API Security

A guide to building and securing APIs from the developer team at Okta

{okta}

with a foreword by

Les Hazlewood

API Security

A guide to building and securing APIs
from the developer team at Okta

Foreword by Les Hazlewood

With chapters by

Lee Brandt

Keith Casey

Brian Demers

Joël Franusic

Sai Maddali

Dave Nugent

Matt Raible

API Security

by the Developer Team at Okta

○ ○ ○ ○ ○

Table of Contents

○ ○ ○ ○ ○

Acknowledgments

Hello World

○ ○ ○ ○ ○

Foreword

by Les Hazlewood

I entered the world of information security almost 20 years ago, as often occurs in our industry, by accident. I was a software engineer excited to be working on a very large software system that was being created for the New York Port Authority - right after 9/11. The system was complex and the problems we were solving were genuinely interesting and intellectually gratifying. And because of our customer, keeping the system and its users secure was of the utmost importance. I went from being somewhat naive about how applications were secured to being thrust into a team that was responsible for securing one of the most important computer systems in the country given events at the time. It made software security very real - in a very concrete in a way I hadn't experienced before. When the project was done, I was proud to have helped in some small way to making New York a safer place.

I learned immensely from that experience, and realized how much technology then and since was advancing at break-neck speeds. The learning curve was really steep and all the while, technology was advancing rapidly, and you constantly had to keep learning. It's easy to forget about security when you've got 20 other things to learn just to get an application out the door! But if I take a step back and look at our industry with a wider perspective, I can't help but be intrigued by this exponential growth and innovation - and how security will always play a part.

Humanity has always had the drive to innovate, but we've also been determined to undermine our own advancements for selfish gain through surreptitious means. As a result, it is incumbent on us, the builders and innovators of the world, to protect ourselves. From the Mesopotamian potter 3500 years ago who wanted to keep his glazing techniques secret from competitors to modern banks who safeguard the world's digital financial transactions - there has always been a need to keep information secret. And there have always been people trying to steal those secrets.

What's important about this dichotomic dance between information holder and information thief is that the dance never ends - a safeguard today will eventually be bypassed tomorrow, which then must be be supplanted by a better safeguard. Unfortunately, even smart, capable people and corporations forget this or even ignore it, which is why we have the Equifax and Yahoo data breaches of the world.

So what does this mean in the current climate of exploding connectivity between millions of devices in the world, and the HTTP APIs that are shared and consumed between them?

To put it simply, it's the Wild West out there!

Of course, no one expects you to wear leather chaps and ride a horse to work (but if that's your thing, you do you, and do it proudly!), but we software developers are constantly looking down the barrel of a hacker's metaphorical six-shooter.

And while a bit tongue-in-cheek, the Wild West metaphor is valid - the Western frontier in the United States' early years was expanding and changing quickly, and law enforcement often wasn't available. Individuals and companies had to protect themselves using the best strategies and technologies at their disposal. Similarly, our computer and information technology industry is soberingly new - the first digital computer was invented only 70 years ago, in the time of a single human life span! Our still-nascent industry clearly reflects the same opportunity to expand and build, and for some, to engage in nefarious activity.

So what about us? The web and mobile application developers? The API developers? How do we address this expanding frontier when even massive companies fall victim every day?

I believe the answer is that we be smart, informed, and proactive. We focus on known best practices and never stop looking for new ones. We implement modern approaches that have been proven successful in real, practical experience. We stay diligent and learn from those who have come before us, thus standing on the shoulders of giants, like the authors of the book in your hands right now.

So you don't have to be afraid of the Wild West. There are amazing opportunities ahead and the Internet is still the Great Equalizer for today's builders and innovators. And for the API builders among you? Armed with the wonderful information in this book, I'm confident you'll be ahead of (and safer than) 99% of all other APIs today.

Head 'em up, move 'em out!

Chapter 1

Transport Layer Security

By Dave Nugent

Any discussion of API security, and more broadly security online, has to start with an understanding of Transport Layer Security (TLS) and its cryptographic underpinnings. Transport security adds privacy and integrity for messages between two parties. In common usage, it's the ability to transmit data over a network without exposing that data to untrusted third parties.

Transport security is critical to modern internet infrastructure, where machines on the public internet exchange sensitive data such as passwords, personal information, financial transactions and classified material. Without efficient and effective transport security, these transactions would be tedious or impossible to complete over a shared network.

The rest of this book dives into detail about best practices for securing your APIs. This chapter focuses on the necessary first step of being able to communicate securely over a network. We'll also cover common pitfalls and best practices for securing your data in transit.

A Brief History of Secure Data Transport

Today we take for granted our ability to send data over the internet securely. When we purchase an item from Amazon or complete our

taxes online, we place our trust in the infrastructure of the internet to securely share our data with only the systems and parties we intend.
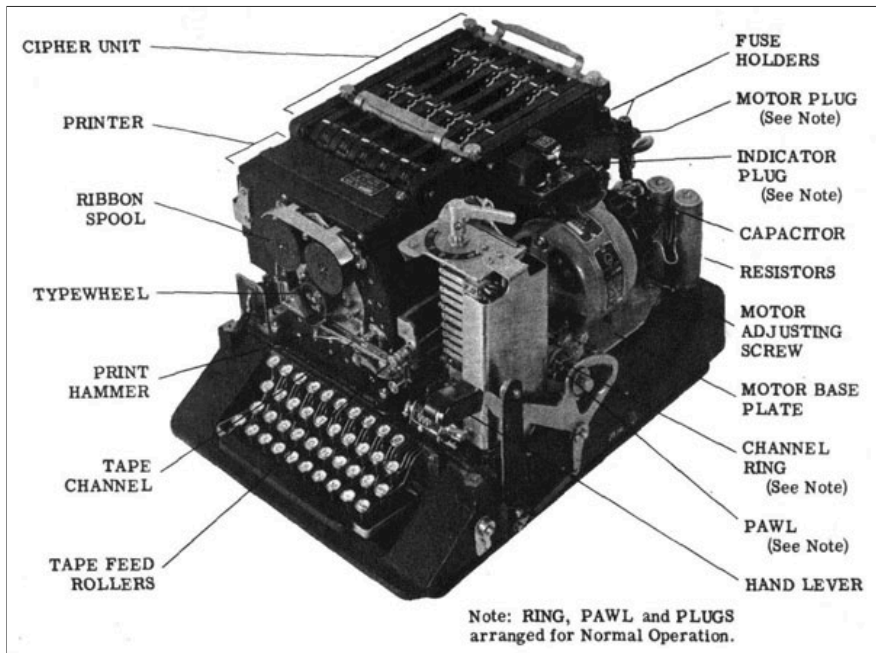
Figure 1-1: The SIGABA, a cipher machine used by the US during World War II

Historically, secure information exchange wasn't simple. Ciphers and cryptography were used in ancient times to encode sensitive messages. These schemes generally had to two main weaknesses: they relied on a shared code book or cipher to encode/decode the messages, and third parties could decode them through pattern analysis.

The problem of a shared code book or cipher persisted for millennia. To ensure that each party would be able to encode/decode the message successfully, a secret "key" needs to be exchanged, for example by courier or diplomatic bag. By working off the same key, the parties to the message exchange would then be able to encode/decode their data. However, if a nefarious actor were

able to gain access to the secret key (for example, by bribing the courier) then all past and future communication could be compromised.

How Does Key Exchange Work Today?

Current hybrid cryptosystems like SSL/TLS use symmetric key algorithms (they are generally faster than asymmetric algorithms.) Symmetric key algorithms require a shared secret, exchanged via key-exchange algorithm.

The most famous cryptographic protocol for key exchange is Diffie–Hellman, published in 1976 by Whitfield Diffie and Martin Hellman. Diffie–Hellman allows the creation of a shared secret between a sender and receiver. This shared secret is unable to be deduced by an eavesdropper who is observing the messages between the sender and receiver, except via a brute force attack. If the keyspace for the shared secret is large enough and the secret generated is sufficiently random, brute force attacks become nearly impossible.

Acronym Party: HTTPS/SSL/TLS

Now that we're up to speed on the basics of key exchange, let's discuss some of the acronyms you'll see throughout this discussion:

- **HTTPS**, also called HTTP over SSL/TLS, is an extension of HTTP which encrypts communication. HTTPS URLs begin with "https://" and use port 443 by default. This is an improvement over HTTP, which is vulnerable to eavesdropping and man-in-the-middle attacks.
- **SSL** or Secure Sockets Layer was released by Netscape in 1995. SSL adoption increased after the redesigned SSL 3.0 was released in 1996. The IETF prohibited SSL 2.0 in 2011. SSL 3.0 was prohibited in 2015 after the IETF identified various security vulnerabilities which affected all SSL 3.0

ciphers.

- **TLS** or Transport Layer Security is the successor to SSL. In fact, the documentation for TLS 1.0 describes it as an "upgrade" of SSL 3.0. The current TLS version is 1.3. Although virtually all HTTPS-secured traffic uses TLS due to problems with SSL, the SSL nomenclature persists in internet culture. These days, when somebody says SSL, it is likely they mean TLS.

In this article, I use "SSL/TLS" to avoid ambiguity that the term SSL causes. However, your implementations should always use TLS.

How Does SSL/TLS Work?

When an SSL/TLS connection needs to be made between a client (a browser, or an application attempting to access an API) and a server (a web server, for example, or an API endpoint) it follows a number of steps. The TLS spec, published by the IETF Transport Layer Security Working Group, gives an overview of these steps.

Here is a visual representation of how the client or "sender" and server or "receiver" set up an SSL/TLS connection:

.

Figure 1-2:

Let's walk through the steps at a high level:

TCP Connection

Your client (browser or application) will initiate a TCP connection with the server. Your client and server can exchange information once this connection is established. Although TLS can work over different transports, by far the most common use case is over TCP/IP, due to its ubiquity, reliability of transport and ability to recover from lost packets and transport errors. In the diagram, SYN, SYN ACK, and ACK denote this sequence of events.

SSL/TLS Handshake

The SSL/TLS handshake takes place once a TCP connection is established.

ClientHello

The client sends a "ClientHello" message, which lists the versions of SSL/TLS the client is capable of, what ciphersuites it has available, and any compression types available.

ServerHello and Certificate Response

The server responds with the same information as the client, and sends the server's certificate back to the client as well.

Certificate Verification

The client verifies that the certificate is valid, and also verifies that the server is authentic and not an impersonator conducting a man-in-the-middle attack. For more information about how certificate verification is accomplished, see "SSL/TLS Certificate Verification" later in this chapter.

ClientKeyExchange

In the previous section, we discussed key exchange, with the example Diffie–Hellman algorithm. This is the step of the handshake where the key exchange actually happens.

Finished/Application Data

Now that the handshake has been established and keys have been exchanged, information may be encrypted and decrypted between the client and server based on the shared secret which has been generated. This symmetric cryptography will secure the remainder of application-to-application communication.

Exposed Data over SSL/TLS

TLS aims to provide data integrity and privacy between two trusted parties. Information exchanged over the latest version of TLS

should be secure from being exposed to third parties in unencrypted form. Additionally, third parties should be unable to modify that information: this is the concept of data integrity, and is the reason an integrity check is performed on each message.

However, even though application data transmitted over a properly-established TLS connection is secure, some metadata and connection information is necessarily exposed to third parties. Without additional obfuscation outside of the scope of SSL/TLS, an observer will be able to discover:

- **The IP addresses of the client and server.** Since the client and server are communicating over TCP/IP, which operates at a lower level than the TLS protocol, these IP addresses are public and are used for routing the encrypted packets.
- **The server certificate**, including the server name. Since the server sends the certificate to the client as a part of the handshake process before encrypted messages are sent, an observer of the handshake will see the certificate in plain text.
- **The approximate length of the URL and payload.** Although the application data is encrypted, an astute observer of an HTTPS connection will be able to deduce the length of the URL requested and the approximate size of any non-cached assets. Cached assets, since they reside on the client already, are not vulnerable to this introspection until the cache expires and they are re-fetched.

In addition to the data listed above, additional information may be inferred based on the timing of network requests. Outdated SSL/TLS versions have additional identified vulnerabilities, and in the future one must anticipate the TLS spec will be versioned to ameliorate any vulnerabilities identified in the future.

SSL/TLS Server Certificates

You've seen that SSL/TLS server certificates are integral to the SSL/TLS handshake. They help the client verify that the server is who they appear to be, which helps prevent third parties impersonating the server. But what are certificates, anyway? How are they generated, and why do the clients trust them?

SSL/TLS Server Certificates are small data files that encapsulate information about the server that owns the certificate. This information is verified through a chain of Certificate Authorities that bridge the gap between the authorities that the browser trusts and the authorities that trust the server.

Certificate Generation

Private or self-signed SSL/TLS certificates are trivial to create. OpenSSL, available for most platforms, allows users to create self-signed SSL/TLS certificates. These consist of a private key and a public key, which a client and server can use to encrypt data and exchange it securely.

The downside to self-signed certificates are that they provide no guarantees of the server's identity. This may not be an issue for corporate networks, where certificates can be exchanged and trusted through internal provisioning.

Trusted Certificates

If anybody can create and self-sign their own certificate, then how is a client (be it a browser or an application hitting an API endpoint) able to verify a server's identity?

On the public internet, trusted certificates are required. Historically, generating these trusted certificates could be expensive. Trust and identity on the web works similarly to meeting individuals in the real world. If somebody wants to verify your identity in person, they may ask for a photo ID generated by a trusted third party, like a

government, school or corporate institution. The person verifying your identity trusts that this third party has verified your identity and that your ID cannot be forged.

This is called a chain of trust, and certificates are verified on the web in the same way. Each client has a list of third parties that they trust to verify certificates. These are called root certificate authorities. Microsoft, Oracle, Mozilla, Adobe and Apple maintain lists of trusted root certificate authorities through their own root programs and include these lists in the operating systems and browsers they produce.

Recently there has been an effort to spread adoption of HTTPS by making generating and installing certificates as easy as possible. Amazon Certificate Manager *(https://aws.amazon.com/certificate-manager/)* and Let's Encrypt *(https://letsencrypt.org/)* are two certificate authorities who make it easy to create and manage trusted certificates for free.

SSL/TLS Certificate Verification

Now that we understand the importance of trusted certificates and why certificate authorities are necessary, let's walk through the missing middle step: how a client verifies a server's SSL/TLS certificate.

First, the client gets the server's certificate as part of the SSL/TLS handshake. (If you are writing an application that is hitting an HTTPS API endpoint, this step happens before any application data is exchanged.)

The client checks to ensure that the server's certificate is not expired and that the domain name or IP address on the certificate matches the server's information. Then, the client attempts to verify that the server's certificate has been properly signed by the certificate authority who authorized it. Due to the nature of asymmetric encryption, the client is able to do this using the

information within the server's response -- without even contacting the certificate authority.

It's unlikely that the server's certificate is signed directly by a root certificate authority that is trusted by the client. However, the client can trust any number of intermediate certificate authorities, as long as the trust chain eventually leads back to one of the client's trusted root certificates:
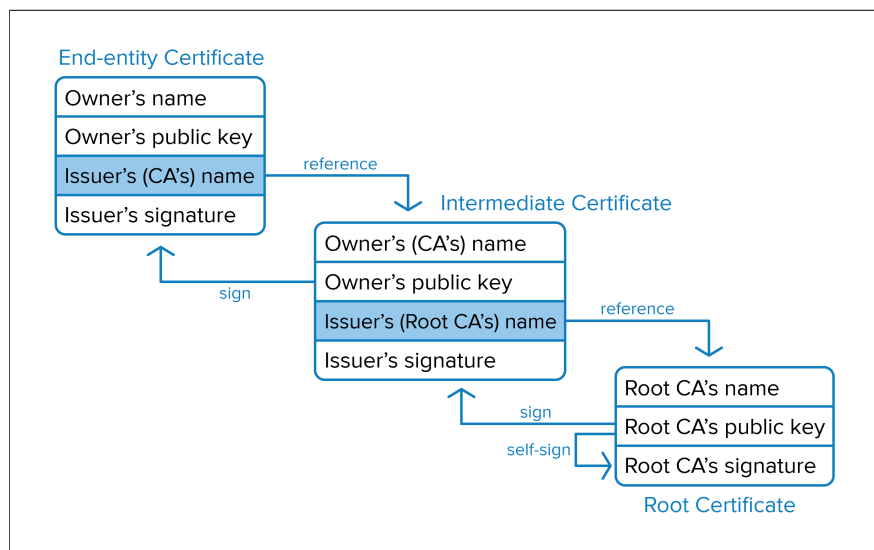
Figure 1-3: *Illustrating the chain of trust from a root CA through an intermediate certificate*

For each intermediate certificate, the client completes the same process: it verifies the issuer's name matches the certificate owner's name, and uses the signature and public key to verify that the certificate is properly signed.

Eventually, in a successful transaction, the client will come to a self-signed root certificate that the client implicitly trusts. At this point, the client has built a cryptographic chain of trust to the server, and the SSL/TLS handshake can proceed.

SSL/TLS Best Practices

Hopefully this chapter has convinced you of the ease and importance of implementing SSL/TLS into your public internet application infrastructure. However, even when using SSL/TLS, organizations can be subject to compromise if best practices are not followed. Let's go over a few of the big ones:

1. Use TLS Everywhere

Having certain pages served over SSL/TLS and some served unencrypted can expose data, such as unencrypted session IDs, to attackers. Similarly, don't allow TLS content to be exposed via non-TLS pages and don't mix TLS and non-TLS content on the same page.

2. Keep Sensitive Data out of the URL and Cache

URLs can be cached in a client's browser history or application logs and sent to another HTTPS site if the user clicks on a link. Setting TLS pages to be uncacheable prevents information leakage from the client cache.

3. Prevent Exposed Data over SSL/TLS

Ensure browsers and applications only access your site via HTTPS by enacting HTTP Strict Transport Security (HSTS). Servers utilizing HSTS send an HTTPS header in their response specifying that requests to their domain should only use HTTPS. An HSTS-complaint client should then make all future requests to that domain over HTTPS, even if HTTP is specified. This helps protect clients from man-in-the-middle and eavesdropping attacks that could be initiated by the client sending sensitive information by making an unsecure HTTP request. For more information, see *https://www.owasp.org/index.php/ HTTP_Strict_Transport_Security_Cheat_Sheet.*

4. Use HTTP Public Key Pinning

While not common in browser-to-web-server communications, HTTP Public Key Pinning is quite useful in API communication. The server will respond with an HTTP header specifying a hash of a valid public key, which helps combat certificate authority compromises. When communicating between two server-side apps, if one server has been compromised and an untrusted certificate authority is trusted, TLS compromise can happen. By having the client download and store a known-valid certificate from the server, the client can "skip" the chain-of-trust verification and instead compare the server's certificate directly to their known-good version, thereby guaranteeing authenticity and preventing any opportunity for man-in-the-middle.

5. Only Support Strong Protocols and Ciphers

Ensure your infrastructure uses the most recent stable version of TLS and the latest recommended ciphers. Due to evolving vulnerabilities, preferred ciphers may change over time.

OWASP maintains a nearly definitive list of best practices for SSL/TLS online at *https://www.owasp.org/index.php/Transport_Layer_Protection_Cheat_Sheet*.

SSL Rating

After implementing SSL/TLS into your API endpoint infrastructure, be sure to run an SSL Rating test to validate your use of the SSL/TLS Best Practices in the section above. While an A+ SSL rating is not a guarantee that your infrastructure is ideally provisioned, any lower rating should raise red flags that you can correct.

To run the SSL Rating test on your public-facing site, visit *https://www.ssllabs.com/ssltest*.

Even if you aren't interested in learning about Diffie-Hellman or Certificate Authorities, you still need to secure your API using TLS/SSL and follow the best practices above.

However, I do encourage you to learn more about the details of TLS/SSL. It can be a daunting task and seemingly never-ending task sometimes, but as with any technology, the more you know about TLS/SSL, the more effective you'll be at building, testing, and reasoning about using it to secure your API. While once could fill an entire book about this topic, hopefully this chapter helped you learn more about the basics of TLS/SSL or helped fill in some gaps in your knowledge.

Chapter 2

DOS Mitigation Strategies

By Lee Brandt

Unfortunately, there's no way to *completely* prevent Denial of Service (DoS) attacks. And that's a problem because they're a plague upon our industry. Between botnets consisting of 6 to 30 million bots *(https://themerkle.com/top-4-largest-botnets-to-date)*, 30,000 distinct DoS attacks per year and growing *(https://www.securityweek.com/internet-sees-nearly-30000-distinct-dos-attacks-each-day-study)*, and 11.2 billion connected "things" in use by the end of 2018 *(https://www.gartner.com/newsroom/id/ 3598917)*, this problem is only going to get worse. As a developer, the only thing you can do is attempt to mitigate the probability and effectiveness of an attack. So, how do you do that? First, you have to understand what a Denial of Service attack is.

What Is a DoS Attack?

A Denial of Service attack occurs when attackers attempt to stop a service from servicing the requests of legitimate users. Most commonly, this is done by flooding the service with requests until it is no longer able to respond as fast as the requests are coming in. By forcing the service to respond to the flood of requests continually, the service denies legitimate traffic.

Why are DoS Attacks So Prevalent?

There are several reasons why someone might perform a DoS attack on a network or service. Most frequently, DoS attacks are carried out for profit. There are several ways to make money by staging a DoS attack. For instance, competitors of Amazon might find it beneficial if Amazon's service were slow or offline. This form of industrial sabotage may encourage customers to search for alternatives, increasing profits for competing businesses who aren't suffering the effects of a DoS attack.

Another way attackers can profit from DoS attacks is by selling access to compromised computers (called "BotNets") that can perform large-scale Distributed Denial of Service (DDoS) attacks. There are many places online where people can contract BotNets to carry out DDoS attacks. They can typically be rented and charge based on the amount of time that they slow or disable the target service; costs range from a few dollars for a fifteen-minute attack, to a few hundred dollars for twenty-four hours. Portals known as "booter" portals offer the BotNets for hire.

Yet another way attackers can make money by performing an attack is by hijacking personal information like credit cards and social security numbers during a DoS attack. For instance, if attackers target a payment provider with a DoS attack, they might be able to take advantage of the broken system to exploit a vulnerability that isn't available under normal circumstances.

It may also be the case that instead of *making* money from a DoS attack, attackers want to cause their targets monetary losses. For instance, if you write an API that sends SMS messages using a service, you might have to pay a few cents per message sent. If attackers flood your service with a hundred thousand API calls that trigger those SMS messages, the financial loss could be severe.

Also, if an attacker knows that a service provider pays for inbound bandwidth, it might just be a matter of sending the service far more (or larger) requests than normal to eat up available bandwidth resources, causing the victim's cloud provider to send them a hefty

monthly bill. Even a few calls made to a particularly data-heavy service can increase a company's cloud-based bandwidth costs. For instance, if a website hosts many large video files, repeatedly downloading massive video files will quickly drive up the provider's bandwidth costs.

Finally, an attack might be carried out for political reasons. This type of attack is akin to a political protest where protesters might block the door of a business that offends their political sensibilities. In 2011, a group known as the Syrian Electronic Army *(https://www.cnn.com/2013/04/24/tech/syrian-electronic-army/ index.html)* worked to support the government of Syrian President Bashar al-Assad by targeting political opposition groups. In October of last year, the Czech Parliamentary election was the target of an attack *(https://sputniknews.com/europe/201710231058456317-czech-election-hit-cyberattack/)* meant to disrupt the counting of votes. Given all these motivations, it's easy to see why so many API services come under attack.

What are the Different Types of Denial of Service Attacks?

There are three main types of DoS attacks:

1. Application-layer Flood

In this attack type, an attacker simply floods the service with requests from a spoofed IP address in an attempt to slow or crash the service, illustrated in Figure 2-1. This could take the form of millions of requests per second or a few thousand requests to a particularly resource-intensive service that eat up resources until the service is unable to continue processing the requests.
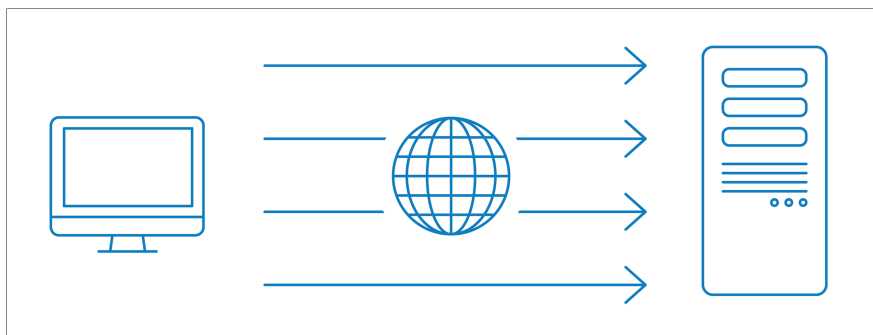
Figure 2-1: An attacker floods the service from a single IP address

Preventing application-layer DoS attacks can be tricky. The best way to help mitigate these types of attacks is to outsource pattern detection and IP filtering to a third party (discussed later).

2. Distributed Denial of Service Attacks (DDoS)

Distributed Denial of Service (DDoS) attacks occur in much the same way as DoS attacks except that requests are sent from many clients as opposed to just one, illustrated in Figure 2-2. DDoS attacks often involve many "zombie" machines (machines that have been previously compromised and are being controlled by attackers). These "zombie" machines then send massive amounts of requests to a service to disable it.

DDoS attacks are famously hard to mitigate, which is why outsourcing network filtering to a third party is the recommended approach. We'll cover this later on.

3. Unintended Denial of Service Attacks

Not all DoS attacks are nefarious. The third attack type is the "unintended" Denial of Service attack. The canonical example of an unintended DDoS is called "The Slashdot Effect *(https://hup.hu/old/ stuff/slashdotted/SlashDotEffect.html)*". Slashdot is an internet news site where anyone can post news stories and link to other sites. If a

linked story becomes popular, it can cause millions of users to visit the site overloading the site with requests. If the site isn't built to handle that kind of load, the increased traffic can slow or even crash the linked site. Reddit and "The Reddit Hug of Death *(https://thenextweb.com/socialmedia/2012/01/17/how-reddit-turned-one-congressional-candidates-campaign-upside-down/)*" is another excellent example of an unintentional DoS.
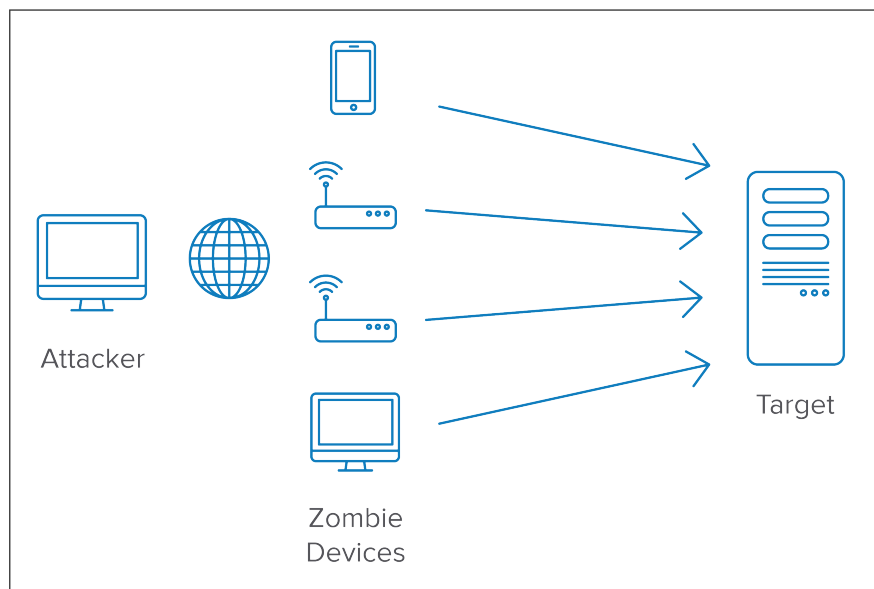
Figure 2-2: An attacker uses zombie machines to launch a DDoS against the target

The only way to prevent these types of unintended DoS attacks is to architect your application for scale. Use patterns like edge-caching with CDNs, HTTP caching headers, auto-scaling groups, and other methods to ensure that even when you receive a large amount of burst-traffic, your site will not go down.

Another type of unintentional DoS attack can occur when servicing low bandwidth areas. For instance, streaming content internationally means that people in certain areas of the world with slow or bad internet connections might cause problems. When your

service attempts to send information to these low-bandwidth areas, packets drop. In an attempt to get the information to the destination, your service will attempt to resend all dropped packets. If the connection drops the packets again, your service may make another attempt. This cycle can cause your service's load to double or triple, causing your service to be slow or unreachable for everyone.

How to Mitigate DoS Attacks

Now that you know what DoS attacks are and why attackers perform them, let's discuss how you can protect yourself and your services. Most common mitigation techniques work by detecting illegitimate traffic and blocking it at the routing level, managing and analyzing the bandwidth of the services, and being mindful when architecting your APIs, so they're able to handle large amounts of traffic.

Attack Detection

The first step of any mitigation strategy is understanding when you are the target of a DoS attack. Analyzing incoming traffic and determining whether or not it's legitimate is the first step in keeping your service available and responsive. Scalable cloud service providers are great (and may even "absorb" a DoS attack transparently) which is fantastic until you receive an enormous bill for bandwidth or resource overuse. Making sure your cloud provider makes scaling decisions based only on legitimate traffic is the best way to ensure your company is not spending unnecessary elasticity dollars due to an attack. Early detection of an attack dramatically increases the efficacy of any mitigation strategy.

IP Whitelisting/Blacklisting

The simplest defense against a DoS attack is either whitelisting only legitimate IP addresses or blocking ones from known attackers. For instance, if the application is meant to be used only by employees of a specific company, a hardware or software rule could be created to disallow any traffic, not from a specific IP range. For example, 192.168.0.0/16 would allow any IP address between 192.168.0.0 and 192.168.255.255. The rule rejects any IP address outside that range. If the software is only meant to be used by US citizens, a rule could be created only to allow access to US IP addresses. Inversely, IP blacklisting adds a rule to reject traffic from specific IP addresses or IP ranges making it possible to create rules to disallow traffic coming from China or Russia.

It is important to remember that blocking IP addresses in this way may prevent legitimate traffic from those countries. Blacklisting IP addresses is also dangerous in that you may end up blacklisting all users sharing an IP address, even if many of those users are legitimate. For example, what would happen if a bad actor used an Amazon EC2 server instance to attack a host and that host blocked all Amazon EC2 IP addresses? While the attack might stop, all legitimate Amazon users are now blacklisted from accessing the service.

Also, this strategy may not be effective against DDoS attacks or DoS attacks using spoofed IP addresses. In the distributed scenario, there may be zombie computers with IP addresses all over the place. Creating a rule to filter them out may become complicated and untenable. For instance, if an attacker is generating many requests to your service using a single spoofed IP address, when you block that address the attacker can start spoofing a new IP address and continue the attack.

Rate Limiting

Rate limiting is the practice of limiting the amount of traffic available to a specific Network Interface Controller (NIC). It can be

done at the hardware or software level to mitigate the chances of falling victim to a DoS attack. At the hardware level, switches and routers usually have some degree of rate-limiting capabilities. At the software level, it's essential to have a limit on the number of concurrent calls available to a specific customer. Giving users strictly defined limits on concurrent requests or total requests over a given duration (50 requests per minute) can be an excellent way to reject traffic and maintain service stability. The rate limit is usually tied to the customer's plan or payment level. For example, customers on a free plan may only get 1,000 API calls, whereas customers at the premium level may get 10,000 API calls. Once the user reaches their rate limit, the service returns an HTTP status code indicating "too many requests" (status code 429).

While rate limiting is useful, depending on it alone is not enough. Using a router's rate limiting features means that requests will still reach the router. Even the best routers can be overwhelmed and DoSed. At the software level, requests still need to reach your service even if a rate-limit has been reached to serve up a 429 status code. This means that your service could still be overwhelmed by requests, even if your service is only returning an error status code.

Upstream Filtering and DDS

One of the best mitigation strategies is to filter requests upstream, long before it reaches the target network. Done effectively, your API never even sees this traffic, so any rate limiting policies are not triggered. There are many providers of "Mitigation Centers" that will filter the incoming network traffic. For example Amazon Shield *(https://aws.amazon.com/shield/)* and Cloudflare *(https://www.cloudflare.com/)* both offer products that allow for protection against DoS and DDoS attacks by checking incoming packet IPs against known attackers and BotNets and attempt to only forward legitimate traffic. Various API gateways have the same capabilities but can also filter based on the requested endpoint, allowed HTTP verbs, or even a combination of verbs and endpoints.

Passing DoS mitigation responsibility to upstream providers can be a great way to reduce liability and risk as mitigation can be incredibly complex and is an ever-changing cat-and-mouse game between service providers and attackers.

These companies typically offer support should your service be currently under attack in an attempt to minimize damages. It then becomes the responsibility of the provider to keep abreast of new DDoS attack vectors and strategies, leaving you to focus on building your service.

Programming for Scale

With the proliferation of easily-scalable cloud services, it's easy to become lazy and not think about efficient development patterns. Sometimes it's easy to spot DoS-vulnerable parts of your application while other times it's not so apparent. It's vital to offload resource-intensive processes to systems that are designed to handle those operations. In some cases, you may even be able to queue expensive work for later batch processing, reducing DoS attack surface area. For instance, uploading or encoding images or video can take a lot of processing power, and it's essential that your application is not affected by those processes. In some cases, a well-configured cache — at the network or application level — can return data previously processed and unchanged. After all, the fastest processing possible is the processing you don't have to perform.

Sometimes, when a startup is first creating their product, the team pays less attention to performance and more attention to shipping features. While this can be okay early on, as a service becomes popular, it's hard to go back and fix performance issues before they cause a widened surface area for attackers. It's good practice to make performance testing part of the development cycle and continuous integration process. By running the Apache Bench command *(https://httpd.apache.org/docs/2.4/programs/ab.html)*, you can get basic performance information about your service. You can also use AB to write automated tests that simulate many users and check that your service responds to requests within a specified time.

These performance tests can be run during the continuous integration process to ensure the application code performs at a level that is satisfactory to your organization.

Parting Shots

API services are becoming a more and more critical part of the overall world economy, and attacks on API services is on the rise. Whether your services are targeted for fun, profit, or political reasons, it's important to know how to protect your assets (and those of your customers). Mitigating the technological and economic effects of a DoS on your internet-based resources is critical to the success of your company's platform.

Using techniques like IP whitelisting and blacklisting, upstream filtering, rate limiting, and good programming practices is your best defense against would-be attackers. It's important to keep abreast of changes in the security landscape and to make performance and load testing a part of your everyday software delivery practices. Whenever possible, offload the responsibility to companies that specialize in those practices so that you can focus on delivering value to your customers.

Overall, build great stuff… and be careful out there!

Chapter 3

Sanitizing Data

By Brian Demers

The inputs to your application represent the most significant surface area of attack for any application. Does your API power forms for user input? Do you display data that didn't originate in your API? Do users upload files through your API?

Any time data crosses a trust boundary - the boundary between any two systems - it should be validated and handled with care. For example, a trust boundary would be any input from an HTTP request, data returned from a database, or calls to remote APIs.

Let's start with a simple example: a user submission to the popular internet forum, Reddit. A user could try to include a malicious string in a comment such as:

```
<img src onerror='alert("haxor")'>
```

If this were rendered as is, in an HTML page, it would pop up an annoying message to the user. However, to get around this, when Reddit displays the text to the user, it is escaped:

```
&lt;img src onerror=&#39;alert(&quot;haxor&quot;)&#39;&gt;
```

which will make the comment appear as visible text instead of HTML, as shown in Figure 3-1.

Figure 3-1: Reddit properly escapes user input

In this example the trust boundary is obvious as any user input should not be trusted.

There are a few different approaches you can use when validating input:

- Accept known good
- Reject bad
- Sanitize
- Do nothing

Accept Known Good

The known good strategy is often the easiest and most foolproof of the given options. With this approach each input is validated against an expected type and format:

- Data type, (Integers are Integers, booleans are booleans, etc)
- Numeric values fall within an expected range (for example: a person's age is always greater than 0 and less than 150)

- Field length is checked
- Specially formatted string fields such as zipcode, phone number, and social security number are valid

Most web frameworks have some type of declarative support to validate input fields built in. For example, in the Node.js world you can use the popular `validator` package to validate different types of input:

```
import validator from 'validator';
validator.isEmail('foobar@example.com');
```

Reject Bad Inputs

Rejecting known invalid inputs is more complicated than only accepting known good inputs (which we talked about above) and far less accurate. This strategy is typically implemented as a blacklist of strings or patterns. This technique may require many regular expressions to be run against a single field which may also affect the speed of your application. It also means that this blacklist will require updates any time a new pattern needs to be blocked.

Take a typical use-case of blocking 'bad-words'. This problem is incredibly complex as language usage varies across locale. These complexities can be demonstrated using the simple word: `ass`. It would be pretty easy to block this word alone, but doing so would also block the proper use of the word referring to donkeys. Then you need to think about both variations of the word and where those letters happen to come together: 'badass,' 'hard-ass,' 'amass,' 'bagasse', the first two are questionable while the second two are fine. Even if you block all of these and the thousands of other words that contain these three letters, there are still variations that would make it past: '4ss', 'as.s,' 'azz,' '@ss,' 'āss,' or '\41\73\73' (escaped characters). As time goes on the list of blocked words would increase raising the total cost of the solution.

Another famous example of this technique is antivirus software. Your antivirus updates every few days to get a new blacklist of items to scan. And we all know how well that works ;)

Sanitize Inputs

Sanitizing inputs can be a good option when the input format is not strict but still somewhat predictable, such as phone numbers or other free-text fields. There are a few different ways to sanitize inputs, you could use a whitelist, a blacklist, or escape input.

Sanitize Input Using a Whitelist

When sanitizing data with a whitelist, only valid characters/strings matching a given pattern are kept. For example, when validating a phone number there are multiple formats people use, US phone numbers could be written as `555-123-1245`, `(555) 123-1245`, `555.123.1245`, or a similar combination. Running any of these through a whitelist that only allows numeric characters would leave `5551231245`.

Sanitize Input Using a Blacklist

A blacklist, of course, is the exact opposite of a whitelist. A blacklist can be used to strip HTML `<script>` tags or other non-conforming text from inputs before using input values. This technique suffers from the same shortcomings of the above section, Rejecting Bad Inputs on page 37. This type of sanitization must be done recursively until the value no longer changes. For example if the value `<scr`**`<script`**`ipt foo bar` is only processed once the result would be still contain `<script`, but if done recursively, the result would be `foo bar`.

Sanitize Input Using Escaping

Escaping input is one of the easiest and best ways to deal with free-form text. Essentially, instead of trying to determine the parts of the input that are safe (as with the above strategies), you assume the input is unsafe. There are a few different ways to encode strings depending on how the value is used:

HTML/XML Encoding

Example Input:
```
<img src onerror='alert("haxor")'>
```

Result:
```
&lt;img src onerror=&#39;alert(&quot;haxor&quot;)&#39;&gt;
```

HTML/XML Attribute Encoding

Example Input:
```
<div attr="" injectedAttr="a value here"><div attr="">
```

Result:
```
<div attr="&quot; injectedAttr&#61;&quot;a value
 here&quot;&gt;&lt;div attr=&quot;">
```

JSON Encoding

Example Input:
```
{"key": "", anotherKey": "anotherValue"}
```

Result:
```
{"key": "\", anotherKey\": \"anotherValue\""}
```

Base64 Encoding

Example Input:
```
any random string or binary data
```

Result:
```
YW55IHJhbmRvbSBzdHJpbmcgb3IgYmluYXJ5IGRhdGE=
```

There are ways to escape just about any format you need SQL, CSV, LDAP, etc.

Do Nothing

The last type of input validation is the no-op. Along with being the easiest to implement it is the most dangerous and most strongly discouraged! Almost every application takes input from an untrusted source. Not validating inputs puts your application and users at risk.

Common Attacks

The examples in this chapter have discussed ways to validate inputs but have only hinted at the type of attacks used when inputs are not properly sanitized. Let's look at those potential attacks, and how to prevent them, now.

SQL Injection Attacks

SQL injection is by far the most common form of data sanitization attack, and remains number one in the OWASP Top 10 *(https://www.owasp.org/images/7/72/ OWASP_Top_10-2017_%28en%29.pdf.pdf)* (a popular list of the most commonly found and exploited software vulnerabilities). It's held the number one spot for over 10 years now.

SQL injection occurs when an attacker is able to query or modify a database due to poor input sanitization. Other query injection attacks are similar, as most are typically a result of string concatenation. In the following example, a simple user query string is built with concatenation.

```
userId = getFromInput("userId");
sql = "SELECT * FROM Users WHERE UserId = " + userId;
```

If the `userId` were `jcoder` the SQL query would be `"SELECT * FROM Users WHERE UserId = jcoder`, however, a malicious attacker might input `jcoder; DROP TABLE ImportantStuff` which would result in two statements being executed:

```
SELECT * FROM Users WHERE UserId = jcoder;
DROP TABLE ImportantStuff
```

Similarly, the user could enter `jcoder OR 1=1` which would query for a user with the ID of `jcoder` OR `true` (`1=1` is always true), this would return all of the users in the system.

The cause of this issue is the use of poor string concatenation. In the example above, the value of the `userId` input crosses a trust boundary and ends up getting executed. The best way around this is to use SQL prepared statements. The syntax for using prepared statements varies from language to language but the gist is that the above query would become `SELECT * FROM Users WHERE UserId = ?`. The question mark would be replaced with the input value and be evaluated as a string instead of changing the query itself.

Most web frameworks and ORM libraries provide tools to protect against SQL injection attacks, be sure to look through your developer library documentation to ensure you're using these tools properly.

XSS - Cross Site Scripting

A cross-site scripting attack (XSS) is an attack that executes code in a web page viewed by a user. There are three different types of XSS attacks:

- **Stored XSS** - A persisted (in a database, log, etc) payload is rendered to an HTML page. For example, content on an forum.
- **Reflected XSS** - Attack payload is submitted by a user, the rendered server response contains the executed code. This differs from Stored XSS where as the attack payload is not persisted, but instead delivered as part of the request, eg. a

link: `http://example.com/`
`login?userId=<script>alert(document.cookie)</script>`
- **DOM based XSS** - The attack payload is executed as the result of an HTML page's DOM changing. With DOM based XSS the attack payload may not leave the victim's browser. The client side Javascript is exploited.

There are tons of resources online *(https://www.owasp.org/index.php/Cross-site_Scripting_%28XSS%29)* that cover this topic in great detail, so I'll only provide a basic example here. Earlier in this chapter the string `<img src onerror='alert("haxor")'>` was posted as a Reddit comment. If this string isn't correctly escaped it would have resulted in an annoying popup, shown in Figure 3-2.
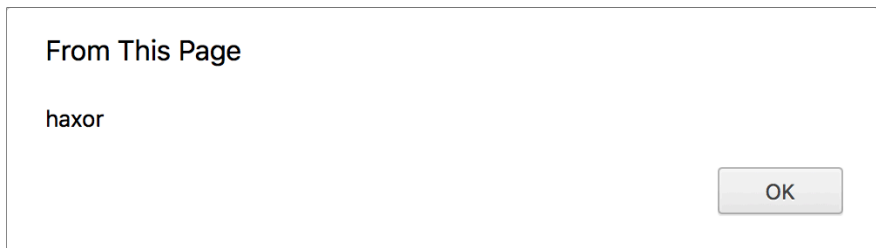
Figure 3-2: A JavaScript alert popup

You may see `alert()` used throughout examples when describing these attacks. The idea is if you can cause an alert to happen in the browser, you can execute other code that does something more malicious like sends your information (cookie, session ID, or other personal info) to a remote site.

File upload attacks

It is common for sites to support file uploads, particularly images such as profile avatars or photos. When uploading files, it is necessary to validate the type, size, and contents of these files. For example, if a user is uploading an avatar image, it's important to ensure the newly uploaded file is actually an image.

If an attacker can upload a PHP file named `avatar.php` instead of an image file, then later retrieve the file, unexpected and disastrous behavior may occur. Imagine what would happen if that file is executed on the server, you could have a remote code exploit on your hands. There are a few things you can do to prevent this type of attack:

Validate expected file types Check that file size is reasonable (if someone is uploading a 1GB image, you might have a problem) If storing the file to disk, do NOT use a user input field as part of the file name, eg: `../../../etc/config.file` Always serve the files with the correct Content-Type header (image/png, audio/mpeg) Run a virus scan on all uploaded files Do not allow uploads of web executed files: php, cgi, js, swf, etc. Process the files - rename, resize, remove exif data, etc - before displaying back to the user

Look For Other Attack Vectors

Inputs are everywhere, often only evident in hindsight. User input and file uploads are just the tip of the iceberg, but what if we consider more than input and instead the code itself? Here are a couple of examples to illustrate this point.

Your Dependencies

Do you trust all of your dependencies? How about all of the transitive dependencies of your application? It is not uncommon to for an application to have a page that lists its dependencies versions and licenses (the later might even be required depending on the license). The popular Node package manager (npm) has had a few projects which have contained maliciously formed license fields *(https://blog.npmjs.org/post/80277229932/newly-paranoid-maintainers)*. In another npm incident, packages ran malicious scripts *(https://iamakulov.com/notes/npm-malicious-packages/)* upon installation automatically that uploaded the user's environment variables to a third party.

Every dependency is code you include from other systems across your trust boundary. Properly inspecting and validating your dependencies is a critical first step of any input sanitation plan. GitHub recently introduced automated security alerting *(https://blog.github.com/2017-11-16-introducing-security-alerts-on-github/)* to let you know when your dependencies might have security issues. Pay attention to these and you can prevent a lot of headaches.

Inbound HTML Requests

Almost all values from an HTTP request can be changed by the sender and need to be handled accordingly. To help illustrate this, here is a simple HTTP POST including numerous headers to `http://example.com/submit-me`:

```
POST /submit-me HTTP/1.1
Host: example.com
Accept: */*
Referer: http://example.com/fake.html
Accept-Language: en-us
Content-Type: application/x-www-form-urlencoded
Accept-Encoding: gzip, deflate
User-Agent: My Fake UserAgent <img src onerror='alert("haxor")'>
Content-Length: 37
Connection: Keep-Alive
Cache-Control: no-cache

foo=bar&key=value
```

You can see right away: request headers are user input too. Imagine for a moment that an HTTP client maliciously changes the User-Agent header. The logged User-Agent may falsely identify a request as coming from a different client application than the one in which it really originated.originated from. While that's unlikely to affect the current request, it might cause confusion in the application's logging and reporting system.

Further, the User-Agent could be visible from an internal web application that doesn't sanitize the User-Agent values before displaying them. In this case, an HTTP client could maliciously modify their User-Agent to any JavaScript code they want which would then be executed in an internal user's browser via XSS.

As these examples illustrate, even sanitizing relatively innocuous inputs is an important part of an overall security strategy.

Best Practices for Secure Data

While this chapter provides an overview of a few common types of attacks; there are many more out there.

First, you don't need to be an expert to prevent these attacks, but you do need to have some knowledge of them. The Open Web Application Security Project at *OWASP.org* is a great source information and examples on how to secure your application, often in multiple programming languages.

One of the most straightforward means of prevention is not to reinvent the wheel, and use an existing framework. Most frameworks contain tools to properly escape values, both on the frontend and backend, when used correctly.

Next, don't forget to monitor your application dependencies. There are mailing lists as well as open source and commercial tools to help you. New CVEs (Common Vulnerabilities and Exposures) are reported all of the time. For example, at the time of this writing a popular Java Web Container, Apache Tomcat 8, has about 60 CVEs *(https://tomcat.apache.org/security-8.html)* reported (and fixed). These reports, and the subsequent releases indicate that the project takes security seriously and updates regularly.

And finally, trust no one! As you have seen, any input into your API is an attack vector. Everything from an HTTP request to data returned from a database query to the files user upload could be

dangerous. Proper data validation and sanitization goes a long way to help mitigate risk.

Chapter 4

Managing API Credentials

By Joël Franusic

A critical part of designing an API is determining how to grant users access to sensitive or important parts of it. While many APIs have publicly accessible endpoints that don't require authentication, the vast majority of APIs require a user to authenticate before any request can be fulfilled.

You need to authenticate with Stripe to charge a credit card, you have to authenticate with Twilio to send an SMS, and you have to authenticate with SendGrid to send an email. There's really no way to avoid authentication.

This chapter will cover two main aspects of managing API tokens:

- Protecting tokens that you use to connect to other APIs
- Advice and suggestions for what sort of API token to use for an API that you are building

Keep Your Credentials Private

No matter whether you are using an API or building your own, the advice applies to you: Never put an API secret into your code. Never ever!

The biggest problem with storing credentials in source code is that once a credential is in source code it's exposed to any developers who have access to that code, including your version control provider.

Many people use GitHub to host their source code — what happens when you push company code containing sensitive credentials? That means GitHub staff can now view your credentials; it's a security risk. If that project is later shared with contractors, partners, customers, or even made public, your secret is no longer secret. The same is true for open source projects — accidentally publishing test API tokens or other sensitive data will cause enormous problems. There are attackers who scan GitHub commits looking for sensitive data like Amazon Web Services API tokens and then use these credentials to do things like mine cryptocurrencies, form botnets, and enable fraud.

If you can't store your credentials in source code, how should you authenticate to your databases or 3rd party APIs? The short answer is to use environment variables instead. What that means is ignoring what you see in sample code (where secrets are entered directly into source) and instead loading secrets from environment variables which are managed from outside your source code and will not be stored in version control and every team member's text editor.

Below are two short snippets of sample code that demonstrate how important it is to use environment variables to store credentials.

Below we have some example Python code from Twilio. In this example, the `AC01a2bcd3e4fg567h89012i34jklmnop5` string is the "username" or "account ID" and the the `01234567890a12b34c567890de123fg4` string is the API token.

```
from twilio.rest import Client

account_sid = "AC01a2bcd3e4fg567h89012i34jklmnop5"
auth_token = "01234567890a12b34c567890de123fg4"
client = Client(account_sid, auth_token)
```

As you can see above, these API credentials are hard-coded into the program's source. This is a big no-no. Instead, as an example of what you *should* do, take a look at this example code from SendGrid, which uses the "os.environ.get" method in Python to grab the SendGrid API token from the SENDGRID_API_KEY environment variable.

```
import os
import sendgrid
from sendgrid.helpers.mail import *

apikey=os.environ.get('SENDGRID_API_KEY')
sg = sendgrid.SendGridAPIClient(apikey)
```

By pulling sensitive credentials from environment variables, which can be managed by configuration management and secret management software, your application code will remain credential-free.

Using environment variables to store secrets is an incredibly important first step to take in securing your code.

"What Kind of API Token Should I Use?"

Now that we've covered general advice for storing and using API tokens, let's review common options for securing your API. We'll cover the different types of API tokens, touch on the advantages and disadvantages of each, and summarize with suggestions and a recommended approach.

Secure Your API

First, let's dig into some best practices:

- If your API is intended to be used to support an end user application, secure it using OAuth 2.0 to act as a Resource Server (RS) as defined by the OAuth 2.0 specification.
- If your API is intended to be used as a service by other

software, secure your API using an "API token" - a string that is unique for each client.

The first approach, implementing an OAuth 2.0 Resource Server, is intended to be used when an application is acting on behalf of the person using the app. The Instagram and Spotify APIs are great examples where every action of the API needs the user's context to make sense.

The second approach, using an API token, is the best approach for automated software. Stripe, Twilio, and SendGrid are examples of this type of an API. While you certainly can, and eventually should consider, implementing OAuth 2.0 access tokens, doing so may be more overhead than telling your users to just use an API token.

Regardless of approach, the following patterns apply:

- Use the `Authorization: Bearer` header for authenticating to your API. The single most important reason is that URL parameters are captured in server access logs and caches. Any "secret" included as a parameter is no longer a secret. As a standard defined by RFC 6750, most HTTP clients have built-in methods for using Bearer tokens.
- Further, instead of using a simple unique string, use a JSON Web Token or JWT as the Bearer token. By using JWT and the claims defined by RFC 7519, you can support a wide range of scenarios using just one authentication method. And, since JWT is such a widely adopted standard, most programming languages and frameworks have first-class JWT decoding and validation built in. You get more flexibility and wider compatibility with less work!

Another important thing to keep in mind is that by using a JWT as a Bearer token, you can support both token types that we describe above. A JWT works equally well as an OAuth 2.0 access token or as an "opaque" generic API token. You could start by just offering authenticating with an API token, but later add support for OAuth 2.0 and give your customers the features that OAuth 2.0 easily

enables like automatic rotation of access tokens. You and your users get a clear, simple upgrade path for the life of your API.

Other Options for Authentication to Your API Service

Now, while I suggest that you have your clients authenticate to your API using JWTs as Bearer tokens, there are other patterns you will encounter. Below is a short list of other approaches to API authentication. Keep in mind that this is by no means an exhaustive list, just a list of the most common approaches and their tradeoffs:

Basic Authentication

This is the good 'ole "username and password" form of authentication method. Some APIs will use other words for "username" and "password" for example, Twilio calls the "username" the "Account SID" and the "password" the "Auth Token" but it works exactly the same.

If you decide to use Basic Auth to secure your API, keep in mind that your "username" and "password" should be random strings and not the same as the account username and password. You can generate these values by using an entropy source from your operating system (`/dev/random` on Unix-like systems or CryptGenRandom on Windows systems)

Opaque Tokens

These were the most common type of API token for APIs designed before OAuth 2.0 was standardized. Companies that use these types of token to secure their APIs include Stripe (with tokens that look like this: `sk_test_ABcdefGHiJkL0MnOpQ1rstU2`), and Okta (with tokens that look like this `00aBCdE0FGHijklmNO1pQ2RStuvWx34Y5z67ABCDEf`). These tokens have

no relationship with the account information and mean absolutely nothing outside those systems.

As with basic auth strings, we suggest that you generate opaque tokens using an entropy source from your operating system.

Another best practice for opaque tokens is to allow multiple tokens to be issued and used at the same time, as this will allow for key rotation.

Signed or Hashed Tokens

These tokens are cryptographically signed or hashed and can either be opaque tokens or contain carry information about themselves. One reason to use a signed or hashed token is to allow your API to validate tokens without the need for a database lookup. The best supported token type in this category is the JWT used for OpenID Connect. Other examples of tokens in this category include PASETO and Hawk. In general, we suggest using JWTs as described above.

Advanced API Token Considerations

Using OAuth 2.0 with OIDC, or just a JWT as a Bearer token is a significant milestone in the ongoing task of keeping your API secure. Depending on use case, type of data, and type of operations your API provides, you may need to consider additional steps to secure your API. Keep in mind that these extra considerations might not be appropriate for every kind of API. If your API is a fan-made API that gives programmatic access to Star Wars data (like swapi.co), simple API keys are probably sufficient. However, if your API deals with things in the real world or any form of sensitive data, you should consider the options below and choose the combination appropriate for your API and fits your compliance requirements.

Implement API Token Rotation

A great next step to take in securing an API is to rotate the API token automatically. Like passwords, regularly changing an API token will limit the damage a leaked or misplaced API token can cause. More importantly, by considering and implementing this from the beginning, if a token is leaked or when an employee leaves the team, you have a process for quickly responding and protecting your systems.

Additionally, one of the great side-effects of frequent API token rotation is that it forces best security practices. Sometimes, when a team is in a rush to deliver a critical feature, corners get cut and hard coding an API token instead of storing it properly may save a few minutes in the short term. If you rotate tokens on a regular basis, developers have to follow the rules, otherwise their code will stop working on the next rotation.

If you are using OAuth 2.0 to secure your API, token rotation is built-in to the OAuth 2.0 standard: An "access_token" always has a limited lifespan and must be rotated periodically using the "refresh_token". As an additional benefit, if you're using an OAuth server such as Okta, when you exchange the refresh_token for a new access_token, your authorization policies are re-evaluated. If a user's API access has been limited, increased, or even revoked, your application will know.

Outside of OAuth 2.0, there isn't an accepted best practice for implementing token rotation. Therefore your best and easiest option is to implement OAuth 2.0. Once you have a system in place to manage your API tokens, it makes sense to start rotating API tokens on a regular basis. Your specific rotation schedule will depend on the use case. For read/write operations in banking or healthcare, rotating every 5 or 10 minutes might be necessary. For read only access to a public Twitter feed, annually is probably sufficient. Regardless, you should always rotate keys after an employee leaves the team to protect against accidental or intentional misuse of API tokens by former employees.

Ideally, key rotation should also be paired with configuring your API to log events into a "Security and Information and Event Management" (SIEM) system that you can use to monitor your API for suspicious events.

Monitor for Token Leaks

In addition to the use of SIEM systems as suggested above, an advanced technique is to scan sites like GitHub and S3 for leaked API keys. No best practices have emerged in this area yet, but a good technique should include automatically disabling and notifying end users when a token has been discovered in public as part of a scan.

Quite a few open source projects can be found that will scan for leaked tokens, a good way to find these services is to search for "github credential scan"

Bind Tokens to TLS Sessions

Finally, an interesting emerging technique that I'm keeping my eye on is the binding of tokens to TLS sessions. This technique is described in RFC 5056 *(https://tools.ietf.org/html/rfc5056)* and RFC 5929 *(https://tools.ietf.org/html/rfc5929)*.

The basic idea with "channel binding" is to tie an API token to a specific TLS session. In practice this would mean writing your API to issue tokens that can only be used in the same TLS session. This way, if an API token is compromised from a client, an attacker can't move that token to another client or machine because they would have a different TLS session for the initial issuer. This still isn't foolproof but the work and effort for the attacker just multiplied.

Key Takeaways for Managing API Credentials

In closing, here is my best advice for managing API credentials:

- Never paste a secret into your code. Never ever!
- Secure your API using OAuth 2.0 by writing your API to act as an OAuth 2.0 "Resource Server"
- Use JSON Web Tokens (JWT) as your tokens to embed additional context
- Use the token as a Bearer token with the Authorization header to prevent leaking your token in logs and caches
- Implement regular token rotation to reduce the damage from leaked keys, poor practices, honest mistakes, and disgruntled employees.
- Monitor your source code for token leaks
- Implement "channel binding" to tie your API tokens to the TLS session they are requested over

Chapter 4: Managing API Credentials

Chapter 5

Authentication

By Matt Raible

Chapter 6

Authorization

By Sai Maddali

Chapter 7

API Gateways

By Keith Casey

An API gateway is a firewall that sits between your API and your users. They range from the simplest proxies which apply throttling and white/blacklisting to fully configurable platforms with fine-grained access mapping individual permissions to specific HTTP verbs and endpoints. Realistically, using an API gateway is not necessary but it makes some things faster, easier, and more reliable, which allows you to focus on your API.

The most prominent gateways are Google's Apigee *(https://apigee.com/api-management/)*, Salesforce's MuleSoft *(https://www.mulesoft.com/)*, the AWS API Gateway *(https://aws.amazon.com/api-gateway/)*, Microsoft Azure's API Management *(https://azure.microsoft.com/en-us/services/api-management/)*, and the Kong API Gateway *(https://konghq.com/)* but the most appropriate gateway for your project will vary depending on context, use cases, and budget.

This section does not make a recommendation for a particular gateway but describes the process and use cases where one may fit.

Most API gateway vendors call themselves API management platforms because gateways are just one part of an overarching API management strategy. With that in mind, there are five key things that most API management platforms provide: Lifecycle

management, interface management, access management, consumption tracking, and business goals.

When you're building and deploying your API, you need to address each of these five areas, which is one of the main reasons API management platforms have taken off in recent years: they make solving these problems tangibly easier.

What Does an API Management Platform Manage?

Lifecycle Management

Effective API management begins long before your first HTTP request. In fact, it likely begins in a document or at a whiteboard with a simple requirement and business needs. It quickly turns into a specification and then a workflow and then a data and interaction model. Eventually, it probably turns into an Open API specification, deployable code, and metrics that are carefully tracked by the team.

The key technical aspect of this entire process is understanding what stage the API is in, how and where it's deployed, and how it's maintained. As a result, many of the gateways integrate with the cloud hosting services blend directly into your devops processes. Further, the process and these integrations are the same whether the APIs are destined for internal, partner, or even consumer use. In this area, that distinction is irrelevant.

Interface Management

While it's pedantic to note, the "I" in API stands for Interface, and it's the only interface our users will ever interact with. A full discussion of API design is beyond the scope of this book, but the result will always be individual URLs or endpoints we need to support, which HTTP verbs are applied and how, and what parameters or properties are required for each.

API management platforms do not help you choose specific endpoints or which verb is best. Instead, once you've made those decisions, the API gateway will allow you to map external URLs to specific endpoints in your API and whitelist specific HTTP verbs, parameters, and even datatypes each endpoint supports. As the first line of defense, this is one of the subtly powerful aspects of a gateway because it limits the surface area where your API can be attacked. That said, all the standard data sanitization practices still apply.

More advanced API management platforms go a step further. Instead of merely configuring the available endpoints via a web interface or even an API, you can upload your API specification document - like Open API, Blueprint, or RAML - and the gateway will parse the document and configure the external interface. You may even be able to integrate this with your continuous integration system to automatically deploy your API into staging, perform the appropriate checks, and prepare to deploy to production.

Access Management

Access management in an API management platform is where things begin to get more complex. Until this point, the platform has dealt with deeply technical challenges the solutions to which are expressed entirely in code. Access management - both authentication and authorization - is a combination of code, the context of the user and their use case, and business policies and practices. This makes it fundamentally more complicated.

At the simplest level, every gateway can use an API key that is checked on every request. While this works, it lacks the fine-grained access most security and compliance teams require.

As an alternative, many API gateways also include a basic OAuth 2.0 server. Your users perform a simple OAuth user flow, receive an access token, and can then use the API. There are two significant drawbacks to this approach. First, the gateway has to keep its own list of users and activate or deactivate them as employees come and

go. As yet another independent system, it's easy for this information to get out of sync. The second drawback is that security teams can't review, audit, and validate the security policies implemented by the gateway. For some organizations, this is troubling at best and catastrophic at worst.

The final approach gateways take is to provide a pluggable interface for an external Identity Provider (IdP). Using an IdP is an easy way to integrate user management with a more extensive system such as Active Directory or Okta's Universal Directory. The single biggest benefit is simplicity: users are activated or deactivated in the API gateway as they are activated or deactivated in the directory. For internal or employee-oriented scenarios, this resolves a major security requirement. Further, by centralizing the issuance of access tokens, the security team can audit and even control the policies independent of API development.

Consumption Tracking

As we move further away from the hard technical implementation and into the business concerns and requirements, the next area is consumption management, or more fundamentally onboarding and engagement. To expose these functions, most API management platforms include a developer portal for documentation and samples, a logging page to show usage and errors for debugging, and sample code to show how to use the API. Every component here has exactly one goal: How can a developer get started with and use your API successfully?

Business Goals

The final aspect that an API management platform addresses are the business goals. On a technical layer, this overlaps with the consumption aspects to track overall API usage but provides a more detailed look at business analytics. Therefore, it's not just overall API consumption but identifying which API calls are the most and least important and how they map to revenue. The most advanced

gateways will also include integration with web analytics platforms to track and understand where your users are struggling on setup and configuration.

What Problems does an API Management Platform Solve?

An API management platform or an API gateway makes basic security easier. With a well-configured gateway, you know exactly which endpoints are open to the world and what parameters they expect. You still have to filter and validate the input according to best practices, but the attack surface is a fraction of what it would be otherwise.

An API management platform handles traffic shaping. Bad actors will misuse and abuse our API. There will also be people who make honest mistakes and run an infinite loop, as well as customers who are really excited about the service. Any one of those can take down our API or drive costs astronomically high. Regardless of the reason, we need to be able to throttle and stop traffic before it hurts us and our customers.

An API management platform lets you worry about other problems. Most teams have enough problems designing, building, documenting, demonstrating, and marketing an API. When we can hand off essential components to reliable third parties, we have to consider it so we can do all the tasks unique to us.

Finally, an API management platform is excellent at logging. One of the biggest challenges for both your customers and your team is understanding "what happened?" The gateways will capture everything and most present it in a clear, consistent manner. A good debugger will save developers - both internal and external - hours of effort and frustration.

What Problems does a API Management Platform not Solve?

An API management platform does not design your API. You still need to understand your users, their goals, and the best way to accomplish those goals. That will require you to determine which use cases you are and are not solving with your API. Further, you have to decide the name and structure of appropriate endpoints and what is required to interact with them.

An API management platform is also not a universal security solution. While it limits your attack surface to pre-defined endpoints with specified parameters, those endpoints still need adhere to security practices regarding data filtering, rate limiting, authentication, and authorization. Various API breaches - such as Equifax and Panera - resulted from attackers using published endpoints in unexpected ways to download entire customer lists, transaction history, and complete credit reports. Rate limiting would have slowed these attacks and monitoring may have detected them, but only strongly defined and enforced authentication and authorization could have stopped them.

The success of your API is also not driven by your API management platform. From a technical perspective, your API still has to be stable and reliable. From a product/market fit perspective, your API still has to solve an important problem for a measurable customer base. And finally, from a user experience perspective, your customers need to find your API and be able to get started quickly enough to solve their problem.

And finally, an API management platform will not establish governance policies for you. When large companies begin an API program or begin to coordinate API efforts, you have to create and enforce policies for tracking APIs lifecycle and development, consistent and predictable naming of endpoints and parameters, understanding and applying security procedures, and publishing them for each audience. These are all leadership and management issues you need to consider in addition to API gateway.

Which API Management Platform is the Best?

This is a much more complex question and depends on your use case, budget, and familiarity. If your infrastructure is entirely on AWS, Azure, or Google Cloud, using their respective gateways is a safe choice.

If you are like most organizations and have various components and systems in various places, the decision becomes more challenging. Mulesoft has its roots in the Enterprise Service Bus (ESB) area, so it is ideal when you have to wrap existing systems and orchestrate components into a single interface. Apigee was born and lives entirely in the cloud so if your architecture is entirely in the cloud or you're just starting, it may be a better fit. Alternatively, Kong and Tyk.io are self-hosted open source gateways which will allow you to deploy them on nearly any architecture. If you're deep into microservices, they may be a better approach to embed directly into the microservice.

Regardless, the original constraints around access management don't go away. Having a central place to create, manage, audit, and deploy access and security policies is key to knowing your people and systems have the right access to the right systems for the right reasons for the right amount of time. Distributing that aspect across servers, teams, and codebases is confusing at best and catastrophic at worst.

About the Authors

Lee Brandt

Keith Casey

Brian Demers

Joël Franusic

Sai Maddali

Dave Nugent

Matt Raible

API Security

The OAuth 2.0 authorization framework has become the industry standard in providing secure access to web APIs. It allows users to grant external applications access to their data, such as profile data, photos, and email, without compromising security.

OAuth 2.0 Simplified is a guide to building an OAuth 2.0 server. Through high-level overviews, step-by-step instructions, and real-world examples, you will learn how to take advantage of the OAuth 2.0 framework while building a secure API.

Written by

Lee Brandt
Keith Casey
Brian Demers
Joël Franusic
Sai Maddali
Dave Nugent
Matt Raible

with a foreword by

Les Hazlewood

What you will learn from this book

- Discover how OAuth 2.0 facilitates authorization and authentication
- Understand how OAuth 2.0 and its extensions relate to each other
- Learn what it takes to build an OAuth server beyond what is written in the spec
- Learn how to securely handle OAuth in native and mobile apps
- Demystify how OAuth 2.0 relates to OpenID Connect

{okta}

developer.okta.com