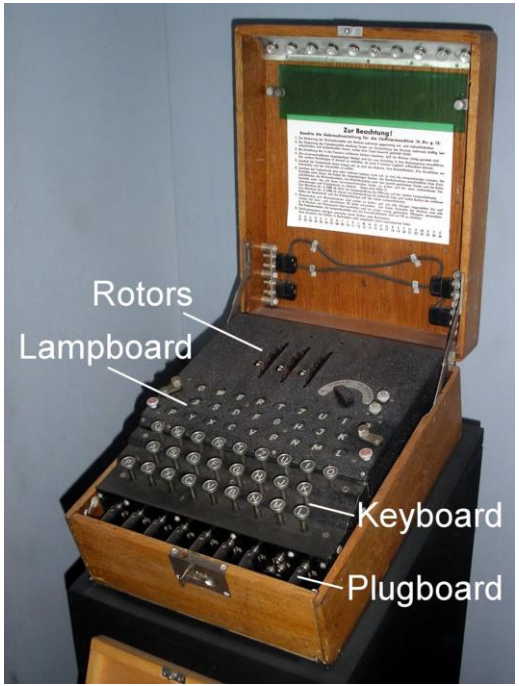
KonvergeAI

CCT 310 - 3 Application Security

UNIT 1 Introduction to Application Security

The History of Software Security

1910s and 1920s, most of which involved tampering with Morse code senders and receivers, or interfering with the transmission of radio waves. However, while these events did occur, they were not common, and it is difficult to pinpoint large-scale operations that were interrupted as a result of the abuse of these technologies.



The type of encryption these Enigma machines used is now known as a symmetric key algorithm, which is a special type of cipher that allows for the encryption and decryption of a message using a single cryptographic key. This family of encryption is still used today in software to secure data in transit (between sender and receiver), but with many improvements on the classic model that gained popularity with the Enigma machine.

If you trace the lineage of modern cryptography far back, you will eventually reach World War II in the 1930s. It's safe to say that the Enigma machine was a major milestone in securing remote communications. From this, we can conclude that the Enigma machine was an essential development in what would later become the field of software security.

The Enigma machine was also an important technological development for those who would be eventually known as “hackers.” The adoption of Enigma machines by the Axis Powers during World War II resulted in extreme pressure for the Allies to develop encryption-breaking techniques. General Dwight D. Eisenhower himself claimed that doing so would be essential for victory against the Nazis.

In September of 1932, a Polish mathematician named Marian Rejewski was provided a stolen Enigma machine. At the same time, a French spy named Hans-Thilo Schmidt was able to provide him with valid configurations for September and October of 1932. This allowed Marian to intercept messages from which he could begin to analyze the mystery of Enigma machine encryption.

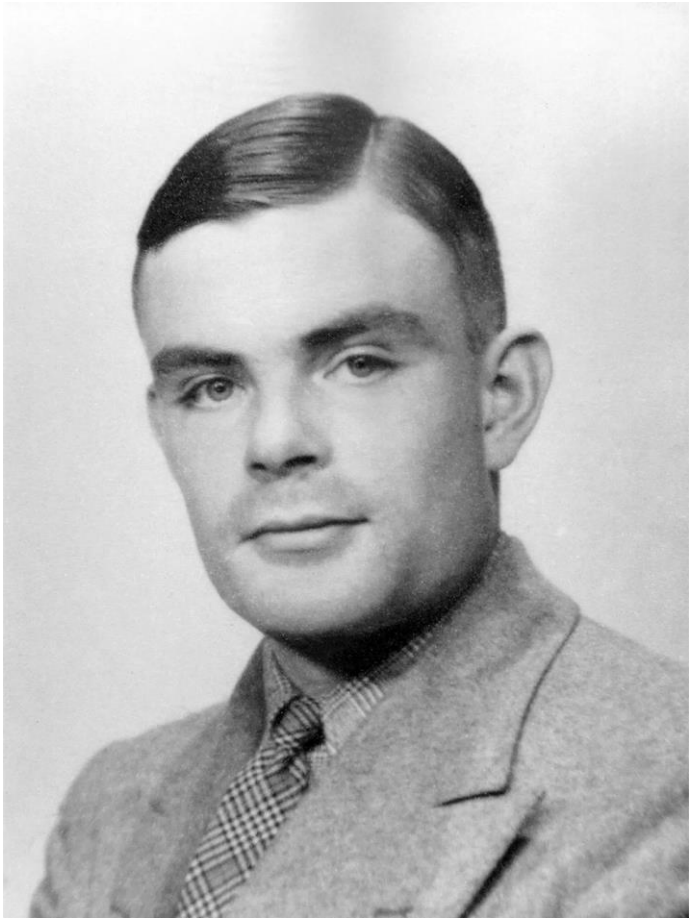
Marian was attempting to determine how the machine worked, both mechanically and mathematically. He wanted to understand how a specific configuration of the machine's hardware could result in an entirely different encrypted message being output.

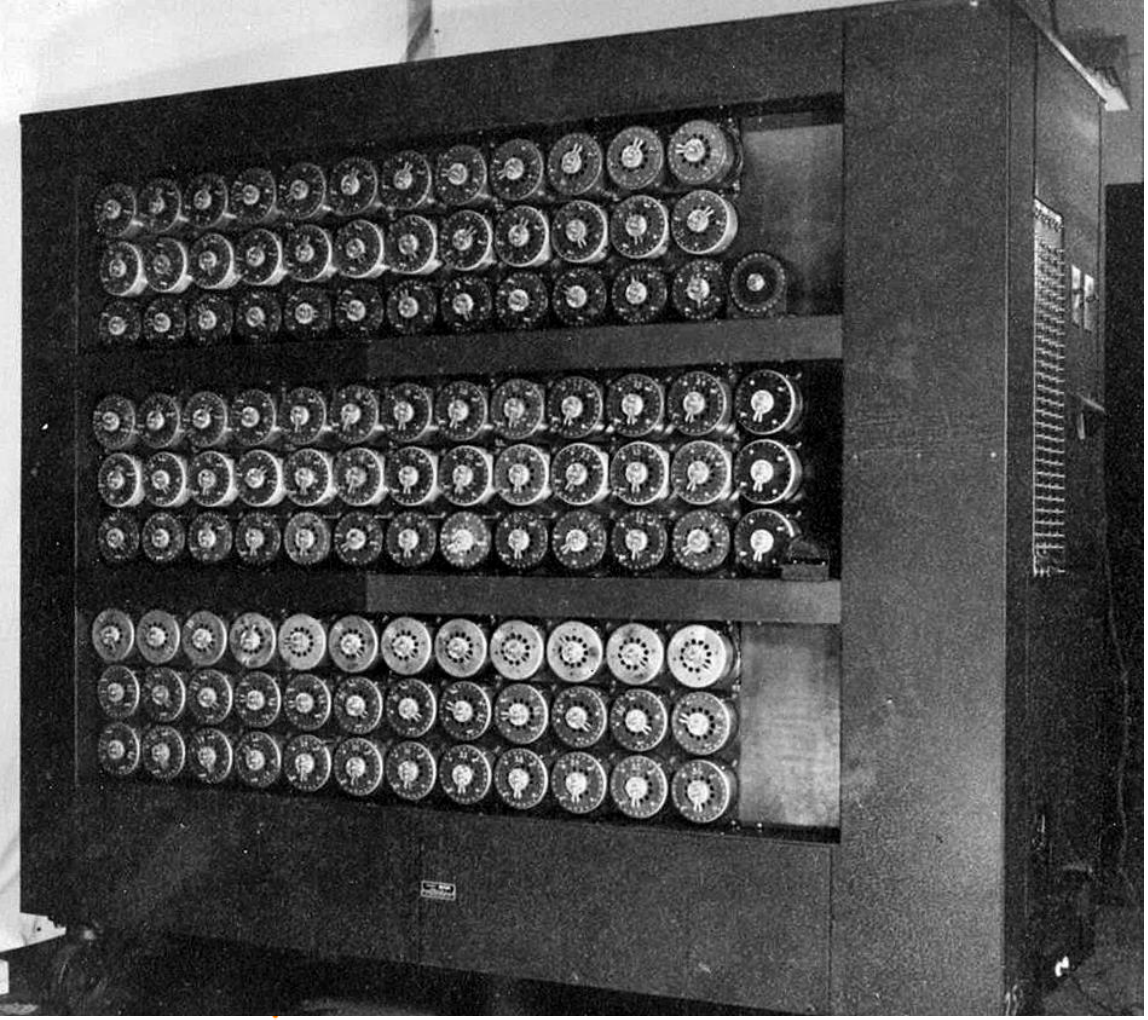
Marian's attempted decryption was based on a number of theories as to what machine configuration would lead to a particular output. By analyzing patterns in the encrypted messages and coming up with theories based on the mechanics of the machine, Marian and two coworkers, Jerzy Różycki and Henryk Zygalski, eventually reverse engineered the system. With the deep understanding of Enigma rotor mechanics and board configuration that the team developed, they were able to make educated guesses as to which configurations would result in which encryption patterns. They could then reconfigure a board with reasonable accuracy and, after several attempts, begin reading encrypted radio traffic. By 1933 the team was intercepting and decrypting Enigma machine traffic on a daily basis.

Much like the hackers of today, Marian and his team intercepted and reverse engineered encryption schemes to get access to valuable data generated by a source other than themselves. For these reasons, I would consider Marian Rejewski and the team assisting him as some of the world's earliest hackers.

In the following years, Germany would continually increase the complexity of its Enigma machine encryption. This was done by gradually increasing the number of rotors required to encrypt a character. Eventually the complexity of reverse engineering a configuration would become too difficult for Marian's team to break in a reasonable time frame. This development was also important, because it provided a look into the ever-evolving relationship between hackers and those who try to prevent hacking.

Marian's team was able to find patterns in the encryption that allowed them to make educated guesses regarding a machine's configuration. But this was not scalable now that the number of rotors in the machine had increased as much as tenfold. In the amount of time required to try all of the potential combinations, a new configuration would have already been issued. Because of this, Alan and Dilly were looking for a different type of solution; a solution that would scale and that could be used to break new types of encryption. They wanted a general-purpose solution, rather than a highly specialized one.





The first bombes were built by the Polish, in an attempt to automate Marian's work. Unfortunately, these devices were designed to determine the configuration of Enigma machines with very specific hardware. In particular, they were ineffective against machines with more than three rotors. Because the Polish bombe could not scale against the development of more complex Enigma machines, the Polish cryptographers eventually went back to using manual methods for attempting to decipher German wartime messages.

Alan Turing believed that the original machines failed because they were not written in a general-purpose manner. To develop a machine that could decipher any Enigma configuration (regardless of the number of rotors), he began with a simple assumption: in order to properly design an algorithm to decrypt an encrypted message, you must first know a word or phrase that exists within that message and its position.

Fortunately for Alan, the German military had very strict communication standards. Each day, a message was sent over encrypted Enigma radio waves containing a detailed regional weather report. This is how the German military ensured that all units knew the weather conditions without sharing them publicly to anyone listening on the radio. The Germans did not know that Alan's team would be able to reverse engineer the purpose and position of these reports.

Knowing the inputs (weather data) being sent through a properly configured Enigma machine made algorithmically determining the outputs much easier. Alan used this newfound knowledge to determine a bombe configuration that could work independently of the number of rotors that the Enigma machine it was attempting to crack relied on.

Alan requested a budget to build a bombe that would accurately detect the configuration requirements needed to intercept and read encrypted messages from German Enigma machines. Once the budget was approved, Alan constructed a bombe composed of 108 drums that could rotate as fast as 120 RPM. This machine could run through nearly 20,000 possible Enigma machine configurations in just 20 minutes. This meant that any new configuration could be rapidly compromised. Enigma encryption was no longer a secure means of communication.

Today Alan's reverse-engineering strategy is known as a known plaintext attack or KPA(Known-plaintext attack). It's an algorithm that is made much more efficient by being provided with prior input/output data. Similar techniques are used by modern hackers to break encryption on data stored or used in software. The machine Alan built marked an important point in history, as it was one of the first automated hacking tools ever built.

In the late 1950s, telecoms like AT&T began implementing new phones that could be automatically routed to a destination number based on audio signals emitted from the phone unit. Pressing a key on the phone pad emitted a specific audio frequency that was transmitted over the line and interpreted by a machine in a switching center. A switching machine translated these sounds into numbers and routed the call to the appropriate receiver.

This system was known as tone dialing, and was an essential development that telephone networks at scale could not function without. Tone dialing dramatically reduced the overhead of running a telephone network, since the network no longer needed an operator to manually connect every call. Instead, one operator overseeing a network for issues could manage hundreds of calls in the same time as one call previously took.

Within a short period of time, small groups of people began to realize that any systems built on top of the interpretation of audio tones could be easily manipulated. Simply learning how to reproduce identical audio frequencies next to the telephone receiver could interfere with the intended functionality of the device. Hobbyists who experimented with manipulating this technology eventually became known as phreakers—an early type of hacker specializing in breaking or manipulating telephone networks. The true origin of the term phreaking is not known, though it has several generally accepted possible origins. It is most often thought to be derived from two words, “freaking” and “phone.”

There is an alternative suggested derivation that I believe makes more sense. I believe that the term phreaking originated from “audio frequency” in response to the audio signaling languages that phones of the time used. I believe this explanation makes more sense since the origin of the term is very close chronologically to the release of AT&T’s original tone dialing system. Prior to tone dialing, telephone calls were much more difficult to tamper with because each call required an operator to connect the two lines.

We can trace phreaking back to several events, but the most notorious case of early phreaking was the discovery and utilization of the 2600 Hz tone. A 2600 Hz audio frequency was used internally by AT&T to signal that a call had ended. It was essentially an “admin command” built into the original tone dialing system. Emitting a 2600 Hz tone stopped a telecom switching system from realizing that a call was still open (logged the call as ended, although it was still ongoing). This allowed expensive international calls to be placed without a bill being recorded or sent to the caller.

The discovery of the 2600 Hz tone is often attributed to two events. First, a young boy named Joe Engressia was known to have a whistling pitch of 2600 Hz and would reportedly show off to his friends by whistling a tone that could prevent phones from dialing. Some consider Joe to be one of the original phone phreakers, although his discovery came by accident.

The first of these hardware devices was known as a blue box. Blue boxes played a nearly perfect 2600 Hz signal, allowing anyone who owned one to take advantage of the free calling bug inherent in telecom switching systems. Blue boxes were only the beginning of automated phreaking hardware, as later generations of phreakers would go on to tamper with pay phones, prevent billing cycles from starting without using a 2600 Hz signal, emulate military communication signals, and even fake caller ID.

From this we can see that architects of early telephone networks only considered normal people and their communication goals. In the software world of today, this is known as “best-case scenario” design. Designing based off of this was a fatal flaw, but it would become an important lesson that is still relevant today: always consider the worst-case scenario first when designing complex systems.

Eventually, knowledge of weaknesses inherit in tone dialing systems became more widely known, which led to budgets being allocated to develop countermeasures to protect telecom profits and call integrity against phreakers.

In the 1960s, phones were equipped with a new technology known as dual-tone multifrequency (DTMF) signaling. DTMF was an audio-based signaling language developed by Bell Systems and patented under the more commonly known trademark, “Touch Tones.” DTMF was intrinsically tied to the phone dial layout we know today that consists of three columns and four rows of numbers. Each key on a DTMF phone emitted two very specific audio frequencies, versus a single frequency like the original tone dialing systems.

This table represents the “Touch Tones,” or sounds, (in hertz) that older telephones made on keypress:

1	2	3	**(697 Hz)**
4	5	6	**(770 Hz)**
7	8	9	**(852 Hz)**
*	0	#	**(941 Hz)**
(1209 Hz)	**(1336 Hz)**	**(1477 Hz)**	

The development of DTMF was due largely to the fact that phreakers were taking advantage of tone dialing systems because of how easy those systems were to reverse engineer. Bell Systems believed that because the DTMF system used two very different tones at the same time, it would be much more difficult for a malicious actor to take advantage of it.

DTMF tones could not be easily replicated by a human voice or a whistle, which meant the technology was significantly more secure than its predecessor. DTMF was a prime example of a successful security development introduced to combat phreakers, the hackers of that era.

The mechanics of DTMF tones are generated are pretty simple. Behind each key is a switch that signals to an internal speaker to emit two frequencies: one frequency based on the row of the key and one frequency based on the column. Hence the use of the term dual-tone.

In 1983, Fred Cohen, an American computer scientist, created the very first computer virus. The virus he wrote was capable of making copies of itself and was easily spread from one personal computer to another via floppy disk. He was able to store the virus inside a legitimate program, masking it from anyone who did not have source code access. Fred Cohen later became known as a pioneer in software security, demonstrating that detecting viruses from valid software with algorithms was almost

1986 ?????

```
Welcome to the Dungeon (c) 1986 Amjads (pvt) Ltd VIRUS_SHOE RECORD V9.0 Dedicated to
the dynamic memories of millions of viruses who are no longer with us today - Thanks
GOODNESS!!! BEWARE OF THE er..VIRUS : this program is catching program follows after
these ....$#@%$@!!
```

A few years later, in 1988, another American computer scientist named Robert Morris was the first person to ever deploy a virus that infected computers outside of a research lab. The virus became known as the Morris Worm, with “worm” being a new phrase used to describe a self-replicating computer virus. The Morris Worm spread to about 15,000 network-attached computers within the first day of its release.

For the first time in history, the US government stepped in to consider official regulations against hacking. The US Government Accountability Office (GAO) estimated the damage caused by this virus at $10,000,000. Robert received three years of probation, four hundred hours of community service, and a fine of $10,050. This would make him the first convicted hacker in the United States.

These days, most hackers do not build viruses that infect operating systems, but instead target web browsers. Modern browsers provide extremely robust sandboxing that makes it difficult for a website to run executable code outside of the browser (against the host operating system) without explicit user permission.

Although hackers today are primarily targeting users and data that can be accessed via web browser, there are many similarities to hackers that targeted the OS. Scalability (jumping from one user to another) and camouflaging (hiding malicious code inside of a legitimate program) are techniques employed by attacks against web browsers.

Today, attacks often scale by distribution through email, social media, or instant messaging. Some hackers even build up legitimate networks of real websites to promote a single malicious website.

Oftentimes, malicious code is hidden behind a legitimate-looking interface. Phishing (credential stealing) attacks occur on websites that look and feel identical to social media or banking sites. Browser plug-ins are frequently caught stealing data, and sometimes hackers even find ways to run their own code on websites they do not own.

Figure 1-4. Apple.com website, July 1997; the data presented is purely informational and a user cannot sign up, sign in, comment, or persist any data from one session to another

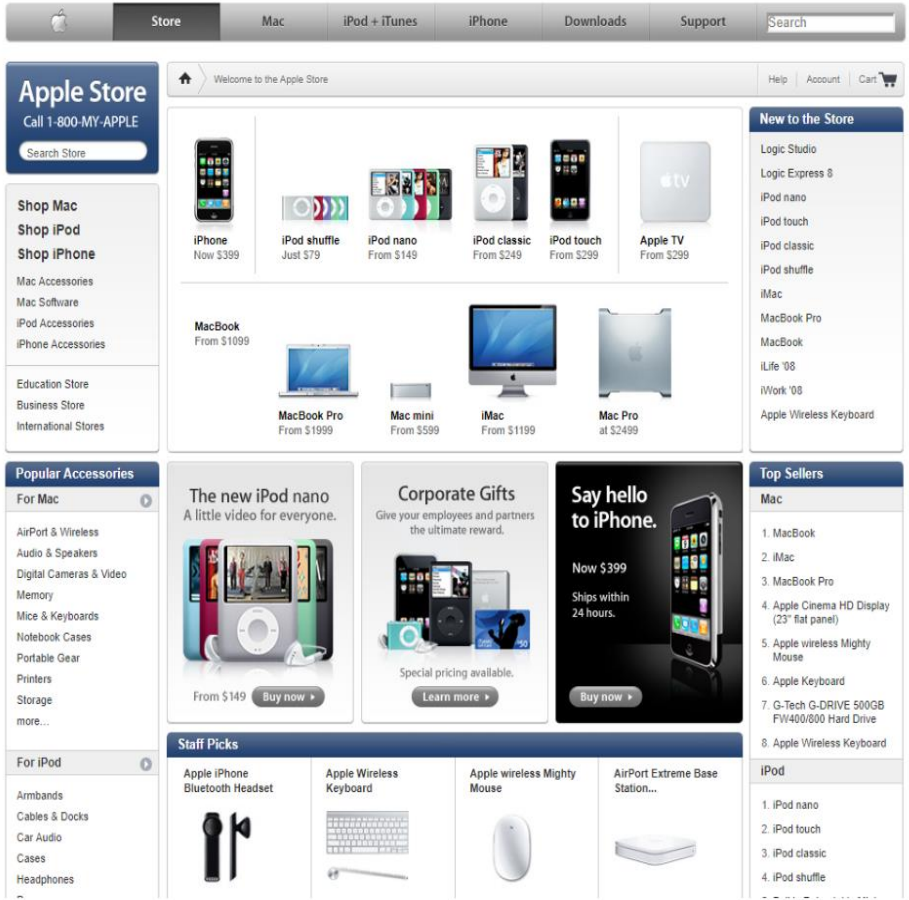


Figure 1-5. Apple.com, October 2007, showing a storefront with items that can be purchased online

In the early 2000s, the first largely publicized denial of service (DoS) attacks shut down Yahoo!, Amazon, eBay, and other popular sites. In 2002, Microsoft's ActiveX plug-in for browsers ended up with a vulnerability that allowed remote file uploads and downloads to be invoked by a website with malicious intentions. By the mid-2000s, hackers were regularly utilizing "phishing" websites to steal credentials. No controls were in place at the time to protect users against these websites.

Cross-Site Scripting (XSS) vulnerabilities that allowed a hacker's code to run in a user's browser session inside of a legitimate website ran rampant throughout the web during this time, as browser vendors had not yet built defenses for such attacks. Many of the hacking attempts of the 2000s came as a result of the technology driving the web being designed for a single user (the website owner). These technologies would topple when used to build a system that allowed the sharing of data between many users.

Introduction to Types of Modern-day Applications

Read and Discuss

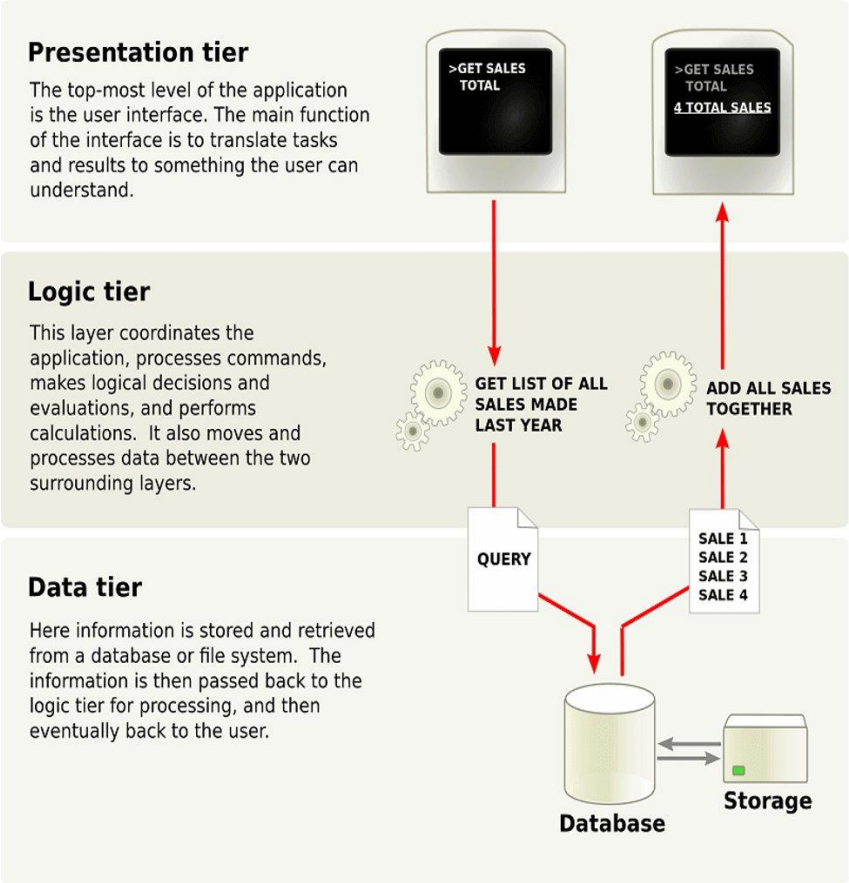
1. https://www.techtarget.com/searchcio/feature/The-rise-of-modern-applications-Why-you-need-them
2. https://azure.microsoft.com/en-us/solutions/modern-application-development/#customer-stories
3. https://www.ibm.com/blogs/cloud-computing/2018/06/20/modern-app-microservices/
4. https://octo.vmware.com/defining-modern-application/

Application Architecture

What is application architecture?

Application architecture is a structural map that provides a guide for how to assemble software applications. This system defines how apps interact with one another to meet a client's needs. This structure comprises software modules, components, systems and the various interactions among them.

Types of Web Application Architectures



3 Tier / N Tier Layered Architecture

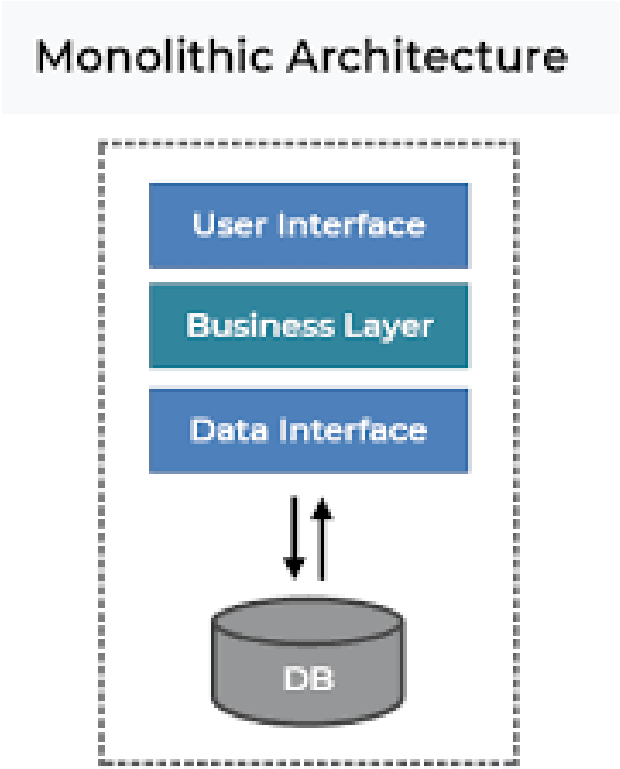
A layered or N-tier architecture is a traditional architecture often used to build on-premise and enterprise apps and is frequently associated with legacy apps.

In a layered architecture, there are several layers or tiers, often 3, but there can be more, that make up the application, each with their own responsibility.

Layers help to manage dependencies and perform logical functions. In a layered architecture, the layers are arranged horizontally, so they are only able to call into a layer below.

A layer can either only call into the layer immediately below it, or it can call into any of the layers below it.

Monolithic Architecture



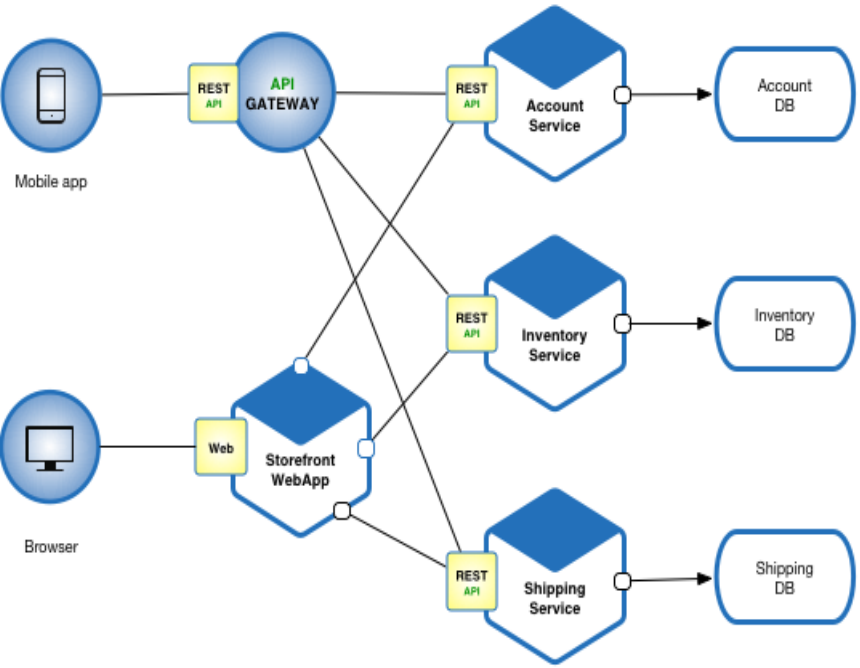
A monolith, another architecture type associated with legacy systems, is a single application stack that contains all functionality within that 1 application. This is tightly coupled, both in the interaction between the services and how they are developed and delivered.

Updating or scaling a single aspect of a monolithic application has implications for the entire application and its underlying infrastructure.

A single change to the application code requires the whole application to be re-released. Because of this, updates and new releases typically can only occur once or twice per year, and may only include general maintenance instead of new features.

In contrast, more modern architectures try to break out services by functionality or business capabilities to provide more agility.

Microservices Architecture



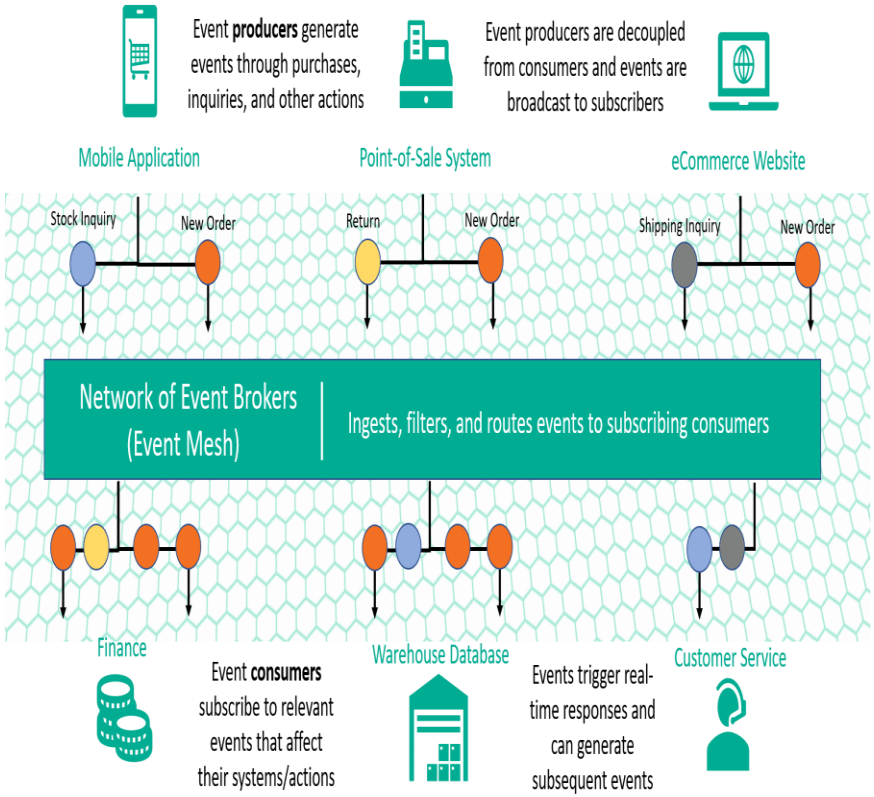
Microservices are both an architecture and an approach to writing software. With microservices, apps are broken down into their smallest components, independent from each other. Each of these components, or processes, is a microservice. Microservices are distributed and loosely coupled, so they don't impact one another. This has benefits for both dynamic scalability and fault tolerance: individual services can be scaled as needed without requiring heavy infrastructure or can failover without impacting other services.

The goal of using a microservices architecture is to deliver quality software faster. You can develop multiple microservices concurrently. And because services are deployed independently, you don't have to rebuild or redeploy the entire app when changes are made.

This leads to more developers working on their individual services at the same time, instead of updating the whole application, resulting in less time spent in development and the ability to release new features more often.

Along with APIs and DevOps teams, containerized microservices are the foundation for cloud-native applications.

Event Driven Architecture



With an event-driven system, the capture, communication, processing, and persistence of events are the core structure of the solution. This differs from a traditional request-driven model.

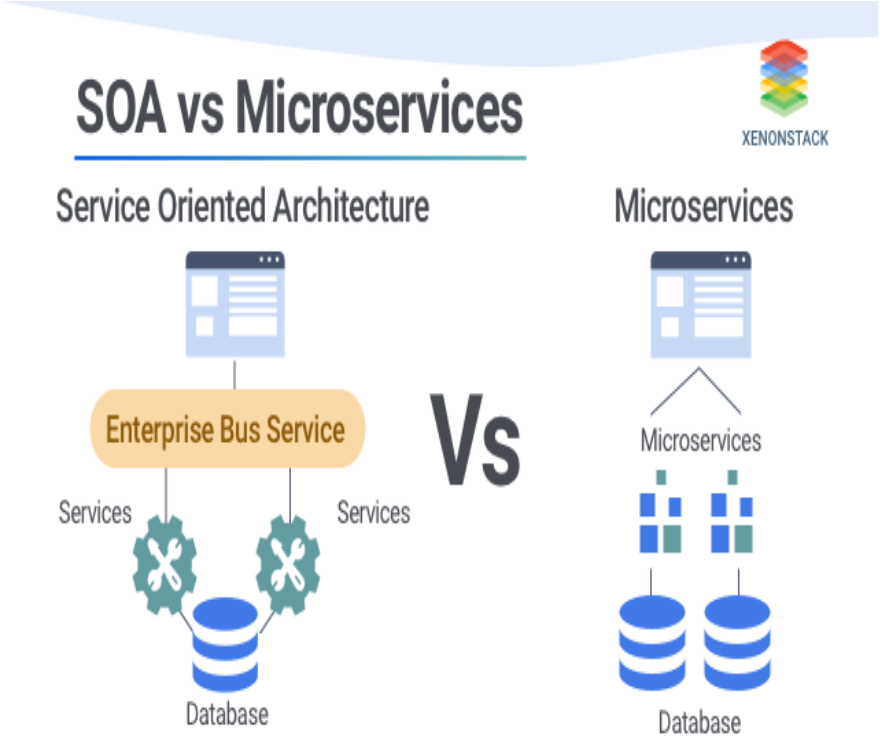
An event is any significant occurrence or change in state for system hardware or software. The source of an event can be from internal or external inputs.

Event-driven architecture enables minimal coupling, which makes it a good option for modern, distributed application architectures.

Event-driven architecture is made up of event producers and event consumers. An event producer detects or senses an event and represents the event as a message. It does not know the consumer of the event, or the outcome of an event.

After an event has been detected, it is transmitted from the event producer to the event consumers through event channels, where an event processing platform processes the event asynchronously.

Service Oriented Architecture



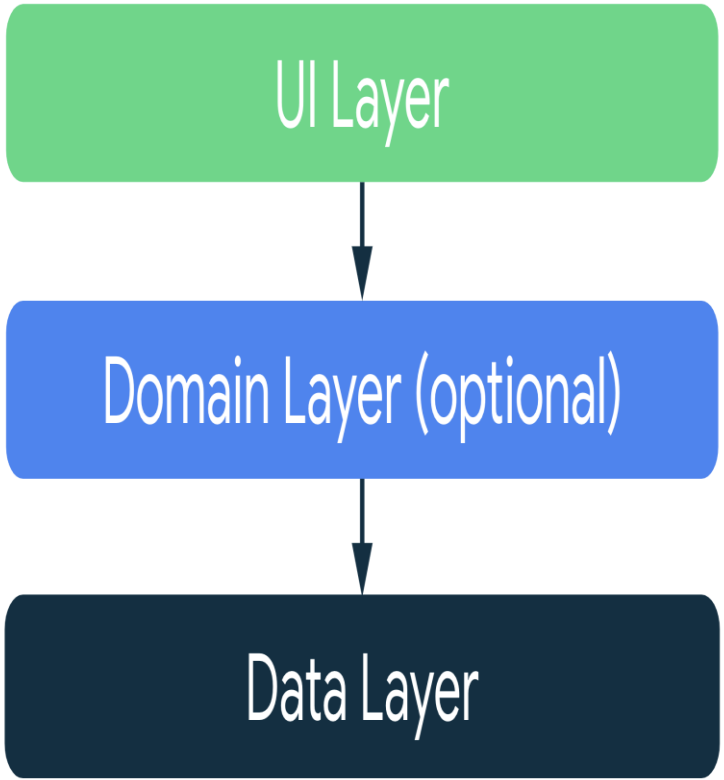
The service-oriented architecture (SOA) is a well-established style of software design, that is similar to the microservices architecture style.

SOA structures apps into discrete, reusable services that communicate through an enterprise service bus (ESB).

In this architecture, individual services, each organized around a specific business process, adhere to a communication protocol (like SOAP, ActiveMQ, or Apache Thrift) and expose themselves through the platform of an ESB. Taken together, this suite of services, integrated through an ESB is used by a front-end application to provide value to a business or customer.

Android App Architecture

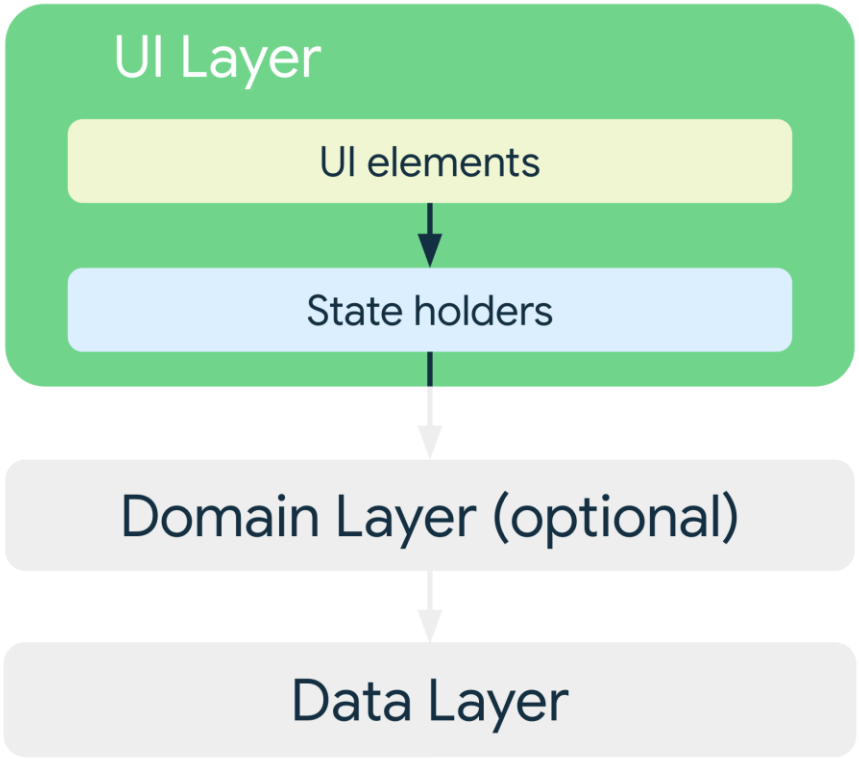
https://github.com/android/architecture-templates/tree/base



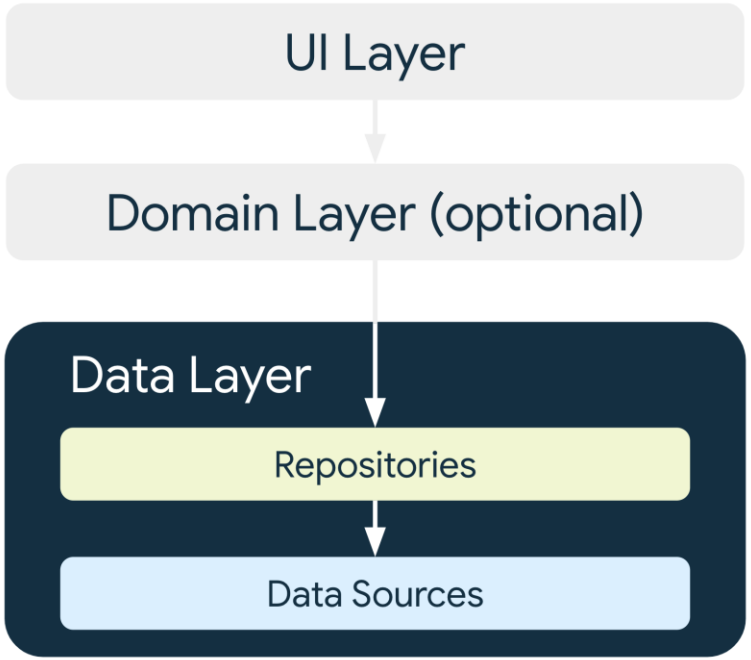
The *UI layer* that displays application data on the screen.

The *data layer* that contains the business logic of your app and exposes application data.

You can add an additional layer called the *domain layer* to simplify and reuse the interactions between the UI and data layers.

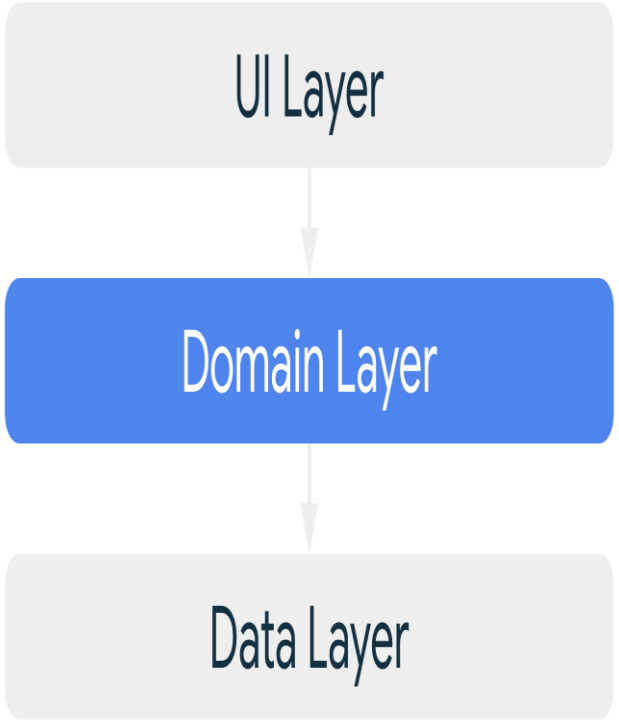


The role of the UI layer (or *presentation layer*) is to display the application data on the screen. Whenever the data changes, either due to user interaction (such as pressing a button) or external input (such as a network response), the UI should update to reflect the changes.
The UI layer is made up of two things:
UI elements that render the data on the screen. You build these elements using Views or Jetpack Compose functions.
State holders (such as ViewModel classes) that hold data, expose it to the UI, and handle logic.



The data layer of an app contains the *business logic*. The business logic is what gives value to your app—it's made of rules that determine how your app creates, stores, and changes data.
The data layer is made of *repositories* that each can contain zero to many *data sources*. You should create a repository class for each different type of data you handle in your app. For example, you might create
a MoviesRepository class for data related to movies, or
a PaymentsRepository class for data related to payments.

Repository classes are responsible for the following tasks:
•Exposing data to the rest of the app.
•Centralizing changes to the data.
•Resolving conflicts between multiple data sources.
•Abstracting sources of data from the rest of the app.
•Containing business logic.
Each data source class should have the responsibility of working with only one source of data, which can be a file, a network source, or a local database. Data source classes are the bridge between the application and the system for data operations.

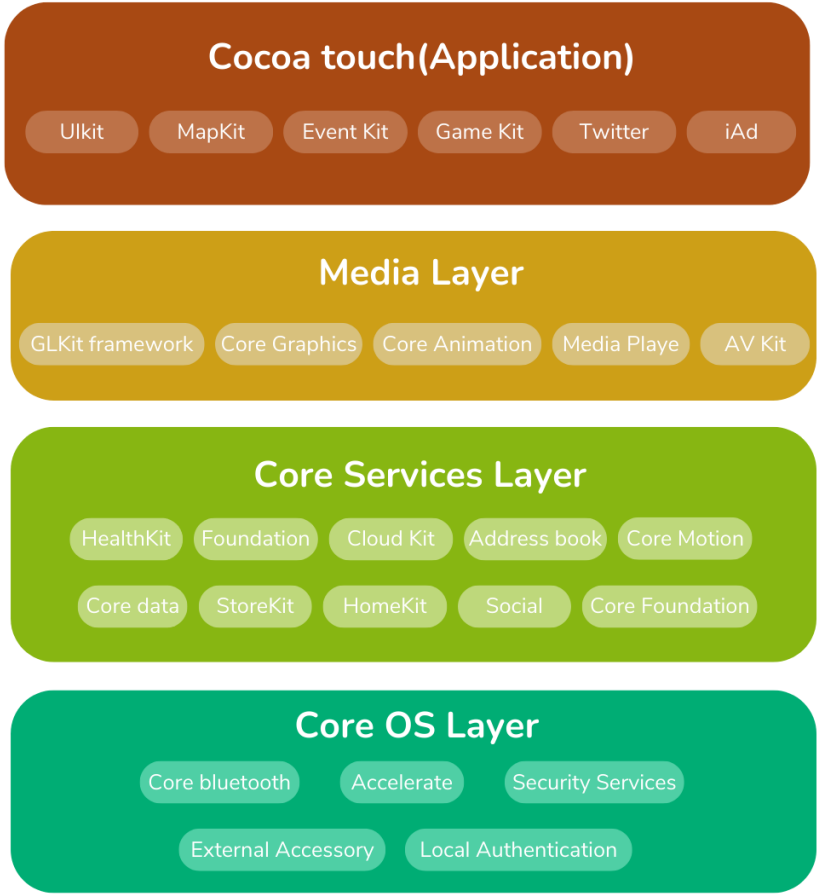


The domain layer is an optional layer that sits between the UI and data layers.

The domain layer is responsible for encapsulating complex business logic, or simple business logic that is reused by multiple ViewModels. This layer is optional because not all apps will have these requirements. You should use it only when needed—for example, to handle complexity or favor reusability.

Classes in this layer are commonly called *use cases* or *interactors*. Each use case should have responsibility over a *single* functionality. For example, your app could have a `GetTimeZoneUseCase` class if multiple ViewModels rely on time zones to display the proper message on the screen.

IOS Application Archietcture

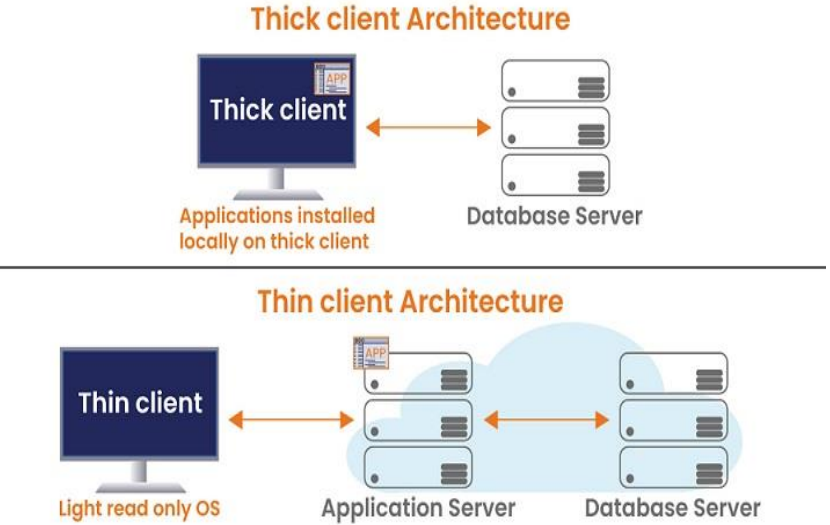


Architecture of IOS is a layered architecture. At the uppermost level iOS works as an intermediary between the underlying hardware and the apps you make. Apps do not communicate to the underlying hardware directly.

Apps talk with the hardware through a collection of well-defined system interfaces. These interfaces make it simple to write apps that work constantly on devices having various hardware abilities.

Lower layers gives the basic services which all application relies on and higher-level layer gives sophisticated graphics and interface related services. Apple provides most of its system interfaces in special packages called frameworks. A framework is a directory that holds a dynamic shared library that is .a files, related resources like as header files, images, and helper apps required to support that library. Every layer have a set of Framework which the developer use to construct the applications.

Thin Client Apps



Thin clients work remotely in environments where most applications and sensitive data reside on servers rather than on local machines. Thin clients access powerful servers with the memory and storage space needed to run applications and internal computing tasks. The purpose of a thin client device is to act as a virtual desktop and utilize the processing power of a networked server, which can be a local or cloud-based system.

Thin clients are an excellent choice for the perfect balance of performance and portability. The idea of thin clients is to limit computing power to necessary applications. Additionally, machine learning solutions can help companies optimize resources by analyzing data across their networks in real-time. Thin clients are easier to control, easier to guard against security risks, and reduce maintenance and licensing costs.

Thin client applications are web-based programs that do not require client installation. It is primarily a gateway to the network. Thin clients are also suitable for the smallest workloads, as they cannot handle large amounts of data processing. The most typical thin client we see today is a web browse

Application Testing Methodologies

Functional Testing

Functional testing involves testing the application against the business requirements.

Unit testing - Unit testing is the first level of testing and is often performed by the developers themselves. It is the process of ensuring individual components of a piece of software at the code level are functional and work as they were designed to.

Integration testing - After each unit is thoroughly tested, it is integrated with other units to create modules or components that are designed to perform specific tasks or activities. These are then tested as group through integration testing to ensure whole segments of an application behave as expected (i.e, the interactions between units are seamless).

Non- Functional Testing

Non-functional testing methods incorporate all test types focused on the operational aspects of a piece of software.

Performance testing
Security testing
Usability testing
Compatibility testing

Functional Testing

System testing - System testing is a black box testing method used to evaluate the completed and integrated system, as a whole, to ensure it meets specified requirements

Acceptance testing - Acceptance testing is the last phase of functional testing and is used to assess whether or not the final piece of software is ready for delivery. I

Performance Testing

- **Load testing** is the process of putting increasing amounts of simulated demand on your software, application, or website to verify whether or not it can handle what it's designed to handle.
- **Stress testing** takes this a step further and is used to gauge how your software will respond at or beyond its peak load. The goal of stress testing is to overload the application on purpose until it breaks by applying both realistic and unrealistic load scenarios. With stress testing, you'll be able to find the failure point of your piece of software.
- **Endurance testing,** also known as soak testing, is used to analyze the behavior of an application under a specific amount of simulated load over longer amounts of time. The goal is to understand how your system will behave under sustained use, making it a longer process than load or stress testing (which are designed to end after a few hours). A critical piece of endurance testing is that it helps uncover memory leaks.
- **Spike testing** is a type of load test used to determine how your software will respond to substantially larger bursts of concurrent user or system activity over varying amounts of time. Ideally, this will help you understand what will happen when the load is suddenly and drastically increased.

Security Testing

With the rise of cloud-based testing platforms and cyber attacks, there is a growing concern and need for the security of data being used and stored in software. Security testing is a non-functional software testing technique used to determine if the information and data in a system is protected. The goal is to purposefully find loopholes and security risks in the system that could result in unauthorized access to or the loss of information by probing the application for weaknesses. There are multiple types of this testing method, each of which aimed at verifying six basic principles of security:

1. Integrity
2. Confidentiality
3. Authentication
4. Authorization
5. Availability
6. Non-repudiation

Usability Testing

Usability testing is a testing method that measures an application's ease-of-use from the end-user perspective and is often performed during the system or acceptance testing stages. The goal is to determine whether or not the visible design and aesthetics of an application meet the intended workflow for various processes, such as logging into an application. Usability testing is a great way for teams to review separate functions, or the system as a whole, is intuitive to use.

Compatibility Testing

Compatibility testing is used to gauge how an application or piece of software will work in different environments. It is used to check that your product is compatible with multiple operating systems, platforms, browsers, or resolution configurations. The goal is to ensure that your software's functionality is consistently supported across any environment you expect your end users to be using.

Whitebox vs Blackbox testing

Black box testing is a software testing methodology in which the tester analyzes the functionality of an application without a thorough knowledge of its internal design. Conversely, in white box testing, the tester is knowledgeable of the internal design of the application and analyzes it during testing

Application Security Testing Lab Setup

1. Understand REST APIs - https://www.paramsid.com/rest-api-playground/
2. OWSAP WebGoat :- https://owasp.org/www-project-webgoat/
3. OWSAP WebGoat GitHub :- https://github.com/WebGoat/WebGoat
4. Web App Pen Testing Tools :- https://www.getastra.com/blog/security-audit/penetration-test-online/