THE PRICE OF HUBRIS: THE PERILS OF OVERESTIMATING THE SECURITY OF YOUR APIS

Summary

This eBook offers a cautionary tale to CISOs and other cybersecurity leaders who believe their APIs are secure when using the wrong security controls for the job. This is a culmination of the empirical data from all my API breaches over the past decade from APIs secured with the wrong control.

Author Information

Alissa Valentina Knight
Partner
Knight Ink
1980 Festival Plaza Drive
Suite 300
Las Vegas, NV 89135
ak@knightinkmedia.com

Publication Information

This eBook is sponsored by Traceable.

Initial Date of Publication: July 12, 2021
Revision: 2.1

TABLE OF CONTENTS

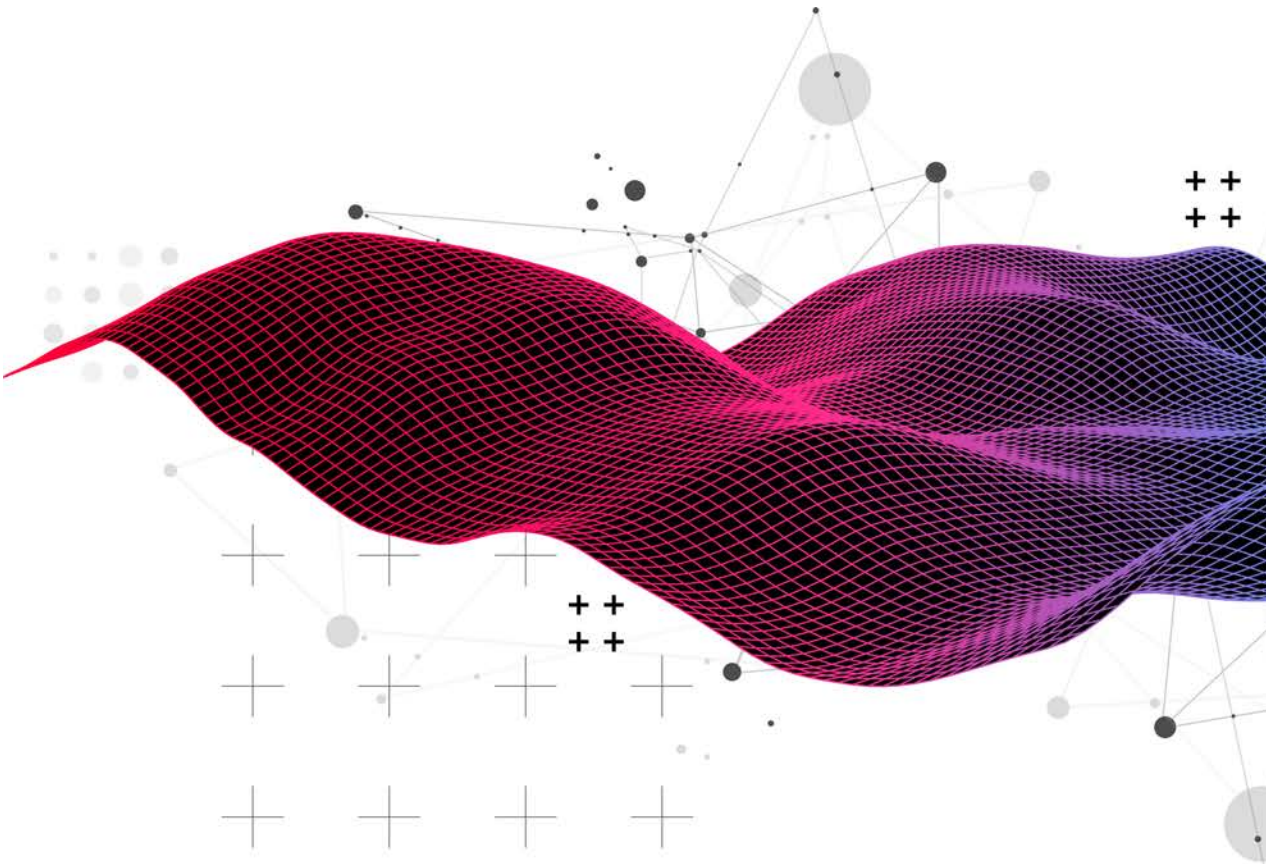


TABLE OF CONTENTS

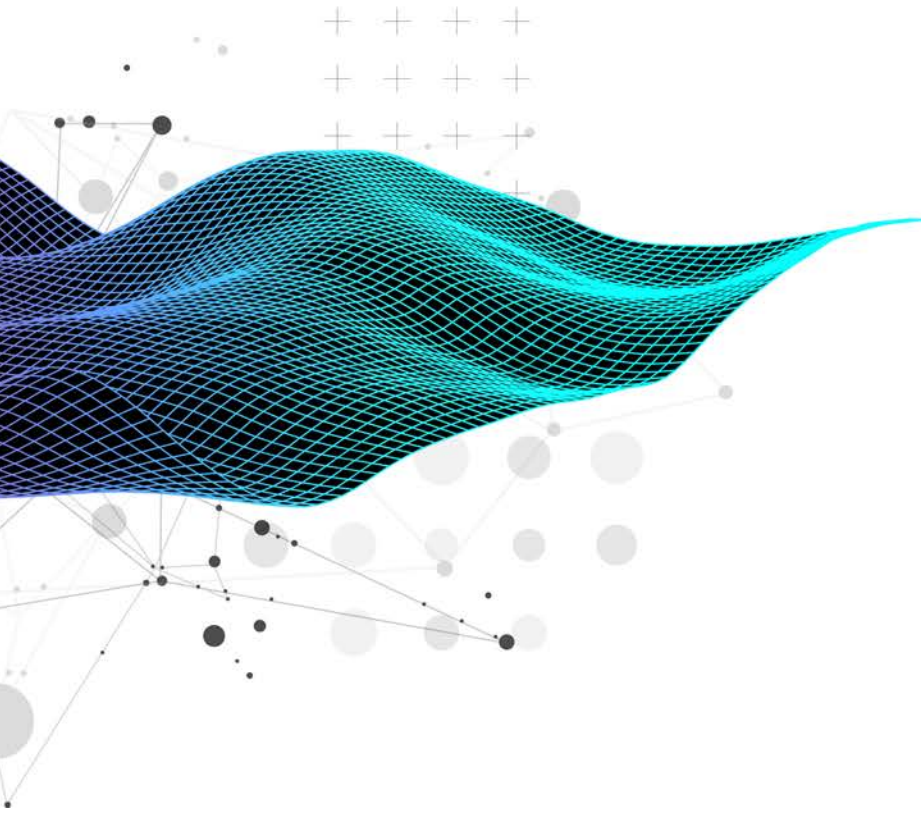


TABLE OF CONTENTS

TABLE OF CONTENTS

INTRODUCTION

"API calls represent 83 percent of web traffic, according to an October 2018 Akamai traffic review detailed in the report." (Akamai, 2019) This effectively means that more than half of the traffic on the internet flowing across one of the largest content delivery networks (CDNs) is no longer human-to-application traffic, but application-to-application traffic.

Web applications, mobile applications, and other services have become a critical cog in how businesses interact with customers, partners, and everyone and everything that they do business with. As such, the security of these applications and services have been a concern for CISOs and IT departments for many years now. And many technologies have been developed and put into production to help secure these important resources.

However, the technology has changed considerably, and with it, how these important resources need to be protected. If you feel secure because you have a WAF in place that doubles as a security defense for both your web servers and APIs then I'm here to tell you that you couldn't be more mistaken. There's a whole new attack surface created by your APIs that you either know are there and aren't adequately protecting, or worse yet, don't even know they're there. Not knowing carries with it huge repercussions when failing to manage the risks they introduce to your data loss prevention efforts.

Purpose of This eBook

CISOs and organizations are exposed and vulnerable to a range of ever expanding and growing API based attacks with each new breach highlighting more sophisticated tactics and techniques. This eBook is a call to action to CISOs to assess and address API vulnerabilities from my vantage point - someone recognized for hacking and securing APIs across dozens of API penetration tests over the past two decades in financial services, automotive, and healthcare.

Who Should Read This eBook

This eBook is for cybersecurity engineers, Chief Information Security Officers (CISOs) and other executive leadership charged with ensuring the confidentiality, integrity, and availability of your organization's data being served by your APIs.

If you answer yes to any of these questions, this eBook was written for you as well:

- I don't have an answer to the hard questions surrounding the security of my organization's APIs, such as how many we have, where they are, and what kind of data each API is serving;
- I've become overly enamored with my cybersecurity program and its resilience to cyberattacks;
- I don't know if my organization even has APIs to secure; or
- I simply secure my APIs with my existing web application firewalls, API management gateways, my CDN, or rely on tokens for security.

WHY YOU SHOULD LISTEN

I've performed dozens of penetration tests against APIs. Each successful breach of an API I've tested has been despite the implementation of authentication and authorization controls, indicating a systemic lack of shift-left and shield-right in API security. With some of the bank targets covered in this eBook, I was able to change the PIN codes of any bank customer and transfer money between accounts I didn't own. In healthcare I was able to login with a clinician account and access patient accounts that weren't assigned to me, or login as a patient and access the electronic health records (EHR) of other patients. In automotive, I was able to take remote control of any vehicle in the fleet by stopping and starting the engine or locking and unlocking the doors. All of these attacks were employed against the APIs the mobile app, IoT device, or web app communicated with.

The most common problem I find among leadership in many organizations is a false sense of security created by the misperception that their APIs are secure. Many of you are also leaning too heavily towards a shield-right security mentality whereupon the API isn't secured until it's put into production and skipping critical hardening efforts while the code is being written. Many organizations are using inadequate tools a la WAFs, API gateways, content delivery network (CDN) security features, or tokens without scopes to secure their APIs. And the rest of you have an over-reliance on shift-left security by implementing it as part of the software development life cycle (SDLC) and then failing to shield-right or shielding right with the wrong tool.

Over the course of my vulnerability research, I've discovered a common theme across the developers I spoke to of the mobile apps and APIs I've tested:

- There is an overall fear of security hardening breaking the application -- bricking their apps through a failure or misconfiguration in those security controls;
- Finger pointing at outsourced developers who wrote the code that they were told was secure (a lack of trust but verify);
- Third-party developers didn't allow the organization to perform penetration testing or review the code, leaving the organization blind to their API risks; and
- Many consultancies who offer outsourced app development didn't provide the client source code.

KEY TAKEAWAYS

This section outlines the salient points from this paper. While it's my hope you'll read this eBook in its entirety, this section attempts to summarize this eBook's key points and research findings but should not be considered all encompassing.

- Securing your APIs only with web application firewalls (WAFs) is a high-risk gamble if you consider it a panacea to your API security risks. You don't have the security you think you have with WAFs due to their inability to understand context and detect logic-based attacks, such as authentication and authorization vulnerabilities, mass assignment, excessive data exposure, and other novel threats facing APIs.
- Combining WAFs, tokens, and API gateways together in order to build a home-grown security stack still doesn't provide the comprehensive security an API threat management solution provides. Many of the APIs I've hacked that are covered in this eBook were "protected" by API gateways, WAFs, and tokens that failed to detect and prevent my attacks.
- Many API gateways are now adding security features and some buyers mistakenly conclude that they have sufficient API security when they don't, which my research has proven. Securing your APIs should be instrumented with purpose-built security solutions from the ground up rather than implementing security as a feature into a management product like an API gateway.
- Out of the mobile apps and APIs I've tested, the most prevalent issue I discovered in those API breaches was broken object level authorization (BOLA) vulnerabilities. My account was properly authenticated but I was able to access data that I wasn't authorized to see.
- In a close second to authorization vulnerabilities in my findings was broken authentication and excessive data exposure vulnerabilities.
- 100% of the mobile apps I've tested contained hard-coded API secrets, private keys, and credentials including API keys and tokens for third-party APIs, such as payment processors and cloud storage buckets.
- The most common issue found across my test targets was use of the wrong or inadequate security controls to secure their APIs, such as WAFs. In some cases, APIs weren't secured with any security controls at all, leaving security only to features offered by API management, only using OAuth2, or other forms of authentication.
- Many of the organizations I successfully breached had failed to shift left by hardening their code as it was being written. They didn't use static and dynamic code analysis, relying only on shield right security controls once the API was placed into production.

RECOMMENDATIONS

- Use in-app protection solutions to obfuscate the code and encrypt it with white box encryption to secure your mobile apps against reverse engineering, especially when API keys and tokens have been hard coded into the apps.
- Implement certificate pinning to protect against adversaries performing woman-in-the-middle attacks that allow for decryption and analysis of traffic between API clients and API endpoints.
- Use both authentication and authorization to secure your API endpoints. Simply using tokens is insufficient if scopes aren't applied to specify what data users are allowed to request. By failing to authorize requests, users can request data that they shouldn't be authorized to see despite them being properly authenticated. Ensure scopes are applied to tokens and that other security controls that apply identity and access management are also implemented as sidecars to your API deployment.
- Hack your own APIs. Regular penetration testing should be performed along with static and dynamic code analysis to ensure vulnerabilities are found before going into and after production deployment.
- Ensure developers are properly trained with secure code training so "shift-left" security is implemented into your software development lifecycle (SDLC).
- Ensure the proper "shield-right" security controls are implemented to protect applications as a part of being placed into production.

APIS RISING

APIs RISING

In the beginning, web applications were built and hosted on the same server and ran as one large application, many times with the backend database running on that same server. This single application architecture is what's known as monolithic applications or colloquially speaking, "monoliths."

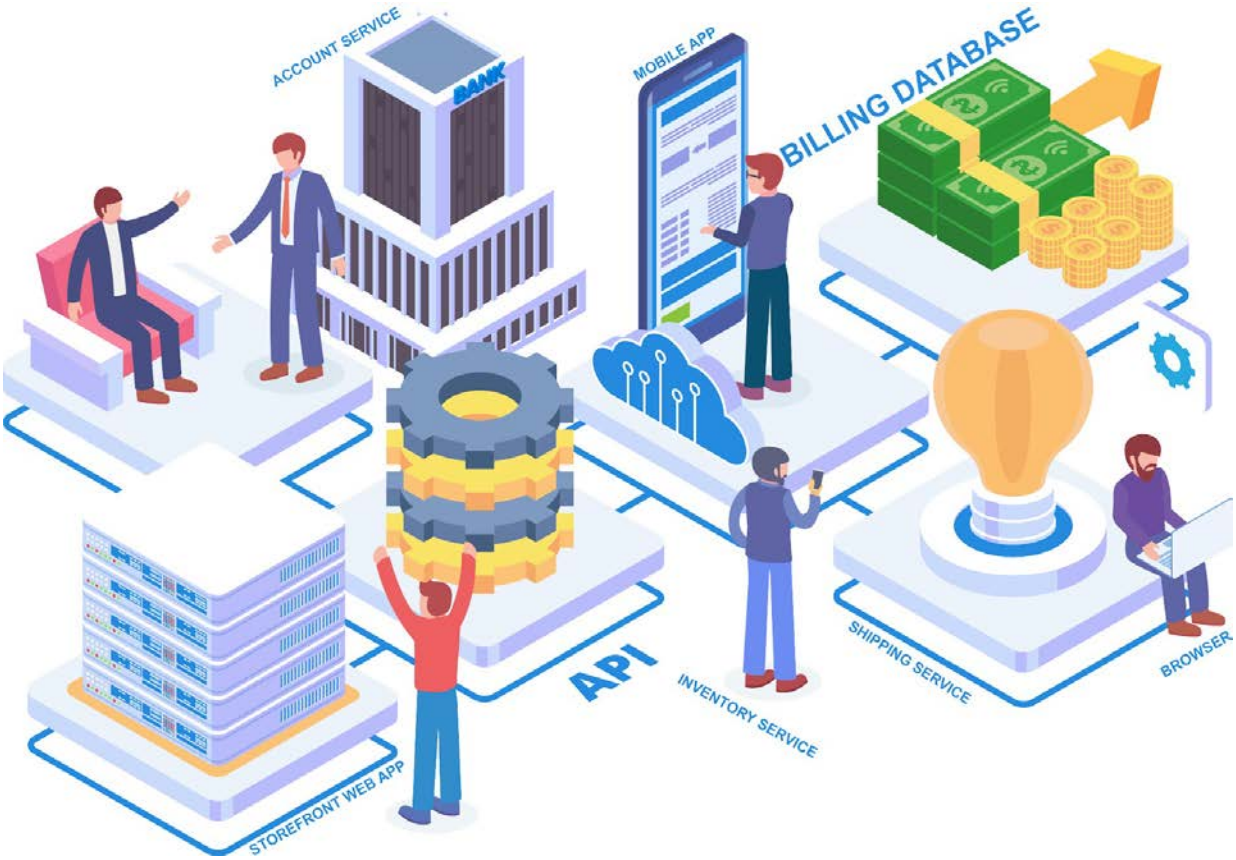
Monoliths posed enumerable challenges in DevOps and DevSecOps such as "too many cooks in the kitchen" where a team of developers would be working sometimes in the same area of a web site or even the same part of the code. Additionally, the entire site would have to be taken offline just to change one part making it unscalable as the site grew.

Microservices and APIs would change this architecture to a distributed one where different parts of the web site could be split up into different microservices gated by APIs.

What Are APIs

API is an acronym for Application Programming Interface and allows for applications and application components (services) to talk to each other. This introduced an evolutionary exodus from monoliths to where we are today with modern applications. Modern applications are broken up into microservices driven by APIs that allow those discrete pieces to talk but seem like a single web application to an end-user. For example, accessing the main page of an ecommerce website and then accessing your shopping cart could be a call to a separate API endpoint. This would seem like it's all running on a single web site despite different parts of the site originating from multiple API endpoints. (**Figure 1**).

Figure 1. An illustration of an eCommerce site running on microservices with APIs



Source: Knight Ink

A good way to think of APIs is to think of them as being the wiring that connects your application to both end users and the inner workings of the application components. APIs are the glue that holds everything together.

While APIs do speak the hypertext transport protocol (HTTP) like a traditional web server, they don't return presentational results complete with HTML like a web site. Rather, APIs will typically send back JSON formatted text to the requestor. Similar to web sites, an API request can be sent with a web browser but if not using a web or mobile application, developers will typically use an API client, such as Postman for sending API requests to different API endpoints. Modern APIs are RESTful (or REST) APIs, meaning they can operate over any protocol (typically HTTP) and don't require third-party libraries or additional software.

Why are they needed

Today's organization, public or private, is producing petabytes of data a day. This requires 24x7 always-on availability and most importantly, data sharing. Data is shared inter-departmentally within organizations and increasingly, outside organizations with partners and customers. Recently, laws such as the UK's mandate for open banking referred to as PSD2 and in the U.S., the CMS rule on patient access have ushered in a new API era where among other things, health and financial data must be made available to the requesting party.

APIs make it possible for anyone to be able to create a client (API consumer) that is capable of accessing the data behind the API.

Where are they used

Not all APIs are created equally, and each can face a different direction for consumption by different types of clients. For example, internal APIs are designed to be hidden from external users on the internet and are designed for data sharing between cross-functional teams or departments. Partner APIs are designed to face third-party partners and are usually restricted to authorized parties only.

Composite APIs allow developers to access several API endpoints in a single API request. Finally, open APIs also referred to as external APIs, are publicly facing and accessible from the internet. They might require registration and use of an API key or may be completely open. Oft-times, the organizations who host open APIs will publish documentation for developers to write an API client capable of consuming that data.

What is the OWASP API Security Top 10

APIs expose application logic and sensitive data, such as personally identifiable information (PII) or even payment card information that must be secured. API security focuses on the strategy and solutions to understand and mitigate the attack surface created by APIs.

In 2013, the Open Web Application Security Project (OWASP) Foundation was formed. Their mission is to improve the security of software through community-led opensource software projects and to publish guidance in support of this mission. OWASP first created the OWASP Top 10 list to provide guidance for securing web applications. This lists the ten most common attacks against web applications.

In 2019, OWASP released the first list of threats against APIs titled the *OWASP API Security Top 10* as the top ten most common threats to APIs.

The OWASP API Security Top 10 lists:

1. Broken object level authorization
2. Broken user authentication
3. Excessive data exposure
4. Lack of resources and rate limiting
5. Broken function level authorization
6. Mass assignment
7. Security misconfiguration
8. Injection
9. Improper assets management; and
10. Insufficient logging and monitoring

More information can be found on the OWASP API Security Top 10 project page.

KILL CHAIN METHODOLOGY

Like many terms in cybersecurity, such as Demilitarized Zone (DMZ), the Kill Chain Model (KCM) is a term plagiarized by the cybersecurity community from military parlance. The KCM defines the steps an enemy follows during the employment of an attack against a target. Clear and concise kill chain steps can be identified throughout history in terrorist attacks, including the Boston marathon bombing and 9/11.

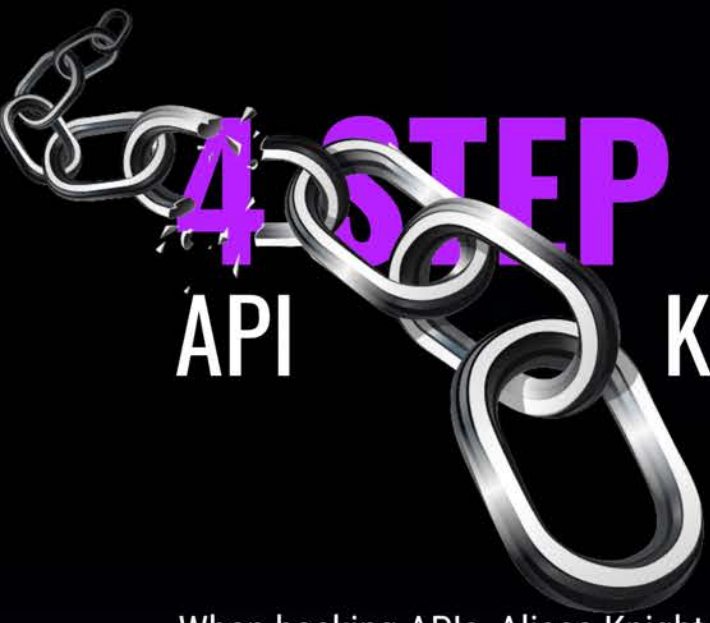
First introduced by the folks at Lockheed Martin in its seminal paper on the subject that it named the Cyber Kill Chain, Lockheed defined the steps as (1) Reconnaissance; (2) Weaponization; (3) Delivery; (4) Exploitation; (5) Installation; (6) Command & Control; and (7) Actions on Objectives.

This entire process defines the steps adversaries take in an advanced persistent threat (APT) attack in the identification, targeting, and exploitation of a victim's host or network.

My Kill Chain Methodology

The entirety of the kill chain I've developed over the past decade of testing APIs has been diagrammed in Figure 3 below into just four steps that I take when performing penetration testing of an API.

TRACEABLE. KNIGHTINK



4 STEP API KILL CHAIN

When hacking APIs, Alissa Knight follows a kill chain she's developed for targeting REST APIs from client to API endpoint that's been used in over 100 penetration tests.

01

Reverse Engineering

Mobile app is downloaded from the Android app store and then extracted off the Android mobile device using APK Extractor. It's then uploaded into MobSF for decompiling so API secrets can be discovered in the code.

02

Network Traffic Analysis

Using SSL/TLS person-in-the-middle attack tools, traffic is decrypted and analyzed to understand how the API client interacts with the API endpoints to better understand legitimate requests. This is done with both mobile or web apps.

03

Map API Behaviors

The API requests are documented along with the responses from the API endpoints for later manual fuzzing.

04

Fuzzing

Manual fuzzing is performed, modifying the API requests in an attempt to find Broken Object Level Authorization, Broken User Authentication, Injection, Excessive Data Exposure, Broken Function Level Authorization, Mass Assignment, and other API vulnerabilities

KNIGHTINK

www.traceable.ai

TRACEABLE.

HACKING APIS STEP-BY-STEP

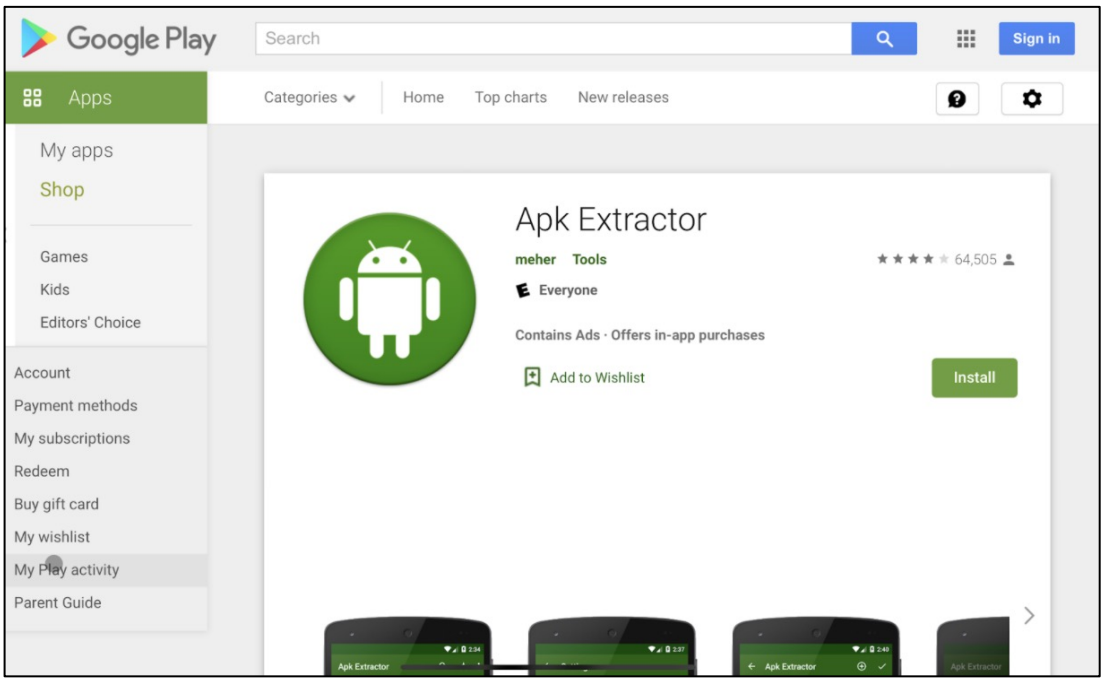
Reverse Engineering

The first step I follow when hacking APIs is to reverse engineer the API client. The purpose of reverse engineering the app is an effort to find hard coded API secrets that can then be used to authenticate with APIs the app talks to. Unfortunately, it's been common practice in the mobile apps I've decompiled for developers to hard code API secrets in their apps as well as usernames and passwords to their own systems and third parties.

In performing reverse engineering of a mobile app, I first download the Android app from the Google Play store onto my Android mobile device. Using APK Extractor (**Figure 2**), I extract the APK package for the app to my cloud drive where I then transfer it to my workstation.

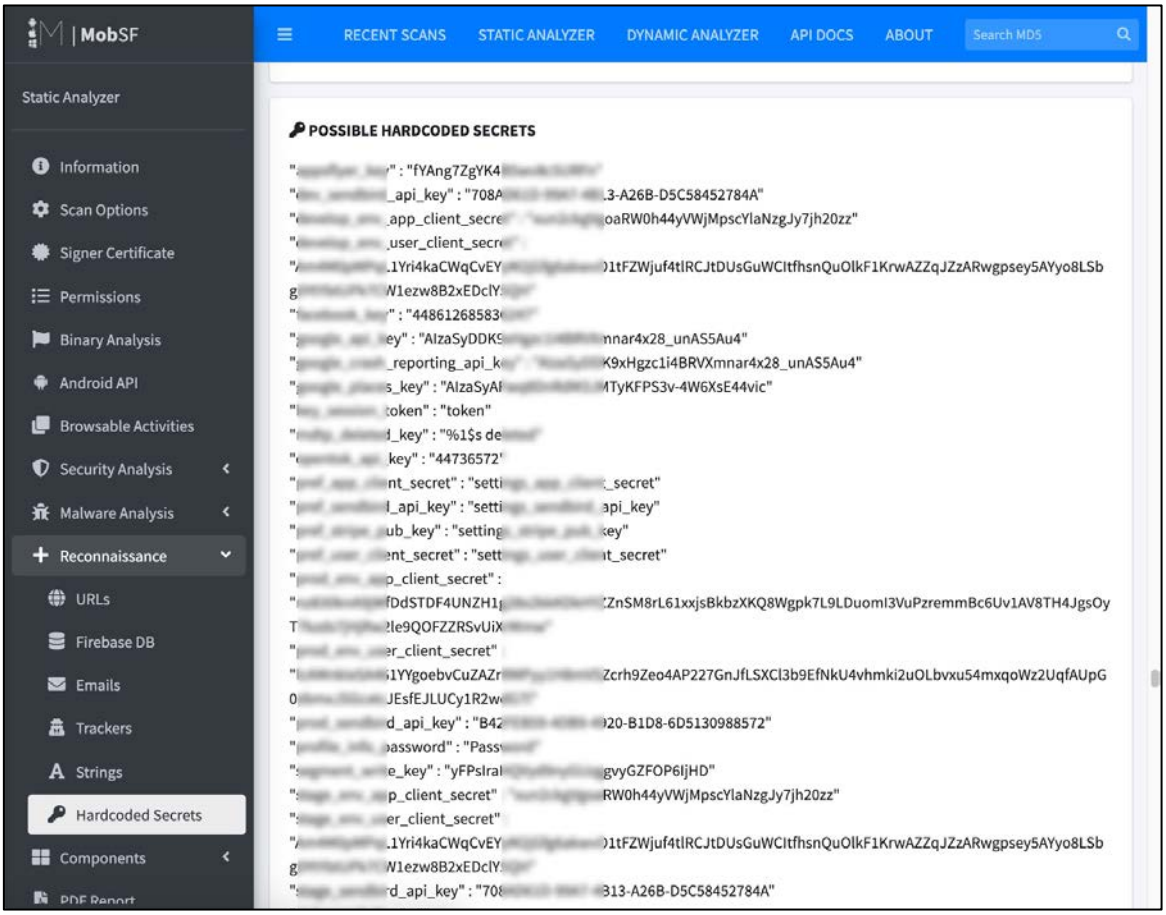
Next, the APK file is imported into Mobile Security Framework (MobSF) (**Figure 3**) where it's automatically decompiled back to its original source code. MobSF makes attempts at finding any hard coded API secrets in the code automatically, but I have been able to find more keys, tokens, and credentials by simply using grep in a terminal window.

Figure 2. APK Extractor in the Google Play Store



Source: Knight Ink

Figure 3. Mobile Security Framework



Source: Knight Ink

The keys, tokens, and any credentials are then carried further into the fuzzing step explained later.

Mitigation Steps

In order to prevent reverse engineering of your mobile app, in-application protection solutions are capable of obfuscating the source code and applying white box encryption that makes it far more difficult for adversaries to reverse it back to the original code.

Any mobile app, whether it contains hardcoded API secrets or not, should be obfuscated. While obfuscation does not prevent reverse engineering, it does make it far more difficult to do and requires a great deal of sophistication to deobfuscate.

Many of the in-app protection tools apply numerous obfuscation methods that makes it far more difficult of an effort causing the adversary to simply move on to another app.

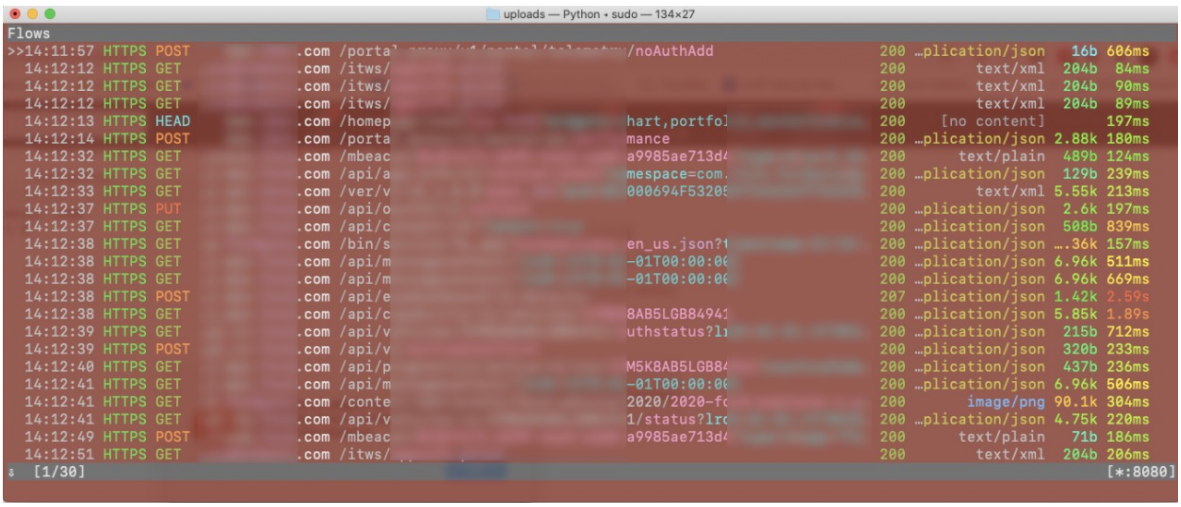
Network Traffic Analysis

Certificate pinning is used to secure the communication between mobile apps and their backend application programming interfaces (APIs) exposed to the internet. Because of a frequent lack of certificate pinning by the targets, adversaries are able to successfully perform woman-in-the-middle (WITM) attacks against the APIs the mobile apps communicate with. This allows an adversary to better understand how the APIs worked since no documentation was published by the organizations.

Every supported action in the mobile app was used and simultaneously captured by my tool Mitmproxy (**Figure 4**).

Certificate pinning restricts which certificates are valid so any certificate presented is rejected if it is not the certificate authority (CA) issuer's public keys that the developer pinned to the endpoint. This prevents an attacker from inserting herself in between the client and API in a WITM attack, that would give the attacker the ability to decrypt and see the data being transmitted between the client and API.

Figure 4. Mitmproxy decrypting SSL traffic in a woman-in-the-middle attack



Source: Knight Ink

As a result of the mobile apps not implementing pinning, an adversary is able to decrypt the SSL sessions between the apps and the APIs in Mitmproxy and then replay the API requests to their API client.

This allows manual fuzzing of the different API requests in an effort to test the API endpoints for vulnerabilities, such as BOLA vulnerabilities, which many were vulnerable to.

Mitigation Steps

Preventing the analysis of traffic between web browser and mobile apps to API endpoints can be done using certificate pinning.

There are however different methods for how this is achieved in web browsers versus mobile apps.

In browsers, the effort was originally implemented using HTTP Public Key Pinning (HPKP). HPKP was a now deprecated security feature that told web clients to associate a specific cryptographic public key with a specific web server. The industry has now moved to Expect-CT (Certificate Transparency), a new field added to the HTTP header. Certificate Transparency is an open framework designed to protect against and monitor for certificate misissuances. (Mozilla, 2021)

Many of the developers I've met with to discuss these issues informed me they don't implement pinning out of fear that a certificate issue will brick their app. Solutions now offer pinning management that better streamlines the effort, eliminating the concern over an expired or bad certificate bricking your app.

Map API Behaviors

The most important step I take when performing penetration tests against APIs is using the API client to map the different *expected* behaviors of the API endpoints.

In this effort, I test every function available to me in the client, document the request in a spreadsheet for example, then document the result. This allows me to then take those different API requests and test them individually in my API client outside of the authorized mobile or web app as well as inspect the responses to see if the authorized client is filtering out all of the results.

Mitigation Steps

None

Fuzzing

Fuzzing is a technique in software testing where the analyst employs malformed/semi-malformed data injection against a target app. Fuzzing is effectively an attempt to employ different permutations of stimulus in requests to the application in order to elicit a response that might be indicative of a vulnerability.

Fuzzing can be automated or manual and attempts different iterations of numbers, characters, metadata, or pure binary sequences against the target application.

When an organization doesn't randomly generate object IDs and implements them in sequential order, it's possible to easily cycle through all of them to find data. The adversary attempts to access different object IDs to a specific end, such as accessing data she isn't authorized for, transfer money between bank accounts, or remotely control automobiles.

Mitigation Steps

Fuzzing, while not impossible, is a bit more difficult to detect and prevent. However, using solutions such as those that require a specific token with scopes (that have a short expiration period) to be sent in the header before it can even talk to the API endpoint will help in defending against synthetic traffic generated by fuzzers.

Additionally, fuzzers work by evaluating the response to certain stimulus, then reply to that with a new stimulus until it receives a successful HTTP code, such as HTTP code 200.

Code 200 is an OK, success response indicating the request was successful. The fuzzer continuously sends API requests in succession at high requests rates. Thus, finding a solution capable of looking for an unreasonable number of API requests per second would be indicative of synthetic versus human traffic as an effective measure to detect fuzzing.

API ATTACKS BY EXAMPLE

API ATTACKS BY EXAMPLE

Types of Vulnerabilities

Let's first define the categories of attacks used in my research in order to demystify terms commonly misunderstood or conflated in vulnerability analysis.

Most vulnerabilities I've found in the APIs I've tested are introduced by failures in authentication or authorization and in some cases excessive data exposure where filtering object properties was left up to the API consumer.

Authentication Vulnerabilities

The goal of an authentication attack is to gain access to an application by obtaining valid credentials to the system. For authentication attacks, adversaries combine dumped credentials from previous breaches in an account takeover (ATO) style attack referred to as credential stuffing, where usernames and passwords are sent to the API until a successful authentication is established.

This is also called brute forcing. Another form of harvesting credentials is purposely designed web forms where a user is phished and social engineered into entering their credentials. Other methods of credential harvesting are breaches of sites containing usernames and passwords and compromised systems that a keylogger has been installed on. Other types of authentication vulnerabilities exist, which I'll cover in a later section.

Authorization Vulnerabilities

An authorization attack attempts to access information or data which the user does not have legitimate access. Simply authenticating with an API with either a legitimate account using Basic Auth or with an API key or token, doesn't mean that the individual is authorized to read or write the data.

An authorization vulnerability is number one on the OWASP API Top 10 list, referred to as BOLA. In the web application vulnerability list, this vulnerability is identified as Insecure Direct Object Reference (IDOR). In this vulnerability a successfully authenticated API request asks to read or write data that doesn't belong to the authenticated user, and the access is mistakenly granted.

Excessive Data Exposure Vulnerabilities

This class of vulnerability is introduced when a developer exposes all object properties without considering the sensitivity of each property, relying on the client to filter out only what the user needs to see. By doing so, this leaves the API endpoint vulnerable to manual API requests sent from API clients that don't implement filtering of those properties.

This vulnerability can have a profound impact when the property is especially sensitive, such as those containing PII, PHI, payment card information, credentials, or other sensitive data that shouldn't be made available.

The API Attacks

Broken Object Level Authorization (API1:2019)

It should come as no surprise that the BOLA vulnerability made the 1st place position in the OWASP API Security Top 10 list. BOLA vulnerabilities have now become the most popular and widely seen method of API abuses in the wild and are a type of authorization attack.

Simply put, a BOLA vulnerability enables an adversary to substitute the ID of a resource with the ID of another. When the object ID can be directly observed in the URI, it opens the endpoint up to ID enumeration that allows an adversary the ability to read objects that don't belong to them. These exposed references to internal implementation objects can point to anything, whether it's a file, directory, database record, or key.

For example:

Substituting patientID 1001 in 'GET /patients/1001/lab_results' with ID 2001 in 'GET /patients/2001/lab_results'

The section that follows contains the successful BOLA attacks I employed against each organization's APIs along with screenshots from the breaches.

Broken User Authentication (API2:2019)

Broken authentication in an API can have devastating effects on confidentiality, integrity, and availability of API endpoints, especially when combined with authorization vulnerabilities, such as BOLA. In this section, I provide empirical data culled from different API breaches where authentication was broken in the implementation allowing no authentication at all or the adversary's own token to perform an attack.

Examples of broken authentication in APIs include being able to successfully receive responses to API requests without sending a cookie. Another example is being able to perform requests on behalf of other users by temporarily or permanently assuming their identity.

Excessive Data Exposure (API3:2019)

Excessive data exposure is introduced when a developer exposes all object properties and relies on the client to perform filtering of that data before displaying it in the API client. This vulnerability can expose highly sensitive data when the request is sent using an API client instead of the legitimate API consumer app. This leads to an excessive data exposure vulnerability where some properties can contain PII, PHI, or other types of highly sensitive or regulated data.

The Banks

A pervasive lack of API security in financial services and FinTech exposes banks and payment service providers (PSPs) to the threat of data loss and fraud.

The Context

Open banking and the API economy are driving a global disruption in the financial services industry. In the U.S., the three major core banking providers FIS, FiServe, and JHA are adopting open banking in the absence of clear regulations. Furthermore, in Europe, adoption of the Revised Payment Services Directive (PSD2) is driving another move to adopt open API led banking services.

Mint.com, a financial aggregator in the United States, staked its claim on the last mile of the customer banking relationship, disrupting and dis-intermediating traditional actors on the US FinServe stage. Mint.com drew worldwide attention to the demand by a new generation of consumers who were hungry for financial account aggregation. This aggregation has created a new economy of Account Information Service Providers (AISPs).

API ATTACKS BY EXAMPLE

Traditional banks attempted to ban services like Mint.com under the intense market pressure of competition, which under PSD2 in Europe is now illegal. As of January 2018, European banks were required under law to provide access to this customer data, such as account information to AISPs.

Welcome to the new API economy in financial services -- the plumbing system to this new intermediated and dis-intermediated banking system that enables open data interchange between FinServe and FinTech companies.

However, in the U.S. it isn't the threat of breaking laws faced by traditional banks in the EU, but by not adapting to these new market demands to compete against challenger banks. Challenger banks are a new market entrant thriving because of their ability to be far more agile and offer a better customer experience in a market increasingly evolving towards digitalization.

As a hacker, this disruption in financial services has created a target-rich environment for me, due to the lack of hardening of the APIs that form the fabric of this new financial services industry.

My Attack

In 2019, I downloaded 30 financial services and financial technology (FinTech) mobile apps and reverse engineered them in order to demonstrate a systemic presence of hardcoded API secrets. These secrets included usernames and passwords, private keys, and API keys and tokens into the APIs and the PSPs they share data with.

The mobile apps in this exercise were extracted off my Android mobile device using APK Extractor. I then decompiled the APK files using Mobile Security Framework (MobSF), where I was then able to search the code for hard coded secrets.

Once the packages were decompiled, I dropped to a terminal shell on my analysis host and ran a series of grep strings against the source code in order to find hard coded API keys, tokens, and even credentials. Many of the hard coded API secrets were not even for the APIs of the app developer, rather, for third-party APIs, such as payment processors and even Amazon Web Services.

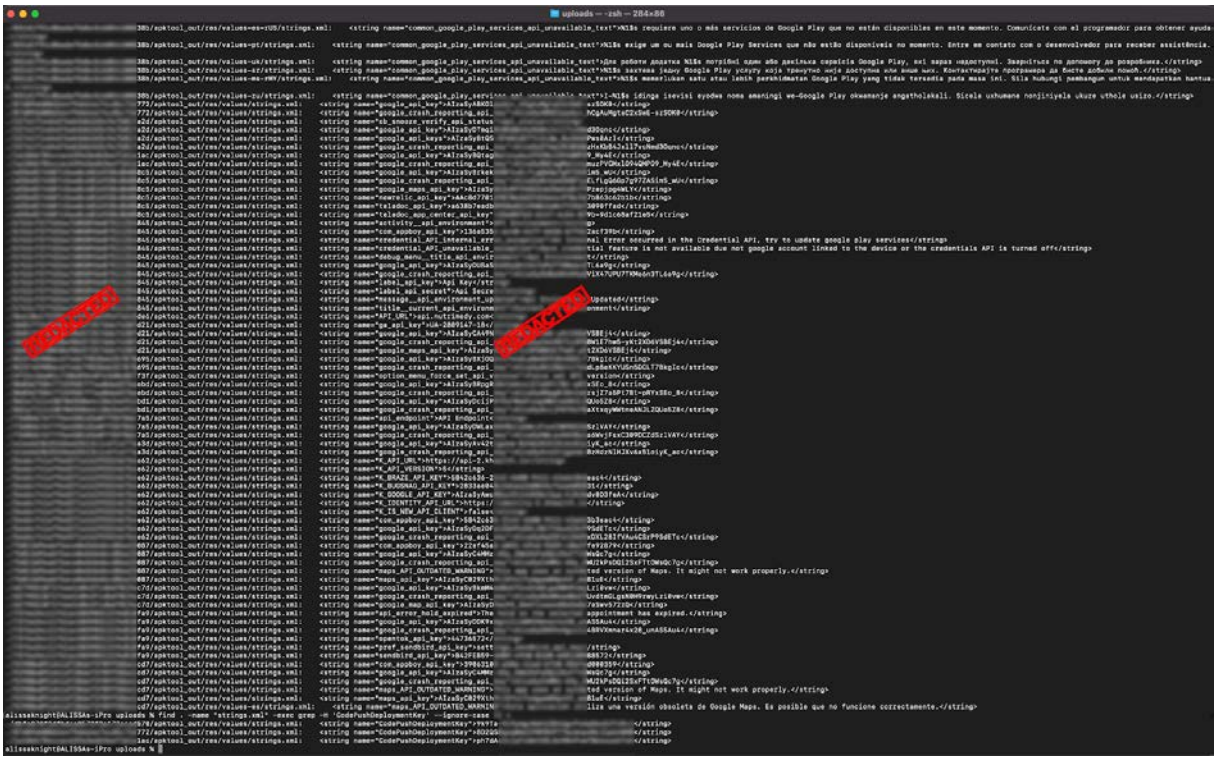
Several of the banks allowed me to take the attack further after I completed the static analysis of their bank mobile apps and then target their APIs.

Once the API secrets are collected, I then move on to analyzing the network traffic between the mobile apps and backend APIs. Because many of the mobile apps fail to implement certificate pinning (explained later), I was able to sit in the middle of the communication between the mobile app and backend API (**Figure 5).** Because I was able to successfully present a self-signed certificate to both ends of the conversation, it was possible to decrypt the SSL/TLS traffic. I then documented every API request to perform fuzzing against the APIs later, attempting to find vulnerabilities allowing me access to the data.

Authorization Vulnerabilities

As a result of a failure to authorize my API requests to its endpoints, I was successful in changing the PIN code of other bank customers' debit cards and move money between my accounts and other bank customers. These vulnerabilities are examples of BOLA vulnerabilities and allowed me to use my own credentials to access the bank data of other customers because of a failure to authorize my API request.

Figure 5. Hacking a bank's API servers



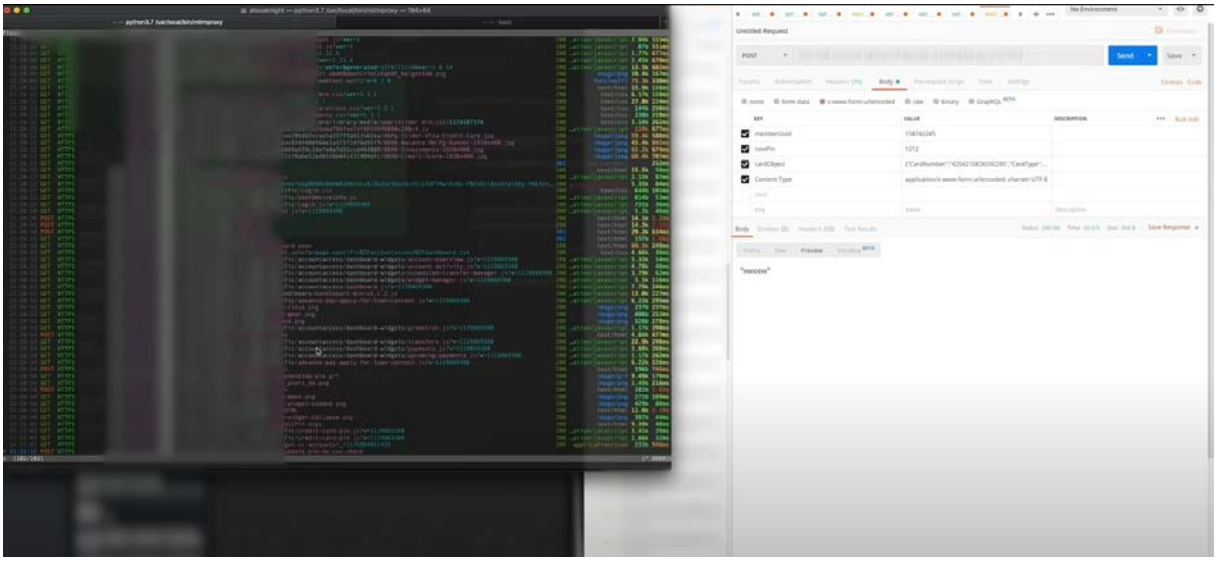
Source: Knight Ink

Authentication Vulnerabilities

In 2018, I discovered a bank's APIs allowed me to successfully move money in and out of bank accounts without having to authenticate. As long as the HTTP header contained the necessary field and the body contained the correct account numbers, it allowed unauthenticated API requests to be performed. This was discovered when I deselected the cookie field once a cookie had been sent to me in Postman, the API client I was using for the engagement (**Figure 6**).

The full video of the bank breach is available here on my YouTube channel.

Figure 6. Hacking a bank account



Source: Knight Ink

The Healthcare Providers

The Context

Just as today's connected consumer is demanding access to their financial information made available through AISPs and other FinTech apps, today's patient is demanding remote access to their data as well. This data is housed in electronic health record (EHR) systems at health and mental wellness providers and payers.

In our new mobile economy where people prefer a cell phone or tablet over a laptop, mobile health (mHealth) represents a very specific type of telehealth driven by our new mobile app economy.

Recently, the World Health Organization (WHO) defined mHealth as "the use of mobile and wireless technologies to support the achievement of health objectives."

Telehealth in the broader sense refers to the use of technology to improve healthcare outcomes. MHealth refers to the specific use of mobile technology and apps for patients to acquire their own health information without the intervention of a clinician.

Accordingly, there are different categories of mHealth apps in the market, to include symptom checkers, clinical records management, self monitoring, rehabilitation programs, prescription filing, communication with a patient's doctor or mental health professional, and more.

My Attack

Authorization Vulnerabilities

In 2020, I downloaded 30 mHealth apps, reverse engineered them, and reported on over 114 hardcoded API keys and tokens, credentials, and private keys in those apps that also led to several healthcare providers giving me access to their APIs to take my attacks further.

One hundred percent of the APIs I targeted in this research were successfully breached as a result of BOLA vulnerabilities.

Videos on my healthcare API vulnerability research are available here.

The Automakers

The Context

As more devices become connected that were previously unreachable, whether Internet of Things (IoT) devices or ground, sea, and air passenger vehicles, the potential for cybersecurity breaches to affect human life and safety becomes more of a present-day reality rather than Hollywood theatrics.

To put the complexity of a modern passenger vehicle into perspective, a Boeing 757 has roughly 6.5 million lines of code behind its avionic and online support systems. The United States' F35 Joint Strike Fighter has over 8 million lines of code giving the fighter jet the capability to take off and land like a helicopter. While Facebook, discounting its backend, runs on roughly 62 million lines of code, today's connected vehicle now runs over 100 million lines of code.

With so many lines of code written by sometimes hundreds of original equipment manufacturer (OEM) parts that all go into a single vehicle, it's easy for vulnerabilities to make their way into what used to be nothing more than a combustion engine on a hand cart.

Today, a modern vehicle is more akin to a smartphone on wheels, with both an internal network and external network that talk over WiFi and GSM giving it the capability to talk to nearby vehicles on the road. Therefore, saying that the passenger doors of a vehicle are the only ingress points into a modern vehicle from the outside is simply no longer true. There are in fact multiple ways for someone from the outside to compromise the vehicle, such as unlocking its doors or starting its engine from their computer thousands of miles away.

My Attack

Authorization Vulnerabilities

Given the severity of these vulnerabilities, in 2020, law enforcement contacted me shortly after my new book, Hacking Connected Cars (Wiley, 2020) was published.

Law enforcement officials were concerned with the growing attack surface they saw being created as more interconnected technologies continue to be added to the cockpit of the vehicles by automakers and the increasing dependence on technology to enhance the crime fighting mission.

Because of a number of vulnerabilities in the automaker's APIs failed to authorize my API requests, I was able to remotely lock and unlock the doors of any vehicle in the fleet or stop and start the engine.

Authentication Vulnerabilities

The automaker in this breach authorized the requests made to its APIs but did not properly authenticate the requests to ensure that the userID of the authenticated individual carried in the JWT payload was the owner of the vehicle.

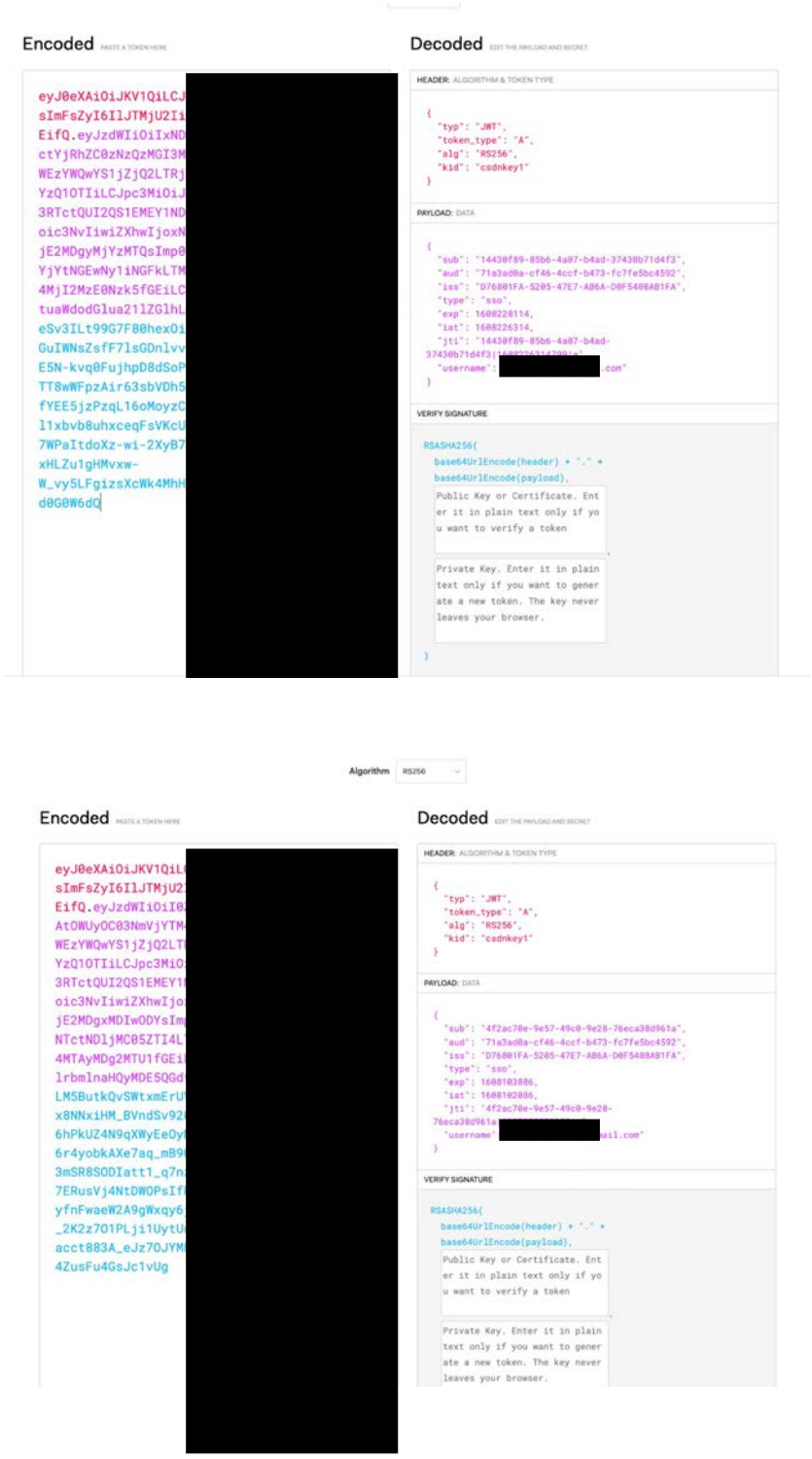
This allows any userID making requests to the API, so long as they have a valid token, to make requests on behalf of the vehicle's owner. (**Figure 7**)

This also allowed the adversary to approve her own request to take remote control of the vehicle despite the fact that the adversary was approving it on behalf of the victim.

Figure 7. Remote control of law enforcement vehicle through API

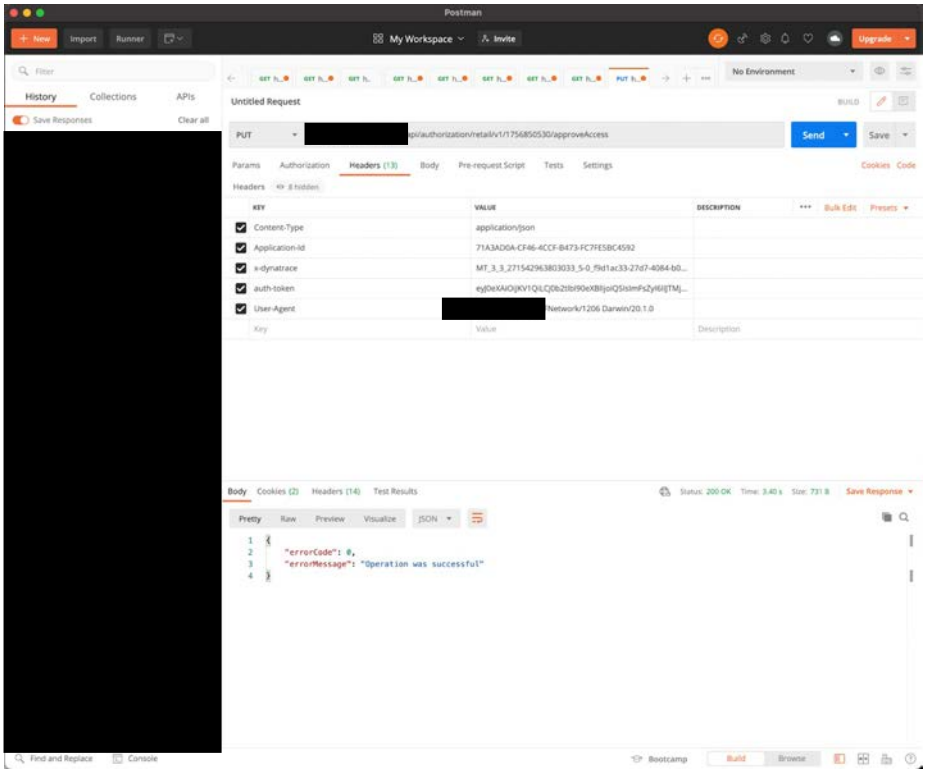
Source: Knight Ink

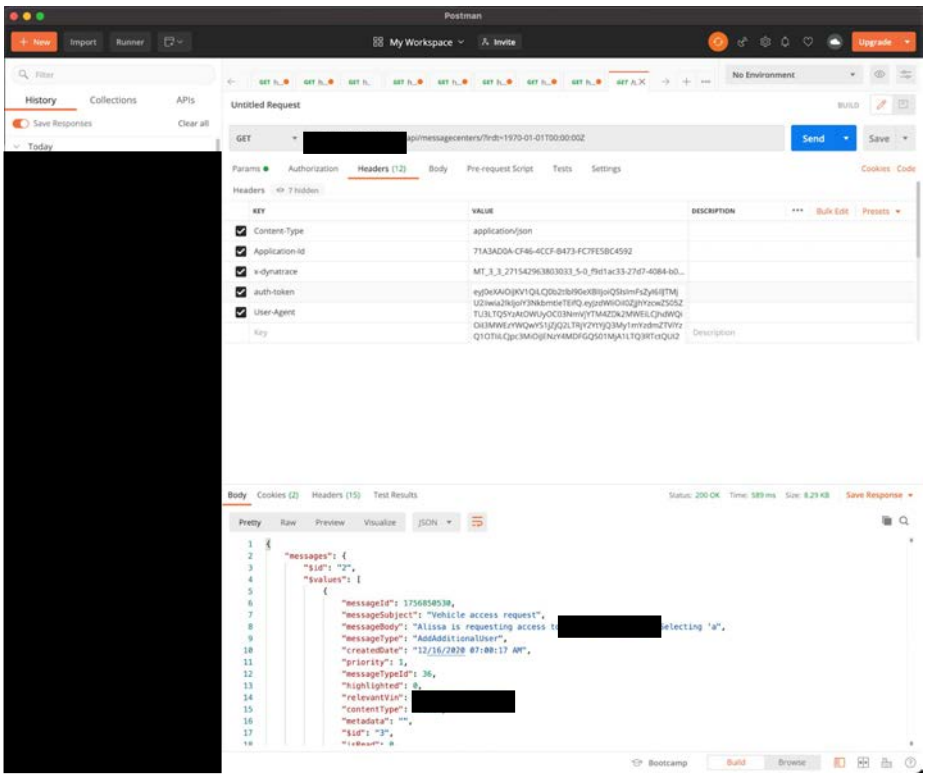
Figure 8. Failure to properly authenticate the logged in user with the vehicle owner



Source: Knight Ink

Figure 9. Failure to properly authenticate the logged in user with the vehicle owner





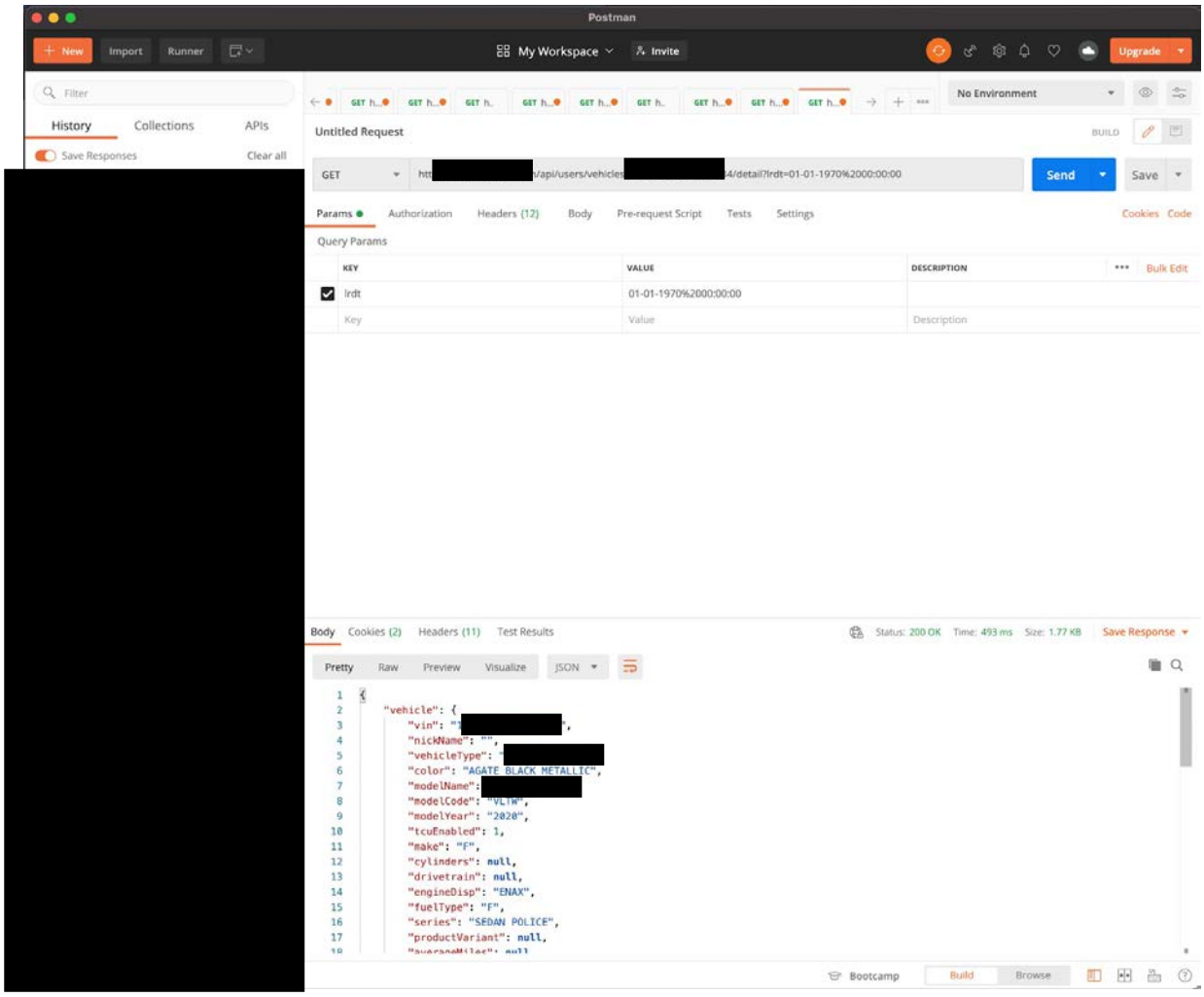
Source: Knight Ink

Excessive Data Exposure

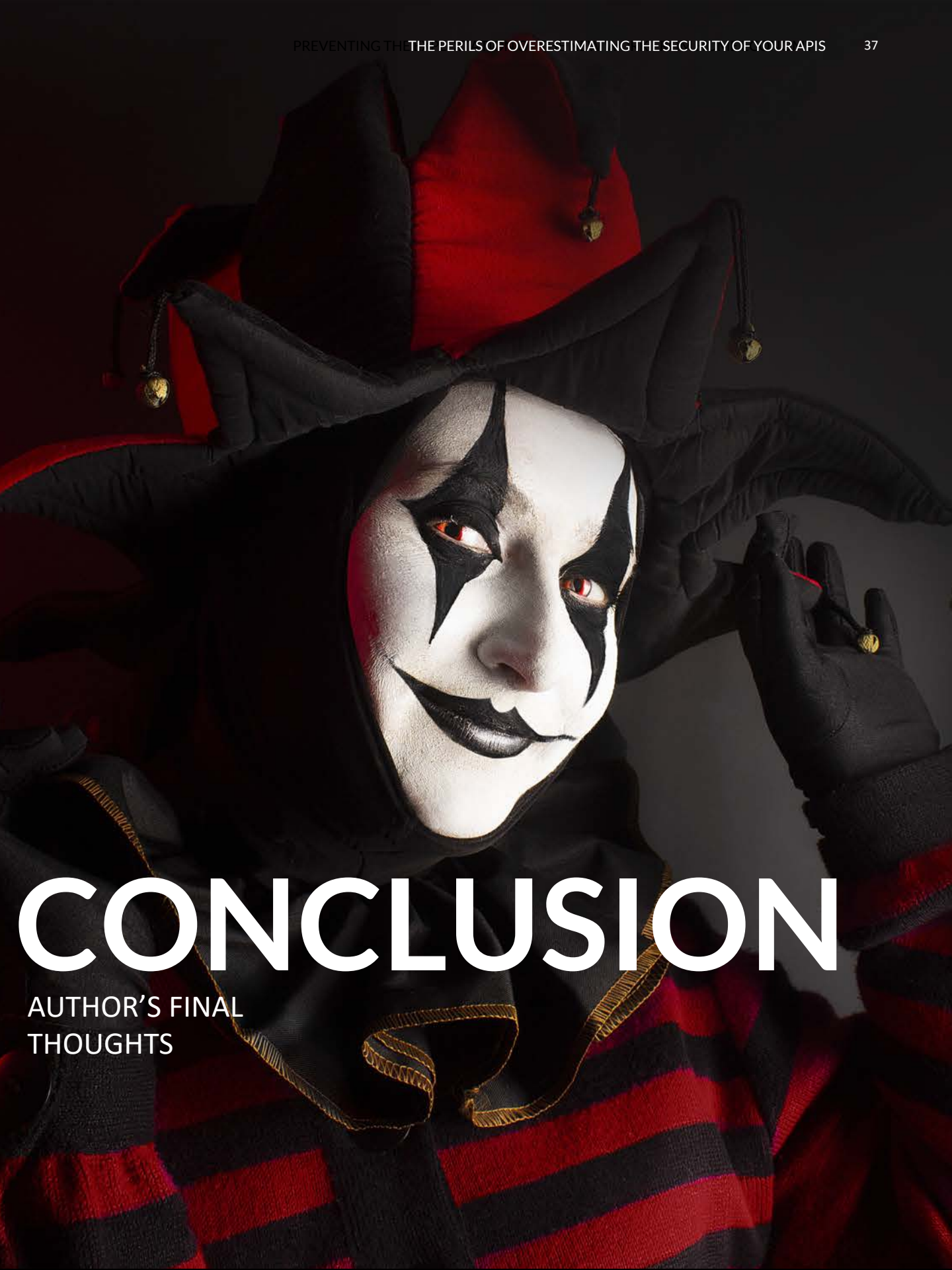
Combined with a failure to authenticate the request, any user can send an API request for all of the details on a vehicle so long as the requestor knows the vehicle identification number (VIN). Due to the fact that the automaker is exposing all of the object properties, all of the details for the vehicle are returned from the database (**Figure 10**).

Videos on my law enforcement vehicle hack are available on my YouTube channel here.

Figure 10. Excessive data exposure



Source: Knight Ink



CONCLUSION

AUTHOR'S FINAL THOUGHTS

The Wrong Way

There are right ways and wrong ways to secure APIs. In every single engagement in the API penetration tests discussed in this eBook, WAFs, API management solutions, CDNs, and tokens were used to secure the APIs. Despite these controls being present, I was still successful in these breaches because the wrong tool was selected for the job.

Web application firewalls (WAFs) are unable to understand the context behind the request. WAFs are great at detecting things that don't require this context such as SQL injection but are incapable of determining whether I, an authenticated user, is authorized to request data that doesn't belong to me.

In summary, securing APIs with these security controls as an alternative to purpose-built API threat management solutions can mean the difference between the prevention of an API breach or being in the wrong story on the nightly news.

The wrong controls to effectively secure APIs:

- Web application firewalls (WAFs)
- API management gateways
- Content Delivery Networks (CDNs)
- Improper use of tokens

The Right Way

In order to secure an API effectively, you want to implement authentication and authorization to ensure that whatever application or device is sending an API request has the right to do so (authentication) and is allowed to read or write the data (authorization).

However, you can't implement authentication and authorization controls and call it a day. Otherwise, an API gateway would suffice just fine. You'll want to ensure you have a comprehensive register of your APIs as you can't protect what you don't know you have.

Additionally, excessive data exposure is a systemic problem across many of the organizations I breached. Ensuring that variable assignments are properly limited to avoid mass assignment vulnerabilities is imperative. This along with the other hardening efforts discussed in this section should be operationalized into your organization.

Authenticating Requests

API requests can be authenticated a number of ways. Unfortunately, some authentication mechanisms are more secure than others and while this is widely known, insecure authentication methods such as the use of Basic Auth or API keys are still being used.

Authorizing Requests

Think of authentication simply as proving who you are while authorization proves you're allowed to see the information being requested. Just because you're authenticated, it doesn't necessarily mean you should be able to see the data you're requesting. For example, requesting the patient records of another patient instead of your own.

One of the most common methods of implementing authorization in an API is using Auth0. Per Auth0 documentation, authorization can be determined through the use of policies and rules, which can be used with role-based access control (RBAC). Regardless of whether RBAC is used, requested access is transmitted to the API via scopes and granted access is returned in the issued Access Tokens.

	How it works	Strength
API Keys	Used to track and control how an API is being used. It can be used as both a unique identifier and a secret token for authentication (IBM, 2020)	Weak: When not secured properly
Basic Auth	A simple challenge and response mechanism which a server can request authentication information (userID and password) from a client). (IBM, 2021)	Weak: Authentication information is base-64 encoded
JSON Web Tokens (JWT)	An open standard defining a compact and self-contained way for securely transmitting information between paties as a JSON object (JWT.IO, N.D.)	Strong: When combined properly with scopes and other controls
OAuth 2	Industry-standard protocol for authorization that enables applications to obtain limited access to user accounts on an HTTP service. It delegates user authentication to the service that hosts the user account and authorizes third-party application to access the user account. (Digital Ocean, 2014)	Because it relies solely on SSL/TLS layer for confidentiality, it's deemed less secure since it passes the client secret with every authentication request.
OpenID Connect	Provides a simple identity layer on top of the OAuth 2 protocol allowing clients to verify the identity of the end-user based on the authentication performed by an authorization server. (OpenID.net, N.D.)	OAuth and OIDC are complicated, and it takes a lot of time and effort to understand and use them properly without opening yourself up to exploitation (Degges, 2019)

SOURCES

Open Banking (PSD2) and the future of Financial services. (n.d.). MuleSoft. Retrieved May 13, 2021, from https://www.mulesoft.com/lp/whitepaper/api/psd2-open-banking-financial-services?utm_source=google&utm_medium=cpc&utm_campaign=g-fins-na-search-psd2&utm_term=psd2&utm_content=g-p-c&gclid=CjwKCAjwj6SEBhAOEiwAvFRuKJ53EyBnnQEKnjS_jA7d58W92D03Cp0cuDVyFy_UlNiAYOMRzTMk5hoCV6wQAvD_BwE

How to Build an Effective API Security Strategy. (n.d.). Gartner. Retrieved May 13, 2021, from https://www.gartner.com/en/documents/3834704

Types of API | Open Rest API Types for Enterprise | Types Of API Calls & REST API Protocol. (n.d.). Stoplight.Io. Retrieved May 13, 2021, from https://stoplight.io/api-types/

OWASP API Security - Top 10 | OWASP. (n.d.). OWASP. Retrieved May 13, 2021, from https://owasp.org/www-project-api-security/

Rowley, J. (2020, July 21). What is Certificate Pinning? DigiCert. https://www.digicert.com/dc/blog/certificate-pinning-what-is-certificate-pinning/

Certificate Transparency - Web security | MDN. (2021, March 20). Mozilla. https://developer.mozilla.org/en-US/docs/Web/Security/Certificate_Transparency

Cyber Kill Chain®. (n.d.). Lockheed Martin Cyber Kill Chain. Retrieved June 12, 2021, from https://www.lockheedmartin.com/en-us/capabilities/cyber/cyber-kill-chain.html

ABOUT THE AUTHOR

Alissa Knight has performed over one hundred penetration tests in her career over the last two decades. Much of her experience has been performing penetration tests of application programming interfaces (APIs), specifically targeting connected cars, medical devices and mobile and web applications via their APIs. Alissa is currently writing a new book on hacking and securing APIs as a published author of Hacking Connected Cars: Tactics, Techniques, and Procedures.

As a public speaker, influencer, and content creator, Alissa is a regular keynote speaker at some of the worlds largest API conferences, including API Days and API World where she provides workshops and guidance on securing APIs and how to perform penetration testing of APIs.

Follow Alissa Knight on Twitter and LinkedIn and subscribe to her YouTube channel to get access to her latest vulnerability research.

ABOUT KNIGHT INK

Firm Overview

Knight Ink is a content strategy, creation, and influencer marketing agency founded for category leaders and challenger brands in cybersecurity to fill current gaps in content and community management. We help vendors create and distribute their stories to the market in the form of written and visual storytelling drawn from 20+ years of experience working with global brands in cybersecurity. Knight Ink balances pragmatism with thought leadership and community management that amplifies a brand's reach, breeds customer delight and loyalty, and delivers creative experiences in written and visual content in cybersecurity.

Amid a sea of monotony, we help cybersecurity vendors unfurl, ascertain, and unfetter truly distinct positioning that drives accretive growth through amplified reach and customer loyalty using written and visual experiences.

Knight Ink delivers written and visual content through a blue ocean strategy tailored to specific brands. Whether it's a firewall, network threat analytics solutions, endpoint detection and response, or any other technology, every brand must swim out of a red sea of competition clawing at each other for market share using commoditized features. We help our clients navigate to blue ocean where the lowest price or most features don't matter.

We work with our customers to create a content strategy built around their blue ocean then perform the tactical steps necessary to execute on that strategy through the creation of written and visual content assets unique to the company and its story for the individual customer personas created in the strategy setting.

Contact Us

Web: www.knightinkmedia.com
Phone: (702) 637-8297
Address: 1980 Festival Plaza Drive, Suite 300, Las Vegas, NV 89135

Knight Ink
1980 Festival Plaza Drive
Suite 300
Las Vegas, NV 89135
ak@knightinkmedia.com