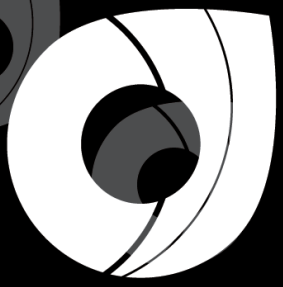
Corndel DevOps Engineering Programme

in association with Softwire

Module 10: Data & Security 2



Corndel Digital.

DB29

Introduction

In Module 9 we considered security in the context of data and databases. In this module we will cover security more broadly, including authentication and authorisation; different classes of exploits and vulnerabilities; how to detect areas of risk; and secure development practices.

In more old fashioned approaches Security was often considered a separate discipline, similar to the distinction between Development and Operations. This could lead to the practice of only considering security once development was completed. Unsurprisingly, this is not an approach we recommend. Modern approaches favour a much more integrated approach, sometimes referred to as **DevSecOps**.

Security is about the journey, not the destination.

Security is not something you can achieve as an add-on or after thought. You should be considering the security of your systems and data from the very start of your project and throughout the entire software development life cycle. You should make secure development an intrinsic part of your team's processes.

Writing secure *code should be as fundamental to your workflow as writing* clean *code.*

Quotes like "Security is everyone's responsibility" may sound trite, but they are nevertheless true.

Table of Contents

Core Material

Additional Material

Chapter 1

Application Security Basics

How are applications secured?

There are two key concepts in application security models:

- **Authentication** means verifying a user's identity;
- **Authorisation** means deciding what actions that user is permitted to take, on what resources.

Authentication

There are many possible **authenticators** that can be used to verify a user's identity, but they typically fall into three categories:

- Something the user **knows**: a piece of private information like a password
- Something the user **has**: a physical token like a chip and PIN card
- Something the user **is**: biometric information like a fingerprint

Something the user knows

Examples include:

1. **Passwords and PINs**: historically the most common way of securing applications, but vulnerable to phishing attacks, data breaches, or brute force attacks (especially if a system permits weak passwords).
2. **Knowledge-based verification (KBV)**: asking the user a question that only they should know the answer to, but not a specific secret like a password. Includes both:
3. *Static* KBV challenges like "Name of your first pet" - these are vulnerable to the challenge response becoming more widely or publicly known
4. *Dynamic* KBV challenges where the answer changes over time, like "Which of these was your statement balance last month?" for a credit card.

Something the user has

Examples include:

- **Time-based One Time Passwords (TOTP)**: a single-use code generated by something the user has in their possession - e.g. a chip and PIN card with a card reader, or a mobile phone using a TOTP app such as Google Authenticator or Duo. These are vulnerable to theft of the physical token (e.g. stolen wallet or phone).
- A single-use verification code or 'magic link' (i.e. a URL containing a unique token), which might be sent to the user by SMS, email or post. These are vulnerable to interception or leak - e.g. SIM swap attacks or breach of email accounts.

These are typically too weak to be used on their own, except for the most low-risk situations. However, they can be very valuable as a secondary authenticator, for example in addition to a password.

Something the user is

Examples include:

- Fingerprint recognition
- Facial recognition
- Verification of a photo, video or signature against a reference

These are typically highly secure because a user's biometric information is unique to them and cannot be stolen or guessed. However, there is some risk that an attacker could impersonate a user - for example by holding up a photo of them, playing a pre-recorded video clip, or using a copy of their fingerprint.

Multi-factor authentication

As seen above, no single authenticator is perfect: they all have their vulnerabilities. However, the vulnerabilities are quite different between the different types (e.g. data theft for a password, vs physical theft for a TOTP device). Therefore, many systems now require the use of multiple authenticators, of different types. For example, combining a password with a TOTP device: the risk of a password leak and theft of the physical TOTP device by the same attacker is very low. This is known as **two-factor authentication (2FA)** or **multi-factor authentication (MFA)**.

Third party authentication providers

Many systems use third-party providers to handle authentication. This means you are relying on a trusted third party in order to leverage latest security best-practice - similar to the "do not roll your own" principle that we saw in the previous module. It also lets the same authentication method be reused across different, often unrelated, applications. This means that users have fewer passwords to remember and helps streamline onboarding & offboarding users across a suite of applications. This is known as **Federated** identity or **Single Sign-On (SSO)**.

The development of authentication standards such as **OAuth 2.0** and **OpenID Connect** has made it much easier to use one (or many) external identity providers. These standards are fundamental in supporting the ubiquitous "Login in Social Media" options. Services like **Okta** and **Auth0** also use these standards to provide easy integrations for a variety of client applications.

In the corporate world, Microsoft's **Active Directory (AD)** and its cloud based equivalent **Azure Active Directory (AAD)** are widespread. However, the latter still supports OpenID Connect and can be treated much like any other identity provider. Alternative cloud solutions include **AWS Cognito** and **Google Identity Platform**.

Authorisation

Once a user has been authenticated, **authorisation** is the process of determining which actions that user is permitted to take, on which resources.

Traditionally, this is thought of in terms of **Access Control Lists (ACL)**: each resource would have an ACL specifying a list of permissions for each user who might interact with it. For example, the ACL for a particular document might look something like `(Amanda: read,write; Barbara: read)`.

In practice, ACLs are difficult to maintain:

1. When users are added or their permissions change, then many resources need to be updated. Similarly, if a new resource is created then an ACL needs to be created detailing its permissions in relation to many users individually.
2. Storage of permissions is fragmented, and it is difficult to get an overall view of what permissions exist, or what actions a user is permitted to take overall.

To get around this, most systems introduce an additional level of abstraction: we consider two examples below.

Role-based access control

In a **role-based access control (RBAC)** system, users are assigned to **roles**, which are themselves granted a set of permissions allowing them to perform certain actions. Any user with a role can carry out the operations that that role has permissions for. Continuing our example from above, Amanda might be granted the `data-writer` role, letting her edit all documents in a system. Barbara would have the `data-reader` role, letting her read the documents, but not edit them.

This solves some of the challenges with simple ACLs:

- Permissions are easier to maintain. If a new user is added, or an existing user granted new permissions, they just need to be added to a new role, rather than having to grant them permission to each resource individually. Similarly, when a new resource of a particular type is created, users in a given role will automatically have permission to interact with that resource.
- Typically, the roles can be aligned to concepts meaningful in the domain of the application (e.g. `hr-administrator`, `finance-administrator`, `auditor`), making it easier to keep an intuitive understanding of the permission system.

Attribute-based access control

In an **attribute-based access control (ABAC)** system, users have *claims* that establish information about them, while resources have *policies* that determine who can interact with them.

This is commonly used in a multi-tenanted application, to ensure that users can only interact with resources associated with the same tenant.

> *Example*: Alice's company uses a multi-tenanted cloud file storage service. The service uses ABAC to ensure that users can only view files from their own company. Alice's user has a claim that says `company: acme-co` and all Acme Co documents have a policy requiring that the user is associated with Acme Co in order to view them.

Role-based and attribute-based access controls may be combined.

> *Example*: For a banking application, customer support staff are granted the `customer-support` role which lets them view all customers' transaction history. But for users with the `customer` role, an attribute-based policy enforces that they can only view their own transactions.

Chapter 2

Areas of Risk

The next two chapters will cover some common risks to application security, drawing on the **OWASP Top 10** web application security risks.

> *The Open Web Application Security Project (OWASP) compiles a list of the top 10 most critical security risks in web applications, based on industry surveys. It is a valuable reference to maintain awareness of common threats and vulnerabilities.*

This chapter will focus on general areas of high risk, and general good practice to maintain security. The next chapter will describe some important specific exploits, and how to mitigate them.

Sensitive data exposure

Web applications must ensure that sensitive data (such as financial, healthcare or personally identifiable information) is protected from attackers.

We touched on this in the previous module, when considering database security (chapter 9.3). The best practices introduced there are all important "hygiene factors" in application security.

When thinking about this from a web application perspective, there are some additional considerations:

- Data must be encrypted in transit between the client and the server, to prevent an attacker from intercepting and eavesdropping on sensitive data;
- Data must be sent to the correct server, to prevent a malicious actor from spoofing the intended server and diverting data to a different location.

The key mitigation here is to use **HTTPS** (Secure HTTP) when communicating over the internet. This ensures that:

- The client is communicating with the correct server, using a certificate issued by the server to prove its identity;
- Data is encrypted in transit, so it can't be intercepted by an eavesdropper.

Broken authentication

Another major risk is that an attacker could exploit the authentication process to impersonate another user - allowing them to do or see anything that user could. Some examples of common authentication vulnerabilities and exploitations are given below.

Beyond the specific mitigations outlined below, a reliable way to ensure your authentication is secure is to follow a tried-and-tested authentication model (don't roll your own), or delegate authentication to a trusted third-party authentication provider - such as Auth0, Azure Active Directory or AWS Cognito. The companies behind these products put a lot of effort into studying security best practices and emergent vulnerabilities, to ensure their authentication model remains secure.

Credential-stuffing

If a user re-uses their password across a number of websites, and the password is leaked through a data breach on another website, then an attacker could use the leaked credentials to impersonate that user to your application - this is known as a **credential-stuffing attack**.

This can be mitigated by enforcing multi-factor authentication (see chapter 10.1), so that leaked credentials alone are not enough to authenticate.

Brute force attack

In a **brute force** attack, an attacker tries out different passwords until they find one that allows them to log in.

This can be mitigated by **rate-limiting** (i.e. limit the number of login attempts in a given time) and preventing users from using known weak passwords.

Weak credential recovery

Any system is only as secure as its weakest point, and attackers can target weak points such as a process for recovering credentials. For example, if a user is allowed to reset their password just by answering an insecure question like "mother's maiden name".

Session expiry

If your application fails to **expire a user's session** after a short time period (i.e. automatically log out after inactivity), then an attacker could gain access to that user's session.

For instance, if a user accesses an application using a public computer, forgets to log out, and their session is not invalidated, then another person using the same computer later on will already be authenticated as the previous user.

Broken authorisation

Just as a flawed authentication system might enable an attacker to impersonate a user, if your authorisation system is flawed then an attacker might be able to take actions that they should not be allowed to.

> *Example*: If an application requires users to be logged in to edit their profile, but doesn't correctly check that the profile being edited is the profile of that user, then an authenticated user could edit any other user's profile.

To help mitigate against authorisation flaws, you should write automated tests covering the authorisation logic in your application.

Don't trust the client

For any application using a client-server model (e.g. a single page web app with a backend API), you should assume that anything in the client could be manipulated by a malicious user. A common flaw in authorisation and authentication models is to trust activity in the client programme. Instead, you should treat all activity in the client as untrusted, and enforce security on the server.

> *Example*: When Alice logs in to her social networking application, the client code running in her browser makes a request to a `/users/alice/profile` endpoint to load her profile data. But the application's flawed authorisation model relies on the client making the correct request. Alice can manipulate the request URL and make a request to `/users/bob/profile` and intercept Bob's profile too.
>
> To mitigate this, you should not rely on restrictions in the client programme as a means of authorisation. Authorisation rules should always be enforced server-side. In this example, the server should check that Alice does not have permission to view Bob's profile, and reject the request to `/users/bob/profile`.

Other examples of manipulated client data include:

1. Malicious users could manipulate authentication tokens - for example, adding an additional claimed role to a JWT. If your application uses JWTs, make sure you verify the signature on each request, to check that the token has not been tampered with.
2. Malicious users may be able to bypass any validation rules present in the client: to be sure that submitted data is valid, it should be verified server-side.

Phishing and Social Engineering

We've discussed many attack vectors, but it's important to remember that the human elements of a system are vulnerable as well. There are many types of attack that depend on manipulating user behaviour to gain unauthorised access, often referred to as 'social engineering'.

Phishing attacks involve sending out messages impersonating an application or organisation in bulk. These will typically contain a link to a fake login page designed to resemble the real one. Any credentials entered into this page can then be used to access the target application

Spear-phishing attacks are a targetted variant, where the attack is tailored to a specific individual. Using personal details to make the attack more convincing can significantly increase the success rate.

Impersonation attacks occur when an attacker impersonates a user, typically through customer support. They can then use personal details of the user to persuade customer support staff to grant them access to that user's account.

Dependencies and configuration

Vulnerabilities in third-party code

Almost any modern software application will rely on a variety of third-party software packages, libraries and tools. It is important to watch out for security vulnerabilities in any third party-code that you rely on.

This can apply to any part of the software development process, including:

1. Libraries that your code relies on
2. Tools used in your build process
3. The infrastructure hosting your application - e.g. servers, databases, VMs

In 2014, a vulnerability which became known as ***Heartbleed*** *was discovered in the widely used cryptography library* OpenSSL*. The bug compromised the security of HTTPS requests in any application using the OpenSSL library - potentially allowing attackers to extract usernames, passwords and other sensitive information.*

Widely-used open source web servers nginx *and* Apache *used OpenSSL, so any application relying on these servers would have been vulnerable. At the time of discovery in 2014, the bug was estimated to have impacted about 17% of secure web servers in the world.*

A fix to the OpenSSL library was released on the same day the vulnerability was publicly announced, and applications could secure themselves by using the latest version of OpenSSL, and ensuring any potentially compromised private keys or certificates were reset.

Fortunately, mitigating against the risk of *known* vulnerabilities in third-party code is relatively straightforward, at least in some languages. Automated dependency checkers compare your project's dependencies to publicly available vulnerability databases; these can be integrated at multiple points in your development process:

CI/CD: Tools like Snyk can scan your code for known vulnerabilities as part of your build pipeline. **Source Control**: GitHub's Dependabot will regularly scan your repository for vulnerabilities, and can be configured to automatically raise pull requests with fixes. **IDE** Tools like Vuln Cost can be installed directly in your IDE, providing real-time checking of dependencies *as you add them to your project*.

> *This should highlight the importance of a well structured and efficient automated deployment process. The longer it takes you to safely release an update to your dependencies, the longer your application could be exposed to a known vulnerability.*

Unfortunately, these tools cannot protect you from *unknown* vulnerabilities. The best approach here is to **minimise risk**. In general, you should remove any unused dependencies, or at least attempt to only import libraries you really need.

In some particularly security-sensitive situations, you may want to go further and actively minimise the amount of third-party code. For example, in a web application, the parts handling particularly sensitive data (passwords, credit card numbers) can be isolated into iFrames with no third-party code (or you can delegate handling this sensitive data to a specialist authentication provider or payment provider).

Companies with especially sensitive systems and with sufficient resources may choose to build their own tooling and rely on internal private package repositories rather than open-source libraries. This is a calculated risk: "rolling your own" deprives you of the benefit of 'battle tested' tools; we wouldn't normally recommend it, but it *might* be the correct choice for you particular circumstances.

Configuring third-party code

As well as security vulnerabilities in third-party code, you should also ensure that any resources you depend on are configured correctly.

The default settings in third-party software may not be optimised for maximum security in your use case: either if the default behaviour is insecure, or simply if unnecessary features are enabled, creating more opportunities for attackers.

You might also need to configure your dependencies differently in different environments.

> *Example: many web application frameworks offer the option to include a full stacktrace in an HTTP error response, showing the root cause of the error in the application code. This can be very helpful for debugging in development environments, but you should never expose stacktraces in public environments, as they might give an attacker helpful clues about the internals of your system.*
>
> *For instance, if Bob makes a request to `/users/alice/pets/cat` and sees an error response showing an authorisation error (rather than a 'not found' error), then Bob will know that Alice has a cat - an unintended leak of private information.*

To mitigate against vulnerabilities introduced by incorrectly configured dependencies, you should:

1. Disable any unused features, to keep the attack surface as small as possible for any malicious actors
2. Write tests or use scanners to verify that infrastructure and libraries are behaving securely
3. Use infrastructure as code tools to keep settings consistent between environments

Chapter 3

Common Exploits

In this chapter we'll look at some specific techniques that can be used to gain unauthorised access to a system.

Injection attacks

Many applications use user inputs as parameters to commands, for example using a user's search term as an input to a database query. Injection attacks exploit weaknesses in how these parameters are used to perform actions not originally intended by the developer.

SQL Injection

Perhaps the most famous example of this type of attack is **SQL Injection**. In this attack user inputs containing SQL keywords are crafted to alter the functionality of the underlying query.

By way of example, consider the following query to search for users by name:

```
SELECT * FROM users WHERE user_name = '<USER_INPUT>'
```

The user enters a search query which is substituted for `<USER_INPUT>` and then executed. Now, consider the impact of a malicious user entering the following search query: `' OR '1 = 1`. When substituted it produces the following query:

```
SELECT * FROM users WHERE user_name = '' OR '1 = 1'
```

Suddenly your search term is effectively ignored and every user in the database is returned. This was not what the developer intended! This vulnerability can be exploited to run arbitrary SQL, for example, the malicious input `'; DROP TABLE users; --` could let an attacker delete all your users!

```
SELECT * FROM users WHERE user_name = ''; DROP TABLE users; --'
```

SQL injection is best defended against by escaping user input *before* combining it with a command, many modern frameworks provide this functionality by default. Limiting the permissions of your applications database user would also help: if your application only need to *read* from the users table don't give your database user permission to drop the table!

Cross-site Scripting (XSS)

Cross-site scripting vulnerabilities occur when untrusted user input is rendered on a web page. This can allow an attacker to trick a user into including malicious scripts on an otherwise trusted website. By way of example, consider a search engine that includes a user's search term in the URL and renders it on the page.

```
www.example.com/?query=<SEARCH_TERM>
```

An attacker could trick a user into clicking a link to a URL that includes malicious JavaScript in the search term. If this isn't properly sanitised then that JavaScript would be executed in the user's browser, in the context of the trusted website. This allows the attacker to impersonate the user, accessing data without permission.

There are many flavours of XSS attack depending on how vulnerable the target system is. Broadly they fall into two distinct categories: **persistent**, where the malicious input is inadvertently stored in a database; and **reflected**, similar to the example above.

Defend against XSS by escaping user input or, more likely, using a framework which does this automatically.

> *Getting the system to perform an unintended action based on user input is a very common pattern for exploits. Several of the other attack types described here will share elements of this approach.*

Exploiting vulnerable language features

Many languages offer powerful features which can create vulnerabilities if employed incorrectly. We shall provide some examples to highlight how even seemingly innocuous features can provide attack vectors.

XXE - XML External Entities

XML is a very powerful mark-up language. One of its features allows data formats to be defined in multiple files, referenced by URI. These are **external entity definitions**. If a user can upload an XML file which contains a reference to external entity definitions, these can cause the XML parser to access files on its machine, access remote URLs (potentially uploading the contents of those files), or even execute arbitrary code in certain XML contexts.

Such attacks can be mitigated by using a simpler mark-up language (e.g. JSON) or ensuring that your XML parser is correctly configured to avoid this attack. Most up-to-date XML parsing libraries will default to disabling this behaviour.

Insecure deserialisation

When data is transmitted between systems it generally needs to be serialised to strings for transmission and deserialised into complex, native objects after reception. Vulnerable systems will allow input strings that can be deserialised in a way the developer was not intending; this may include changing the type of object that is created, tampering with its properties, or in some cases allow arbitrary code execution.

This applies to both a language's native serialisation format but also to common formats like JSON or XML. For example, in 2013 a bug in Ruby on Rails' XML parser allowed arbitrary code execution (see CVE-2013-0156).

Mitigate against this sort of attack by preferring to deserialise to a defined type and don't assume that serialised data from your client (e.g. cookies) is trustworthy.

File inclusion

This occurs where user input determines which application files are loaded in a scripted language. By modifying their input, an attacker may be able to execute arbitrary files on the server (local file inclusion), or even their own malicious code hosted elsewhere (remote file inclusion). Don't decide which files to execute based on user input.

Request forgery/proxying attacks:

These are attacks where the attacker tricks another user or a remote server into executing a malicious request on their behalf.

CSRF - Cross Site Request Forgery

This attack occurs when a user is tricked by an attacker into sending a malicious request to the application (using a link, or from a different site altogether). It takes advantages of the fact that browsers will forward cookies associated with a site automatically. If the user is logged in via a cookie, this will automatically be used to authenticate them.

For example, my malicious site but innocent looking site (`looks-innocent.example.com`) could make a request in the background to (`your-bank.com`). Because the request is coming from *your* browser, it would appear to be coming from you and could be acted upon without your knowledge.

These attacks can be mitigated against in multiple ways. Servers can generate unique, secret, unpredictable values called **CSRF tokens** and then reject any requests that do not include this token. This makes it harder for attacks to create a valid request. Modern browsers also support **Cross-origin resource sharing (CORS)** headers and will refuse to send a request from `looks-innocent.example.com` to `your-bank.com` unless the latter has explicitly approved the former.

> *CORS is controlled by the `Access-Control-Allow-Origin` header. Setting a value of `Access-Control-Allow-Origin: *` allows requests from **any** other domain. Avoid doing this!*

Common Vulnerabilities and Exposures

It is considered a common good that when vulnerabilities in software or systems are discovered that they are fixed and disclosed quickly. To facilitate this common database of vulnerabilities are maintained, making it easy to reference specific vulnerabilities. Perhaps the most famous example is CVE, references in the form `CVE-2013-0156` refer to this database and are widely used.

SSRF - Server Side Request Forgery

Server Side Request Forgery is where an application makes external HTTP requests to a URL supplied by an attacker. This is particularly problematic when the server has access to privileged resources (e.g. a database server or an internal corporate network) that a regular user does not does not.

SSRF can be mitigated by restricting what your application has access to and the permissions it have. Ideally it should only be able to access resources that are essential to its operation, no more. If you **need** to make requests to arbitrary user-supplied URLs (e.g. like `downdetector.co.uk`), do this in a context with minimal permissions so that no privileged access to internal resources is available.

Defending against exploits

We've discussed some mechanisms for dealing with specific exploits already. While modern browsers are increasingly good at protecting users, and cloud technology like **web application firewalls (WAFs)** can automatically detect many known exploits, this is not sufficient. As a general rule you should always be *extremely cautious* of inputs originating beyond your system.

Bad actors can be exceedingly creative and opportunistic. If you leave yourself exposed to a common exploitation, it will be found and exploited.

Buffer Overflows

A very well known exploit that does not appear in the OWASP Top 10 is the 'buffer overflow'. This occurs when data is written outside the bounds of a 'buffer' of temporary storage. For example, by writing to index 15 of a 10-element array. This can then overwrite other data in the temporary storage; for example the application's call stack, allowing execution of arbitrary code.

In practice, buffer overflows can be difficult to discover and exploit but they have been used in the wild a number of times - most famously by the Morris Worm, the first major internet attack.

More generally, buffer overflows belong to a class of attacks where data is read or written from outside the expected area. Many modern languages will perform automatic bounds-checking on memory access by default, to help guard against these kinds of attacks.

Chapter 4

Detecting Vulnerabilities

Threat modelling

To build a more secure system you first need to be aware of its potential insecurities. **Threat modelling** is an attempt to identify the kinds of threats that the application will be subject to and to assess the magnitude of the risk each of those threats poses. The goal of the exercise is to identify realistic threats to the system and appropriately prioritise fixes and/or mitigations.

STRIDE

Developed at Microsoft in the late 1990s **STRIDE** is both an approach to theatre modelling and a mnemonic for different categories of threats:

1. **Spoofing** - impersonating another user or application
2. **Tampering** - modifying input to execute an attack (e.g. the injection attacks from previous chapters)
3. **Repudiation** - performing an action that cannot definitively be traced to the attacker
4. **Information disclosure** - leak of sensitive data, a data breach
5. **Denial of service** - rendering the application unavailable
6. **Elevation of privilege** - gaining access to resources that would otherwise be protected

Assessing Risk

Once threats have been identified it is important that these risks are examined so that their relative importance can be understood. For each threat you should estimate its **likelihood**, the chance of it happening; and its **severity**, how bad it would be. This is very similar to how Health & Safety Risk Assessments assess risk.

> *The severity of a threat is dependant on the nature of your system: five minutes of unplanned downtime on a Friday evening would be inconsequential to many corporate websites; extremely embarrassing to the BBC; but exceedingly expensive to Just Eat.*

Penetration testing

The goal of penetration testing is to expose your system or application to some of the tools and techniques used by real attackers, in order to reveal vulnerabilities and establish how easily your system could be compromised. Consider a penetration test as a pre-authorised, *simulated* attack on your system.

The exact form of the penetration test will depend on the system you are testing and can be conducted both with or without visibility of the internals of that system. In some cases it may be appropriate to include the physical security of servers, back-ups or other resources in the scope of the tests.

> *While it is possible to conduct penetration testing against your own systems it is better to use a specialist third party, to enforce separation of duties.*

Automated tools

Various automated tools exist to help identify known classes of security problems. We already touched on these in Chapter 2 with dependency checking tools that run before the application is deployed. However, some problems can only become apparent in a deployed environment. The following tools help identify vulnerabilities after the code is deployed.

Network scanning

Computer networks are ubiquitous and there are billions of networked devices in existence. However, they are also a means of attack. Network scanning tools (e.g. nmap) can uncover a lot of information about devices on a network, including host addresses, open ports, and which operating systems are being used. This can expose vulnerabilities for attackers to exploit (or for system owners to secure).

Automated vulnerability scanning

DAST - Dynamic Application Security Testing

This is a technique where an automated tool attempts to find vulnerabilities in an application by performing standard attacks against its UI or API. This is a form of **black box** testing as the DAST tool has no access to (or knowledge of) the internals of the application. While DAST is similar to how an attacker might try to compromise your application, it will not detect logical issues in your application or draw any specific information from analysing the application code.

SAST - Static Application Security Testing

SAST tools scan an application's code to detect vulnerabilities. It is a form of **white box** testing and **static analysis**, a concept we covered earlier in the course. Because SAST scans the whole codebase it can provide much better coverage and can also be applied earlier in the software development life cycle, including in the IDE. It can, however, produce lots of **false positives**, and won't be able detect runtime issues or misconfigured integrations.

IAST - Interactive Application Security Testing

A form of **runtime analysis**, IAST tools are embedded inside an application to attempt to detect potential vulnerabilities when code is executed. Unlike DAST and SAST, these tools require installation and can have an impact on performance. They are typically only used within CI or QA environments to complement other automated or manual testing.

Clear or Opaque Testing

The terms **white box** and **black box** are used to different approaches to testing a system. The former describes situations where the internals are visible to the tester; the latter where the internals are not visible. There are benefits and drawbacks to both approaches. The term **grey box** can be used for scenarios where the tester has partial knowledge of the internals of the system, e.g. access to documentation but not the code.

Detecting attackers

We've discussed various various of detecting vulnerabilities in your system; however, this is not enough. If these vulnerabilities were exploited, would you know about it?

Preventing all attacks is ideal. However, if someone were attempting to attack your system or, worse, if you system were to be compromised, how would you know about it? It is important to be able to detect an attack quickly and respond appropriately. Being aware of an attack can let you assess the scale of the damage and take remedial action (e.g. resetting any leaked credentials); it also lets you identify and fix the vulnerability, to prevent repeated intrusions.

Without sufficient monitoring, a breach can go undetected for long periods of time - increasing the opportunity for the attacker to conduct malicious activity & extract sensitive data, and exacerbating the consequences of the attack. An undetected breach can also lead to further attacks in future using leaked credentials or elevated access.

To mitigate this, you should add automated alerting on suspicious activity to your system to ensure that any attack would not go undetected. DevOps-style monitoring practices can help to detect patterns of suspicious behaviour here. We'll cover these in more detail in a future module dedicated to logging and monitoring.

Chapter 5

Secure Development

As we said at the start of this module, security is a journey, not a destination. Any approach to secure development is about creating a culture and process where it is easy to create secure systems. In practice this means incorporating security into your development process as far "left" as possible: both in terms of project life cycle, and the software development life cycle.

Practices like threat modelling and risk assessment should be done before and during the development process, rather than afterwards. Vulnerability scanners and dependency checking can be built into IDEs and CI/CD pipelines; enabling known exploits to be avoided long before they could make it to production. Teams should incorporate security in their agile **definition of done**.

DevOps monitoring practices can be used to provide better security; logs can be automatically parsed to flag up suspicious behaviour, or alerts can be triggered when high-risk actions are taken.

> *Security: think about it early; think about it often.*

In this chapter we will cover aspects of best practice for secure development.

Standards of security

The approach to security you take will, in part, be determined by the nature of your project. The standards of security required to protect extremely sensitive financial or medical data are not necessarily the same standards required for a freemium browser based game.

Deciding your approach to security at the start of the project allows you to apply it consistently throughout the project. This allows your standards to be embedded in your development processes, your architecture and even the onboarding messaging for new team members. Setting up automated tooling at the start of your project can help detect issues early, potentially preventing painful remedial work later in the project.

Demonstrating security

> *How do you* know *your application is secure?*

Security is not just a technical and cultural challenge, it can be a political one as well. Many organisations have to demonstrate a certain standard of security to stakeholders: perhaps to meet regulatory requirements or to inspire confidence in their clients. Effectively demonstrating this can be a challenge. Establishing your security standards at the start of the project and agreeing them with key stakeholders can make this process significantly easier.

> *Ideally you should be working to industry recognised, "battle tested standards"; standards that have been widely used and demonstrated to work.*

For example, the UK's Open Banking Standards made use of the existing Open ID Connect standards for security, rather than creating their own.

Minimising Risk

All systems, no matter how thoroughly tested, are at risk of undetected vulnerabilities being discovered and exploited by attackers. This risk cannot be eliminated, but can be reduced and mitigated.

Reducing attack surface area

Every part of your system is potentially vulnerable to attack. The larger the system, the greater the risk that some part of it is vulnerable to attack. By reducing the size of the system, where possible, you will reduce the area available to be attacked.

Minimise the amount of code that is actually running; both by writing cleaner code but also ensuring your application only loads the dependencies it actually needs.

Consider eliminating low value features. Every bit of additional complexity increases the size of your application. If a feature is used infrequently or doesn't provide benefit consider removing it.

'0-trust' security

Traditional security models focus on securing a network perimeter, with an assumption that activities within the perimeter were safe. This can be more difficult to enforce in modern architectures, especially with cloud-based web apps where resources primarily exist outside the corporate network. The lack of **defence in depth** also means that if an attacker compromises one aspect (the perimeter), they have access to everything.

This has led to the '0-trust' model, which emphasises securing any access to resources, from within or outside the network. Any resource access should involve verifying the consumer's identity and checking whether they are authorised to view or access that resource. In addition, all consumers should be granted permissions according to the **least privilege security principle**. This means granting users the absolute minimum access necessary for the tasks they are going to perform. This mitigates against both compromised accounts but also the risk of a trusted individual secretly being a **bad actor**.

> *Organisations might proactively monitor particularly sensitive permissions, for example assuming any use of elevated administrator permissions is suspicious until and unless proven otherwise.*

Additional Material

HTTPS

We've talked about how HTTPS can make applications more secure - but what is it? An extension to HTTP, using the TLS protocol to encrypt traffic and authenticate identities. A trusted third-party ('certificate authority') signs a certificate validating that the holder is the owner of the domain used by the website. This certificate contains the domain and a public key associated with the website, which can be used to encrypt messages. The website uses the corresponding private key for to decrypt them.

When the user visits the site, their browser uses the third-party's signature to confirm the certificate is valid. It then generates a shared key, encrypts it using the website's public key, and sends it back to the website. The website uses its private key to decrypt this message and read the shared key. All further communication is encrypted using this shared key.

This achieves some important results. The third party signature means that websites cannot create their own certificates (technically they can, but browsers won't trust them by default). If a different website tries to reuse the certificate, they won't be able to communicate with the browser because they don't have the private key to decrypt the browser's messages. Finally, only the browser and website are able to read the shared key, so future messages are encrypted, preventing any third parties in between the two from reading the contents of the messages (a 'man-in-the-middle' attack).

Multi-Tenancy

Multi-tenanted applications are applications that support multiple customers with a single instance of the application. Most SaaS platforms are multi-tenanted. This makes release and environment management across multiple customers much simpler, as only a single copy of the application must be deployed. However, multi-tenanted applications must be carefully structured to ensure that users from one customer can only see data relating to that customer.

Identity and security in web apps

For a web application (software running in a web browser) and communicating with a back-end (server), it is effectively essential to be able to confirm a user's identity once (e.g. when they log in), and reuse that identity between HTTP requests. (This prevents the user from having to enter their password on every request, or the app having to store the password insecurely and send it with every request).

Session cookies

The traditional approach to the problem of "logging in" to a web app and reusing the authenticated identity is based on **sessions** and **cookies**. The user logs in, sending their authentication credentials to the server for validation. If the authentication is valid, then the server creates and stores a record of this *session* with a unique identifier. This unique id is communicated to the client by including a `Set-Cookie` header in the response to the client, setting a cookie in the user's browser which contains the unique identifier for the session.

For each subsequent HTTP request made by the web app to its server the browser automatically sends the authentication cookie, for any request to the same server hostname. The server verifies that the session identifier in the cookie matches an active session. When the user logs out, the cookie is unset and the session is invalidated on the server.

It is good practice also to expire sessions: the server will invalidate the session after a certain amount of time, so that another person using the same browser at a later time cannot gain access to the previously authenticated user's account.

However, this approach has a major disadvantage in that the server must store a record of all active sessions and check this data store for every request; this can make it hard to scale to high numbers of concurrent authenticated users without degrading performance.

Signed tokens

To solve this problem, instead of storing the session data server side and using the session cookie just as an identifier of the session, the cookie may be used to store the required session data itself.

One approach to this is using **signed tokens**. When the user authenticates, the server generates a token which contains information

about the user's identity; the user's roles and permissions; and an expiry time for the token. The server has a private key which it uses to **sign** the token (i.e. it adds a signature which is effectively the token data encrypted using an asymmetric cryptography algorithm and the server's private key). This proves that the token originates from the server and has not been tampered with.

The server returns the signed token to the client (either as a cookie, or to be stored in memory by the application). In all subsequent requests the browser sends the signed token back to the server, which verifies that the token is valid (i.e. the signature is correct - proving that the data in the token has not been altered), and it is not past the token's expiry time.

This has several advantages. There is no need for the server to store data about active sessions (sessions are *stateless* from the server's point of view). Logging out can be done purely in the client application, by deleting the session token; this means a user can still log out even if they have no Internet connection (unlike when the session data is stored server-side, and an HTTP request is needed to invalidate the session). Because the token is signed using asymmetric cryptography, it can be verified by any application with the public key. This means the token can be reused across different backends. For example, a web application can interact with multiple different APIs at different origins. One backend acts as authentication provider and signs the token. The web application sends the token in a request to another API, which uses the public key to verify the token. Alternatively, an application provider creates and signs a token which can be reused across multiple different applications which use the public key to verify the claimed information in the token.

This approach is typically used for **Single Page Applications (SPAs)**. SPAs typically use a RESTful API design: the server does not store any information about the state of the client (e.g. what page the user is on). In each request, the client sends all the information required for the server to process the request. The signed token provides a secure way for the client to include information about authentication & authorisation in the request, maintaining the stateless design.

The most common standard for implementing this mechanism is using **signed Json Web Tokens (signed JWTs)**.

Further Reading

The OWASP Top 10 has further details on many of the vulnerabilities described here:

- https://owasp.org/www-project-top-ten/

MITRE also publishes a Top 25 threats list:

- https://cwe.mitre.org/top25/archive/2020/2020_cwe_top25.html
- These tend to be a little more specific than the OWASP equivalents (e.g. different types of injection attack are counted separately).

Some guides on approaches to threat modelling

- https://securityintelligence.com/posts/a-guide-to-easy-and-effective-threat-modeling/
- https://martinfowler.com/articles/agile-threat-modelling.html

OWASP maintain a list of tools useful for testing for different vulnerabilities:

- https://owasp.org/www-project-web-security-testing-guide/stable/6-Appendix/A-Testing_Tools_Resource.html

Single Page Apps

Single Page Apps (SPAs) are web applications that exist entirely on one page, using code running in the browser and HTTP requests to rewrite the page as the user navigates and performs actions. This allows them to provide a rich UX closer to that of a desktop or native mobile application. Downsides can include longer initial load times, and poorer accessibility

The application frontend is usually written in a JavaScript framework, such as Angular or React. There are different models for these, but some typical features they provide are: - Templating, where code can be embedded in HTML defining the structure of the page - Data binding - either 'one-way', where the display is updated based on the value of a field in code, or 'two-way', where the display can also update the value in code - Routing, where different code is run and different HTML is rendered depend on the URL

As WebAssembly (a new low-level browser language designed to be easy for other languages to compile down to) has been adopted by major browsers, new frameworks in other languages have also begun to emerge - e.g. Blazor (C#) and Yew (Rust).