their token. Under the delegation system, "http://mail.corporate.org" would look to see what rights they have been given and who they are acting on behalf of. As the user is coming from "http://profile.corporate.o and has already logged in on that page using an authenticated account with elevated privelages, the mail server allows access.

2.6 Holistic Security vs. Singular Approach

Most important to all of these considerations is the way we treat the fundamental security system we are implementing. Far too often, developers fall into the trap of considering security to be **one-sided**. An average user might consider having three authentication devices to be a good security check; but if that authentication server were to ever go offline, you could have an infinite number of authentication-based security systems and your network would still be exposed. By thinking of security as a singular approach rather than a holistic one, you are placing your API and your system in **far more danger** than is necessary.

Consider security from the constraints of our story concerning Lancelot, and put yourselves in the rather silky, comfortable shoes of the noble and wise King Arthur.

You know invaders are coming; in fact, you can see them crossing the mountain now, preparing to invade. Examine your security, and really contemplate your entire **API Stronghold**.

Would you consolidate all your jewels and gold in one fortress and defend with all men on a single wooden gate? — OR — would you rather spread your wealth across multiple fortresses each with an impassable moat, a manned wooden gate, a manned metal gate, and armed warriors waiting just beyond? Operationally, costs may be the same, but the security is drastically different. In the first scenario, the enemy would only have to destroy the wooden gate once to get into your castle, whereas in the second scenario, they would have to pass four separate and daunting obstacles to even get a peek at a single inner keep. A multi-layered stronghold is how you must consider security in the API space.

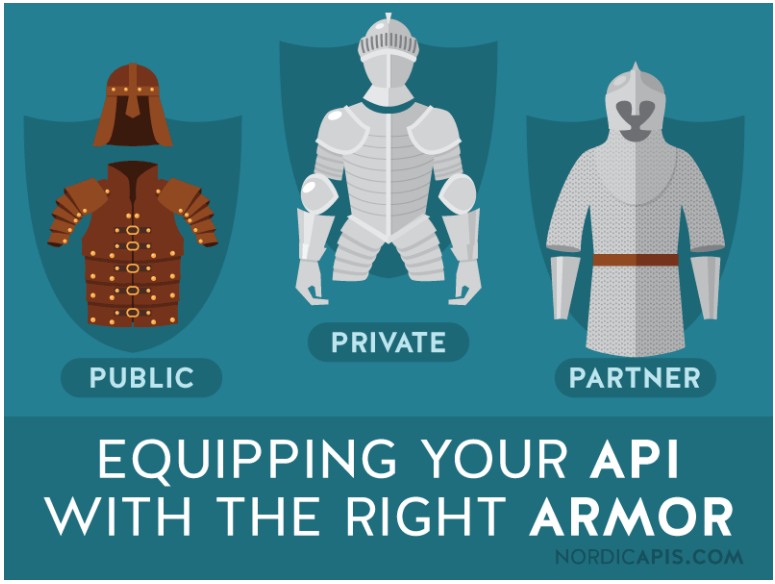
2.7 Application For APIs

When establishing a security system for your API, understanding **Authentication**, **Authorization**, **Federation**, and **Delegation** is vitally important. Deciding the access and specific circumstances behind sharing your resources will help establish a security shield to protect internal assets and solve many security issues before they arise.

For the modern system administrator, there is a wide range of tools and services available that make implementation of any of these types of security relatively pain-free. Services like OAuth and OpenID Connect can be integrated early in the development cycle of the API, and third-party authentication services can even be implemented after the API is deployed.

While all the above systems can work in tandem with one another, knowing when and where each are best applied makes for a better system on the whole, improving your security, usability, and longevity. In the following chapters we embark on a more specific journey to define these actors and workflows.

3. Equipping Your API With the Right Armor: 3 Approaches to Provisioning



APIs are as vast and varied as the systems that depend on them; while some systems may handle client data, payments, and collaborative research, other APIs may handle less important data such as social media and image sharing.

Due to the variations in application and usage, APIs face

a unique set of considerations and issues arising from their availability that no other system faces - the difference between having an API for public consumption or one intended for internal use, in certain circumstances, can mean the difference between megalithic success and catastrophic failure.

Understanding the specific business objectives involved in making your API private or public is paramount. In this chapter, we will identify the difference between Public, Private, and Partner APIs, and when each type makes sense in different scenarios. We'll also tackle a few examples of failure in API restriction and documentation, as well as a few success in design and development.

3.1 Differences In API Approaches: Private, Public, & Partner APIs

When considering the interaction between APIs and their intended markets, there are three general approaches to licensing and availability.

Private APIs are APIs intended to streamline internal operations, translating calls in a hidden way. While they may be documented, this documentation is largely limited and is meant for internal purposes only. Development is handled entirely by the creators of the API, and no derivations are allowed. While this type of development and implementation cycle can be very effective in creating and maintaing accelerated growth, your API is limited by what you have designed it to do — your API will not evolve with the times

and needs of the marketplace unless you specifically force it to, which can be both time and money intensive.

Public APIs are APIs that have certain public access, working more transparently and allowing other users, services, and systems to tie into parts of the underlying interface. These APIs are usually heavily documented, as they will be used publicly by developers to utilize the API for various services. For some very transparent Public APIs, "Forking" the API — creating new derivations of the API — is open to any who wish to do so, making it evolve organically. This is rare though, as most public web APIs are still closed-source, allowing a company to maintain full ownership. In general, Public APIs allow a third-party developer ecosystem to be created with your data or services, an organic evolution with new posibilities and services, potentially resulting in explosive and sustained accelerated growth. The downside of a Public API is that it could decrease the value of your main distribution channels, and, in some situations, direct monetization opportunities that would arise from a more locked-down Private or Partner API.

Partner APIs are B2B APIs that are exposed through agreements between the developer and the client, outlining the application and use of the API. Documentation is usually thorough, but only for those interface calls that will be used by the business as defined in the contractual license; other functionality is locked down, undocumented, or requires authorization and authentication to utilize. Such an API is highly effective and can be extremely lucrative. Deriva-

tions can be contracted out, but all of the API development is synthetic and directed rather than organic.

3.2 Considerations and Caveats

What, then, is the choice that drives one business to the Public API model, and another to the Partner API? When considering the type of API licensing and availability, several considerations should be made at the beginning of the API Lifecycle, as they can fundamentally change the approach to the development of your API by changing the methodology of implementation.

Firstly, consider the nature of the data and systems that your API will reference. When designing an API that taps into secure, personal data, such as social security numbers, payment records, and general identification services, the amount of security needed for your API shifts the balance drastically towards a consideration of either private or business-to-business licensing.

This is largely due to the vulnerability of public-facing API code and documentation. Think of your licensing as a suit of armor. While plate armor is heavy and makes movement hard, it is extremely secure. Leather armor, on the other hand, is extremely light and flexible, granting extreme agility. Your licensing is much the same — Private APIs being plate armor, and open-source Public APIs being leather. Business-to-business Partner APIs are like chain mail— somewhere in the middle in terms of granting agility and protection. While the use of Public APIs can be somewhat controlled (or tanned in the spirit of our armor metaphors), they will never be as secure as a closed-source and plated Private API.

Secondly, consider the amount of upkeep you wish to

perform. All API types can have arduous, long, and expensive development cycles, relying on proprietary systems and interfaces. Costs and resources may be extended for Public APIs in order to maintain public-facing developer portals and upkeep external support systems such as marketing channels, hackathons, and other API product operations.

Finally, monetization should be considered. While closed-source Private APIs can be monetized through improved operational efficiency and content promotions, Partner APIs can be licensed at a fee for direct revenue; open-source Public APIs can also be monetized through the direct sale of data and other methodologies.

3.3 So Where Is The Middle Ground?

So where, then, is the line? What is the ideal combination of security, access conditions, and monetization for your specific situation? Functionally, your API should only be as public-facing as it is required to be by its functions, its designs, the limitations of your business, and the objectives of its implementation.

For example, if one were developing an API that would use data provided by third party vendors to collate data for analytics, what would be an appropriate amount of exposure? Given the nature of the data, and the security requirements likely put in place by the vendors, a Partner API would be appropriate; after all, other partners will have to tie in to your API to deliver data, and you will need to provide a certain amount of documentation for that to occur.

Conversely, if we were designing an API that delivers social media or RSS feeds to a user, what would be an

appropriate type of exposure? In this case, because the data is entirely open, is not tying into systems, and utilizes public data, a Public API is appropriate. In this scenario, open-source is perfect, as it allows the maximum number of users to utilize the API and improve it over time, utilizing collaboration services like GitHub and Bitbucket to revise and fork the API into bigger and better systems; if the API were ever to deal with confidential data, third-party vendors, etc., this would change, but as it is, the level of exposure is absolutely appropriate to the use-case scenario.

Your considerations should be a broad spectrum of monetization opportunities, upkeep cost, and more, but in terms of security, developers should always follow the rule of simplicity — **expose only what needs to be exposed**.

3.4 Real-World Failure

In late 2014, social media giant Snapchat faced a large security failure in their undocumented API discovered by security firm GibSec. Because of the nature of their API — B2B and semi-private — third party apps were able to connect to the service in such a way that the servers and systems connected were made vulnerable.

One vulnerability in the API, utilizing an authenticated, authorized call to the `/ph/find_friends` request, opened the Snapchat system to hackers in a way the developers never even considered; after calculating the number of Snapchat users and the amount of requests that could be completed in a minute, GibSec estimated that all 8 million users could have their personal data, phone numbers,

and even images scraped by an average server in approximately 26.6 hours (which was a conservative estimate).

So how did the API fail? Functionally, there was a mismatch between the documented access and the actual restrictions on those features. While the API was designed to be a Partner API, it was documented with these partners in the same way a Public API is.

While it's true that if the API were more public vulnerabilities could have been detected and fixed before it became a larger issue, the fact of the matter is that the API did not need to be open in the first place. By not properly managing the data and allowing unvetted companies and developers access to the Snapchat API and internal systems, Snapchat was unnecessarily exposed and made vulnerable.

Additionally, the fact that Snapchat did not adhere to a singular approach was extremely detrimental. If the system were 100% internal, a team would likely have been monitoring the API, and would have come across the issue far before it was actually discovered. If the API were B2B alone, the vulnerability would have not been an issue, as the calls would have required authentication and authorization far above that of a normal user, preventing the leak from occurring. Finally, if the API were completely public, it is likely that the vulnerability would have been discovered earlier, perhaps through the efforts of so-called "white hat hackers" testing vulnerabilities for fun or by common usage.

Neither plate, leather, or chain, the Snapchat failed because it was occupying a strange space between API approaches. Since the necessary preparation in defining the API premise was not performed, API access was not well-maintained, leaving a gaping vulnerability present.

3.5 Two Real-World Successes

Lego

LEGO, the famous brick-toy company from Denmark, has spent most of its existence making sure that their toys were fun, creative, and safe. As LEGO launched into the modern era, they began to seriously consider developing their own API, focusing largely on extending that creativity and security into the world wide web. Below, Dennis Bjørn Petersen discusses the development, challenges, and successes of their closed-source Private API.

The success of the LEGO story demonstrates exactly why an API needs to have determined limits when it comes to source disclosure. Due to the nature of the data LEGO is handling, and the young age of its target audience, the API that was developed was locked down in a Private API environment. This development, isolated from exposure, made sure that the system was safe and effective.

Carson City, Nevada

Another great API success story is that of Carson City, Nevada. Whereas the success of the LEGO API was due to its closed nature and purposeful design, for Carson City, the integration of an API utilizing both the assets of city infrastructure systems and the collaboration of various private and public partners demonstrates success in the Partner API space.

When Carson City set out to design their smart city system, they formed a partnership between the Carson City Public Works Department, the citizens of Carson City, and Schneider Electric. The city packaged open data gener-

ated by various smart devices around the city into a Partner API designed specifically for optimization, reporting, and planning. Carson City officials were able to accomplish some pretty amazing things, converting their city into a true smart city. According to the final report filed by Schneider Electric, there were many benefits:

- Management of the city's power plants was streamlined
- The water management system was drastically optimized.
- Maintenance operations were cut from a 5 to a 4 day week.
- Staff hours were saved in the form of reduced "drive time" between locations By designing their API to function specifically for the purpose intended — in a limited circumstance with secure, defined, and vetted partners — Carson City set themselves up for massive success with real lucrative results.

3.6 Conclusion

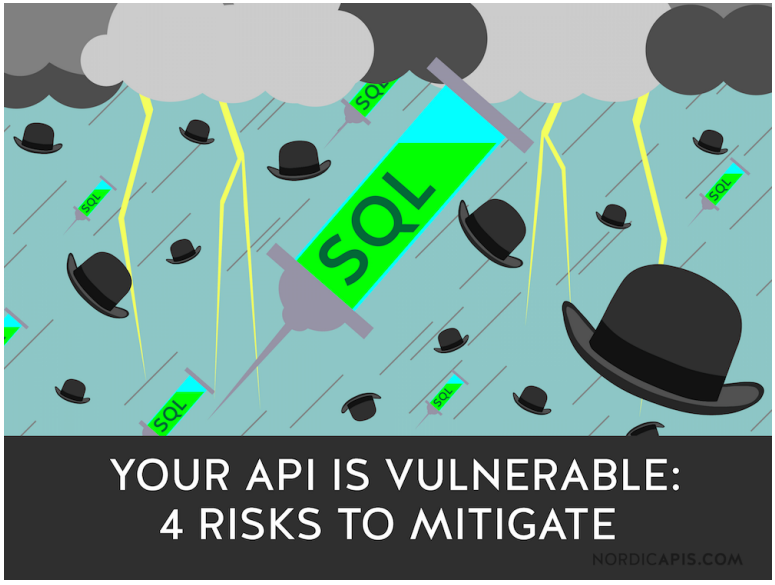
Contemplating the licensing, accessibility, and fundamental availability of your API should be your first step in the API Lifecycle. The complexity of situations presented by your choice in this initial step can impact your APIs success, longevity, and usefulness. While security is, as always, a chief concern in initial considerations, the monetization of your API, the availability of the system, and the user-friendliness of that system as a whole is dependent entirely on the approach taken.

Let's review the benefits of each approach:

- **Leather Armor Public APIs** can have a wide adoption rate and potential to spread brand awareness.
- **Plated Armor Private APIs** allow for increased security and internal management. Keep it secret, keep it safe.
- **Chain Mail Partner APIs** are perhaps the Goldilockian Mithril for your B2B integration situation, allowing a mix of both security and defined partnership potential.

More than any other consideration, your particular purpose for developing an API will determine the choice you make. Enter the API economy brandishing the armor of your choosing.

4. Your API is Vulnerable: 4 Top Security Risks to Mitigate



It's an unfortunate reality that if a system faces the publicly-served internet, chances are it will be attacked at some point within its lifecycle. This is simply a matter of statistics — given the number of users utilizing APIs, the Internet of Things, and the dynamic World Wide Web, an attack is statistically more likely than unlikely. According to a report from The Hill, cyber-attacks increased 48% during 2014.

Despite this grim reality, API providers often consider security to be "someone else's problem". Weak points in an API construct can expose customer data, backend server appliances, and even monetary systems to unauthorized access, putting your API and business at risk.

So what should one look out for to avoid an attack? In this chapter we present four **top security risks** and concerns that **every** API provider needs to consider — along with how to mitigate them.

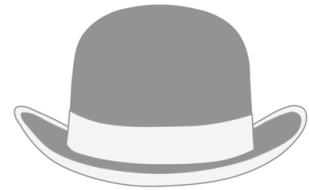
4.1 Gauging Vulnerabilities

Before we can truly appreciate the errors most commonly committed by API developers, we need to understand what constitutes a **vulnerability** and how they are measured.

When identifying vulnerabilities, an API provider or developer should ask themselves, "does this expose something that shouldn't be exposed?" It can be a simple, small exposure, such as allowing someone to access the root file path on a server, or something as complex as failing to plan for complex denial of service methods that utilize exposed resources to deny authentication services to legitimate users.

In the last chapter, we used armor as a metaphor for the types of licenses inherent in the API system. This metaphor holds true for vulnerabilities as well — while you might have a wonderful suit of steel armor, replete with plates, spines, and reinforcing braces, if you lack a simple Gorget, a series of plates protecting the throat, you would be vulnerable to defeat with a single blow. An entire system can only be considered as secure as its weakest part.

4.2 Black Hat vs. White Hat Hackers



Not all possible "exploits" are identified internally. There are a great deal of hackers throughout the World Wide Web who try to gain access to a service, but do so *not* for the purpose of malicious use.

In the hacking world, there are two categories into which the majority of hackers can be broadly categorized — **Black Hats** and **White Hats**.

The Black Hat hacker is most commonly discussed within the security community, and are thus part of the public consciousness. When people think of hackers, they are usually thinking of Black Hats — hackers typing away at a server console screen, attempting to illicitly bypass security measures for personal gain or enjoyment.

These dangerous hackers may immediately utilize an exploit — many Black Hats will find an exploit, and bide their time until either authorization or authentication roadblocks can be similarly bypassed or a zero day exploit embedded in the system is utilized.

Of more use to API developers, however, is the White Hat hacker. These hackers utilize the same tools and techniques as Black Hats, but for a wildly different purpose — to *increase* security. These hackers, often hired by companies to test security, identify exploits and vulnerabilities

for the sole purpose of reporting them to the API provider and/or public in the hopes that they may be patched before they are illicitly used.

Do not underestimate the value of hiring outside members to commit **penetration testing** on your API. To quote an old turn of phrase, "you can't see the forest from the trees". Outside security testing can help identify vulnerabilities, and rectify them before they become full-on exposures.

4.3 Risk 1 - Security Relies on the Developer

Many feel that security is as much the responsibility of the developer consumer as it is a responsibility of the API provider. In some cases, this is true due to the nature of APIs — authentication exchanges, physical access, and so forth should be partially secured by the one who is requesting usage, especially in the case of B2B or Public APIs.

However, as security is best implemented at the lowest possible level, it should be the problem of the API developer (the API provider), and the API developer alone. All other security solutions should spread from the base of the API developer's approach and guidance, and not the other way around.

The philosophy of fostering an internal culture of security encompassess a wide variety of solutions and responsibilities. Developer-centric security means correctly overseeing versioning and dependencies, and properly handling authentication and authorization along with delegation and federation — a huge issue within the API space, as

improper authorization and authentication policies can lead to massive security breaches through the exposure of administrator or sysop credentials. Improper session handling and failure to properly verify delegation and federation routines can lead to session capture and replay attacks.

The API developer consumer and end user can not help with any of these in a meaningful way. Utilizing HTTPS, SSL, SSH, and so forth is helpful, but cannot be done without the API host first designing the system to utilize these solutions.

Also of consideration is the fact that there is a stark difference between API Developers and API Providers. An API Provider is the single body or group of bodies that creates the initial API; an API Developer is a developer who ties into this API, extends upon it, or otherwise implements it in a service. Security, in this case, is easiest to implement within the scope of the Provider, and should thus be the approach taken when an API is in its infancy.

The takeaway - *Secure Your System - It's YOUR System!*

4.4 Risk 2 - "Just Enough" Coding

Perhaps the biggest vulnerability is one that originates earliest in the API development lifecycle — improper coding. Regardless of language of choice, poor error handling, value checking, memory overflow prevention, and more can lead not only to massive vulnerability issues, but to fundamental issues of usefulness and functionality. One phrase that sums up this style of coding is "Just Enough" — the code in question is just enough for functionality, or just enough for usability, without the greater concern of how the code ties into the platform as a whole,

what security concerns are addressed or not addressed, and how potential threats are handled.

The easiest way to present this is to show some basic code snippets that, despite their usefulness to the API functionality, expose the API in some pretty significant ways.

First, let's use an example of an image sharing API written in Python. This API uploads images to a specific root of a specific server for sharing amongst friends utilizing an automatically generated URL system:

```
1  from django.core.files.storage import default_storage
2  from django.core.files.base import File
3  ...
4  def handle_upload(request):
5   files = request.FILES
6   for f in files.values():
7     path = default_storage.save('upload/', File(f))
8  …
```

This is a working service — but upon inspection, it fails several important security checks. Firstly, there is no **type checking**; a file of any type can be uploaded, meaning that anybody can upload an executable and utilize the automatically generated URL to run the application natively on the server. Secondly, the path for the uploaded file is specified to the service directory root, meaning that the application will have broad level access to the server.

Let's look at another snippet. This example will be an error report generated from improper command usage in COBOL:

```
...
EXEC SQL
 WHENEVER SQLERROR
 PERFORM DEBUG-ERR
SQL-EXEC.
...
DEBUG-ERR.
 DISPLAY "Error code is: " SQLCODE.
 DISPLAY "Error message is: " SQLERRMC.
...
```

While error handling and notification is vitally important, the way this is handled above is flawed. When a Black Hat attempts to utilize a malformed command, they will instantly know two things:

- If the error is served via console (the default behavior of many servers), the connection to the service, despite being malformed, will be kept open.
- Secondly, that the server is vulnerable to SQL injection. This type of error is known as an **external system information leak**.

These may seem trivial to an outsider, but to a hacker, this is a treasure-trove of information. Imagine being a burglar who, through a simple phone call, and find out which door is unlocked, which window improperly seated, where everyone is sleeping, and where the valuables are. The information gleaned above is tantamount to this wealth of information, and serves the hacker a tremendous benefit.

These types of errors are grave, indeed — simply understanding how your code functions and how the average user will utilize system functionality can go a long way towards securing these vulnerabilities and not exposing system details.

4.5 Risk 3 - Misunderstanding Your Ecosystem

The API world is a rapidly shifting place. As API architectures evolve, and new, more expansive methodologies for microservice development and management emerge, the security issues inherent with each choice in the API lifecycle naturally evolve alongside.

Unfortunately, many developers seem to adopt these new technologies without fully understanding what they mean for API security. It's so easy to adopt the "brightest and best", but without a similarly evolved development mindset, vulnerabilities can quickly become far larger than they have any right to be.

For example, the innovations in cloud computing, especially in terms of the evolution of System-as-a-Service, Platform-as-a-Service, and Infrastructure-as-a-Service platforms, has led to a massive amount of data and computing being handled off-site. What was once an environment of local resources and servers utilizing an API to communicate with the outside world is becoming increasingly decentralized.

While this is a good thing, it comes with some pretty heavy caveats and vulnerabilities. For example, utilizing side-

channel timing information, a virtual server sharing the same physical space as that of an API virtual server could theoretically hijack private cryptographic keys, according to a research paper from the University of North Carolina.

Barring such virtual attacks, there is still the issue of physical threats. Because a cloud server is not local to the API developer, physical security cannot be maintained, meaning hardware hosting secure password hashes or user account files can be vulnerable to environmental damage or malicious use.

Much of this can be negated. Proper server file security, backups, and more can go a long way towards improving security vulnerabilities of this type. But these solutions can only be implemented if the issues are known — and for many API developers who have moved legacy or classically designed APIs to the cloud space without first understanding the ecosystem, this is easier said than done.

4.6 Risk 4 - Trusting the API Consumer With Too Much Control

While much has been said about maintaining positive user experience through the API lifecycle, average end-users are the number one largest security risk an API can ever have.

Regardless of the API type, language and syntax, or development approach, an API is functionally secure until it's actually used.

Until your API receives its first request for information, it lives in a veritable island of security — isolated, alone, but ultimately untouchable. The second it receives that

request, however, it's wide open. When developing an API, developers often trust the user far too much, allowing too much data manipulation, not limiting password complexity and usage, and sometimes even allowing repeat session ID tokens. This is a huge issue.

That's not to say you should treat every user like a Black Hat — most users are going to be legitimate, are going to use the service how it was intended, and will not be a security vulnerability in and of themselves. But, assuming that most of your users are "good" users, that still leaves a good deal of "unknowns" who can use anything gleaned through average use to break your system and access it illicitly.

Thus, you must assume every user is a vulnerability. Implement password strength requirements, session length and concurrent connection limitations, and even require periodic re-authentication and authorization for continued use. Track usage and account metrics, and respond to any deviation. Assume that even your best user is subject to hijacking and malicious redirection, and plan accordingly.

4.7 Conclusion

By understanding these basic security risks and adequately responding, API security risks can be largely mitigated. While no system is ever going to be truly perfect, they can at least be complex enough and complete enough to deter all but the most ardent and dedicated hackers.

5. Deep Dive into OAuth and OpenID Connect

OAuth 2 and OpenID Connect are fundamental to securing your APIs. To protect the data that your services expose, you must use them. They are complicated though, so in this chapter we go into some depth about these standards to help you deploy them correctly.

5.1 OAuth and OpenID Connect in Context

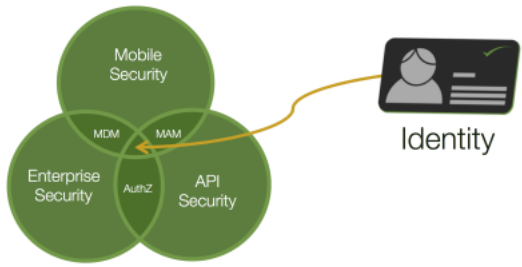
Always be aware that OAuth and OpenID Connect are part of a larger information security problem. You need to take additional measures to protect your servers and the mobiles that run your apps in addition to the steps taken to secure your API. Without a holistic approach, your API may be incredibly secure, your OAuth server locked down, and your OpenID Connect Provider tucked away in a safe enclave. Your firewalls, network, cloud infrastructure, or the mobile platform may open you up to attack if you don't also strive to make them as secure as your API.

Enterprise Security

API Security

Mobile Security

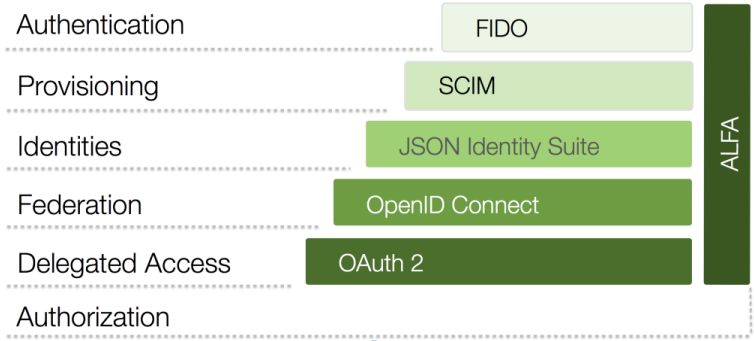
To account for all three of these security concerns, you have to know who someone is and what they are allowed to do. To authenticate and authorize someone on your servers, mobile devices, and in your API, you need a complete Identity Management System. At the head of API security, enterprise security and mobile security is identity!



Only after you know who someone (or something) is can you determine if they should be allowed to access your data. We won't go into the other two concerns, but don't forget these as we delve deeper into API security.

5.2 Start with a Secure Foundation

To address the need for Identity Management in your API, you have to build on a solid base. You need to establish your API security infrastructure on protocols and standards that have been peer-reviewed and are seeing market adoption. For a long time, lack of such standards has been the main impediment for large organizations wanting to adopt RESTful APIs in earnest. This is no longer the case since the advent of the Neo-security Stack:



This protocol suite gives us all the capabilities we need to build a secure API platform. The base of this, OAuth and OpenID Connect, is what we want to go into in this blog post. If you already have a handle on these, learn more about how the other protocols of the Neo-security Stack fit together.

5.3 Overview of OAuth

OAuth is a sort of "protocol of protocols" or "meta protocol," meaning that it provides a useful starting point for other protocols (e.g., OpenID Connect, NAPS, and UMA). This is similar to the way WS-Trust was used as the basis for WS-Federation, WS-SecureConversation, etc., if you have that frame of reference.

Beginning with OAuth is important because it solves a number of important needs that most API providers have, including:

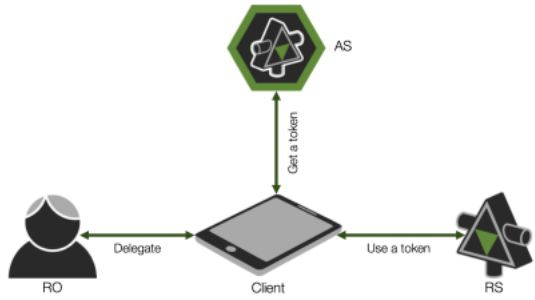
- Delegated access
- Reduction of password sharing between users and third-parties (the so called "password anti-pattern")
- Revocation of access

When the password anti-pattern is followed and users share their credentials with a third-party app, the only way to revoke access to that app is for the user to change their password. Consequently, all other delegated access is revoked as well. With OAuth, users can revoke access to specific applications without breaking other apps that should be allowed to continue to act on their behalf.

5.4 Actors in OAuth

There are four primary actors in OAuth:

1. **Resource Owner (RO)**: The entity that is in control of the data exposed by the API, typically an end user
2. **Client**: The mobile app, web site, etc. that wants to access data on behalf of the Resource Owner
3. **Authorization Server (AS)**: The Security Token Service (STS) or, colloquially, the OAuth server that issues tokens
4. **Resource Server (RS)**: The service that exposes the data, i.e., the API



5.5 Scopes

OAuth defines something called “scopes.” These are like permissions or delegated rights that the Resource Owner wishes the client to be able to do on their behalf. The client may request certain rights, but the user may only grant some of them or allow others that aren’t even requested. The rights that the client is requested are often shown in some sort of UI screen. Such a page may not be presented to the user, however. If the user has already granted the client such rights (e.g., in the EULA, employment contract, etc.), this page will be skipped.

What is in the scopes, how you use them, how they are displayed or not displayed, and pretty much everything else to do with scopes are not defined by the OAuth spec. OpenID Connect does define a few, but we’ll get to that in a bit.

5.6 Kinds of Tokens

In OAuth, there are two kinds of tokens:

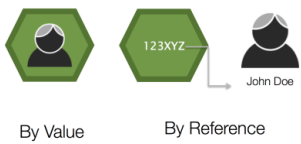
1. **Access Tokens**: These are tokens that are presented to the API
2. **Refresh Tokens**: These are used by the client to get a new access token from the AS

(Another kind of token that OpenID Connect defines is the ID token. We’ll get to that in a bit.)

Think of access tokens like a session that is created for you when you login into a web site. As long as that session is valid, you can continue to interact with the web

site without having to login again. Once that session is expired, you can get a new one by logging in again with your password. Refresh tokens are like passwords in this comparison. Also, just like passwords, the client needs to keep refresh tokens safe. It should persist these in a secure credential store. Loss of these tokens will require the revocation of all consents that users have performed.

5.7 Passing Tokens



As you start implementing OAuth, you'll find that you have more tokens than you ever knew what to do with! How you pass these around your system will certainly affect your overall security. There are two distinct ways in which they are passed:

1. By value
2. By reference

These are analogous to the way programming language pass data identified by variables. The run-time will either copy the data onto the stack as it invokes the function being called (by value) or it will push a pointer to the data (by reference). In a similar way, tokens will either contain all the identity data in them as they are passed around or they will be a reference to that data.

If you pass your tokens by reference, keep in mind that you will need a way to dereference the token. This is typically done by the API calling a non-standard endpoint exposed by your OAuth server.

5.8 Profiles of Tokens

There are different profiles of tokens as well. The two that you need to be aware of are these:

1. Bearer tokens
2. Holder of Key (HoK) tokens

You can think of bearer tokens like cash. If you find a dollar bill on the ground and present it at a shop, the merchant will happily accept it. She looks at the issuer of the bill, and trusts that authority. The saleswomen doesn't care that you found it somewhere. Bearer tokens are the same. The API gets the bearer token and accepts the contents of the token because it trusts the issuer (the OAuth server). The API does not know if the client presenting the token really is the one who originally obtained it. This may or may not be a bad thing. Bearer tokens are helpful in some cases, but risky in others. Where some sort of proof that the client is the one to who the token was issued for, HoK tokens should be used.

HoK tokens are like a credit card. If you find my credit card on the street and try to use it at a shop, the merchant will (hopefully) ask for some form of ID or a PIN that unlocks the card. This extra credential assures the merchant that the one presenting the credit card is the one to whom it was issued. If your API requires this sort of proof, you will need HoK key tokens. This profile is still a draft, but you should follow this before doing your own thing.

NOTE: You may have heard of MAC tokens from an early OAuth 2 draft. This proposal was never finalized, and this profile of tokens are never used in practice. Avoid this unless you have a very good reason. Vet that rational on

the OAuth mailing list before investing time going down this rabbit trail.

5.9 Types of Tokens

We also have different types of tokens. The OAuth specification doesn't stipulate any particular type of tokens. This was originally seen by many as a negative thing. In practice, however, it's turned out to be a very good thing. It gives immense flexibility. Granted, this comes with reduced interoperability, but a uniform token type isn't one area where interop has been an issue. Quite the contrary! In practice, you'll often find tokens of various types and being able to switch them around enables interop. Example types include:

- WS-Security tokens, especially SAML tokens
- JWT tokens (which I'll get to next)
- Legacy tokens (e.g., those issued by a Web Access Management system)
- Custom tokens

Custom tokens are the most prevalent when passing them around by reference. In this case, they are randomly generated strings. When passing by val, you'll typically be using JWTs.

JSON Web Tokens

JSON Web Tokens or JWTs (pronounced like the English word "jot") are a type of token that is a JSON data structure that contains information, including:

- The issuer
- The subject or authenticated uses (typically the Resource Owner)
- How the user authenticated and when
- Who the token is intended for (i.e., the audience)

These tokens are very flexible, allowing you to add your own claims (i.e., attributes or name/value pairs) that represent the subject. JWTs were designed to be light-weight and to be snuggly passed around in HTTP headers and query strings. To this end, the JSON is split into different parts (header, body, signature) and base-64 encoded.

If it helps, you can compare JWTs to SAML tokens. They are less expressive, however, and you cannot do everything that you can do with SAML tokens. Also, unlike SAML they do not use XML, XML name spaces, or XML Schema. This is a good thing as JSON imposes a much lower technical barrier on the processors of these types of tokens.

JWTs are part of the JSON Identity Suite, a critical layer in the Neo-security Stack. Beyond JWT, the JSON identity suite of protocols includes:

- **JWK:** JSON web key is a data structure that sets up the protocols for defining asymmetric and symmetric keys
- **JWE:** JSON web encryption is a compact web encryption format, ideal for space-constrained environments
- **JWS:** JSON web signatures enable multiple signatures to be applied to the same content

- **JWA:** JSON web algorithms register cryptographic algorithms
- **Bearer Tokens:** An auxiliary specification used to define how to use these tokens and to put a JWT into an HTTP authorization header for an API to receive

These together with JWT are used by both OAuth (typically) and OpenID Connect. How exactly is specified in the core OpenID Connect spec and various ancillary specs, in the case of OAuth, including the Bearer Token spec.

5.10 OAuth Flow

OAuth defines different "flows" or message exchange patterns. These interaction types include:

- The code flow (or web server flow)
- Client credential flow
- Resource owner credential flow
- Implicit flow

The code flow is by far the most common; it is probably what you are most familiar with if you've looked into OAuth much. It's where the client is (typically) a web server, and that web site wants to access an API on behalf of a user. You've probably used it as a Resource Owner many times, for example, when you login to a site using certain social network identities. Even when the social network isn't using OAuth 2 per se, the user experience is the same. Checkout this YouTube video at time 12:19 to see how this flow goes and what the end user experiences:

We'll go into the other flows another time. If you have questions on them in the meantime, ask in a comment below.

5.11 Improper and Proper Uses of OAuth

After all this, your head may be spinning. Mine was when I first learned these things. It's normally. To help you you orient yourself, I want to stress one really important high-level point:

- **OAuth is not used for authorization**. You might think it is from it's name, but it's not.
- **OAuth is also not for authentication**. If you use it for this, expect a breach if your data is of any value.
- **OAuth is also not for federation**.

So what is it for?

It's for delegation, and delegation only!

This is your plumb line. As you architect your OAuth deployment, ask yourself: In this scenario, am I using OAuth for anything other than delegation? If so, go back to the drawing board.

Consent vs. Authorization

How can it *not* be for authorization, you may be wondering. The "authorization" of the client by the Resource

Owner is really consent. This consent may be enough for the user, but not enough for the API. The API is the one that's actually authorizing the request. It probably takes into account the rights granted to the client by the Resource Owner, but that consent, in and of its self, is not authorization.

To see how this nuance makes a very big difference, imagine you're a business owner. Suppose you hire an assistant to help you manage the finances. You *consent* to this assistant withdrawing money from the business' bank account. Imagine further that the assistant goes down to the bank to use these newly delegated rights to extract some of the company's capital. The banker would refuse the transaction because the assistant is not authorized – certain paperwork hasn't been filed, for example. So, your act of delegating your rights to the assistant doesn't mean squat. It's up to the banker to decide if the assistant gets to pull money out or not. In case it's not clear, in this analogy, the business owner is the Resource Owner, the assistant is the client, and the banker is the API.

5.12 Building OpenID Connect Atop OAuth

As I mentioned above, OpenID Connect builds on OAuth. Using everything we just talked about, OpenID Connect constrains the protocol, turning many of the specification's SHOULDs to MUSTs. This profile also adds new endpoints, flows, kinds of tokens, scopes, and more. OpenID Connect (which is often abbreviated OIDC) was made with mobile in mind. For the new kind of tokens that it defines, the spec says that they must be JWTs, which were also designed for low-bandwidth scenarios. By building on

OAuth, you will gain both delegated access and federation capabilities with (typically) one product. This means less moving parts and reduced complexity.

OpenID Connect is a modern federation specification. It is a passive profile, meaning it is bound to a passive user agent that does not take an active part in the message exchange (though the client does). This exchange flows over HTTP, and is analogous to the SAML artifact flow (if that helps). OpenID Connect is a replacement for SAML and WS-Federation. While it is still relatively new, you should prefer it over those unless you have good reason not to (e.g., regulatory constraints).

As I've mentioned a few times, OpenID Connect defines a new kind of token: ID tokens. These are intended for the client. Unlike access tokens and refresh tokens that are opaque to the client, ID tokens allow the client to know, among other things:

- How the user authenticated (i.e., what type of credential was used)
- When the user authenticated
- Various properties about the authenticated user (e.g., first name, last name, shoe size, etc.)

This is useful when your client needs a bit of info to customize the user experience. Many times I've seen people use by value access tokens that contain this info, and they let the client take the values out of the API's token. This means they're stuck if the API needs to change the contents of the access token or switch to using by ref for security reasons. If your client needs data about the user, give it an ID token and avoid the trouble down the road.

The User Info Endpoint and OpenID Connect Scopes

Another important innovation of OpenID Connect is what's called the "User Info Endpoint." It's kinda a mouthful, but it's an *extremely* useful addition. The spec defines a few specific scopes that the client can pass to the OpenID Connect Provider or OP (which is another name for an AS that supports OIDC):

- openid (required)
- profile
- email
- address
- phone

You can also (and usually will) define others. The first is required and switches the OAuth server into OpenID Connect mode. The others are used to inform the user about what type of data the OP will release to the client. If the user authorizes the client to access these scopes, the OpenID Connect provider will release the respective data (e.g., email) to the client when the client calls the user info endpoint. This endpoint is protected by the access token that the client obtains using the code flow discussed above.

: An OAuth client that supports OpenID Connect is also called a Relying Party (RP). It gets this name from the fact that it on the OpenID Connect Provider to assert the user's identity.

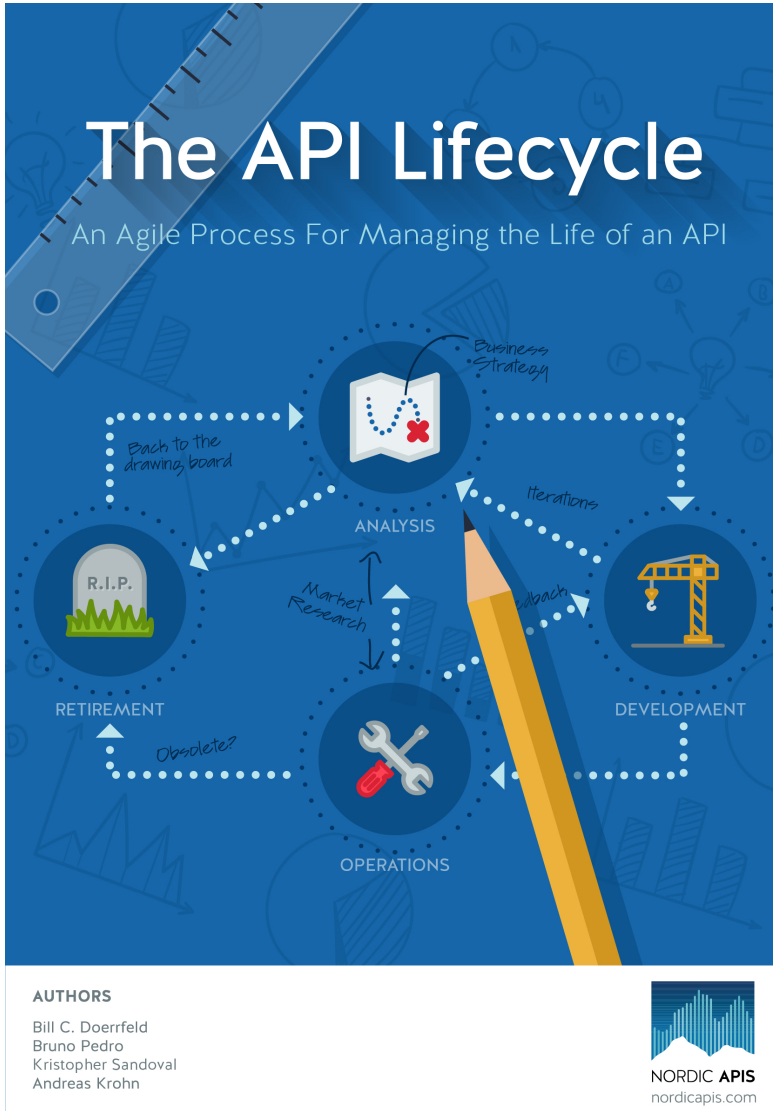
Not Backward Compatible with v. 2

It's important to be aware that OpenID Connect ***is not*** backward compatible with OpenID 2 (or 1 for that matter). OpenID Connect is effectively version 3 of the OpenID specification. As a major update, it is not interoperable with previous versions. Updating from v. 2 to Connect will require a bit of work. If you've properly architected your API infrastructure to separate the concerns of federation with token issuance and authentication, this change will probably not disrupt much. If that's not the case however, you may need to update *each and every* app that used OpenID 2.

5.13 Conclusion

In this chapter, we dove into the fundamentals of OAuth and OpenID Connect and pointed out their place in the Neo-security Stack. In truth, we've only skimmed the surface. Anyone providing an API that is protected by OAuth 2 (which should be all of them that need secure data access), this basic knowledge is a prerequisite for pretty much everyone on your dev team. Others, including product management, operations, and project management should know some of the basics described above.

If you have questions beyond what we covered here, checkout this video recording, this Slideshare presentation.



Also by Nordic APIs: *The API Lifecycle: A holistic guide to API development and operations*

6. Unique Authorization Applications of OpenID Connect

OpenID Connect, an identity layer on top OAuth, is one of the most important ways that access authorization and information is passed between two parties online. The OAuth2 protocol decides how a client receives a token from a consenting user and then uses that token on an API call.

Paul Madsen of Ping Identity argues that even though OAuth2 passes identity by way of that token's permissions, the protocol lacks in that it does not withold useful user information. By layering OpenID Connect on top of OAuth2, the identity semantic comes into play and OAuth2 becomes *identity aware*, enabling things like single sign-on and personal profile information sharing.

OpenID Connect Key Identity Extensions:

- **UserInfo Endpoint**: The OAuth protected endpoint that provides user identity attributes, which limits registration form drop-off.
- **ID Tokens**: A structured, secure, signed information object that carries information about the user in question, like *when* they authenticated and *how*.

If widely adopted, OpenID could transform identity control by enabling single sign-on, increasing information

security, and helping to manage identity throughout the Internet of Things. Within this post we'll dive into these **three use cases** on using OpenID to securely manage user identity.

6.1 How OpenID Connect Enables Native SSO

OpenID Connect enables **native single sign-on**, usually referred to as Native SSO. As native apps continue to grow in popularity due to their ease of use and ease of distribution, there comes a greater demand for default OAuth in native environments. But the burden of managing authentication throughout a sea of various native apps falls on the end user, who must know which login is for which app, which needs to be re-authenticated, among other nuisances.

Forecasts show an increase in native app usage within the foreseeable future. In 2014, 86 percent of time spent on smartphones was spent within native apps, not web browsers. One way to get an edge in this increasingly crowded market is to increase usability with a single sign-on for multiple apps published by the same owner. This can be accomplished using OpenID paired with OAuth.

The process involves the implementation of an **Authentication Agent** (AZA) which is either an agent installed on the device's operating system or is it's own separate mobile app. The mobile device user authorizes the AZA agent to retrieve tokens automatically from other native mobile apps that it's authorized to use.

This case has an obvious appeal for businesses to enable and control SSO access to certain enterprise-grade applications, for both web and native apps, as well as for

bundles of B2C apps that were created by the same brand that wants to give users easy access to them all.

The next step is establishing and standardizing the protocols under which this Native SSO can be permitted and tokens can be shared. The OpenID Foundation currently is collaborating on developing the specifications for native app SSO via OpenID Connect 1.0.

6.2 How to Use OpenID Connect to Enable Mobile Information Management and BYOD

Major security breaches have unfortunately been omnipresent in recent news. Various security management strategies are being employed to avoid these breaches with a determined focus on offering enterprises the confidence that their employees can use any mobile to access sensitive business apps while still having a standard of security, irrespective of mobile device in this bring-your-own-device (BYOD) world. Strategies include:

- **Mobile Device Management** (MDM) aims to protect the whole device, regardless if it's used for both business and pleasure. When applied to BYOD, privacy and property rights are then called into question.
- **Mobile Application Management** (MAM) goes granular, with enterprise IT departments focusing just on securing business apps, compartmentalizing data and apps.
- **Mobile Information Management** (MIM) then acts as the next step in this granularity. The enterprise imposes its management policy directly on the data

> itself. Business data is wrapped with security mechanisms and policy that constrain and determine how that data can be used.

MIM is key management. Using MIM, before data is passed to the device, it is encrypted with a corresponding key and a policy that encompasses how the data is passed down as well as how the encryption key is bound to the data. Decryption keys are then released only under strict conditions— the right user with the right key using the right app.

Madsen argues that OpenID Connect is significant to that key distribution model, via ID token and a user info API, as it follows an OpenID flow with key distribution steps layered on top of it. The app itself doesn't have a decryption key but it can use its access token to send its license back up to the key distribution server. That distribution server can from there determine who the user is, in what context, and under what licensed circumstance should a key be released. When approved, it hands back a key along with a license that constrains what can be done with the authorized data.

There's no doubt that Mobile Network Operators (MNOs) are becoming increasingly aware of how OpenID Connect interacts with identity services currently being used online, including payments and log-ins. The widespread assumption is that OpenID Connect can mitigate pain points within existing services by offering a public key encryption-based authentication framework. This takes the burden of identity verification from the hands of the average user and puts it into the hands of expert service providers.

Of course, applying MIM using OpenID is theoretical, and has not yet been proven. However, if the one-two punch

of OpenID and mobile information management (MIM) is widely adopted, it is assumed that it could increase the security of the entire Internet dramatically.

6.3 How OpenID Connect Enables the Internet of Things

Just as identity verification and authentication are priority topics for MNOs, these themes similarly must extend to the rapidly expanding world of the Internet of Things (IoT). As the number of IoT services increase, so will the number of passwords as mechanisms for sharing and trusting identities. "The importance of interoperability amongst identity solutions is that it will enable individuals to choose between and manage multiple different interoperable credentials," said chief domain architect at BT Charles Gibbons.

According to Madsen, the Internet of Things (IoT) really assumes the *Identity* of Things because all these Things will be operating on *our* behalf, managing *our* data. There is a definite need to distinguish among different connected Things and their identities, as well as a need for a way to authenticate how these Things will collect data.

Via OpenID Connect, a Thing can obtain a token to use in an API call. The user is actively involved in the issuing of that token, and has the power to impose policies as to when that token can be used and how his or her data is shared.

Because OpenID Connect standardizes mechanisms by which users can control the sharing of the identity that they use, Madsen believes OpenID Connect will become a critical asset in the further development of usable, personalized, and secure IoT applications.

This topic was presented at a Nordic APIs conference by Paul Madsen from PingIdentity. View his complete presentation on YouTube.

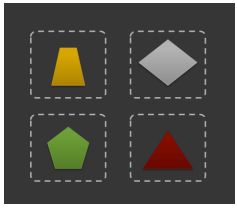
7. How To Control User Identity Within Microservices

Properly maintaining identity and access management throughout a sea of independent services can be tricky. Unlike a traditional monolithic structure that may have a

single security portal, microservices pose many problems. Should each service have it's own independent security firewall? How should identity be distributed between microservices and throughout my entire system? What is the most efficient method for the exchange of user data?

There are smart techniques that leverage common technologies to not only authorize but perform delegation across your entire system. In this chapter we identify how to implement OAuth and OpenID Connect flows using JSON Web Tokens to achieve the end goal of creating a **distributed authentication mechanism for microservices** — a process of managing identity where everything is self-contained, standardized, secure, and best of all — easy to replicate.

7.1 What Are Microservices, Again?

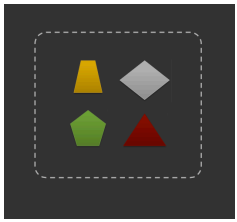


Microservice Architecture

For those readers not well-versed in the web discussion trends of late, the microservice design approach is a way to architect web service suites into independent specialized components. These components are made to satisfy a very targeted function, and are fully independent, deployed as separate environments. The ability to recompile individual units means that development and scaling can be vastly easier within a system using microservices.

This architecture is opposed to the traditional **monolithic** approach that consolidates all web components into a single system. The downside of a monolithic design is that version control cycles are arduous, and scalability is slow.

The entire system must be continuously deployed since it's packaged together.



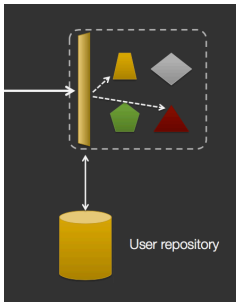
Monolithic Design

The move toward microservices could have dramatic repercussions across the industry, allowing SaaS organizations to deploy many small services no longer dependent on large system overhauls, easing development, and on the user-facing side allowing easy pick-and-choose portals for users to personalize services to their individual needs.

7.2 Great, So What's The Problem?

The problem we're faced with is that microservices don't lend themselves to the traditional mode of identity control. In a monolithic system security works simply as follows:

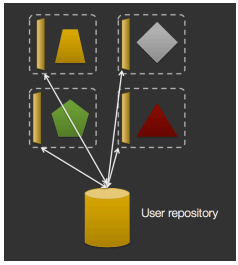
1. Figure out **who** the caller is
2. Pass on credentials to other components when called
3. Store user information in a data repository



Simplified monolithic flow

Since components are conjoined within this structure, they may **share** a single security firewall. They share the state of the user as they receive it, and may also share access to the same user data repository.

If the same technique were to be applied to individual microservices, it would be grossly inefficient. Having an independent security barrier — or request handler — for each service to authenticate identity is unnecessary. This would involve calling an Authentication Service to populate the object to handle the request and respond in every single instance.



The problem with microservice security

7.3 The Solution: OAuth As A Delegation Protocol

There is a method that allows one to combine the benefits of isolated deployment with the ease of a federated identity. Jacob Ideskog of Twobo Technologies believes that to accomplish this OAuth should be interpreted not as Authentication, and not as Authorization, but as *Delegation*.

In the real world, **delegation** is where you delegate someone to do something for you. In the web realm, the underlying message is there, yet it also means having the ability to offer, accept, or deny the exchange of data. Considering OAuth as a Delegation protocol can assist in the creation of scalable microservices or APIs.

To understand this process we'll first lay out a standard OAuth flow for a simple use case. Assume we need to access a user's email account for a simple app that organizes a user's email — perhaps to send SMS messages as notifications. OAuth has the following four main actors:

- **Resource Owner (RO)**: the user

- **Client**: the web or mobile app
- **Authorization Service (AS)**: OAuth 2.0 server
- **Resource Server (RS)**: where the actual service is stored

7.4 The Simplified OAuth 2 Flow

In our situation, the app (the Client), needs to access the email account (the Resource Server) to collect emails before it can organize them to create the notification system. In a simplified OAuth flow, an approval process would be as follows:

1. The **Client** requests access to the **Resource Server** by calling the **Authentication Server.**
2. The **Authentication Server** redirects to allow the user to authenticate, which is usually performed within a browser. This is essentially signing into an authorization server, not the app.
3. The **Authorization Server** then validates the user credentials and provides an **Access Token** to client, which can be use to call the **Resource Server**
4. The **Client** then sends the **Token** to the **Resource Server**
5. The **Resource Server** asks the **Authentication Server** if the token is valid.
6. The **Authorization Server** validates the **Token**, returning relevant information to the **Resource Server** i.e. time till token expiration, who the token belongs too.
7. The **Resource Server** then provides data to the **Client**. In our case, the requested emails are unbarred and delivered to the client.

An important factor to note within this flow is that the Client — our email notification app — knows nothing about the user at this stage. The token that was sent to the client was completely opaque — only a string of random characters. Though a secure exchange, the token is itself useless. The exchange thus supplies little information. What if our app client had a backend with multiple sessions, for example? Wouldn't it be nice to receive additional user information?

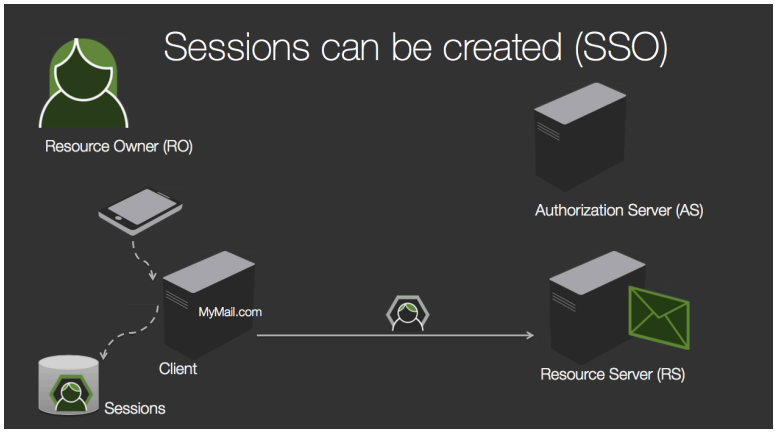
7.5 The OpenID Connect Flow

Let's assume that we're enhancing the email service client so that it not only organizes your emails, but also stores them and translates them into another language. In this case, the client will want to retrieve additional user data in order to create and store it's own user sessions. The **Resource Owner** is now split between a mobile app as well as a backend. The system needs to create session between client app and the client backend.

To enable this, use an alternative flow with OpenID Connect. In this process, the Authorization Server sends an **ID Token** along with the Access Token to the client, allowing the client to store it's own user sessions. The flow is as follows:

1. The **Client** requests access to the **Resource Server** by calling the **Authentication Server.**
2. The **Authentication Server** redirects to allow the user to authenticate.
3. The **Authorization Server** then validates the user credentials and provides an **Access Token** AND an **ID Token** to the client.

4. The **Resource Owner** then uses this ID to create it's own sessions.
5. The **Client** then sends this to the **Resource Server**
6. The **Resource Server** responds, delivering the data (the emails) to the **Client**.



The ID token contains information on the user, such as whether or not they are authenticated, the name, email, and any number of custom data points on a user. This ID token takes the form of a JSON Web Token (JWT), which is a coded and signed compilation of JSON documents. The document includes a header, body, and a signature appended to the message. Data + Signature = JWT.

Using a JWT, you can access the public part of certificate, validate the signature, and understand that this authentication session was issued — verifying that the user has authenticated. An important facet of this approach is that ID tokens establish **trust** between the Authentication Server and the Client.

7.6 Using JWT For OAuth Access Tokens

Even if we don't use OpenID Connect, JWTs can be used for many things. A system can standardize by using JWTs to pass user data among individual services. Let's review the types of OAuth access tokens to see how to smartly implement secure identity control within microservice architecture.

By Reference: Standard Access Token

This type of token contains no information outside of the network, simply pointing to a space where information is located. This opaque string means nothing to user, and as it is randomized cannot easily be decrypted. This is the standard form of an access token — without extraneous content, simply used for a client to gain access to data.

By Value: JSON Web Token

This type may contain necessary user information that the client requires. The data is compiled, and inserted into the message as an access token. This is an efficient method because it erases the need to call again for additional information. If exposed over the web, a downside is that this public user information can be read easily read, exposing the data to an unnecessary risk of decryption attempts to crack codes.

The Workaround : External vs. Internal

To limit this risk of exposure, Iskedog recommends splitting the way the tokens are used. What is usually done is as follows:

1. The **Reference Token** is issued by the **Authentication Server.** The client sends back when it's time to

call the API.
2. In the middle: The **Authentication Server** authenticates and responds with a **JWT**.
3. The **JWT** is then passed further along in the network.

In the middle we essentially create a firewall, an Authentication Server that acts as a token translation point for the API. This stateless point could be a full fledged API firewall. It can cache, and should rest on the edge of a network, porting data into a user repository to create and manage user sessions.

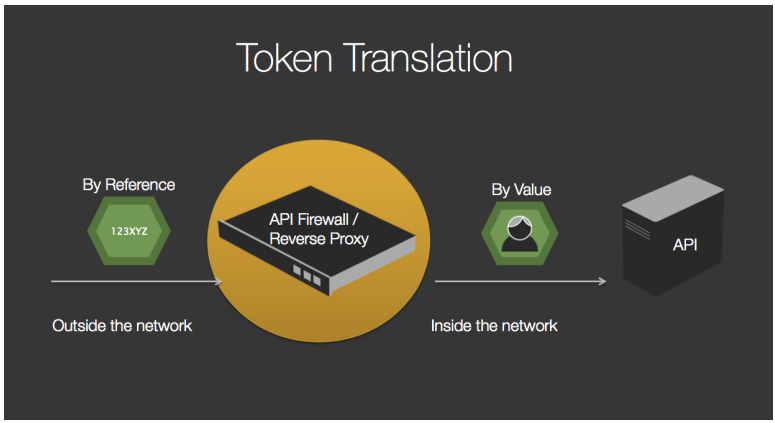
7.7 Let All Microservices Consume JWT

So, to refresh, with microservice security we have two problems:

- **We need to identify the user multiple times**: We've shown how to leave authentication to OAuth and the OpenID Connect server, so that microservices successfully provide access given someone has the right to use the data.
- **We have to create and store user sessions**: JWTs contain the necessary information to help in storing user sessions. If each service can understand a JSON web token, then you have distributed your authentication mechanism, allowing you to transport identity throughout your system.

In microservice architecture, an access token should not be treated as a request object, but rather as an *identity* object. As the process outlined above requires translation, JWTs should be translated by a front-facing stateless

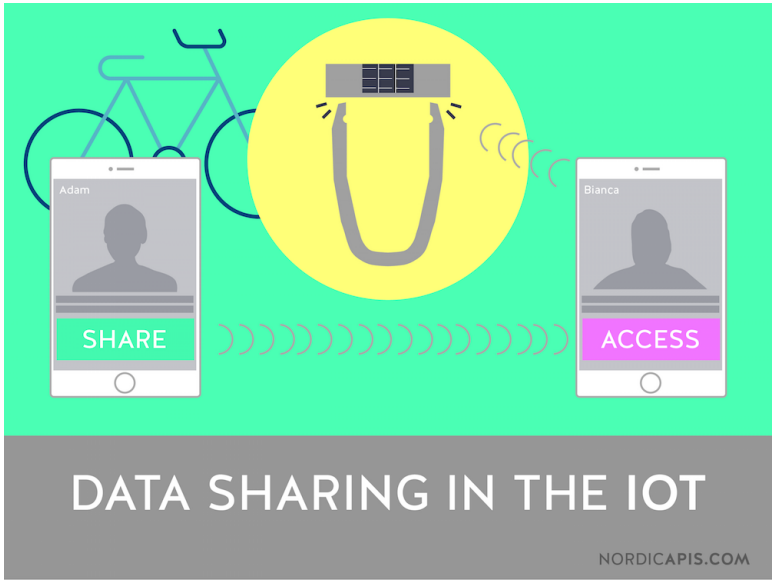
proxy, used to take a reference token and convert it into a value token to then be distributed throughout the network.



7.8 Why Do This?

By using OAuth with OpenID Connect, and by creating a standards based architecture that universally accepts JWTs, the end result is a distributed authentication mechanism that is self contained and easily to replicate. Constructing a library that understands JWT is a very simple task. In this environment, access as well as user data is secured. Constructing microservices that communicate well and securely transport user information can greatly increase agility of the whole system, as well as increase the quality of the end user experience.

8. Data Sharing in the IoT



We consistently put our personal data at risk. API security concerns sky rocket as the user's desire to pass access to others steadily increases. With the rise of Internet of Things (IoT) devices, the ability to share the use of physical connected devices makes **access management** an increasing concern.

Whether sharing a collaborative document or car location, users routinely pass data and access to family members, friends, coworkers, clients, or fellow app users — most often using APIs. But how can these data exchanges be accomplished in a secure and efficient way using existing standards?

In a session with Nordic APIs, Jacob Ideskog demonstrates how this is possible using OAuth enabled with OpenID Connect. In this chapter we'll examine how to implement secure data sharing for a new **bike lock IoT device** — a real world example of introducing a best practice approach to identity management in the IoT realm.

8.1 A New Economy Based on Shared, *Delegated* Ownership

In addition to dramatically altering the way enterprise business is done, APIs are fueling a major consumer shift. With the dawn of smartphone ubiquity, sensors, and IoT devices, new subscription services and businesses have quickly emerged around the concept of sharing access to **personal property**.

Homeowners are becoming part time business owners with **Airbnb**, citizens with a car and free time are making extra income driving with **Uber** or **Lyft**. Apps enable users to share their office or work space, or use communal cars with **Car2Go** or **ZipCar**. Put simply, the Anything-as-a-Service (XaaS) model is the wave of the future.

Sharing ownership comes with inherent risks. Risks on the side of the party using the property, and risks within the data exchange process. So how does an app avoid security risks to ensure ownership is delegated correctly? What specific technologies and workflows offer the best route to access management?

8.2 Connected Bike Lock Example IoT Device

Take Skylock, a crowdfunded initiative to develop a solar powered connected smart bike lock. It's loaded with features — a built in anti-theft feature, accelerometer to analyze movement, and automatic safety response alerts. Skylock also allows keyless entry, and has an ability to **share access remotely with friends** using the Skylock app.

The Skylock lock and app likely utilize many first or third party APIs to power location, mapping, safety response, and more. But out of all the software powering the scenarios presented in their promo video, the most complex part of this system is likely the **sharing** of bike lock access that occurs between the main character and his girlfriend.

8.3 How This Works

In a scenario like this one, the user typically authorizes the app to access the API. We'll call the guy Adam, and his girlfriend Bianca. Let's say Adam wants to share access so that Bianca's app can access Adam's account via the API. In this case, sharing *is* delegating use.

Can't OAuth be used for that? Not exactly. OAuth is really about the User delegating access from the User to the Application, i.e. **user-to-app delegation**. Typically, in OAuth the app requests access to an API and the user grants that access. OAuth then gives the app a token.

Rather, this new situation is more complicated. Since Adam wants to allow Bianca to access *his* account, delegating user responsibility to Bianca, we are dealing with

user-user delegation. There are two ways to do this. Either you set up a database or table around the API for Bianca to gain access to in order to retrieve data, or, the system grants Bianca an access token that belongs to **someone else**, i.e. Adam, granting Bianca equivalent powers.

8.4 Option #1: Access Tables

In the first approach, we use the API to retrieve data from an access table. The flow is as follows:

1. Bianca's app contacts the OAuth server.
2. The OAuth server challenges Bianca to enter her credentials (username/password).
3. The OAuth server accepts, and issues her own account's access token.
4. Bianca uses this token to send a request to the API. At this point, she must somehow instruct the API that she intends to perform on other resources — she must distinguish between users to access Adam's account. (This step is unfortunate as it requires tangling identity management within the API itself).
5. The API talks to the database to verify that the access is real, and validates access.
6. The API then responds with Adam's data to the app.

Looking at it, this is architecturally the simplest flow, but however, implementation is the hardest. This is especially difficult when building microservices that plugin to many APIs. We may be building many small APIs that need to

implement and contact services to repeatedly allow data access.

These microservices may be doing many separate things. Their communication protocols may not be similar, and on top of that, if you have user info tangled into that data, you're going to have to program a lot — a lot to leave up to developers to handle.

8.5 Option #2: Delegated Tokens: OpenID Connect

OpenID Connect is a companion protocol to OAuth. Enabling an OAuth server with OpenID Connect adds an identity layer on top. Now our API not only knows what access is being given, but it knows *who* is accessing that data.

This is processed using the OpenID Connect Userinfo endpoint, a simple endpoint that can be called using a GET verb with an authorization token. The Userinfo endpoint responds with a JSON document, containing basic user information such as name, phone, email, etc. With a user token you can retrieve Userinfo to access user information — a pretty simple process.

We can take this response, and change the meaning by adding access tokens in this response. This is how we list the delegations. We'll use the Userinfo endpoint to reissue tokens, or downgrade tokens. In the case of our bike lock, Bianca receives an access token for the Resources Owner, Adam, containing meaningful data and scopes to specify who the authenticator is — in our case, Bianca.

The new flow is like this:

1. The app requests access from the OpenID Connect enabled OAuth server.
2. The server challenges Bianca to enter her credentials (username/password).
3. Bianca receives her own access token.
4. Bianca then calls the Userinfo endpoint with her access token.
5. The OpenID Connect enabled server takes this access token.
6. The server responds with a second token (Adam's access token). This came from JSON document.
7. BIanca now has 2 tokens. When Bianca needs to operate on Adams data she uses the token she got from the UserInfo endpoint. If she needs to do stuff on her on account, she uses the initial access token she got.
8. The API will at all times receive one token, and can look at it to see which user it should operate on.

Sharing Access Between Microservices

As we learned in How to Control User Identity Within Microservices, handling user data within microservices is a very similar process. We send in an access token, and it terminates that token. If we want to access someone else's account, we simply use a different token. To regulate this process we can place a gateway at this intersection. It doesn't really matter — regular or delegated access tokens are acted upon in the same way. Our normal access case won't have to be altered — meaning that modifying the backend is not necessary.

Ideskog admits a possible drawback is that if an app must maintain a multitude of tokens. When delegating tons of

tokens with many people involved in a single flow, you may run into system bloat.

8.6 Review:

Thanks to smart approaches to delegating access, Adam can remotely share his bike lock with Bianca in a secure and confident manner. These methodologies also transfer into identity management within web apps, APIs, or IoT situations in which **users must grant functionality to other users**. As a review, to accomplish this there are two alternatives:

Access Table Lookup Approach: Identity is built into the API, and a database is stored access lookup is stored.

- Easy to implement form an ADMIN perspective
- Every API needs to know about delegation
- Every API needs to resolve access rights
- With microservices this becomes a LOT heavier.

Delegated Tokens Approach: Identity is removed form API, thrown into OpenID Connect server and OAuth server.

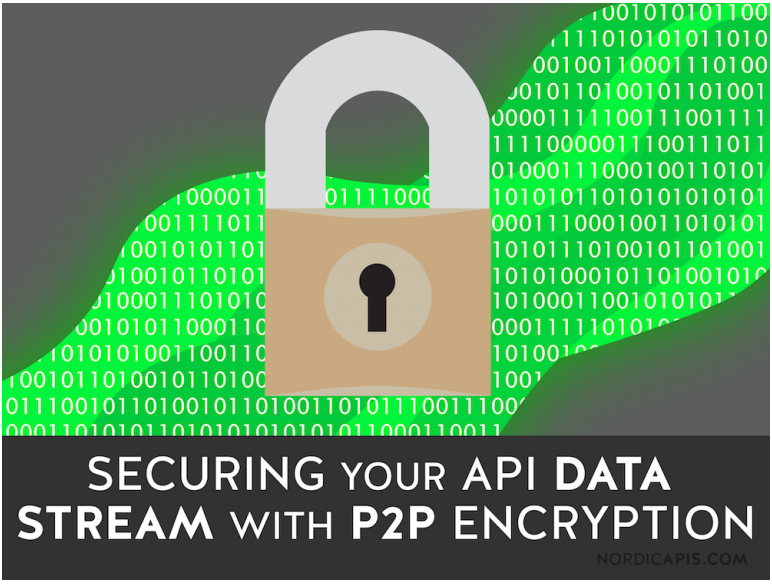
- Easy to implement on the API side
- APIs work the same for regular access and delegated access
- App must maintain multiple tokens

It's important to remember that OAuth is **user-to-app delegation**, not user-to-user delegation. However, if we throw on OpenId Connect and use the Userinfo endpoint, we can add **user-to-user delegation** and be complete with quality access sharing standards.

Disclosure:

- *Skylock is just used as an illustrative example, and we have no affiliation with the company and don't know if they used this specific technique in their system.*

9. Securing Your Data Stream with P2P Encryption



A system is only as secure as its weakest part — the most expensive chain in the world wrapped around deeply sunk steel columns is worthless if tied together with zip ties. The same holds true for security in the API space. The most secure, authenticated, stable system is only as secure as its weakest point. By and large, this weakness lies in the **data stream**.

In this chapter, we discuss the importance of securing

your API data stream, the various technologies necessary to do so, and the benefits and drawbacks of various offerings in the point-to-point (P2P) encryption industry.

9.1 Why Encrypt Data?

Systems are tricky things — because a system is by definition a collection of objects working towards a singular goal or collection of goals, their efficiency and security are directly tied to one another. Think of a network or an API server cluster as a suit of armor — you might have the best plate steel, the most intricately laid bronze work, and the most reinforced shield, but without a sturdy and protective helmet, you might as well be walking into battle nude.

This is a basic concept in network security, but it bears repeating (and remembering) - **your system is only as strong as its weakest component.**

Enter encryption. Encryption can take an old decrepit server running out of data software and encrypt the outgoing data stream into a nigh-unbreakable juggernaut. Compare encrypted traffic breaking to non-encrypted traffic breaking.

Let's say Bob sends a remote URI call to a server unencrypted. This call carries his key to the server, which then relays the information requested. With a single node in the middle listening to and capturing traffic (launching a Man-in-the-Middle Attack, this information can then be used in a session replay attack, compromising the integrity and confidentiality of your system.

Let's say that Bob did the same remote URI call, but instead utilized a 2048-bit SSL certificate encryption system

from DigiCert. Even if that node in the middle was able to completely capture the traffic, DigiCert estimates that it would take 6.4 quadrillion years to crack the data using a 2.2Ghz based system with 2GB of RAM — that's longer than the total time the universe has existed.

This amount of security is clearly valuable, and it's easy and cheap to implement — infrastructure/network solutions such as HTTPS, SSL, and TLS utilize ports and network technologies to secure the data stream as it leaves and enters the network, and these solutions are typically already built into most modern servers and workstations.

Third party solutions are also generally low in cost, with some solutions falling into the "open software" methodology of releasing programs for free to garner community support and iteration.

Keep in mind that encryption solutions are far different from methodologies that derive from encryption. For instance, API Keys are one way to secure your datastream, but in and of themselves, they do not encrypt traffic — this is in fact one of the greatest weaknesses of keys. Do not assume any solution that is not expressly encryption oriented will encrypt traffic.

9.2 Defining Terms

Before we dive too deeply into encryption solutions, we need to understand a few terms unique to encryption. These terms can have different meanings used in different industries, so it is very important to remember that these definitions may only apply to this very specific use-case.

In this piece, we are specifically discussing **P2P encryption**. While this often has connotations in the API space

of payment processing, in the larger cryptanalysis field, P2P has a looser definition. P2P simply means the point-to-point **cycle** consisting of a provider of data, the data stream carrying the data, and a consumer of data.

When discussing encryption, the concept of the **"data stream"** is also very important to understand. A data stream is the flow of data from a generating point to a receiving point — that is, from the API itself to the client requesting the data. Think of it like the postal service — the process from when a letter is put into a mailbox to when it arrives in the mailbox it was addressed to is akin to a data stream.

Most important, **encryption** is a method by which data is codified or obfuscated, preventing unauthorized access, viewing, or modification. This can be done in a variety of ways, but all of these possibilities to date fall into two distinct categories — Block Encryption and Stream Encryption.

Block Encryption is exactly what it sounds like — the data stream is put through an encryption method in blocks or chunks of data, usually in a set size dictated by the encryption method. For example, in the AES (aka Rijndael) method of block encryption, these blocks are limited to 128 bits of information, meaning that only 128 bits of data can be encrypted at a time.

Stream Encryption, on the other hand, encrypts the data stream in real time — data is not encrypted in blocks, but rather as a series of binary signals. This method is resource intensive, and is often easily cracked due to the fact that encryption methods can be detected by simply listening to the stream and finding patterns in the ciphertext output. For this reason, Stream Encryption is not used much.

9.3 Variants of Key Encryption

There are similarly two types of encryption commonly employed in the network space.

The first, **Public-key**, uses algorithms like the RSA Asymmetric Block Cipher or ElGamal to encode communications. This method uses a key pair, from which both keys are derived, with one key functioning to encrypt the data, and the other to decrypt. The decrypting key is kept secret, and is typically tied to the recipient, whereas the encryption key is public, allowing anyone to encrypt data for that particular recipient.

This is the most secure version of encryption due to the fact that the codes are separate, but defined against one another. The private key is generated from a public key, and that public key is in itself a secure key. By setting up the system this way, an unauthorized user or node could intercept the full decrypted public key and still have no access to the user system; likewise, they could intercept the full decrypted private key, and again have no access to the server.

Another form of encryption is the **Symmetric-key** methodology, which utilizes the same two keys to both encrypt and decrypt. This method is secure for local use (such as encrypting password databases on a root server), but should not be used over a network in its plain form without encrypting the session key or data stream using a secondary assymetric encrypted protocol.

The Symmetric-key method is considered to be a less-secure solution than the Public-key method due to the fact that both the encryption and decryption key are the same key — because of this, a theoretical unauthorized user could obtain either the server key or the client key,

and access any resource he or she wanted to. Public keys should not be inherently trusted either — a user must know who owns the private part of the key to ensure security. Nonetheless, Asymmetric-key encryption is typically a stronger solution despite this.

9.4 Built-in Encryption Solutions

The most common and easily implemented solution is **HTTPS**. HTTPS, standing for Hyper Text Transfer Protocol Secure, is a protocol by which HTTP, the language of the internet, is encrypted using a variety of algorithms, including RSA, ECDH (Elliptic curve Diffie–Hellman), and Kerberos.

This encrypted data is then sent over a secure protocol to the end user for decryption. This transportation happens over two main protocols — SSL and TLS.

SSL, or Secure Sockets Layer, utilizes a Public-key or Asymmetric algorithm to transmit Symmetric encrypted data — this two-part process provides incredible security and agility in encryption. This protocol uses an SSL Certificate that is issued in response to a Certificate Signing Request (CST), which is then validated. SSL is a complex system, but this complexity is largely hidden from consumers. This hidden complexity makes the user-experience that much better, improving the impression of your service.

Likewise, HTTPS can also utilize the **TLS** protocol. TLS, or Transport Layer Security, is the updated version of SSL, and is considered the modern protocol to use over SSL whenever possible. TLS functions nearly identical to SSL, with very few exceptions — notably the key derivation system and key exchange process.

It doesn't matter which method you choose (though many administrators would suggest you utilize the version just before the most up-to-date protocol version to ensure functionality) — just remain consistent! Consistency is a key part of success in API development, so whatever you choose, stick with it.

9.5 External Encryption Solutions

While HTTPS is a very flexible system, there are certain use-cases that may require third party point-to-point encryption methods designed for very specific purposes. Luckily, the API space is populated with encryption providers utilizing a variety of technologies and systems to ensure a secure ecosystem.

Take for example the CyberSource payment encryption service. This service utilizes card-reader software tied into hardware security modules to send data remotely over secure channels, increasing security and data loss prevention success rates.

For a more general solution, a system like OAuth 1.0a pairs authorization with data encryption. Utilizing a signature, typically HMAC-SHA1, OAuth 1.0a sends this data over plain communication lines (though TLS/SSL can be used). Because OAuth 1.0a does not send the password in transit directly, this all but removes the threat of sniffed traffic and captured sessions.

9.6 Use-Case Scenarios

Given that encryption of point-to-point data transmission is so vitally import for APIs transferring secure data, such

as passwords, health records, payment processing, etc., view encryption as an **absolute requirement** in any situation where there is a fiscal value assigned to data. If your data could be sold, it should be protected. If there's money involved, you can be sure there are people somewhere who want to take that stream and use it for their own purposes.

In some other cases, encryption might be required. Federal contracts, government-to-civilian client implementation, and healthcare APIs may be required to encrypt traffic as part of contractual negotiations or laws such as HIPAA (the Health Insurance Portability and Accountability Act of 1996).

That's not to say, however, that encryption is only useful for secure situations. As users increasingly value their privacy, implementing encryption standards can be as much a security feature as a selling point to potential customers.

9.7 Example Code Executions

By and large, these methods are incredibly simple for the user and the API provider to implement. For instance, in curl, you can simply use the following path command:

... curl -3 -capath –ssl https://api.website.com ...

And with that small piece of code, you're now utilizing a local certificate to sign and encrypt your data connection.

For consumers, using your web app or API in a secure mode is as simple as connecting over HTTPS. For example, when connecting to the Facebook web application, a variety of APIs are used to connect user accounts to data. Simply logging in under https://www.facebook.com

rather than the conventional http://www.facebook.com will trigger a secure HTTPS connection utilizing the encryption chosen by the provider.

9.8 Conclusion

Security is an important thing to consider — it should be one of the first things you consider in the API lifecycle. Regardless of whether your API is public or private, first or third party, a well-deployed security solution will make your API more useful, secure, and attractive.

10. Day Zero Flash Exploits and Versioning Techniques

This month, a day-zero Flash exploit was disclosed, exposing potentially millions of users' data. After an emergency patch was rushed out, two more exploits were quickly discovered, leading to a vocal demand for an end-of-life date for Flash.

This isn't just a problem that affects media outlets and entertainment sites — vulnerabilities from services such as Flash and other dependencies commonly used in the

production, use, and implementation of APIs are often subject to the same day-zero exploits and disclosures as the beleaguered Adobe Flash platform.

Here we discuss the nature of security in the modern dependency-centric design methodology, the importance of versioning, and some basic steps developers can take to secure their product throughout the API Lifecycle.

10.1 Short History of Dependency-Centric Design Architecture

Unauthorized access to resources on a network is as old as networking itself — in 1976 and 1978, for example, John Draper (aka Captain Crunch) was indicted for using a cereal-box toy whistle to generate tones for free calling around the world. Vulnerabilities in phone systems, cable television systems, and early internet-based subscription systems have long been the focus of skilled, intelligent, and persistent hackers.

In the old days, hacking into systems was a difficult prospect — once systems began to move away from tone generation (as in the phone system) and basic unencrypted passwords and usernames, gaining access to these systems became more difficult. Larger connected systems soon began to implement proprietary security solutions, managing Authentication and Authorization using internal services behind a secure connection to the outside world.

As the concept of user-oriented services such as payment processing, online shopping, and internet media providers began to take hold, however, a certain amount

of standardization was required. With the burgeoning World Wide Web taking the world by storm on the standardized TCP/IP protocol stack, developers soon began to create systems that they could tie into their own code for further functionality.

The code created with these programs depended on the standardized programs to function — hence the term **dependencies**. Shockwave, Flash, and other such systems enabled developers to stop "re-inventing the wheel" each time they created a new product, and assured compatibility between disparate systems.

This shift in architecture design has created many of the problems API developers face today — the change from **internal design** to **dependency-centric design** has created a situation where dependency vulnerabilities are causing increased security concerns.

10.2 The Hotfix — Versioning

Seeing the issue with dependencies and the constant need to update and revise, developers soon implemented a solution known as **versioning**. With versioning, an API or service developer can elect to use the most up-to-date version, beta versions, and even old versions of a dependency for their code.

While this was largely meant for compatibility, allowing services to utilize older dependencies on machines without access or permissions to utilize newer dependencies, this also created a security situation where developers of the dependencies could ensure security by **pushing out patches** to fix vulnerabilities identified through testing and real-world use cases. While this made development

simpler for the developers, it came with its own set of caveats.

Within the context of modern API development, versioning has two basic approaches, each with strengths and weaknesses.

The First Approach — Update and *then* Review

The first and most common approach is to Update and Review as dependency versions are released. By taking this approach, an API developer will immediately install updated dependencies or systems, and then review the update through changelogs and real-life use. This type of versioning often ignores beta or test releases, and focuses on stable releases.

While this seems like a good approach, there are some huge weaknesses. By not first looking at the code of the newest patch for the dependency, an API developer is placing **blind faith** in the dependency developer. This shifts responsibility from personal to external, creating a situation where the security of your ecosystem is placed squarely on the shoulders of someone without a vested interest in keeping your system safe.

Furthermore, this approach could have an **inverse impact** — by installing an untested or unreviewed version, an API developer could potentially open themselves up to more exploits than the previous unpatched dependency. For many, this is simply a caveat too big to ignore — it's easier to fight a battle with an enemy you know than to wage a war with untold enemies you don't.

The Second Approach — Review and *then* Update

The second and less common approach is to Review and Update. Unlike the first approach, this necessitates the developer first reviews the code of the dependency update, comparing it to previously known code. The developer installs only when entirely convinced of the update's veracity and completeness.

This might seem counter-intuitive — "increase your security by not installing new updates". The truth is that you dramatically increase your security by using this approach simply due to the fact that you will know what you are fighting.

Let's say you use a dependency such as Flash for the web portal for your API. You know a set of exploits exist that can force a buffer overflow. Knowing this, what is the best solution — update this dependency to a new version with unproven, unreviewed code, or handle the buffer overflow with a prevention system and self-patch while you review the new update?

Simply put, **an API developer should review each and every piece of code before implementation** — or at the very least, compare it to the already existent code to ensure compatibility, security, and safety amongst a wide variety of their use cases.

10.3 Dependency Implementation Steps: EIT

So what then is the proper process an API developer should take to ensure that something like the recent day-zero Flash exploits don't occur? This process can be summed up in three simple letters: **EIT**.

- **E** — Examine. Examine the code of the dependency patch you are intending to install. Check it against previous patches — has the code changed dramatically? If so, do you understand how it has changed? Use changelogs and bug trackers to examine issues with new patches. Use these resources to dig deep into how the dependency functions and why the patch had to be created.
- **I** — Implement. Once you have thoroughly examined the code of the dependency update, the dependency should be implemented. Ensure that compatibility between security systems and the updated dependency is maintained.
- **T** — Test. Test your API. Use penetration testing tools, try and trigger buffer overflows, and send un-validated calls and requests to see how your API responds. If you trigger any failures in the new patch, add your result and methods to duplicate to the bug tracker for your dependency after repairing your ecosystem to ensure it doesn't continue to be an issue.

By following this set of guidelines, an API developer can ensure that their API dependencies and complete ecosystem functions as intended, ensuring long-term security and functionality throughout patching.

10.4 Lessons Learned

Dependencies are a wonderful thing — to be able to quickly create, implement, and patch, API developers must use a certain amount of dependencies. When these dependencies inevitably are exposed, a trend towards blaming the dependency developer quickly blooms.

This is a poor response — the fact of the matter is that the vulnerability in the API is not the fault of the dependency developer, but the fault of the API developer who adopted that dependency. Security is solely the domain of the developer — shifting responsibility to the user or to another developer is simply bad practice, and will lead to a less secure ecosystem.

10.5 Conclusion

Though the recent exploits are focused on Flash, this is not just an issue exclusive to platforms, either — vulnerabilities in methodologies and workarounds in code such as Scala, Go, and other such languages can lead to buffer overflows, memory leaks, and poor encryption. Issues within the cloud computing stack can expose huge security flaws. Even lack of effective encryption could result in the complete exposure of network resources which may betray your security ecosystem.

The takeaway? **Check your dependencies, and check your code — security for your API is solely your responsibility.**

11. Fostering an Internal Culture of Security

Fostering an internal **culture** of security is paramount to an organization's success. This means adopting API security and developer responsibility as cultural **norms** within your organization. In this chapter, we talk about why this culture is so important, and what steps to take to improve your internal approach to promote sustainability and growth.

11.1 Holistic Security — Whose Responsibility?

There is a mindset amongst many novice developers (and, unfortunately, many seasoned veteran developers) that security is the responsibility of the user. After all, it is the user that holds the keys to the kingdom in the form of a username and password, along with authentication/authorization profiles and usage needs.

This is fundamentally flawed, however, if only for one reason — the user does not have complete access to the system they are requesting access to. By its very nature, an API is restrictive to those remotely accessing it when compared to those with physical access to its server. Thus, **both the the API user *and* the API developer have a large security responsibility**.

Think of it this way — a person invites a friend to house sit for a week and gives them a key. At that moment, the key is the friend's responsibility. In an API environment, the username, password, or token are similarly the user's responsibility.

But whose responsibility is the front door? Who decided the type of material the door was made of? Who needed to change the lock when it stopped working properly? Who had the keys made? The homeowner did. In the API space, providers must similarly take responsibility to ensure security within their system.

A user can only be responsible for that which they have

— the methods by which they authenticate and authorize themselves. Federation, delegation, physical security, and internal data security is within the purview of the API developer for the simple fact that they are the ones most able to ensure these systems are secure.

11.2 The Importance of CIA: Confidentiality, Integrity, Availability

An ideal system balances Confidentiality, Integrity, and Availability in harmony with security solutions and access requirements. With so many APIs functioning on a variety of platforms, and with many modern systems utilizing cloud computing and storage, internal security balanced with external security is of incredible importance.

Confidentiality

Confidentiality is the act of keeping information away from those who should not be accessing it. In the API space, a division needs to be made between **external** and **internal** confidentiality. External confidentiality is, obviously, the restriction of external access to confidential materials. This includes access to API functionality not needed for the user's specific requirements, and restricted access to password databases.

While confidentiality is often handled utilizing encryption to obscure information, there is a great deal of information that cannot be encrypted — the information held by the developers of the API. This internal confidentiality is often far more dangerous than any external confidentiality issue could ever be.

As an example, assume a developer is using a flat database system for passwords that is protected by an internal authentication service. This service, hosted on a Linux server, requires a username and password on the root level to access the authentication tables.

An Internal security culture restricts data to only those who have the rights to see it

A hacker is attempting to access a confidential server, and has a direct connection to your systems and servers through an API. He calls the developer's office, and states that he is the hardware provider for the server calling to issue a patch for a massive vulnerability, and that he needs a private, unrestricted session with the server.

The developer creates a root username/password combination, which the hacker is able to use to enter the service unrestricted, and steal the authentication tables for nefarious purposes.

This is called **phishing**, and it's a huge risk that many people have fallen afoul of. Promoting an internal culture of security in this realm means ensuring that data is kept secure, that developers follow policies ensuring security of authentication and authorization protocols, and that things like phishing are avoided.

In addition to ensuring that a culture of security exists, make developers aware of these threats. Have a properly utilized and understood system of Authentication, Authorization, Federation, and Delegation to ensure unau-

thorized external access granted by internal developers becomes a non-threat.

Integrity

Integrity in the API space means ensuring data **accuracy** and **trustworthiness**. By maintaining data streams and employing secure workstations, unauthorized changes to data in transit should not occur, and the alteration of hard-coded information becomes a non-threat.

An internal security culture guarantees data is only changed by those authorized to do so

Unlike confidentiality, threats in this category are often internal. In the corporate world, disgruntled employees, faulty servers, and even poor versioning can lead to the change of data during the transit cycle. Poorly coded software can return values that should not be returned, vulnerabilities that should be secured by software can be breached, and physical transmission of code can result in captured sessions and man-in-the-middle attacks.

One of the best ways to manage the integrity of a system is to educate developers on exactly what their API traffic should look like. This can be done with a range of API Metric solutions which can show the rate of traffic, the type of data requested, the average connection length, and more.

Know which services are most often attacked and through what method, and take steps to secure these resources

(through bastion workstations, DMZ zones, etc.). Preemptively stop these problems by educating developers on secure data transmission, protocol requirements, and what is a "normal" and "abnormal" data stream.

Adopt a culture that places prime importance on **risk management** — especially when it comes to integrity — one of the harder things to maintain. Balance risk management with effectiveness of the service, however, ensuring that integrity exists alongside ease of use and access for clients and users.

Linda Stutsman put it best in an interview with Andrew Briney of Information Security Magazine:

> "It's said time and time again, but it's absolutely true: You have to get to the point where risk management becomes part of the way you work. That starts with good policies driven by the business — not by security. Communication is absolutely the top factor, through policies and training programs. Then it's determining the few significant metrics that you need to measure."

As an aside, the integrity of an API is not wholly dependent on the software or code transmission factors — a lot can be said for the physical network the API is planning on running through, and the limitations inherent therein.

For instance, if an API developer is creating an API for local area record transmission, such as a hospital setting, knowing whether the signal will be transmitted through coaxial or fibre-optic cable, whether these cables will be running near power transmission causing data loss, and even whether the data will be exiting to the wider Internet,

will inform the developer as to error-checking, packet-loss mitigation, and integrity-increasing features that might be required and maintained.

Availability

An internal security culture focuses on high uptime and ease of access

While it's important to ensure that your API has great data confidentiality and integrity, perhaps the most important attribute of an effective culture of security is ensuring availability. After all, if an API cannot be used by its users, then is it really an API?

Whether an API is Private, Public or Partner-centric, ensuring your API is accessible is incredibly important. This can be balanced in a handful of ways, but all of these techniques can be broadly summed up in two categories — ensuring availability through **developer activity** and through **user activity**.

Let's look at **developer activity**. First and foremost, developers should understand that every single thing they do to an API or the server an API runs on will affect the availability of the system. Updating server firmware, changing the way an API call functions, or even accidentally bumping into a power strip can result in the failure of availability for many users.

Additionally, some changes that are considered simple are actually catastrophic for the end-user. Consider **versioning** — while updating to the newest version of a

dependency might deliver the most up-to-date content for your user, if this update does not support legacy systems or services, an API might be fundamentally broken. Changes should be balanced through the **lifecycle** of the API, regardless of whether the API in question is a First or Third Party API.

User activity is far easier to handle. Threats to availability from users often spring from poorly formed requests in the case of non-malicious threats, and in port-scanning or traffic flooding (especially UDP flooding) in malicious threats. These user threats are easier to handle, and can often be taken care of simply by choosing the correct architecture type and implementing solutions such as buffer overflow mitigation, memory registers, and error reporting.

11.3 4 Aspects of a Security Culture

So far we've covered high-level concepts — let's break down what an effective culture of security is in four bullet points. These points, when implemented fully, should not only create an effective culture of security, but lead to growth and stability over time.

A culture of security entails:

- **Awareness of Threats** - developers should be aware of potential threats to their system, and code their APIs accordingly;
- **Awareness of Vulnerabilities** - developers should recognize vulnerabilities inherent in their system, servers, or devices. This includes vulnerabilities arising from their own code and systems as well as those from third party vendors, services, or servers;

- **Awareness of Faults** - developers should be conscious of their own personal faults. If a developer has a history of misplacing thumbdrives, sharing passwords, or worse, they should not be responsible for internally managed secure services;
- **Awareness of Limitations** - developers should know the network that is being utilized for their API, and what limitations it represents. Security solutions for a close intranet running on fibre-optic cable will be different than solutions for an Internet-facing connection running on coaxial or twisted-pair;

11.4 Considering "Culture"

It's important to consider the function of culture within an organization. All of the topics discussed in this piece are applicable in a huge range of situations, environments, and organizations, due to the nature of security. Security concepts are universal, and scale directly with the size of the data being protected.

The way a culture of security is built and perpetuated is directly influenced by the type of organization which adopts it. For instance, in a governmental organization, this culture can be directly enforced through policy, law, and guidelines, whereas in a non-profit, this information must be disseminated through classes or instructional guidelines to workers who may be unfamiliar with such stringent policies.

In a corporate environment, much of this security can be managed directly through limiting privileges and abilities.In a corporate environment, each server or service might have its own administrator, and by limiting powers,

knowledge, and abilities to only those that need it to function, you maintain a culture of security.

In a small startup or non-profit, however, one person may need access to ten different services and servers. In this environment, where the success of the company directly controls the well-being of its employees in a very granular way, reaching out verbally or via email can be extremely effective, as there is a personal stake in security.

11.5 All Organizations Should Perpetuate an Internal Culture of Security

Fundamentally, fostering an internal culture of security is easiest to do in the earliest stages — beginning with a strong security-focused mindset ensures that you can revise, expand, and reiterate while staying safe against current attacks. Additionally, preparing your systems for known attacks and being aware of any vulnerabilities ensures that any system can stay secure long into the future against new, unforseen attacks.

By acting on these points, you make your API a more effective service for the user. An insecure service might be functional, but a secure service is fundamentally useful.

Resources

API Themed Events

To learn about upcoming Nordic APIs meetups, conferences, and seminars, subscribe to our newsletter or check our event calendar. Here is a list of Nordic APIs sessions referenced in this e-book, by order of appearance. Also follow our YouTube channel for more videos.

API Security Talks:

- The Nuts and Bolts of API Security: Protecting Your Data at All Times, Travis Spencer, Twobo Technologies
- Integrating API Security Into A Comprehensive Identity Platform, Pamela Dingle, Ping Identity
- Pass On Access: User to User Data Sharing With OAuth2, Jacob Ideskog, Twobo Technologies
- Building a secure API: Overview of techniques and technologies needed to launch a secure API, Travis Spencer, Twobo Technologies
- OpenID Connect and its role in Native SSO Paul Madsen, Ping Identity
- OAuth and OpenID Connect for Microservices, Jacob Ideskog, Twobo Technologies

Follow the Nordic APIs Blog

Much of our eBook content originates from the Nordic APIs blog, where we publish in-depth API-centric thought pieces and walkthroughs twice a week. Sign up to our newsletter to receive blog post updates via our Weekly Digest, or visit our blog for the latest posts!

More eBooks by Nordic APIs:

Visit our eBook page to download any of the following eBooks for free:

- **The API Lifecycle**: A holistic approach to maintaining your API throughout it's entire lifecycle, from conception to deprecation.
- **Developing The API Mindset** : Details the distinction between Public, Private, and Partner API business strategies with use cases from Nordic APIs events.
- **Nordic APIs Winter Collection**: Our best 11 posts published in the 2014 - 2015 winter season.
- **Nordic APIs Summer Collection 2014**: A handful of Nordic APIs blog posts offering best practice tips.

Endnotes

Nordic APIs is an independent blog and this publication has not been authorized, sponsored, or otherwise approved by any company mentioned in it. All trademarks, servicemarks, registered trademarks, and registered servicemarks are the property of their respective owners.

- Select icons made by Freepik and are licensed by CC BY 3.0
- Select images are copyright Twobo Technologies and used by permission.
- The Mobile/Enterprise/API Security Venn diagram was created by [Gunnar Peterson][gunnar] and also used by permission.]*

Nordic APIs AB Box 133 447 24 Vargarda, Sweden

Facebook | Twitter | Linkedin | Google+ | YouTube

Blog | Home | Newsletter | Contact