Secure Coding

Dr. Chuck Easttom, M.Ed, MSDS, MBA, MSSE, Ph.D.[2], D.Sc.

Case 6

This system was developed to handle case files for the FBI. It was eventually abandoned while still in the development stage, after costing 170 million dollars. It was considered so poorly designed and inadequate as to be completely unusable in real world conditions. It failed even the most basic systems and failed to meet basic requirements. A detailed report regarding the project's failure listed several problems including:

1. Repeated changes to specifications.
2. Poor architectural decisions.
3. Scope creep.
4. Managers of the project with little or no computer science training.

Software Product Defects

Software Design Defects

- Designing software with incomplete or erroneous decision-making criteria
- Failing to program as designed
- Failure to validate data

Data Defects

- Incomplete data used by automated decision-making applications.

DoD DevSecOps Testing

Continuous integration executes unit tests, such as Static and Dynamic Application Security Test (SAST), verify the integrity of the work in the broader context of the artifact or application. The CI assembly line is solely responsible at this point for guiding the subsystem, including dependency tracking, regression tests, code standards compliance, and pulling dependencies
from the local artifact repository, as necessary. When the CI completes, the artifact is automatically promoted to the **test environment**.

DoD Enterprise DevSecOps Fundamentals Version 2.0

DoD DevSecOps Testing

Once the code and artifact(s) reach the integration environment, the continuous deployment (CD) assembly line is triggered. *More* tests and security scans are performed in this environment, including operational and performance tests, user acceptance test, additional security compliance scans, etc. Once all of these tests complete without issue, the CD assembly line releases and delivers the final product package to the released artifact repository.

Released is never equivalent to Deployed! This is a source of confusion for many. A released artifact is *available* for deployment. Deployment may or may not occur instantly. A laptop that is powered off when a security patch is pushed into production will not immediately receive the artifact. Larger updates or out-of-cycle refreshes like anti-virus definition refreshes often require the user to initiate. The *deployment* occurs later. While this is a trivialized example, it effectively illustrates that released is never equivalent to deployed.

Reasons for Software Defects

- IT improperly interprets requirements
- Users specify the wrong requirements
- Requirements are incorrectly recorded
- Design specifications are incorrect
- Errors in program coding
- Data entry errors
- Testing errors
- Tests falsely detect an error
- The corrected condition causes another defect

CERT Security Coding Standards

- Rule 00. Input Validation and Data Sanitization (IDS)
- Rule 01. Declarations and Initialization (DCL)
- Rule 02. Expressions (EXP)
- Rule 03. Numeric Types and Operations (NUM)
- Rule 04. Characters and Strings (STR)
- Rule 05. Object Orientation (OBJ)
- Rule 06. Methods (MET)
- Rule 07. Exceptional Behavior (ERR)
- Rule 08. Visibility and Atomicity (VNA)
- Rule 09. Locking (LCK)
- Rule 10. Thread APIs (THI)
- Rule 11. Thread Pools (TPS)
- Rule 12. Thread-Safety Miscellaneous (TSM)
- Rule 13. Input Output (FIO)
- Rule 14. Serialization (SER)
- Rule 15. Platform Security (SEC)
- Rule 16. Runtime Environment (ENV)
- Rule 17. Java Native Interface (JNI)
- Rule 49. Miscellaneous (MSC)
- Rule 50. Android (DRD)

https://wiki.sei.cmu.edu/confluence/display/java/SEI+CERT+Oracle+Coding+Standard+for+Java

CERT Security Coding Recommendations

- Rec. 00. Input Validation and Data Sanitization (IDS)
- Rec. 01. Declarations and Initialization (DCL)
- Rec. 02. Expressions (EXP)
- Rec. 03. Numeric Types and Operations (NUM)
- Rec. 04. Characters and Strings (STR)
- Rec. 05. Object Orientation (OBJ)
- Rec. 06. Methods (MET)
- Rec. 07. Exceptional Behavior (ERR)
- Rec. 13. Input Output (FIO)
- Rec. 15. Platform Security (SEC)
- Rec. 18. Concurrency (CON)
- Rec. 49. Miscellaneous (MSC)

https://wiki.sei.cmu.edu/confluence/display/java/SEI+CERT+Oracle+Coding+Standard+for+Java

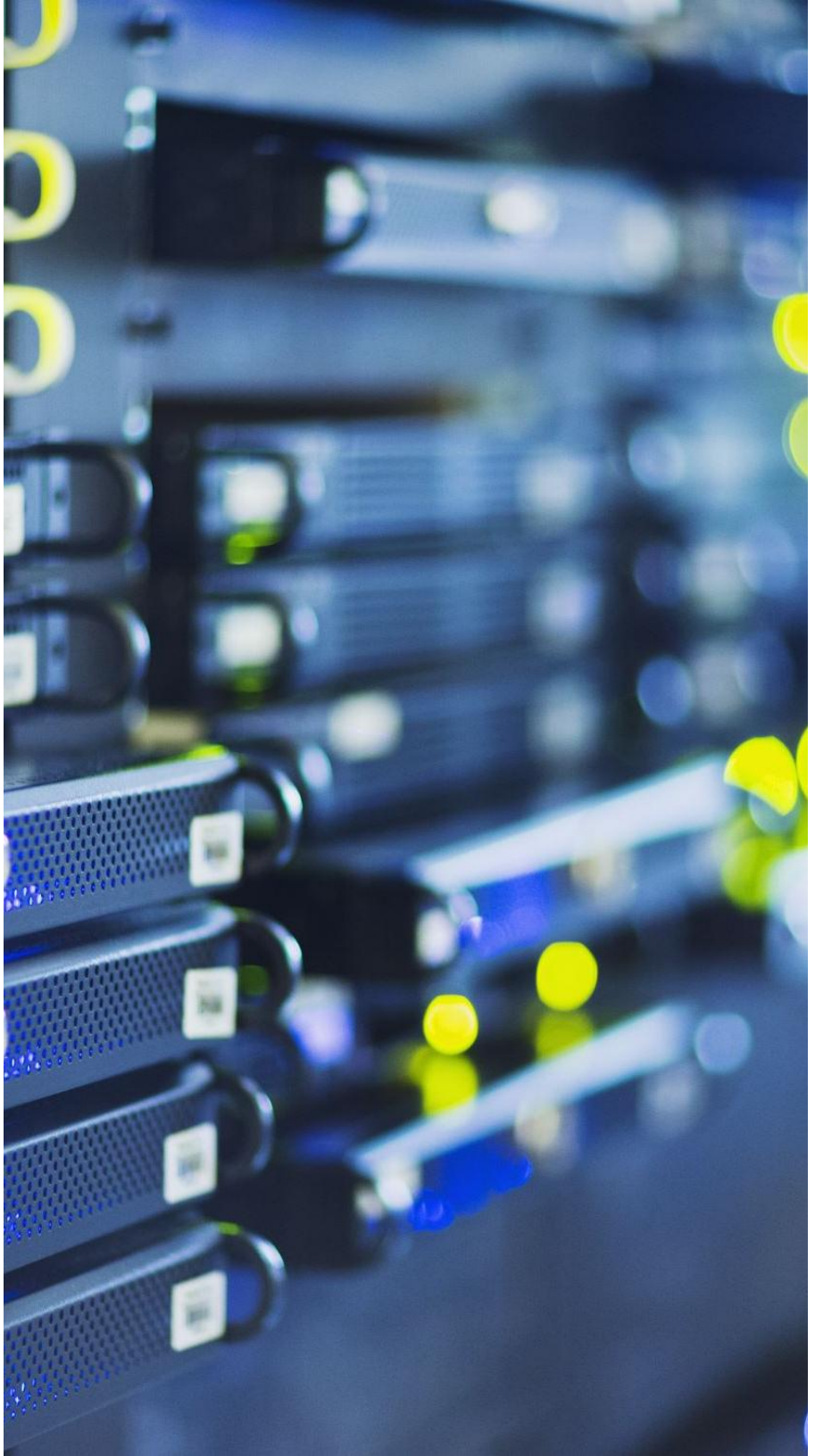
UDIs and CDIs

Unconstrained Data Items (UDIs)

- Unfiltered User Input
- May even be from an attacker

Constrained Data Items (CDIs)

- Data that has been verified and is now guaranteed to be valid
- Data that is "safe"



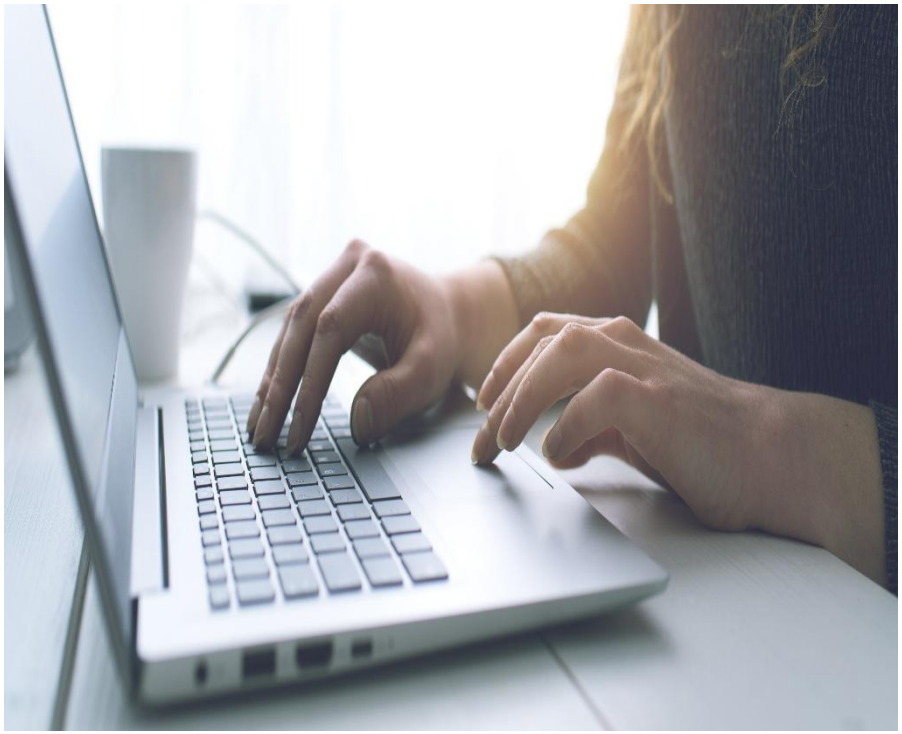
Secure coding

SECURE CODING

MURPHY'S LAW

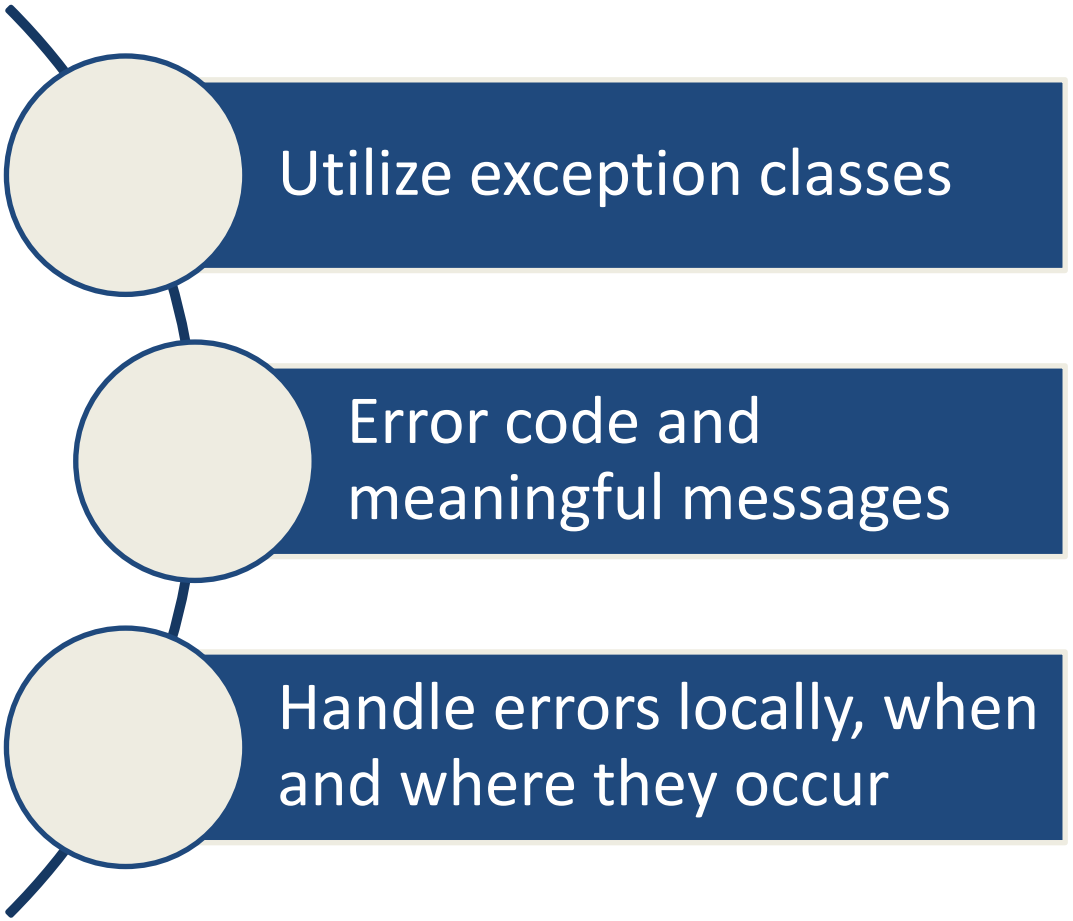
DEFENSIVE

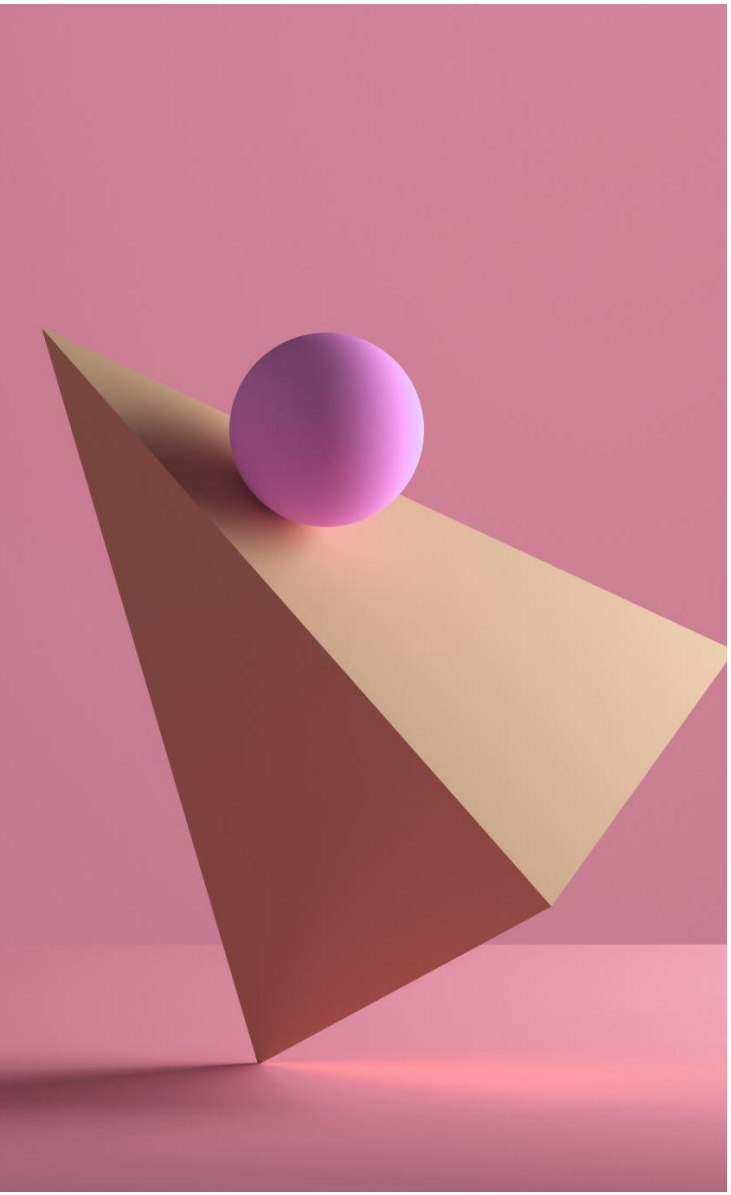
General guidelines for software Development



- Filter input
- Stored Procedures
- Error Handling

Error Handling Rules





Defensive Coding

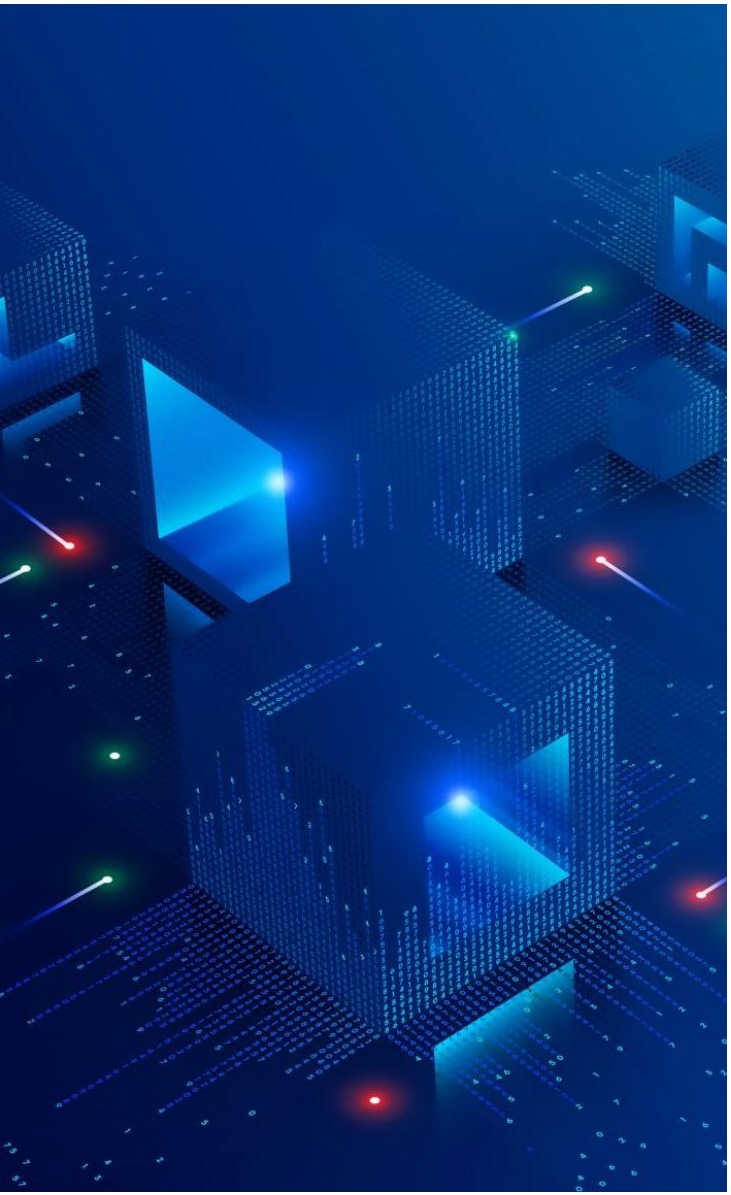
Always be wary of input (regardless of its source)
Plan for success
Stop errors before they happen (data validation)
Hard to have too much exception handling
Develop clean code
Test early and test often

SANS top 25 software vulnerabilities

Rank	ID	Name
1	CWE-787	Out-of-bounds Write
2	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
3	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
4	CWE-416	Use After Free
5	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
6	CWE-20	Improper Input Validation
7	CWE-125	Out-of-bounds Read
8	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
9	CWE-352	Cross-Site Request Forgery (CSRF)
10	CWE-434	Unrestricted Upload of File with Dangerous Type
11	CWE-862	Missing Authorization
12	CWE-476	NULL Pointer Dereference
13	CWE-287	Improper Authentication
14	CWE-190	Integer Overflow or Wraparound
15	CWE-502	Deserialization of Untrusted Data
16	CWE-77	Improper Neutralization of Special Elements used in a Command ('Command Injection')
17	CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer
18	CWE-798	Use of Hard-coded Credentials
19	CWE-918	Server-Side Request Forgery (SSRF)
20	CWE-306	Missing Authentication for Critical Function
21	CWE-362	Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')
22	CWE-269	Improper Privilege Management
23	CWE-94	Improper Control of Generation of Code ('Code Injection')
24	CWE-863	Incorrect Authorization
25	CWE-276	Incorrect Default Permissions

Checklist Sections

Data Validation
Authentication and Password Management
Authorization and Access Management
Session Management
Sensitive Information Storage or Transmission
System Configuration Management
General Coding Practices
Database Security
File Management
Memory Management



Object reuse

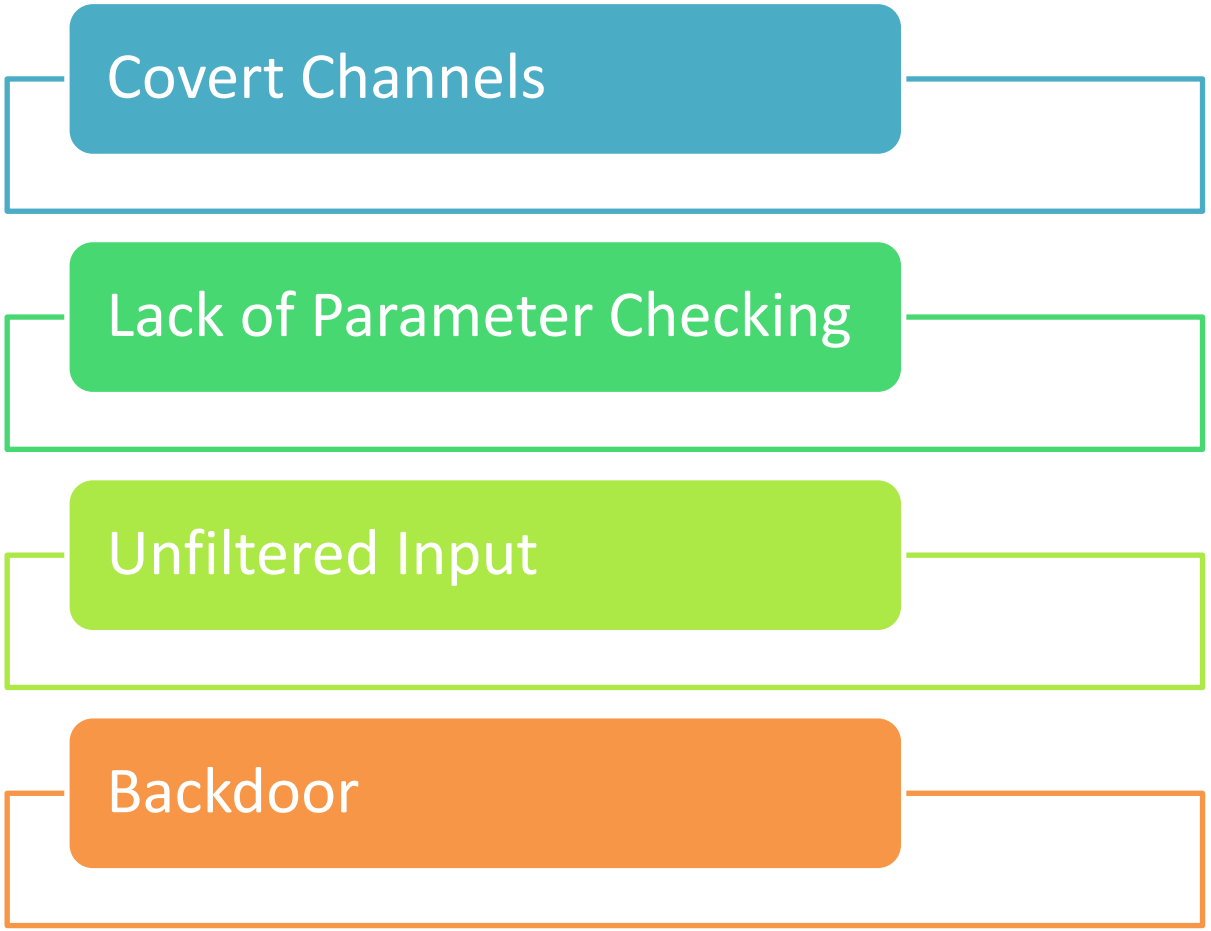
Use of a resource belonging to another process, including:

- Memory, databases, file systems, temporary
- files, and paging space

Object reuse countermeasures

- Application isolation
- Server virtualization
- Developer training

Application Security Issues



The CERT Oracle Secure Coding Standard for Java

- Rule 00. Input Validation and Data Sanitization (IDS)
- Rule 01. Declarations and Initialization (DCL)
- Rule 02. Expressions (EXP)
- Rule 03. Numeric Types and Operations (NUM)
- Rule 04. Characters and Strings (STR)
- Rule 05. Object Orientation (OBJ)
- Rule 06. Methods (MET)
- Rule 07. Exceptional Behavior (ERR)
- Rule 08. Visibility and Atomicity (VNA)
- Rule 09. Locking (LCK)
- Rule 10. Thread APIs (THI)
- Rule 11. Thread Pools (TPS)
- Rule 12. Thread-Safety Miscellaneous (TSM)
- Rule 13. Input Output (FIO)
- Rule 14. Serialization (SER)
- Rule 15. Platform Security (SEC)
- Rule 16. Runtime Environment (ENV)
- Rule 17. Java Native Interface (JNI)
- Rule 49. Miscellaneous (MSC)
- Rule 50. Android (DRD)

The CERT Top 10 Secure Coding Practices

Top 10 Secure Coding Practices

1. **Validate input.** Validate input from all untrusted data sources. Proper input validation can eliminate the vast majority of software vulnerabilities. Be suspicious of most external data sources, including command line arguments, network interfaces, environmental variables, and user controlled files [Seacord 05].
2. **Heed compiler warnings.** Compile code using the highest warning level available for your compiler and eliminate warnings by modifying the code [C MSC00-A, C++ MSC00-A]. Use static and dynamic analysis tools to detect and eliminate additional security flaws.
3. **Architect and design for security policies.** Create a software architecture and design your software to implement and enforce security policies. For example, if your system requires different privileges at different times, consider dividing the system into distinct intercommunicating subsystems, each with an appropriate privilege set.
4. **Keep it simple.** Keep the design as simple and small as possible [Saltzer 74, Saltzer 75]. Complex designs increase the likelihood that errors will be made in their implementation, configuration, and use. Additionally, the effort required to achieve an appropriate level of assurance increases dramatically as security mechanisms become more complex.
5. **Default deny.** Base access decisions on permission rather than exclusion. This means that, by default, access is denied and the protection scheme identifies conditions under which access is permitted [Saltzer 74, Saltzer 75].
6. **Adhere to the principle of least privilege.** Every process should execute with the the least set of privileges necessary to complete the job. Any elevated permission should be held for a minimum time. This approach reduces the opportunities an attacker has to execute arbitrary code with elevated privileges [Saltzer 74, Saltzer 75].
7. **Sanitize data sent to other systems.** Sanitize all data passed to complex subsystems [C STR02-A] such as command shells, relational databases, and commercial off-the-shelf (COTS) components. Attackers may be able to invoke unused functionality in these components through the use of SQL, command, or other injection attacks. This is not necessarily an input validation problem because the complex subsystem being invoked does not understand the context in which the call is made. Because the calling process understands the context, it is responsible for sanitizing the data before invoking the subsystem.
8. **Practice defense in depth.** Manage risk with multiple defensive strategies, so that if one layer of defense turns out to be inadequate, another layer of defense can prevent a security flaw from becoming an exploitable vulnerability and/or limit the consequences of a successful exploit. For example, combining secure programming techniques with secure runtime environments should reduce the likelihood that vulnerabilities remaining in the code at deployment time can be exploited in the operational environment [Seacord 05].
9. **Use effective quality assurance techniques.** Good quality assurance techniques can be effective in identifying and eliminating vulnerabilities. Fuzz testing, penetration testing, and source code audits should all be incorporated as part of an effective quality assurance program. Independent security reviews can lead to more secure systems. External reviewers bring an independent perspective; for example, in identifying and correcting invalid assumptions [Seacord 05].
10. **Adopt a secure coding standard.** Develop and/or apply a secure coding standard for your target development language and platform.

OWASP TOP 10 Web Programming Vulnerabilities

A01 Broken Access Control

A02 Cryptographic Failures

A03 Injection

A04 Insecure Design

A05 Security Misconfiguration

A06 Vulnerable and Outdated Components

A07 Identification and Authentication Failures

A08 Software and Data Integrity Failures

A09 Security Logging and Monitoring Failures

A10 Server Side Request Forgery (SSRF)

A01:2021 – Broken Access Control

- Moving up from the fifth position, 94% of applications were tested for some form of broken access control with the average incidence rate of 3.81%, and has the most occurrences in the contributed dataset with over 318k. Notable Common Weakness Enumerations (CWEs) included are CWE-200: Exposure of Sensitive Information to an Unauthorized Actor, CWE-201: Insertion of Sensitive Information Into Sent Data, and CWE-352: Cross-Site Request Forgery.
- Access control enforces policy such that users cannot act outside of their intended permissions. Failures typically lead to unauthorized information disclosure, modification, or destruction of all data or performing a business function outside the user's limits. Common access control vulnerabilities include:

- Violation of the principle of least privilege or deny by default, where access should only be granted for particular capabilities, roles, or users, but is available to anyone.
- Bypassing access control checks by modifying the URL (parameter tampering or force browsing), internal application state, or the HTML page, or by using an attack tool modifying API requests.
- Permitting viewing or editing someone else's account, by providing its unique identifier (insecure direct object references)
- Accessing API with missing access controls for POST, PUT and DELETE.
- Elevation of privilege. Acting as a user without being logged in or acting as an admin when logged in as a user.
- Metadata manipulation, such as replaying or tampering with a JSON Web Token (JWT) access control token, or a cookie or hidden field manipulated to elevate privileges or abusing JWT invalidation.
- CORS misconfiguration allows API access from unauthorized/untrusted origins.
- Force browsing to authenticated pages as an unauthenticated user or to privileged pages as a standard user.

A02:2021 – Cryptographic Failures

The first thing is to determine the protection needs of data in transit and at rest. For example, passwords, credit card numbers, health records, personal information, and business secrets require extra protection, mainly if that data falls under privacy laws, e.g., EU's General Data Protection Regulation (GDPR), or regulations, e.g., financial data protection such as PCI Data Security Standard (PCI DSS). For all such data:

- Is any data transmitted in clear text? This concerns protocols such as HTTP, SMTP, FTP also using TLS upgrades like STARTTLS. External internet traffic is hazardous. Verify all internal traffic, e.g., between load balancers, web servers, or back-end systems.
- Are any old or weak cryptographic algorithms or protocols used either by default or in older code?
- Are default crypto keys in use, weak crypto keys generated or re-used, or is proper key management or rotation missing? Are crypto keys checked into source code repositories?
- Is encryption not enforced, e.g., are any HTTP headers (browser) security directives or headers missing?
- Is the received server certificate and the trust chain properly validated?
- Are initialization vectors ignored, reused, or not generated sufficiently secure for the cryptographic mode of operation? Is an insecure mode of operation such as ECB in use? Is encryption used when authenticated encryption is more appropriate?
- Are passwords being used as cryptographic keys in absence of a password base key derivation function?
- Is randomness used for cryptographic purposes that was not designed to meet cryptographic requirements? Even if the correct function is chosen, does it need to be seeded by the developer, and if not, has the developer over-written the strong seeding functionality built into it with a seed that lacks sufficient entropy/unpredictability?
- Are deprecated hash functions such as MD5 or SHA1 in use, or are non-cryptographic hash functions used when cryptographic hash functions are needed?
- Are deprecated cryptographic padding methods such as PKCS number 1 v1.5 in use?
- Are cryptographic error messages or side channel information exploitable, for example in the form of padding oracle attacks?

A03:2021 – Injection

- An application is vulnerable to attack when:
- User-supplied data is not validated, filtered, or sanitized by the application.
- Dynamic queries or non-parameterized calls without context-aware escaping are used directly in the interpreter.
- Hostile data is used within object-relational mapping (ORM) search parameters to extract additional, sensitive records.
- Hostile data is directly used or concatenated. The SQL or command contains the structure and malicious data in dynamic queries, commands, or stored procedures.
- Some of the more common injections are SQL, NoSQL, OS command, Object Relational Mapping (ORM), LDAP, and Expression Language (EL) or Object Graph Navigation Library (OGNL) injection. The concept is identical among all interpreters. Source code review is the best method of detecting if applications are vulnerable to injections. Automated testing of all parameters, headers, URL, cookies, JSON, SOAP, and XML data inputs is strongly encouraged. Organizations can include static (SAST), dynamic (DAST), and interactive (IAST) application security testing tools into the CI/CD pipeline to identify introduced injection flaws before production deployment.

A04:2021 – Insecure Design

Insecure design is a broad category representing different weaknesses, expressed as “missing or ineffective control design.” Insecure design is not the source for all other Top 10 risk categories. There is a difference between insecure design and insecure implementation. We differentiate between design flaws and implementation defects for a reason, they have different root causes and remediation. A secure design can still have implementation defects leading to vulnerabilities that may be exploited. An insecure design cannot be fixed by a perfect implementation as by definition, needed security controls were never created to defend against specific attacks. One of the factors that contribute to insecure design is the lack of business risk profiling inherent in the software or system being developed, and thus the failure to determine what level of security design is required.

A05:2021 – Security Misconfiguration

- The application might be vulnerable if the application is:
- Missing appropriate security hardening across any part of the application stack or improperly configured permissions on cloud services.
- Unnecessary features are enabled or installed (e.g., unnecessary ports, services, pages, accounts, or privileges).
- Default accounts and their passwords are still enabled and unchanged.
- Error handling reveals stack traces or other overly informative error messages to users.
- For upgraded systems, the latest security features are disabled or not configured securely.
- The security settings in the application servers, application frameworks (e.g., Struts, Spring, ASP.NET), libraries, databases, etc., are not set to secure values.
- The server does not send security headers or directives, or they are not set to secure values.
- The software is out of date or vulnerable (see A06:2021-Vulnerable and Outdated Components).
- Without a concerted, repeatable application security configuration process, systems are at a higher risk.

A06:2021 – Vulnerable and Outdated Components

You are likely vulnerable:

If you do not know the versions of all components you use (both client-side and server-side). This includes components you directly use as well as nested dependencies.

If the software is vulnerable, unsupported, or out of date. This includes the OS, web/application server, database management system (DBMS), applications, APIs and all components, runtime environments, and libraries.

If you do not scan for vulnerabilities regularly and subscribe to security bulletins related to the components you use.

If you do not fix or upgrade the underlying platform, frameworks, and dependencies in a risk-based, timely fashion. This commonly happens in environments when patching is a monthly or quarterly task under change control, leaving organizations open to days or months of unnecessary exposure to fixed vulnerabilities.

If software developers do not test the compatibility of updated, upgraded, or patched libraries.

If you do not secure the components' configurations (see A05:2021-Security Misconfiguration).

A07:2021 – Identification and Authentication Failures

- Confirmation of the user's identity, authentication, and session management is critical to protect against authentication-related attacks. There may be authentication weaknesses if the application:
- Permits automated attacks such as credential stuffing, where the attacker has a list of valid usernames and passwords.
- Permits brute force or other automated attacks.
- Permits default, weak, or well-known passwords, such as "Password1" or "admin/admin".
- Uses weak or ineffective credential recovery and forgot-password processes, such as "knowledge-based answers," which cannot be made safe.
- Uses plain text, encrypted, or weakly hashed passwords data stores (see A02:2021-Cryptographic Failures).
- Has missing or ineffective multi-factor authentication.
- Exposes session identifier in the URL.
- Reuse session identifier after successful login.
- Does not correctly invalidate Session IDs. User sessions or authentication tokens (mainly single sign-on (SSO) tokens) aren't properly invalidated during logout or a period of inactivity.

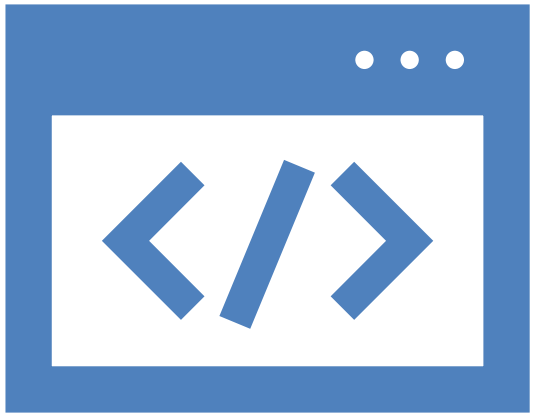
A08:2021 – Software and Data Integrity Failures

- Software and data integrity failures relate to code and infrastructure that does not protect against integrity violations. An example of this is where an application relies upon plugins, libraries, or modules from untrusted sources, repositories, and content delivery networks (CDNs). An insecure CI/CD pipeline can introduce the potential for unauthorized access, malicious code, or system compromise. Lastly, many applications now include auto-update functionality, where updates are downloaded without sufficient integrity verification and applied to the previously trusted application. Attackers could potentially upload their own updates to be distributed and run on all installations. Another example is where objects or data are encoded or serialized into a structure that an attacker can see and modify is vulnerable to insecure deserialization.

A09:2021 – Security Logging and Monitoring Failures

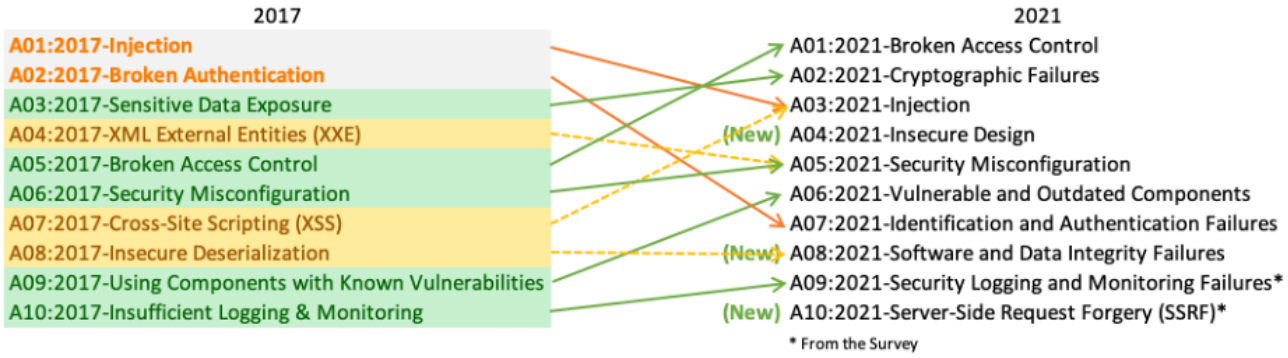
- Returning to the OWASP Top 10 2021, this category is to help detect, escalate, and respond to active breaches. Without logging and monitoring, breaches cannot be detected. Insufficient logging, detection, monitoring, and active response occurs any time:
- Auditable events, such as logins, failed logins, and high-value transactions, are not logged.
- Warnings and errors generate no, inadequate, or unclear log messages.
- Logs of applications and APIs are not monitored for suspicious activity.
- Logs are only stored locally.
- Appropriate alerting thresholds and response escalation processes are not in place or effective.
- Penetration testing and scans by dynamic application security testing (DAST) tools (such as OWASP ZAP) do not trigger alerts.
- The application cannot detect, escalate, or alert for active attacks in real-time or near real-time.
- You are vulnerable to information leakage by making logging and alerting events visible to a user or an attacker (see A01:2021-Broken Access Control).

A10:2021 – Server-Side Request Forgery (SSRF)



- SSRF flaws occur whenever a web application is fetching a remote resource without validating the user-supplied URL. It allows an attacker to coerce the application to send a crafted request to an unexpected destination, even when protected by a firewall, VPN, or another type of network access control list (ACL).
- As modern web applications provide end-users with convenient features, fetching a URL becomes a common scenario. As a result, the incidence of SSRF is increasing. Also, the severity of SSRF is becoming higher due to cloud services and the complexity of architectures.

OWASP TOP 10



OWASP Top 10 – 2010
A1 – Injection
A2 – Cross Site Scripting (XSS)
A3 – Broken Authentication and Session Management
A4 – Insecure Direct Object References
A5 – Cross Site Request Forgery (CSRF)
A6 – Security Misconfiguration (NEW)
A7 – Failure to Restrict URL Access
A8 – Unvalidated Redirects and Forwards (NEW)
A9 – Insecure Cryptographic Storage
A10 - Insufficient Transport Layer Protection

OWASP TOP 10 IoT (2018)

I1 Weak Guessable, or Hardcoded Passwords

I2 Insecure Network Services

I3 Insecure Ecosystem Interfaces

I4 Lack of Secure Update Mechanism

I5 Use of Insecure or Outdated Components

I6 Insufficient Privacy Protection

I7 Insecure Data Transfer and Storage

I8 Lack of Device Management

I9 Insecure Default Settings

I10 Lack of Physical Hardening

OWASP TOP 10 API

API1:2023 - Broken Object Level Authorization	APIs tend to expose endpoints that handle object identifiers, creating a wide attack surface of Object Level Access Control issues. Object level authorization checks should be considered in every function that accesses a data source using an ID from the user.
API2:2023 - Broken Authentication	Authentication mechanisms are often implemented incorrectly, allowing attackers to compromise authentication tokens or to exploit implementation flaws to assume other user's identities temporarily or permanently. Compromising a system's ability to identify the client/user, compromises API security overall.
API3:2023 - Broken Object Property Level Authorization	This category combines API3:2019 Excessive Data Exposure and API6:2019 - Mass Assignment, focusing on the root cause: the lack of or improper authorization validation at the object property level. This leads to information exposure or manipulation by unauthorized parties.
API4:2023 - Unrestricted Resource Consumption	Satisfying API requests requires resources such as network bandwidth, CPU, memory, and storage. Other resources such as emails/SMS/phone calls or biometrics validation are made available by service providers via API integrations, and paid for per request. Successful attacks can lead to Denial of Service or an increase of operational costs.
API5:2023 - Broken Function Level Authorization	Complex access control policies with different hierarchies, groups, and roles, and an unclear separation between administrative and regular functions, tend to lead to authorization flaws. By exploiting these issues, attackers can gain access to other users' resources and/or administrative functions.
API6:2023 - Unrestricted Access to Sensitive Business Flows	APIs vulnerable to this risk expose a business flow - such as buying a ticket, or posting a comment - without compensating for how the functionality could harm the business if used excessively in an automated manner. This doesn't necessarily come from implementation bugs.
API7:2023 - Server Side Request Forgery	Server-Side Request Forgery (SSRF) flaws can occur when an API is fetching a remote resource without validating the user-supplied URI. This enables an attacker to coerce the application to send a crafted request to an unexpected destination, even when protected by a firewall or a VPN.
API8:2023 - Security Misconfiguration	APIs and the systems supporting them typically contain complex configurations, meant to make the APIs more customizable. Software and DevOps engineers can miss these configurations, or don't follow security best practices when it comes to configuration, opening the door for different types of attacks.
API9:2023 - Improper Inventory Management	APIs tend to expose more endpoints than traditional web applications, making proper and updated documentation highly important. A proper inventory of hosts and deployed API versions also are important to mitigate issues such as deprecated API versions and exposed debug endpoints.
API10:2023 - Unsafe Consumption of APIs	Developers tend to trust data received from third-party APIs more than user input, and so tend to adopt weaker security standards. In order to compromise APIs, attackers go after integrated third-party services instead of trying to compromise the target API directly.

OWASP TOP 10 Mobile (2023)

M1: Insecure Authentication/Authorization
M2: Insecure Communication
M3: Inadequate Supply Chain Security
M4: Inadequate Privacy Controls
M5: Improper Credential Usage
M6: Insufficient Input/Output Validation
M7: Security Misconfiguration
M8: Insufficient Cryptography
M9: Insecure Data Storage
M10: Insufficient Binary Protections

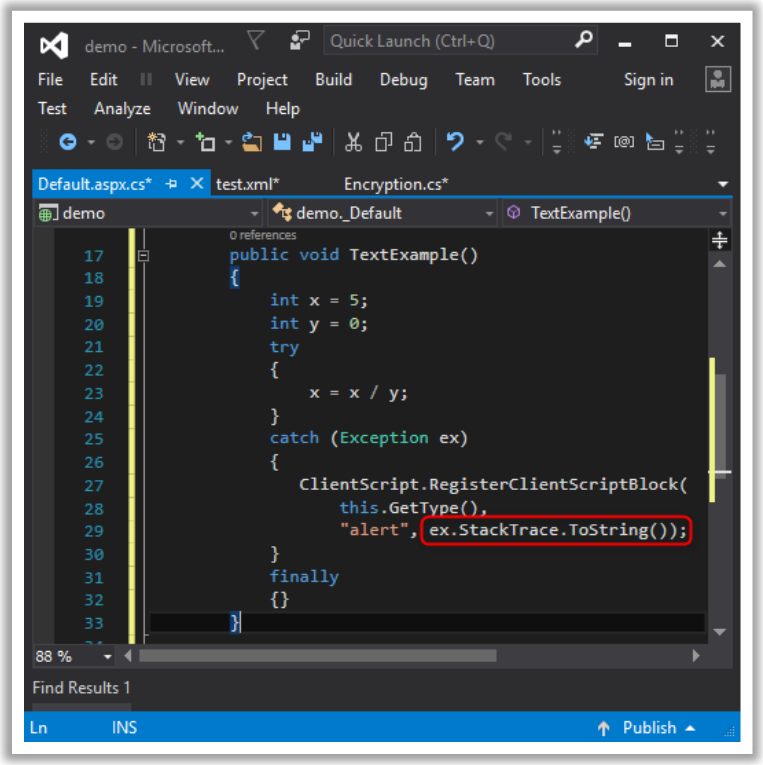
Avoid Displaying Detailed Error Messages

The stack trace gives detailed information about the exception occurred that helps the attacker in determining loopholes and security flaws in an application

For example, the database error message will give hints to the attacker that the application is vulnerable to a SQL injection attack

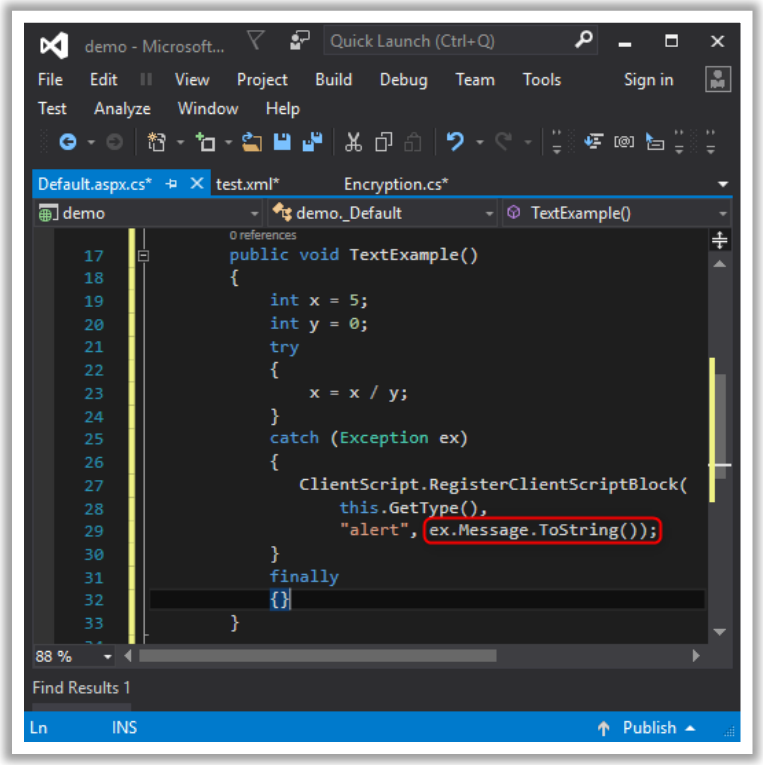
To remove this vulnerability in exception handling, avoid printing the stack trace when the exception is caught

Vulnerable Code



It prints detailed error message

Secure Code



It prints only error message

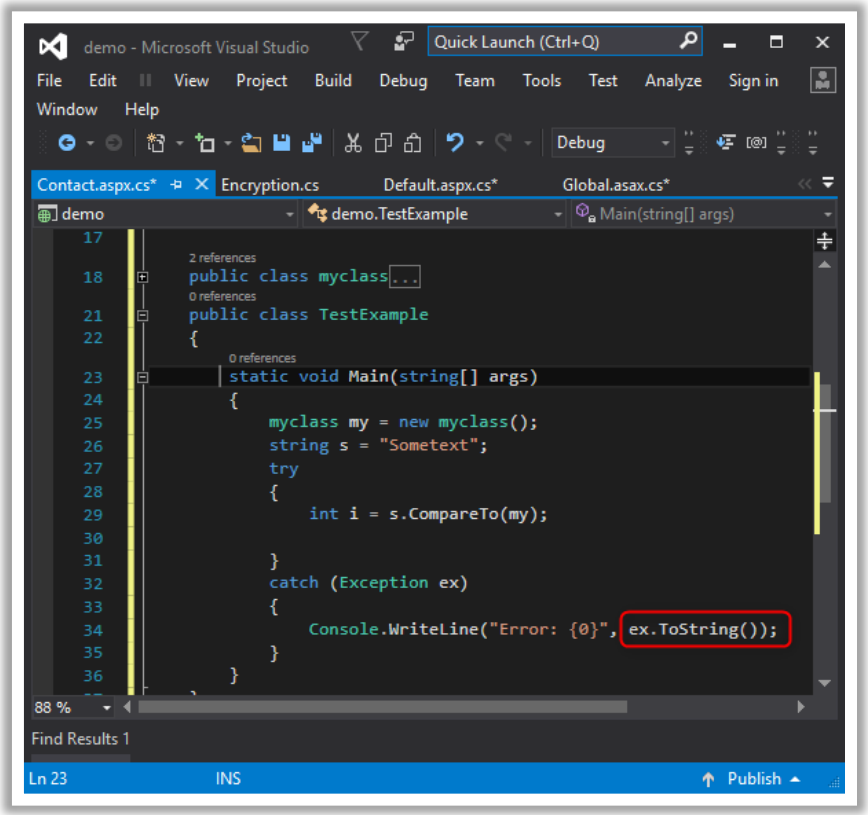
Avoid Displaying Detailed Error Messages

The **Exception.ToString()** method should not be used when the information about the exception is to be shown to the user

This method prints a **stack trace** and may give internal information about the application, making it vulnerable to attack

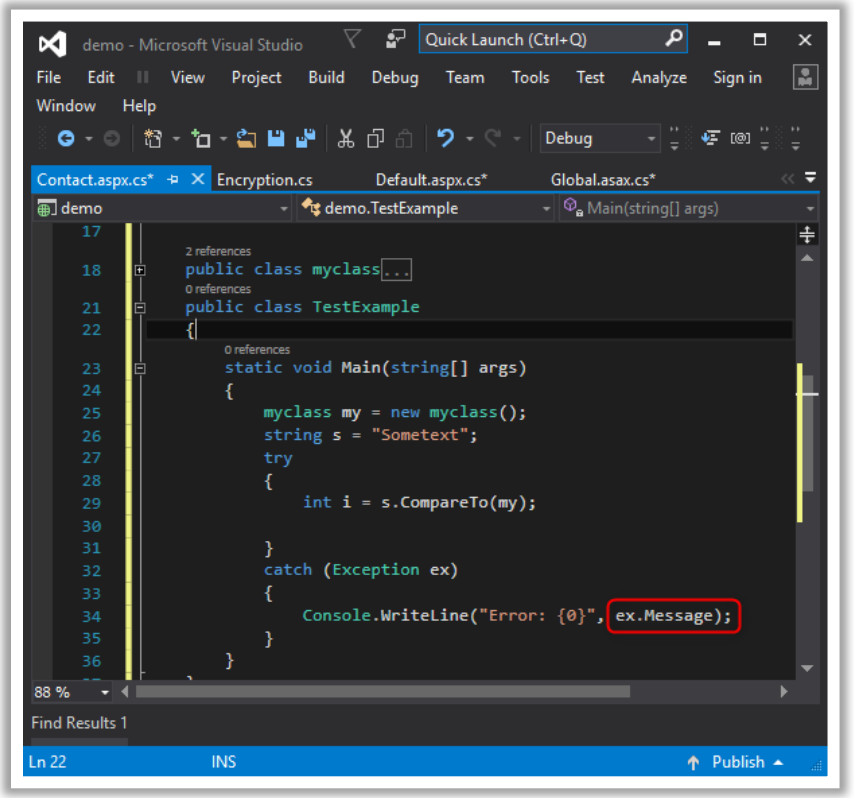
Instead, **Exception.Message** should be used, which gives a simple description about the exception

Vulnerable Code



`ex.ToString()` prints detailed error message

Secure Code



`ex.Message` prints only error message

Checklist for Proper Exception Handling

- Structured exception-handling provides an **efficient**, **clean**, and **unobtrusive** way of handling exceptions in the application
- Here are some of the techniques for handling exceptions properly:

Use single try and many catches	Try to use a single try and many **catch** statements in a program, which may increase the readability of your code
Arrange the catch blocks from specific to general	Make sure that the different types of **exceptions** are placed from most **specific** to most general
Always reach to valid state	Ensure that the objects in code are returned to a **valid** state after an exception. This can be done by implementing the necessary cleanup under the finally block
Throw the closest match	While handing more than one exception for representing the same errors, use closely matching **error conditions**
Do not throw exceptions for normal occurring events	Do not use exceptions for commonly occurring events; instead, throw exceptions only when something **out of the ordinary happens**
Extend the ApplicationException class, not the Base Exception class	Never extend the base class **System.Exception**; always try to extend the **ApplicationException** class
Use the inner exception property when re-throwing	Use the base **System.Exception** class to wrap another new exception from throwing that checks the Inner Exception property, determining the cause of the problem
Do not create new exception	Use the existing exceptions that are well defined in the .**NET Framework** rather than creating new exceptions
Name appropriately	After creating any Exception class, name it with the word ending 'Exception' so that there will not be any **conflicts**

C++ Exceptions

- Exception handling - catch errors before they occur
- Deals with synchronous errors (i.e., Divide by zero)
- Does not deal with asynchronous errors - disk I/O completions, mouse clicks - use interrupt processing
- Used when system can recover from error
- Exception handler - recovery procedure
- Typically used when error dealt with in different place than where it occurred
- Useful when program cannot recover but must shut down cleanly
- Exception handling should not be used for program control
- Not optimized, can harm program performance

C++ Exceptions

- If none of the catch handlers for a try block matches a thrown exception the exception moves to the next enclosing try block.
- If there is no match in any enclosing try block the exception is uncaught. An uncaught exception also occurs if a new exception is thrown before an existing one is handled. Cleanups may fail to occur with an uncaught exception, so this is an error.
- If an exception is uncaught the special function terminate() is called.
- Uncaught exceptions can always be avoided by enclosing the contents of main in a try block with an ellipsis handler.

ISOCCPP Error Handling Rules

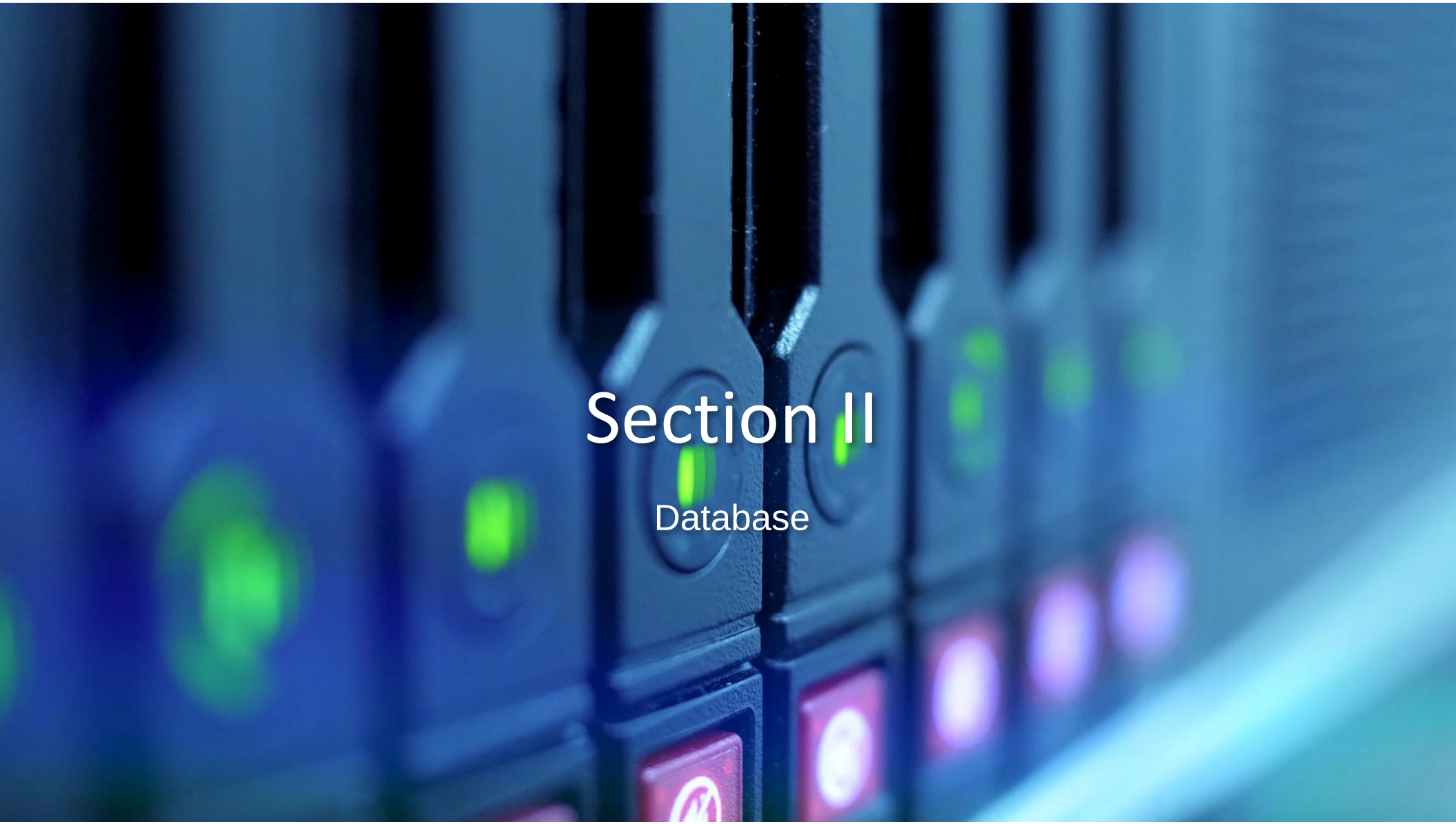
- E.1: Develop an error-handling strategy early in a design
- E.2: Throw an exception to signal that a function can't perform its assigned task
- E.3: Use exceptions for error handling only
- E.4: Design your error-handling strategy around invariants
- E.5: Let a constructor establish an invariant, and throw if it cannot
- E.6: Use RAII to prevent leaks
- E.7: State your preconditions
- E.8: State your postconditions
- E.12: Use noexcept when exiting a function because of a throw is impossible or unacceptable
- E.13: Never throw while being the direct owner of an object
- E.14: Use purpose-designed user-defined types as exceptions (not built-in types)
- E.15: Throw by value, catch exceptions from a hierarchy by reference
- E.16: Destructors, deallocation, and swap must never fail
- E.17: Don't try to catch every exception in every function
- E.18: Minimize the use of explicit try/catch
- E.19: Use a final_action object to express cleanup if no suitable resource handle is available
- E.25: If you can't throw exceptions, simulate RAII for resource management
- E.26: If you can't throw exceptions, consider failing fast
- E.27: If you can't throw exceptions, use error codes systematically
- E.28: Avoid error handling based on global state (e.g., errno)
- E.30: Don't use exception specifications
- E.31: Properly order your catch-clauses

•http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#S-errors

ISOCCPP Philosophy

- P.1: Express ideas directly in code
- P.2: Write in ISO Standard C++
- P.3: Express intent
- P.4: Ideally, a program should be statically type safe
- P.5: Prefer compile-time checking to run-time checking
- P.6: What cannot be checked at compile time should be checkable at run time
- P.7: Catch run-time errors early
- P.8: Don't leak any resources
- P.9: Don't waste time or space
- P.10: Prefer immutable data to mutable data
- P.11: Encapsulate messy constructs, rather than spreading through the code
- P.12: Use supporting tools as appropriate
- P.13: Use support libraries as appropriate

- http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#S-philosophy



Section II

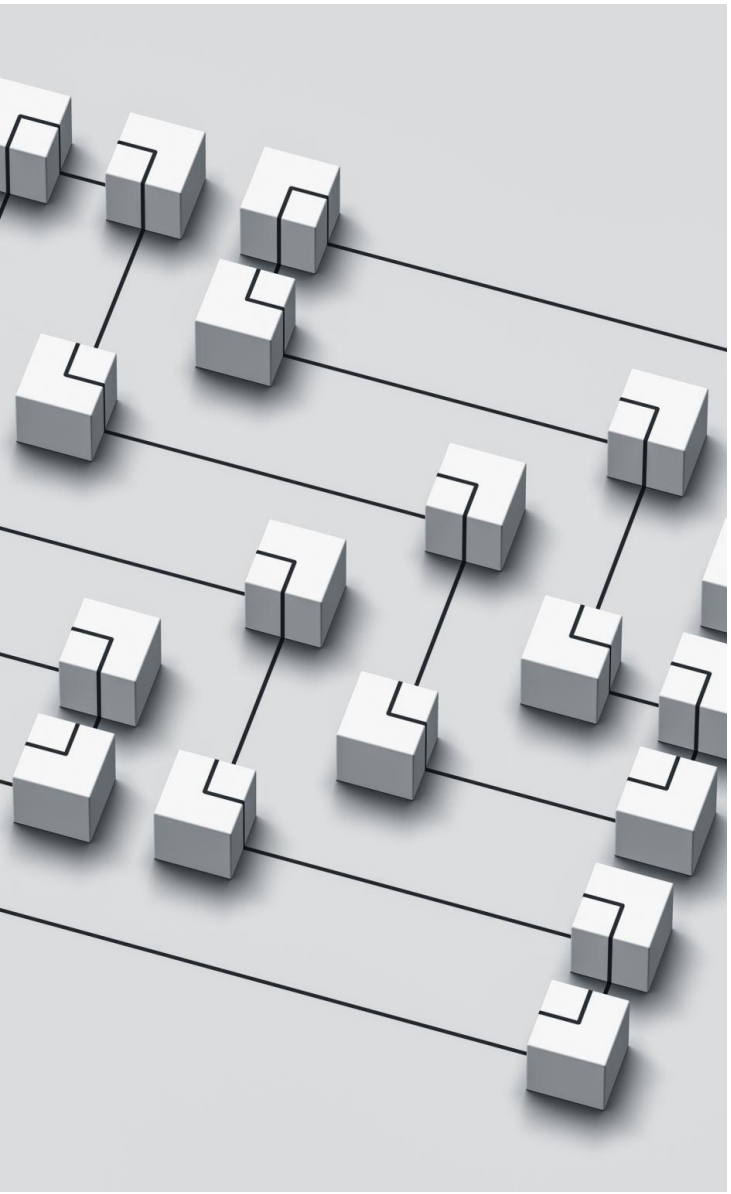
Database

Dr. Chuck Easttom, M.Ed, MSDS, MBA, MSSE, Ph.D.[2], D.Sc.

Database Security

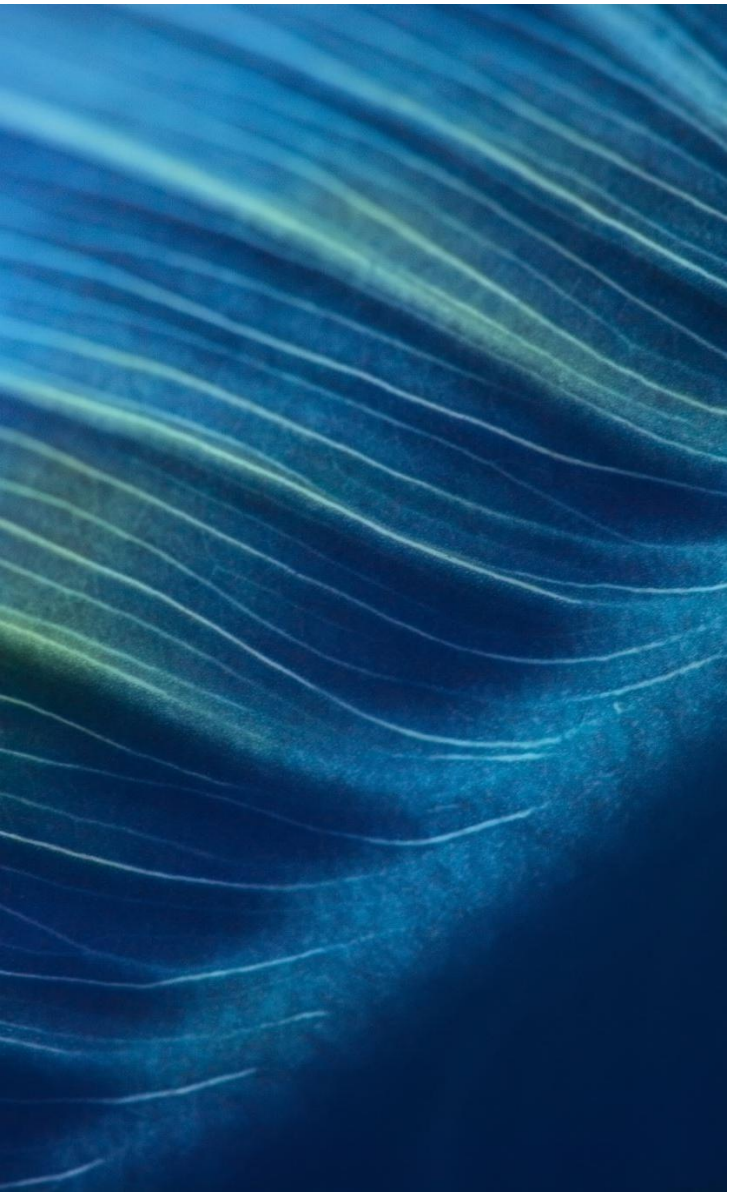
- Use strongly typed parameterized queries. Parameterized queries keep the query and data separate through the use of placeholders. The query structure is defined with place holders and then the application specifies the contents of each placeholder.
- Utilize input validation and if validation fails, do not run the database command.
- Ensure that variables are strongly typed.
- Escape meta characters in SQL statements.
- The application should use the lowest possible level of privilege when accessing the database.
- Use secure credentials for database access.
- Do not provide connection strings or credentials directly to the client. If this is unavoidable, encrypted them.
- Use stored procedures to abstract data access.
- Turn off any database functionality (e.g., unnecessary stored procedures or services).
- Eliminate default content.

- Disable any default accounts that are not required to support business requirements.
- Close the connection as soon as possible.
- The application should connect to the database with different credentials for every trust distinction (e.g., user, read-only user, guest, administrators).



Database types

Relational databases use tables that are related via keys and communicates with Structure Query Language. SQL Server, Oracle, MySQL, and Postgres are the most common examples.
A *NoSQL database* is not a relational database and does not use SQL. MongoDB is the most common example.
Hierarchical Database Management Systems
Object-Oriented Database Management Systems
Object-Relational Database Management Systems



RDMS

- Relational databases are based on relations
- The structure includes tables, primary and foreign keys, and relations.
- Each row represents a single entity
- Each column represents a single attribute
- Each record is identified by a unique number called a primary key.
- Tables are related by foreign keys. A foreign key is a primary key in another table.

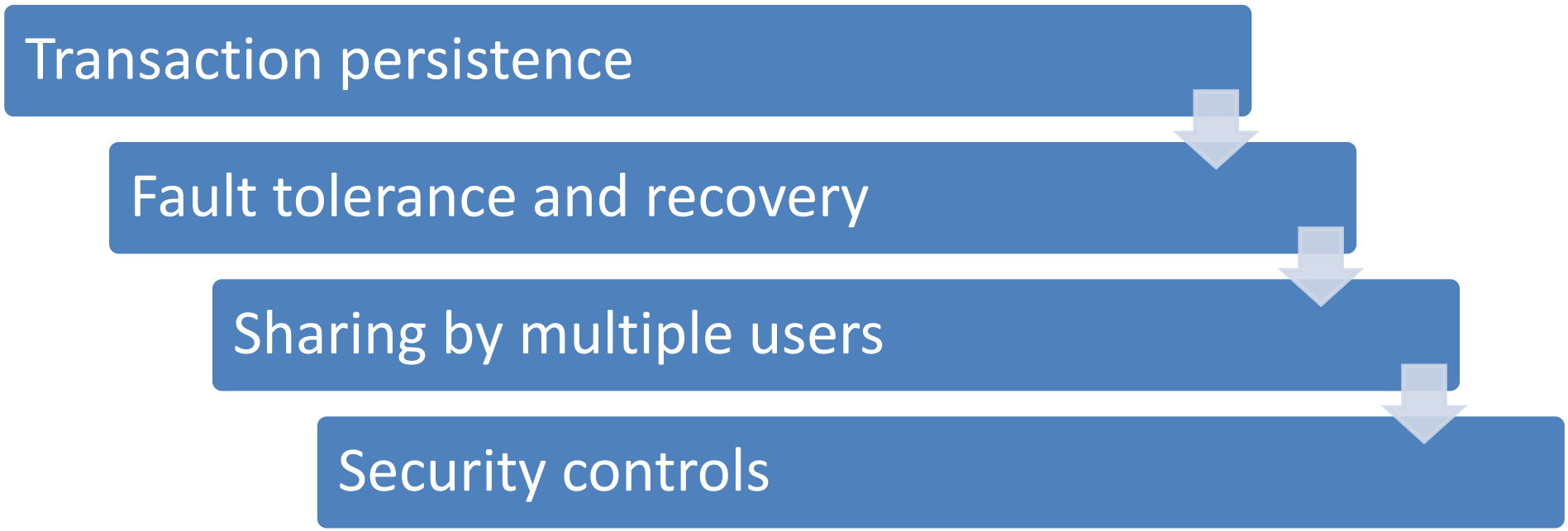
Relational DB

SQL Server, Oracle, MySQL, PostGres, MS

PK	LNAME	FNAME	JobCode	Hire Date
1	Smith	Jane	2	1/10/2010
2	Perez	Juan	2	1/14/2011
3	Kent	Clark	1	3/2/2005
4	Euler	Leo		
5	Plank	Ma		

PK	Job Name	Min Edu.	Min Salary	Max Salary
1	Super hero	None	100,000	1,000,000
2	Programmer	BA/BS	70,000	95,000
3	Math / Scientist	Ph.D.	80,000	110,000
4	Manager	BA/BS	140,000	220,000

DBMS Should Provide:



Database Security Issues

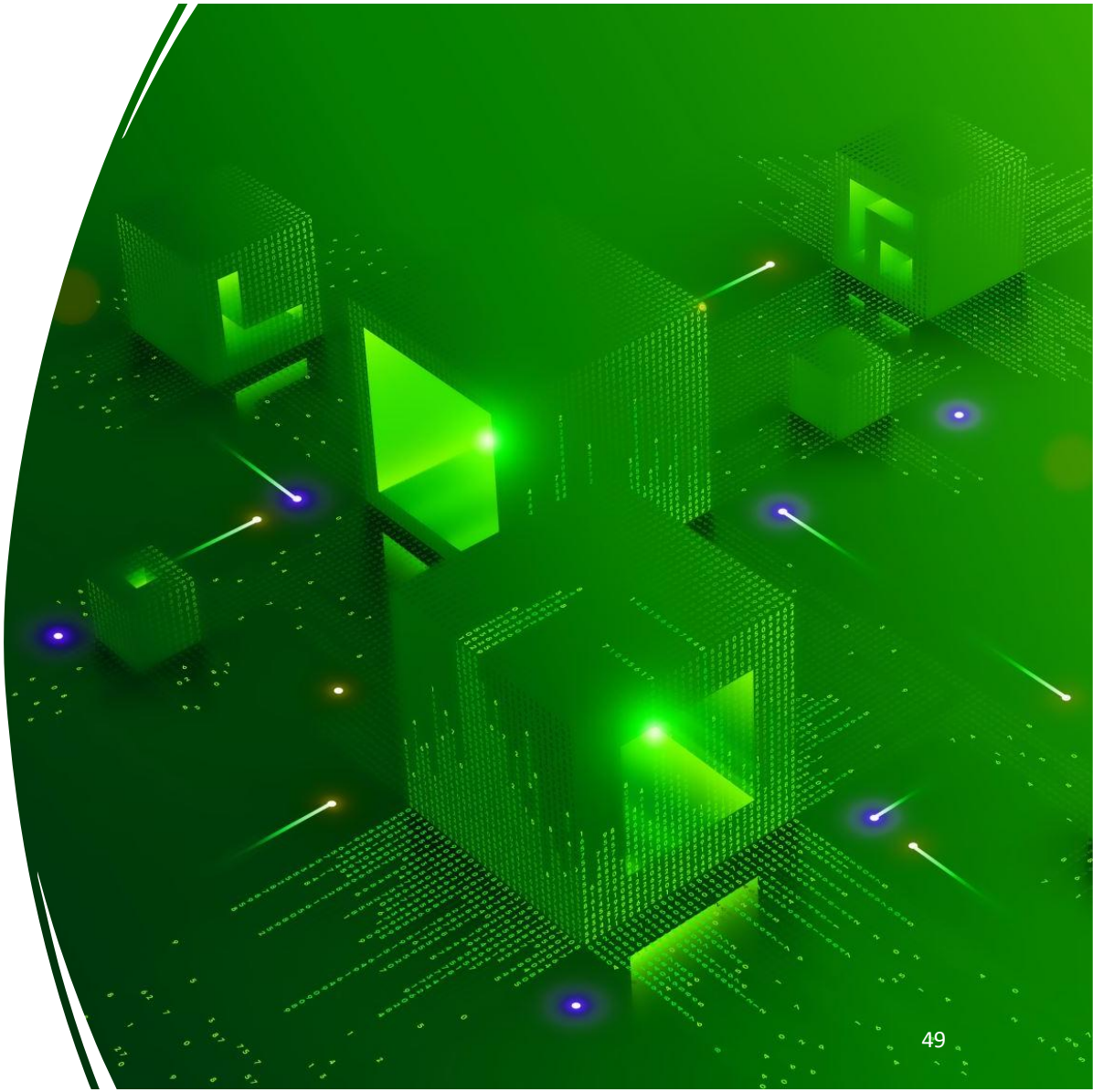
- Unauthorized Access
- Improper Modification of Data
- Access Availability
- Query attacks
- Bypass attacks
- Interception of data
- Data contamination

Database Interface

Standardized access methods that provide an interface to the database.

Examples include:

- Open Database Connectivity (ODBC)
- Object Linking and Embedding (OLE)
- ActiveX Data Objects (ADO)
- Java Database Connectivity (JDBC)



Database Transactions

Records retrieval
Records update
Records creation
Transactional integrity
- Nested or complex transactions executed as a unit
- Begin work… <transactions> …end work

Database Security Controls

Access controls

- Userids, passwords
- Table / row / field level access control
- Read-only or read/write

Views

- Virtual tables that are a subset of individual tables, or a “join” between tables
- Permission given to views just like “real” tables

Lock Controls

Used to control read and write access to specific rows of data in relational systems, or objects in object-oriented systems.

Locks ensure only one user at a time can alter data.

Better programming logic and testing reduce deadlocking problems.

Lock Controls - the ACID Test

Atomicity

- either all changes take effect or none do.

Consistency

- a transaction is allowed only if it meets owner/system-defined integrity constraints.

Isolation

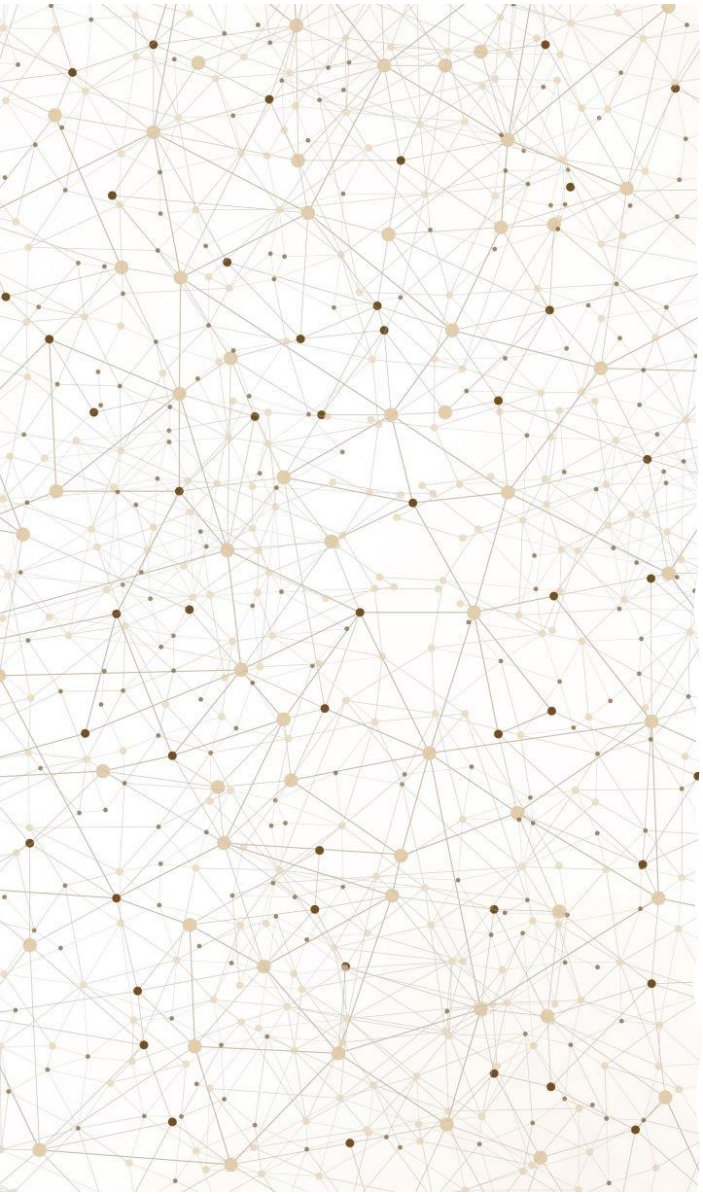
- the results of the transaction are not visible until the transaction is complete.

Durability

- a completed transaction is permanent.

What is NoSQL

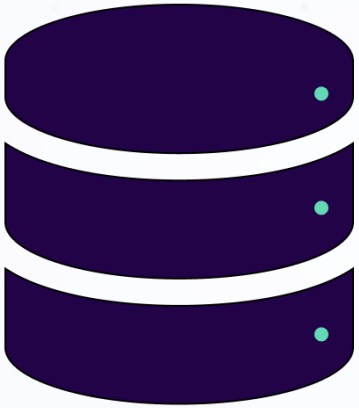
- Some say it means Not Only SQL others say Not SQL. Which you prefer is up to you.
- It is a database architecture not based on relations and that does not depend on the use of structured query language.
- At its heart it is polyglot persistence. Polyglot persistence is defined as using different data stores in different circumstances. Basically, a much more malleable structure than RDMS/SQL has.
- It is more of a neoligsm than a definition. The term “NoSQL” is applied to a variety of data storage implementations.



NoSQL Definition

From www.nosql-database.org:

> **Next Generation Databases mostly addressing some of the points: being non-relational, distributed, open-source and horizontal scalable. The original intention has been modern web-scale databases. The movement began early 2009 and is growing rapidly. Often more characteristics apply as: schema-free, easy replication support, simple API, eventually consistent / BASE (not ACID), a huge data amount, and more.**



NoSQL

NoSQL History - Continued

NoSQL of today traces back to 2009. Essentially a number of developers were interested in alternative data storage. There where examples such as BigTable, Dynamo, and Hadoop. This group of developers came up with the name "NoSQL". According to the book "NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence", the term was suggested by Eric Evans, a developer with RackSpace.

NoSQL Databases

NoSQL databases are not based on the relational data model.

There is no standard NoSQL model.

Many different data models are grouped under the NoSQL umbrella, including document stores, graph stores, column stores, and key-value stores.

The word stores is used often in lieu of database. Store indicates that these models permanently store data.

There is no schema in NoSQL

NoSQL often uses Aggregates

NoSQL Databases

Asynchronous Inserts & Updates

Schema-less

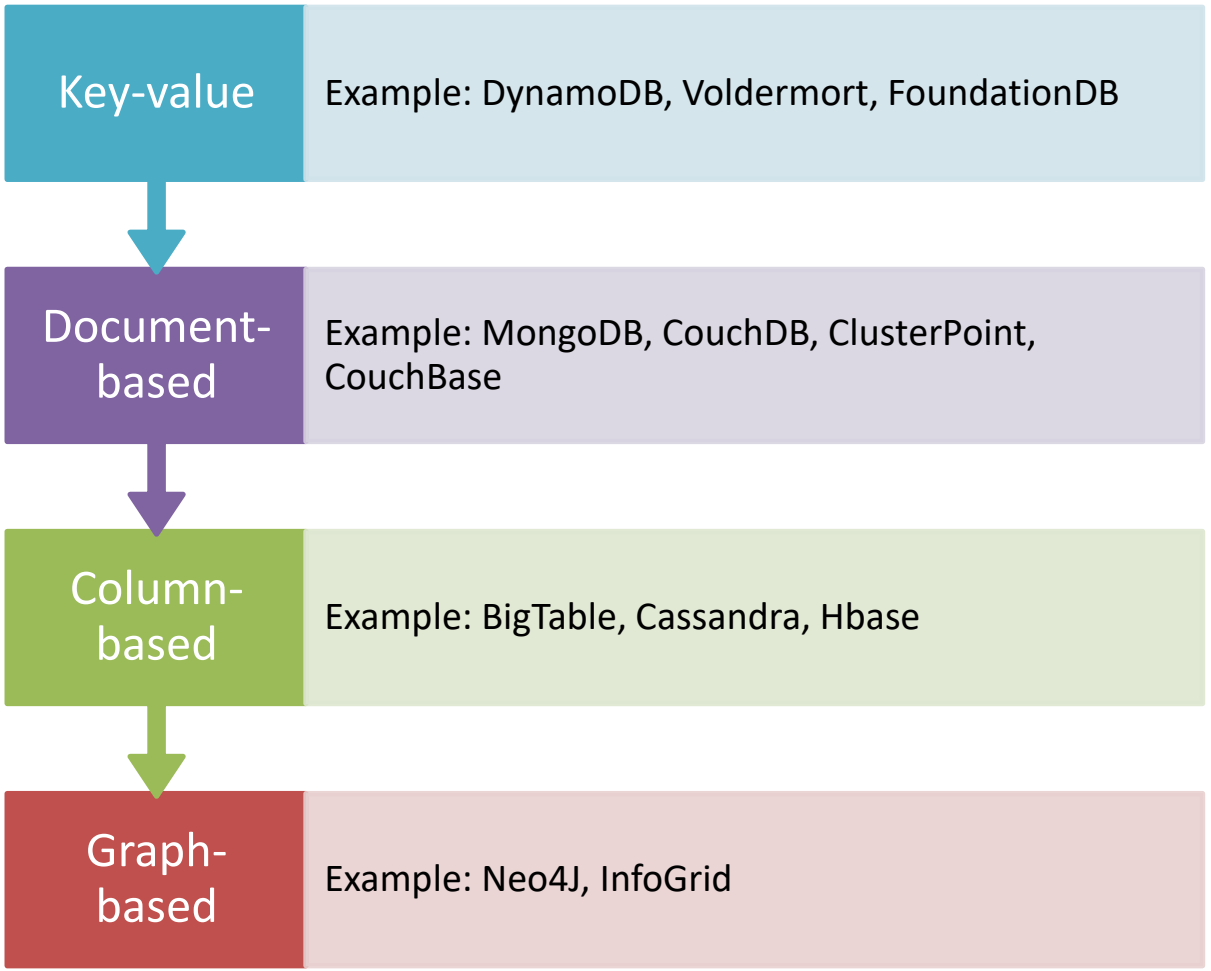
ACID transaction properties are not needed – BASE

CAP Theorem

Open Source (usually)

Most NoSQL databases are geared toward performance rather than transaction consistency

NOSQL categories



Key-value

Focus on scaling to huge amounts of data

Designed to handle massive load

Based on Amazon's dynamo paper

Data model: (global) collection of Key-value pairs

Dynamo ring partitioning and *replication*

Example: (DynamoDB)

- *items* having one or more attributes (name, value)
- An *attribute* can be single-valued or multi-valued like set.
- items are combined into a *table*

Document-based

Can model more complex objects

Inspired by Lotus Notes

Data model: collection of documents

Document: JSON (**J**ava**S**cript **O**bject **N**otation is a data model, key-value pairs, which supports objects, records, structs, lists, array, maps, dates, Boolean with **nesting**), XML, other semi-structured formats.

NoSQL

Column-based

One column family can have variable numbers of columns

Cells within a column family are sorted “physically”

Very sparse, most cells have null values

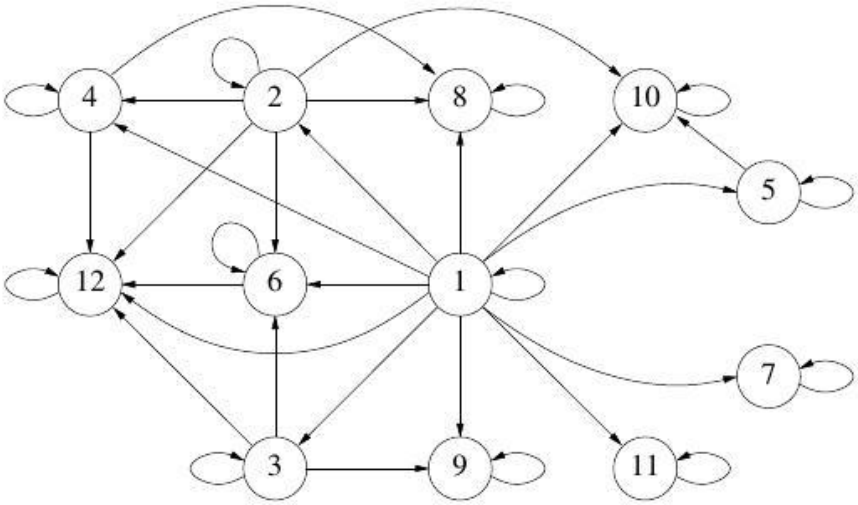
Comparison: RDBMS vs column-based NOSQL

Query on multiple tables

RDBMS: must fetch data from several places on disk and glue together

Column-based NOSQL: only fetch column families of those columns that are required by a query (all columns in a column family are stored together on the disk, so multiple rows can be retrieved in one read operation data locality)

Graph-based



Focus on modeling the structure of data (*interconnectivity*)

Scales to the complexity of data

Inspired by mathematical Graph Theory (G=(E,V))

Data model:

- (Property Graph) nodes and edges
  - Nodes may have properties (including ID)
  - Edges may have labels or roles
- Key-value pairs on both

Interfaces and query languages vary

NoSQL Query Answers

Answers returned quickly
Answers more important than inserts
Data is replicated to multiple nodes
Can query closest node
    Improved response time

Inserts and Updates

Inserts and updates are asynchronous
You might not get the latest inserts in your query
You might not get the latest updates in your query
Consistency is eventual, not immediate

Typical NoSQL API

Basic API access:

get(key) -- Extract the value given a key.

put(key, value) -- Create or update the value given its key.

delete(key) -- Remove the key and its associated value(s).

execute(key, operation, parameters) -- Invoke an operation.

BASE Transactions

Acronym contrived to be the opposite of ACID

- **Basically Available,**
- **Soft state,**
- **Eventually Consistent**

Basically Available

NoSQL is all about availability
Basically available, means the data is available even with multiple failures.
Accomplished by data replications

Soft state

The state of the data may not be consistent
Consistency is not handled by the database/datastore
Consistency is the problem of the programmer

Eventually Consistent

Consistency is not always current

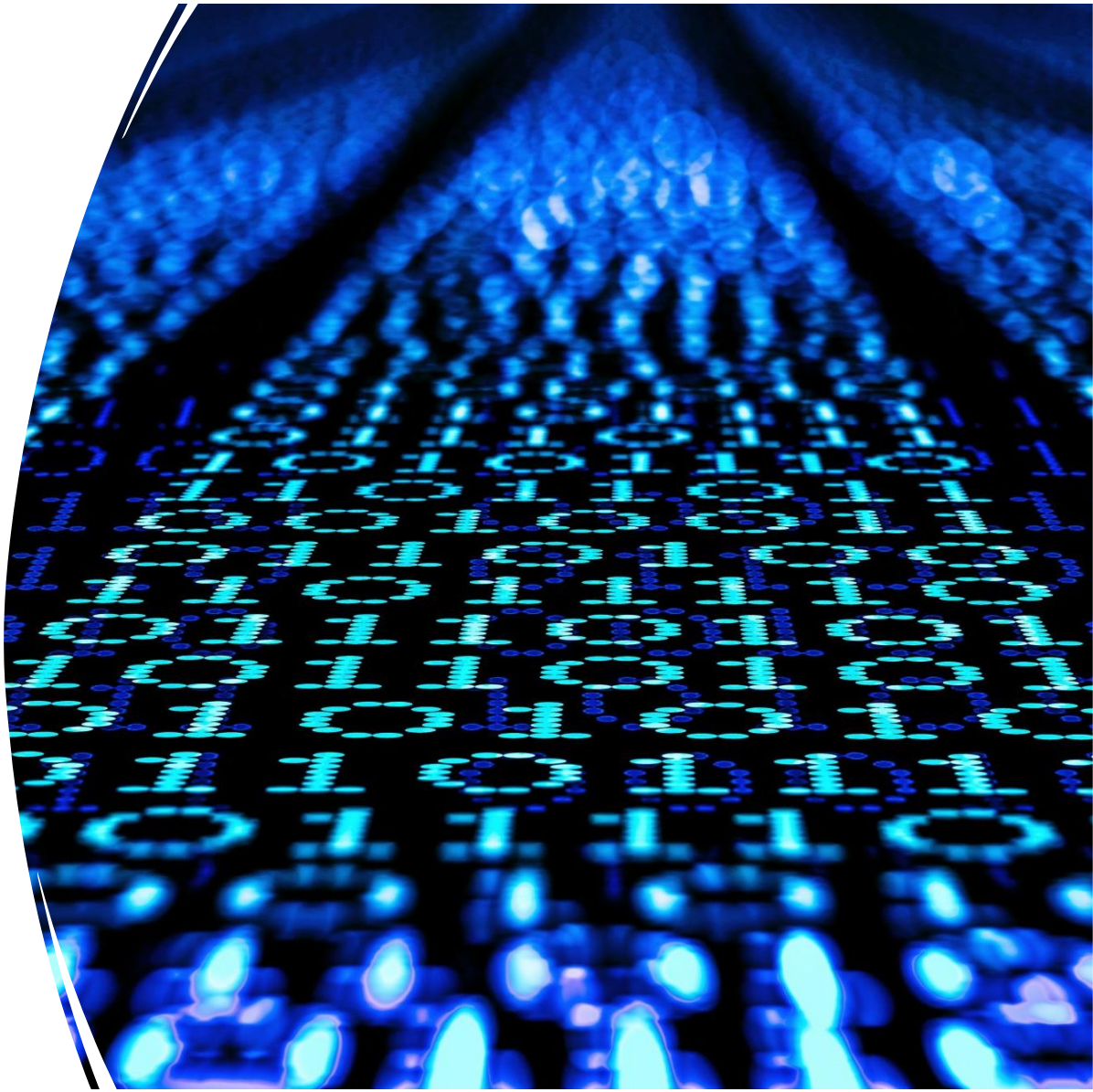
It must *eventually* be consistent

BASE Characteristics

- **Weak consistency**
- **Best effort**
- **Approximate answers**
- **Simpler and faster**

PART IIII

Web Programming

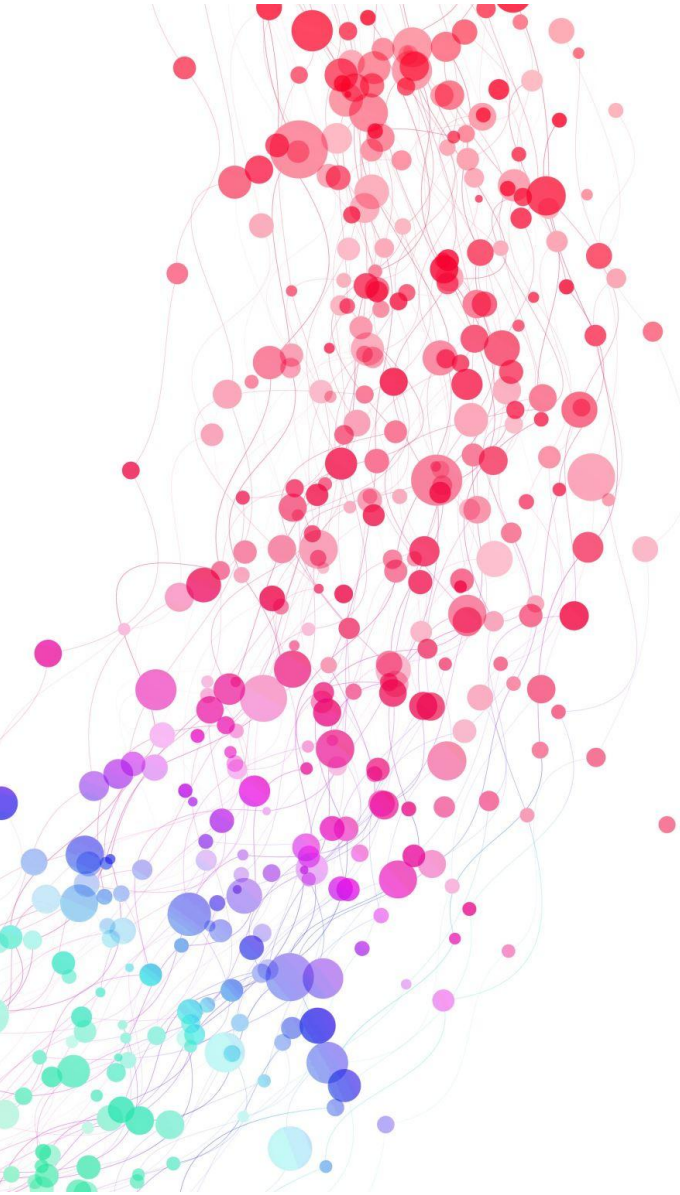


Dr. Chuck Easttom, M.Ed, MSDS, MBA, MSSE, Ph.D.[2], D.Sc.

Web Traffic

Web traffic uses the hyper text transfer protocol, or HTTP. That protocol normally operates on port 80. The primary means of communication is via messages. This table gives you a summary of the basic HTTP messages a web page might send to a web server.

GET	**Request to read a Web page**
HEAD	Request to read a Web page
PUT	Request to write a Web page
POST	Request to append to a page
DELETE	Remove the Web page
LINK	Connects two existing resources
UNLINK	Breaks an existing connection between two resources



Web Traffic

•The most common are GET, HEAD, PUT, and POST. In fact, you might see only those four during most of your analysis of web traffic. Link and Unlink are a lot less common. You should know that the GET command actually is the server getting information. To quote the WWC "The GET method means retrieve whatever information (in the form of an entity) is identified by the Request-URI. If the Request-URI refers to a data-producing process, it is the produced data which shall be returned as the entity in the response and not the source text of the process, unless that text happens to be the output of the process. "

•There are important differences between GET and POST
•GET requests can be cached POST requests are never cached
•GET requests remain in the browser history(by default), POST requests do not remain in the browser history
•GET requests can be bookmarked POST requests cannot be bookmarked
•GET requests have length restrictions POST requests have no restrictions on data length
•GET requests should never be used when dealing with sensitive data

•http://www.w3schools.com/tags/ref_httpmethods.asp

RFC details on GET and POST

- RFC 2616 defines HTTP
- Limitations on the request body size are set by the server and in some cases proxies. However, section 10.4.15 of RFC2616 specifies response code 414 Request-URI Too long in the event that the URL length exceeds the server limit.

- The 413 Request Entity Too Large response status code indicates that the request entity is larger than limits defined by server; the server might close the connection or return a Retry-After header field.

RFC details on GET and POST

The HTTP protocol does not place any a priori limit on the length of a URI. Servers MUST be able to handle the URI of any resource they serve, and SHOULD be able to handle URIs of unbounded length if they provide GET-based forms that could generate such URIs. A server SHOULD return 414 (Request-URI Too Long) status if a URI is longer than the server can handle

- -https://www.ietf.org/rfc/rfc2616.txt

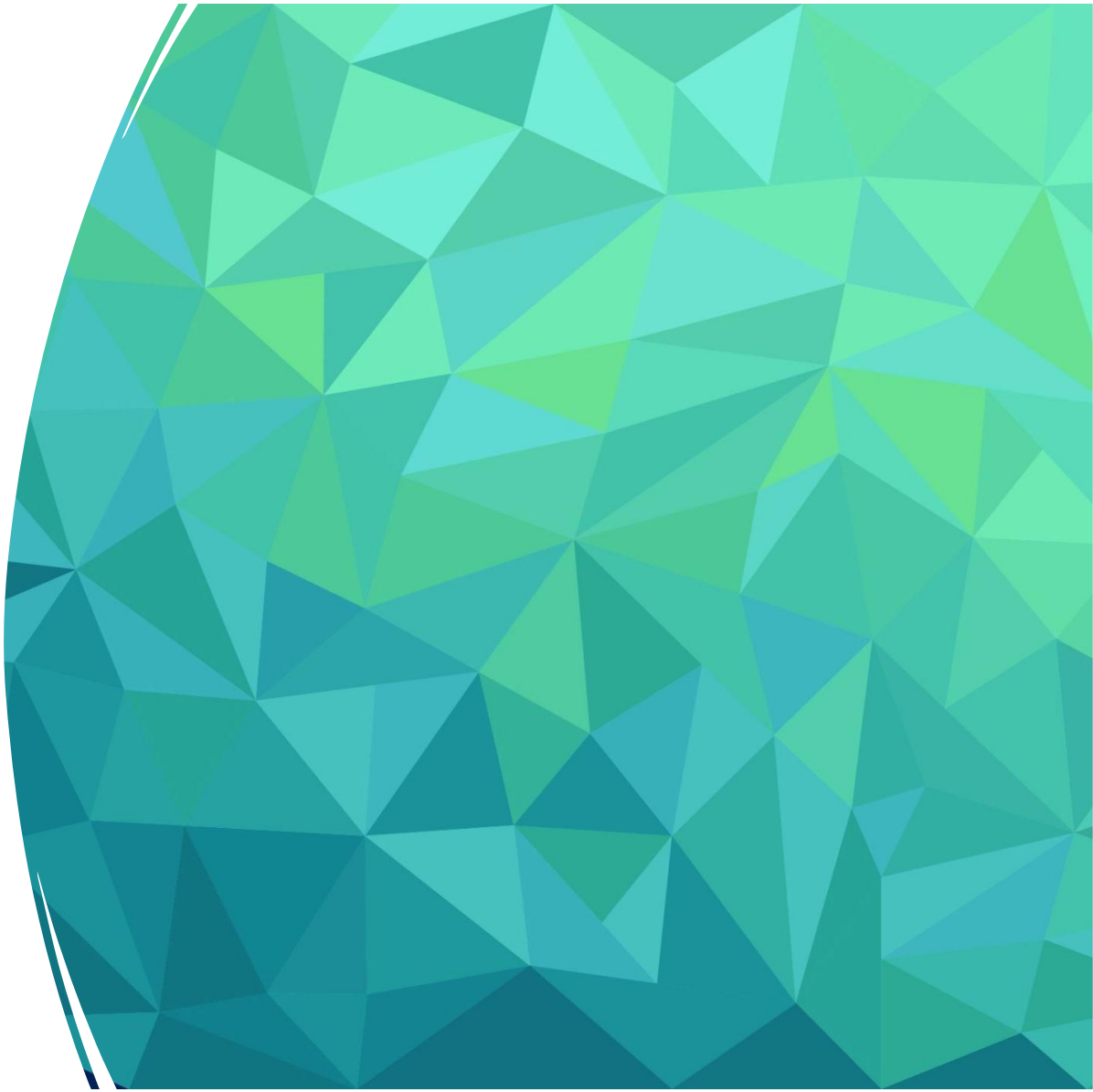
Message Range	Meaning
100	These are just informational. The server is telling your browser some information, most of which will never be displayed to the user. For example when you switch from http to https, a 101 message goes to the browser telling it that the protocol is changing.
200-299	These are basically 'OK' messages, meaning that whatever the browser requested, the server successfully processed. Your basic HTTP messages like POST, GET, HEAD, etc. should, if everything is working properly, get a 200 code in response.
300-399	These are redirect messages telling the browser to go to another URL. For example 301 means that the requested resource has permanently moved to a new URL, but the message code 307 indicates the move is temporary.
400-499	These are client errors, and the ones most often shown to the end user. This might seem odd since, for example 404 file not found means that the server could not find the file you asked for. However, the issue is that the server functioned properly, just that file does not exist. Therefore the client request was in error.
500-599	These are server side errors. For example 503 means the service requested is down, possibly overloaded. You will see this error frequently in DoS attacks.

Web Traffic

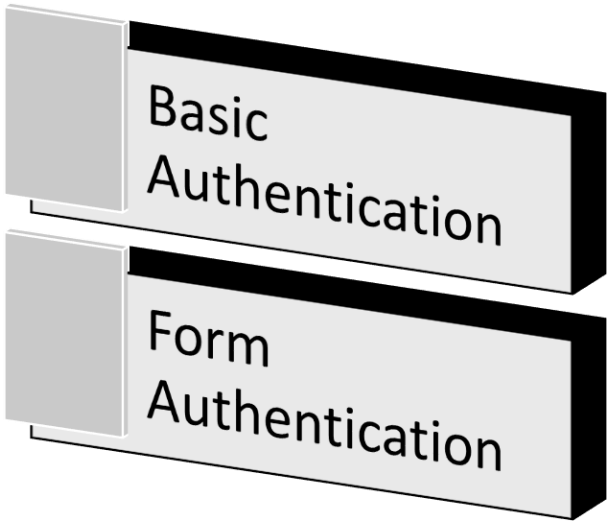
These error messages can be important for web security. Some of the reasons are shown in the table above. For example, multiple 503 errors could indicate a Denial of Service attack. Error code 305 is also interesting. It states that the requested source is only available via a proxy. That gives information about the architecture of that web server. Message 407 is related, but it states that authentication with the proxy is required.

Redirects

- Quote from OWASP
- "The best way to find out if an application has any unvalidated redirects or forwards is to:
- Review the code for all uses of redirect or forward (called a transfer in .NET). For each use, identify if the target URL is included in any parameter values. If so, verify the parameter(s) are validated to contain only an allowed destination, or element of a destination.
- Also, spider the site to see if it generates any redirects (HTTP response codes 300-307, typically 302). Look at the parameters supplied prior to the redirect to see if they appear to be a target URL or a piece of such a URL. If so, change the URL target and observe whether the site redirects to the new target.
- If code is unavailable, check all parameters to see if they look like part of a redirect or forward URL destination and test those that do. "

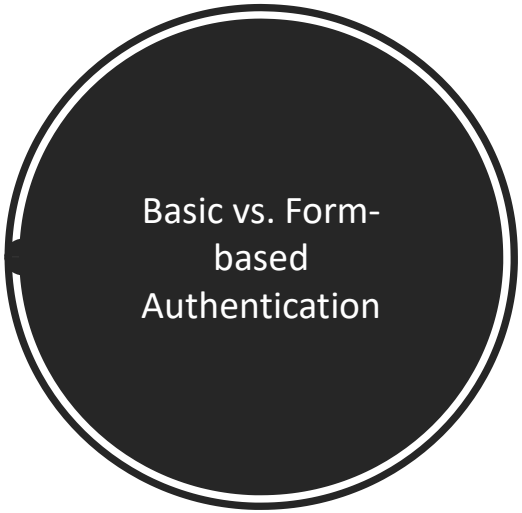


Declarative security authentication



BASIC Authentication

- For declarative security and basic authentication:
- the directory as protected in <url-pattern> in web.xml (for restricted servlet, also declare it in <url-pattern>)
- The server will pop up a standard authentication window asking for username & password



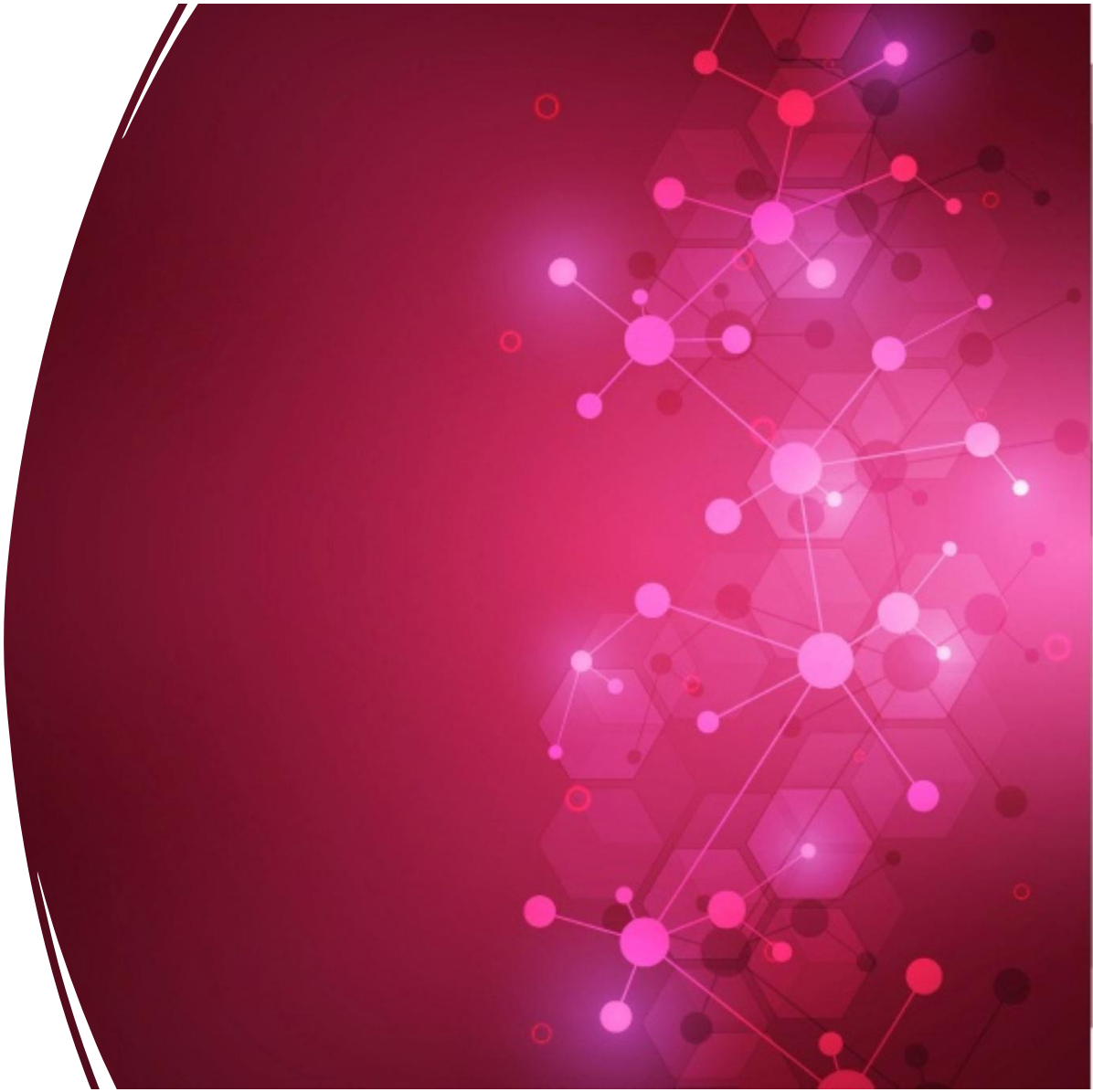
Basic	Form-based
Get username and password by using browser provided dialog box	Get username and password by using a customized login page
Only username and password can be collected	Customized data can be collected (literally whatever the programmer wants)
HTTP Authentication header is used to convey username and password	Form data is used to convey username and password
It is automatic	Programmer has to do the work

OWASP HTML Sanitizer Project https://www.owasp.org/index.php/OWASP_Java_HTML_Sanitizer_Project

- HTML Sanitizer written in Java which lets you include HTML authored by third-parties in your web application and protects against XSS
- Easy to use..
- Maintained by Google's AppSec team
- This is code from the Caja project that was donated by Google. It is rather high performance and low memory utilization.
- Download from https://search.maven.org/#search%7Cga%7C1%7Cowasp%20html%20sanitizer

JavaScript html sanitizer

- https://code.google.com/archive/p/google-caja/wikis/JsHtmlSanitizer.wiki



HTTP Strict transport security

HTTP Strict Transport Security (HSTS) is an opt-in security enhancement that is specified by a web application through the use of a special response header. Once a supported browser receives this header that browser will prevent any communications from being sent over HTTP to the specified domain and will instead send all communications over HTTPS. It also prevents HTTPS click through prompts on browsers**.**
-
https://www.owasp.org/index.php/HTTP_Strict_Transport_Security

Examples
Strict-Transport-Security: max-age=31536000; includeSubDomains

Supported by IE 11, Firefox 4, Opera 12, Safari with OS X 10.9 and Chrome 4.0.211

Encoding Unsafe Output using HtmlEncode

Html encoding is done when the data reads from user **input**, **database**, or **local file**

The attacker uses unsafe characters in the input field to perform injection attacks

The **HtmlEncode** method is used to convert the **unsafe input characters** to their **HTML-encoded** equivalent

HtmlEncode converts **unsafe characters** as follows:

< is converted to **<**

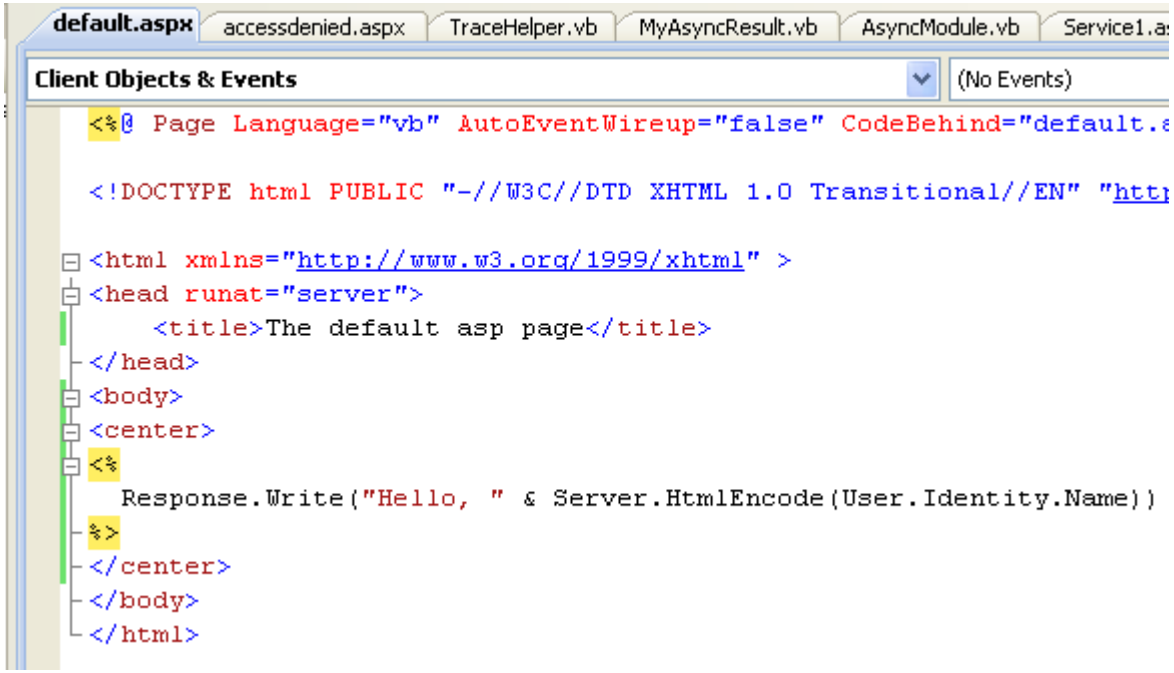
> is converted to **>**

& is converted to **&**

(") is converted to **"**

Encoding Unsafe Output using HtmlEncode

Illustration of `HtmlEncode` Method

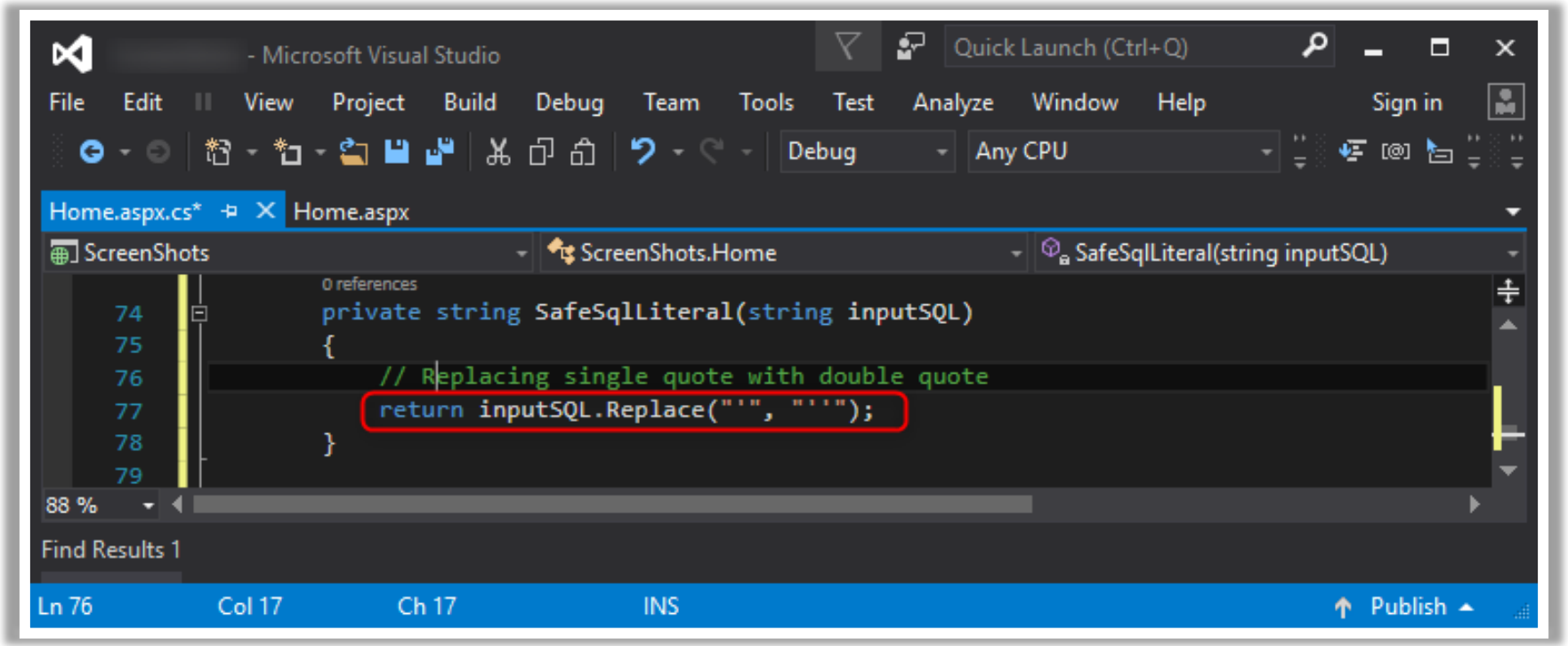


If we run this page and enter some **HTML code** in the input text box, it will produce **safe output**

For example, **<script>say hello;</script>** input is given to the textbox, it will produce safe output as **<script> say hello;</script>** only instead of running **"say hello"** script

Using Escape Routines to Handle Special Input Characters

- This technique is used to escape special characters from **user input** before supplying them to **query**
- It is used when **parameterized queries** or **stored procedures** cannot be used and have no other option besides using **dynamic SQL query**
- In such a situation, it is necessary to safeguard against **special user input characters** supplied that have special meaning to **SQL Server**; if not handled, a character such as (‘) may cause **SQL injection**
- Escape routines are defined to replace the escape characters with characters having special meaning to **SQL Server** thereby avoiding **harmful characters** from being supplied to the **query**



Using Parameterized Queries

Secure Code (Parameterized Query)

```
1  String customer = request.getParameter("username");
2  // Parameterizing
3  String query = "SELECT * FROM users WHERE username = ?";
4  PreparedStatement query_prepared = connection.prepareStatement(query);
5  query_prepared.setString(1, customer);
6  ResultSet results = query_prepared.executeQuery();
```

This parameterized approach of the query helps in preventing the change in the intent of the query, thus preventing an attack

Implementing SSL to Encrypt Cookies

The cookies should be encrypted using SSL whenever they are transmitted over the network in order to prevent them from being stolen

Set the cookieRequireSSL element to true to use SSL for communication in the Web.config file

```
<system.web>
    ...
    <httpCookies httpOnlyCookies="true" requireSSL="true" />
</system.web>
```

Setting a Limited Time Period for Expiration

- If the time period of cookies is set to be less, then the attacker will get less time to steal the cookies, thus reducing the risk of stealing the cookies of that session
- Reduce the cookies' time period in the roleManager properties in the Web.config file

```
<html>
   <head>
      <script>
         <!--
            function WriteCookie() {
               var now = new Date();
               var minutes = 30;
               now.setTime(now.getTime() + (minutes * 60 * 1000));
               cookievalue = escape(document.myform.customer.value) + ";"

               document.cookie="name=" + cookievalue;
               document.cookie = "expires=" + now.toUTCString() + ";"
               document.write ("Setting Cookies : " + "name=" + cookievalue );
            }
         //-->
      </script>
   </head>
   <body>
      <form name="myform" action="">
         Enter name: <input type="text" name="customer"/>
         <input type="button" value="Set Cookie" onclick="WriteCookie()"/>
      </form>
   </body>
</html>
```

Secure JavaScript

1. Avoid Eval() The eval() function in JavaScript is used to evaluate a string as JavaScript code and execute it. While this function can be convenient for some dynamic programming tasks, it poses a significant security risk when misused. Arbitrary Code Execution: If you're using eval() to execute code from an untrusted source, you're giving that source the ability to run arbitrary code on your application. This can lead to various attacks, including code injection and data theft. If you're using eval() to parse JSON data, consider using JSON.parse() instead, which is a safer and faster alternative. If you must use eval(), sanitize and validate the input string to ensure it does not contain malicious code.
2. Use Content Security Policy (CSP) Content-Security-Policy: default-src 'self';
3. Use HTTPS

```
if (window.location.protocol !== "https:") {
window.location.protocol = "https:"; }
```

1. Secure Cookies: document.cookie = "sessionId=12345; Secure; HttpOnly; SameSite=Strict";
2. Avoid Global Variables. Other scripts can modify global variables, leading to potential security issues.

DoD DevSecOps Testing

Automate as much developmental and operational testing and evaluation (OT&E), including functional tests, security tests, and non-functional tests, as possible.

DoD Enterprise DevSecOps Fundamentals Version 2.0

OWASP SAMM

The mission of OWASP Software Assurance Maturity Model (SAMM) is to be the prime maturity model for software assurance that provides an effective and measurable way for all types of organizations to analyze and improve their software security posture. OWASP SAMM supports the complete software lifecycle, including development and acquisition, and is technology and process agnostic. It is intentionally built to be evolutive and risk-driven in nature.

The original model (v1.0) was written by Pravir Chandra and dates back from 2009. Over the last 10 years, it has proven a widely distributed and effective model for improving secure software practices in different types of organizations throughout the world. Translations and supporting tools have been contributed by the community to facilitate adoption and alignment. With version 2.0, we further improve the model to deal with some of its current limitations.

OWASP SAMM

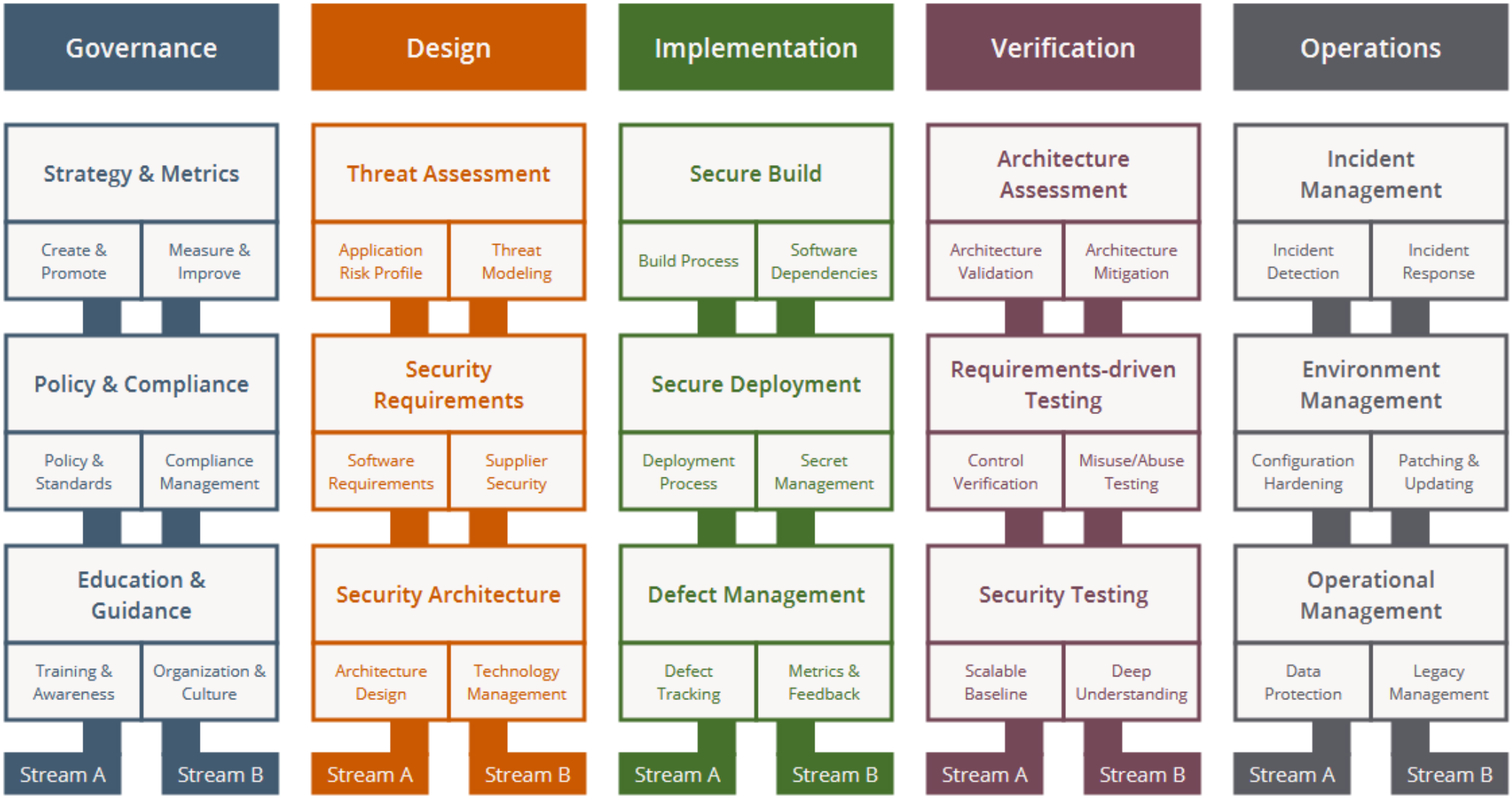
SAMM is based around 15 security practices grouped into 5 business

functions. Every security practice contains a set of activities, structured into 3 maturity levels. The activities on a lower maturity level are typically easier to execute and require less formalization than the ones on a higher maturity level.

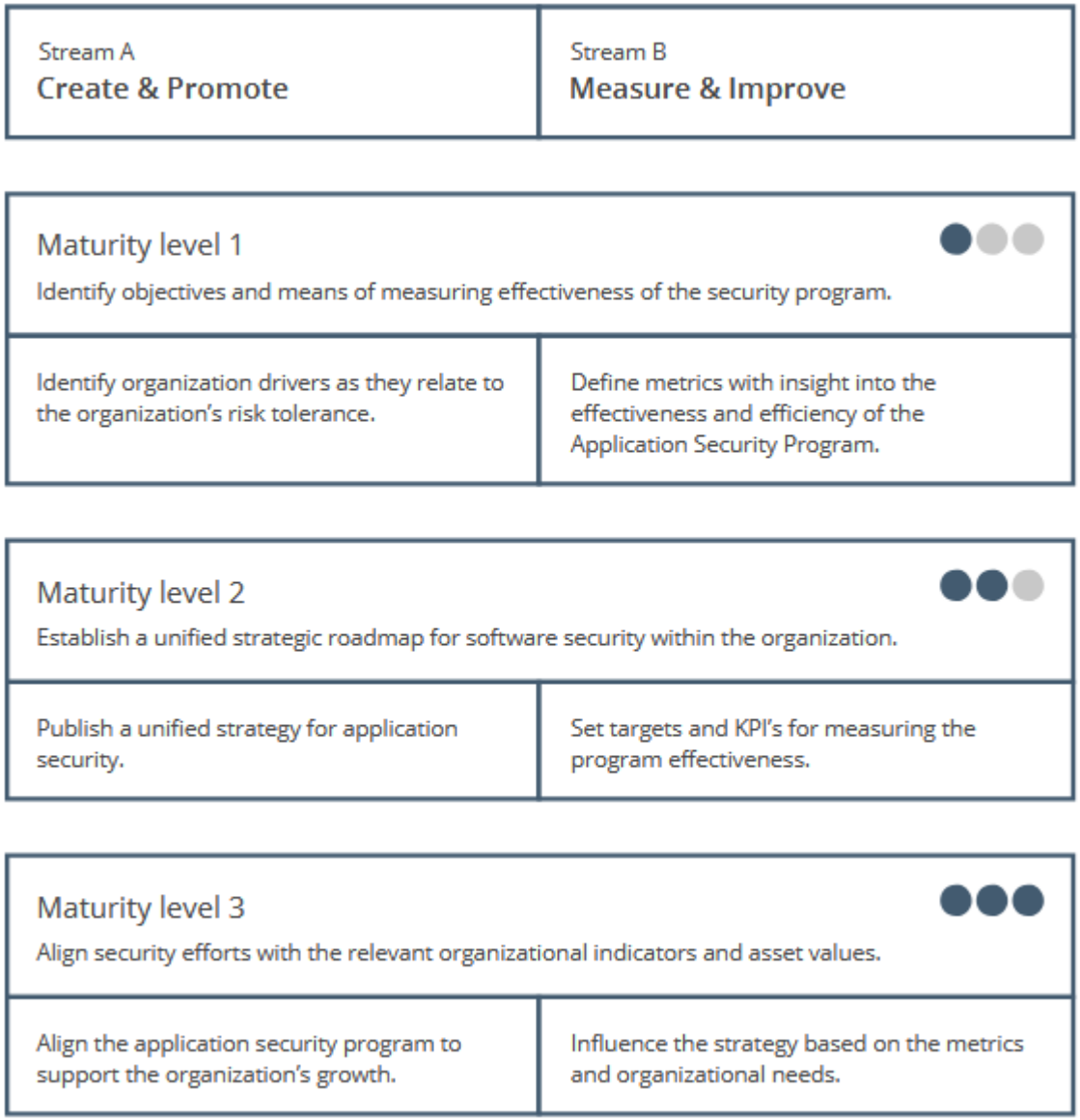
At the highest level, SAMM defines five business functions. Each business function is a category of activities that any organization involved with software development must fulfill to some degree.

Each business function has three security practices, areas of security-related activities that build assurance for the related business function.

OWASP SAMM


Governance
Design
Implementation
Verification
Operations
Strategy & Metrics
Create & Promote
Measure & Improve
Policy & Compliance
Policy & Standards
Compliance Management
Education & Guidance
Training & Awareness
Organization & Culture
Stream A
Stream B
Threat Assessment
Application Risk Profile
Threat Modeling
Security Requirements
Software Requirements
Supplier Security
Security Architecture
Architecture Design
Technology Management
Stream A
Stream B
Secure Build
Build Process
Software Dependencies
Secure Deployment
Deployment Process
Secret Management
Defect Management
Defect Tracking
Metrics & Feedback
Stream A
Stream B
Architecture Assessment
Architecture Validation
Architecture Mitigation
Requirements-driven Testing
Control Verification
Misuse/Abuse Testing
Security Testing
Scalable Baseline
Deep Understanding
Stream A
Stream B
Incident Management
Incident Detection
Incident Response
Environment Management
Configuration Hardening
Patching & Updating
Operational Management
Data Protection
Legacy Management
Stream A
Stream B

OWASP SAMM


Stream A
Create & Promote
Stream B
Measure & Improve
Maturity level 1
Identify objectives and means of measuring effectiveness of the security program.
Identify organization drivers as they relate to the organization's risk tolerance.
Define metrics with insight into the effectiveness and efficiency of the Application Security Program.
Maturity level 2
Establish a unified strategic roadmap for software security within the organization.
Publish a unified strategy for application security.
Set targets and KPI's for measuring the program effectiveness.
Maturity level 3
Align security efforts with the relevant organizational indicators and asset values.
Align the application security program to support the organization's growth.
Influence the strategy based on the metrics and organizational needs.

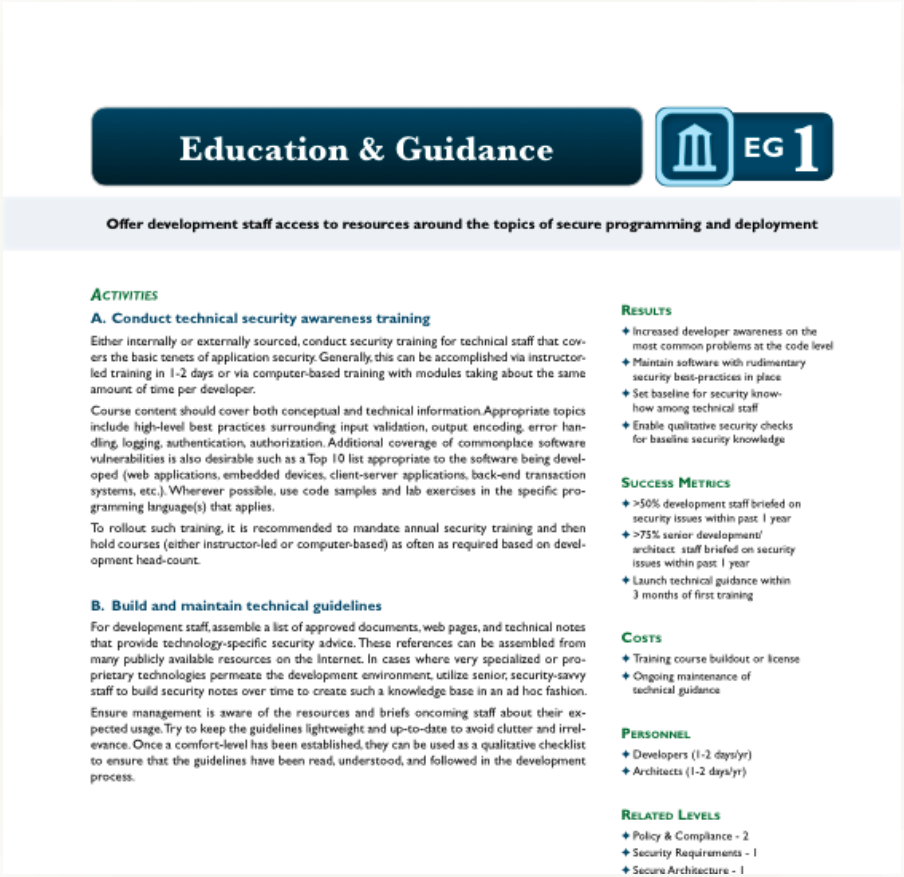
Under each Security Practice

- Three successive Objectives under each Practice define how it can be improved over time
  - This establishes a notion of a Level at which an organization fulfills a given Practice

- The three Levels for a Practice generally correspond to:
  - (0: Implicit starting point with the Practice unfulfilled)
  - 1: Initial understanding and ad hoc provision of the Practice
  - 2: Increase efficiency and/or effectiveness of the Practice
  - 3: Comprehensive mastery of the Practice at scale

Per Level, SAMM defines

- Objective
- Activities
- Results
- Success Metrics
- Costs
- Personnel
- Related Levels



Education & Guidance

EG 1

Offer development staff access to resources around the topics of secure programming and deployment

Activities

A. Conduct technical security awareness training

Either internally or externally sourced, conduct security training for technical staff that covers the basic tenets of application security. Generally, this can be accomplished via instructor-led training in 1-2 days or via computer-based training with modules taking about the same amount of time per developer.

Course content should cover both conceptual and technical information. Appropriate topics include high-level best practices surrounding input validation, output encoding, error handling, logging, authentication, authorization. Additional coverage of commonplace software vulnerabilities is also desirable such as a Top 10 list appropriate to the software being developed (web applications, embedded devices, client-server applications, back-end transaction systems, etc.). Wherever possible, use code samples and lab exercises in the specific programming language(s) that applies.

To rollout such training, it is recommended to mandate annual security training and then hold courses (either instructor-led or computer-based) as often as required based on development head-count.

B. Build and maintain technical guidelines

For development staff, assemble a list of approved documents, web pages, and technical notes that provide technology-specific security advice. These references can be assembled from many publicly available resources on the Internet. In cases where very specialized or proprietary technologies permeate the development environment, utilize senior, security-savvy staff to build security notes over time to create such a knowledge base in an ad hoc fashion.

Ensure management is aware of the resources and briefs oncoming staff about their expected usage. Try to keep the guidelines lightweight and up-to-date to avoid clutter and irrelevance. Once a comfort-level has been established, they can be used as a qualitative checklist to ensure that the guidelines have been read, understood, and followed in the development process.

Results

- Increased developer awareness on the most common problems at the code level
- Maintain software with rudimentary security best-practices in place
- Set baseline for security know-how among technical staff
- Enable qualitative security checks for baseline security knowledge

Success Metrics

- >50% development staff briefed on security issues within past 1 year
- >75% senior development/ architect staff briefed on security issues within past 1 year
- Launch technical guidance within 3 months of first training

Costs

- Training course buildout or license
- Ongoing maintenance of technical guidance

Personnel

- Developers (1-2 days/yr)
- Architects (1-2 days/yr)

Related Levels

- Policy & Compliance - 2
- Security Requirements - 1
- Secure Architecture - 1