



Introduction to Software Security

Backend: Security Best Practices

Fandi Gunawan
Senior Engineer
Teknologi Nasional Indonesia Siber
fandi.septian@tnisiber.id

Download and View Presentation



<https://s.id/softsec-arkavidia>

About Fandi Gunawan

Senior engineer specialized in creating cyber security solutions with secure architecture.

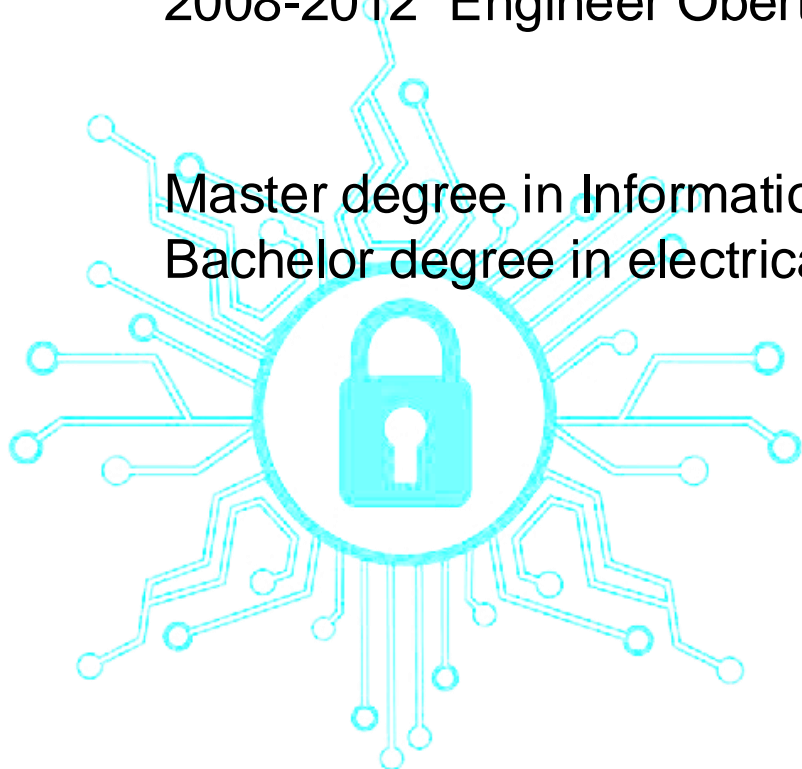
2018-now Senior Engineer Teknologi Nasional Indonesia Siber

2012-2017 Engineer NSS Focus

2008-2012 Engineer Oberthur Card Systems

Master degree in Information Technology – Universitas Indonesia

Bachelor degree in electrical Engineering – President University



Software Insecurity Creates Havoc

WIRED

SECURITY POLITICS GEAR THE BIG STORY BUSINESS SCIENCE CULTURE IDEAS MERCH

SIGN IN

SUBSCRIBE

LILY HAY NEWMAN

MATT BURGESS

ANDY GREENBERG

SECURITY

JUL 19, 2024 10:46 AM

How One Bad CrowdStrike Update Crashed the World's Computers

A defective CrowdStrike update sent computers around the globe into a reboot death spiral, taking down air travel, hospitals, banks, and more with it. Here's how that's possible.

CYBERCRIME

Millions of User Records Stolen From 65 Websites via SQL Injection Attacks

The ResumeLooters hackers compromise recruitment and retail websites using SQL injection and XSS attacks.

BSSN Akui Data Lama INAFIS Bocor di Dark Web

CNN Indonesia

Senin, 24 Jun 2024 17:28 WIB



Massive IT outage causes global chaos

A global tech outage is disrupting companies around the world. The US is issuing a ground stop on all its flights worldwide, delays are hitting Berlin and Amsterdam's Schiphol airport — among many others — and banks are also affected. In the UK, the London stock exchange was temporarily affected along with the Sky News channel which was unable to broadcast for several hours.

TECHNOLOGY | 07/19/2024 | 01:17 min

KEJAHATAN SIBER

Situs Kementerian Pertahanan Diduga Diretas, Dokumen Rahasia Berpotensi Bocor

Data Kementerian Pertahanan sebesar 1,64 terabita berpotensi bocor setelah situs resmi kementerian itu dibobol peretas.

PHP Vulnerability Exploited to Spread Malware and Launch DDoS Attacks

Jul 11, 2024 Ravie Lakshmanan

Cyber Attack / Vulnerability



Ingatlah, bahwa kechilafan satu orang sahaja
tjukup sudah menjebabkan keruntuhan negara

dr. Roebiono Kertopati, pendiri Lemsaneg/BSSN



A chain is only as strong as its weakest link.





people

process



technology



What is Software Security (McGraw)

“Software Security is the ability of the software to resist, tolerate, and recover from events that intentionally threaten its dependability”

“Software Security is about building secure software: designing software to be secure, making sure that software is secure, and educating software developers, architects, and users about how to build secure things”.

“The idea of engineering software that continues to function correctly under malicious attack”

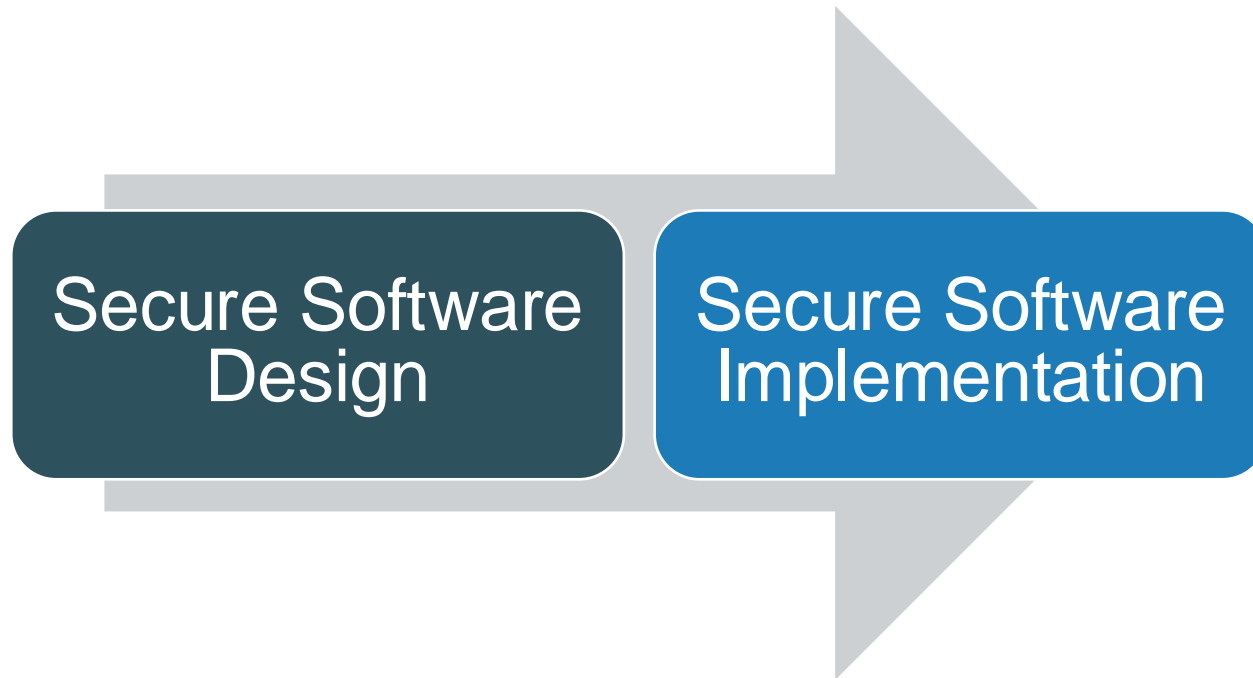
“The process of designing, building, and testing software for security”

“Defends against software exploits by building software to be secure in the first place, mostly by getting the design right (which is hard) and avoiding common mistakes”

“Software Security is a system-wide issue that takes into account both security mechanisms (such as access control) and design for security (such as robust design that make software attacks difficult)”

What is Software Security

Building secure software which is actually means to **design** and **implement** secure software from the **beginning** of **software development**.



SDLC vs Secure SDLC

SDLC Process



Secure SDLC Process



Secure Software Design Principles (1/2)

System

- Session Management
- Exception Management
- Configuration Management

Secure Design

- Least Privilege
- Separation of Duties
- Defense in Depth
- Single Point of Failure

Secure Software Design Principles (2/2)

Security Model

- Access Control Models
- Integrity Models
- Information Flows Models

Adversaries

- Threat landscape
- Adversary groups and type

Secure Software Requirements (1/3)

Functional Requirements

- Role and User Definitions
- Activities and Actions
- Secure Coding Standards

Operational and Deployment Requirements

- Logging
- Security Monitoring
- DevSecOps

Secure Software Requirements (2/3)

Data Management

- Data types and usage
- Data lifecycle
- Data ownership
- Encryption

Privacy

- Privacy Policy
- Data Masking
- Anonymization
- Personally Identifiable Information (PII)

Secure Software Requirements (3/3)

Regulations and Standards

- ISO 27001
- GDPR
- PCI DSS
- UU ITE dan UU PDP
- Aturan BI/OJK

Threat Modeling

- Threat Model Development
- Attack Surface
- Threat Intelligence
- Threat Hunting

Secure Software Architecture

Software Architecture

- Security Control
- Software Model
 - Distributed
 - Service Oriented
 - Embedded
 - Cloud
 - Mobile Applications

Secure Software Implementation

Programming Language

- Compiled vs Interpreted
- Native vs VM
- Static vs dynamic type
- Frameworks maturity
- Type safe
- Error handling

Human Aspects

- Capabilities
- Lessons learned
- Personnel trainings

Secure Software Implementation

Cryptography

- Securing credentials
- Encryption type (symmetric vs assymetric)
- Hash
- Implementation of secure cryptography

Input Validation

- Enforcing input type
- Escaping strings
- Buffer overflow preventions

Secure Software Implementation

Code Analysis and Review

- Static Application Security Testing
- Dynamic Application Security Testing
- Bug hunting
- Manual code review

Security Controls

- Coding standards and lint
- Anti tampering
 - Code signing
 - Code obfuscation
- API Security Gateway

Secure Software Implementation

Security Test

- Security Test Cases and Development
- Attack Surface Evaluation and Simulation
- Penetration test
- Bug hunting program

Secure Deployment, Ops, and Monitoring

- Continuous Integration/Continuous Deployment
- Hardening
- Monitoring
- Continuous code and artifacts audits



Open Web Application Security Project (OWASP)

- Nonprofit foundation dedicated to improving software security
- Operates under an “open community” model, meaning that anyone can participate in and contribute to OWASP-related online chats, projects, and more

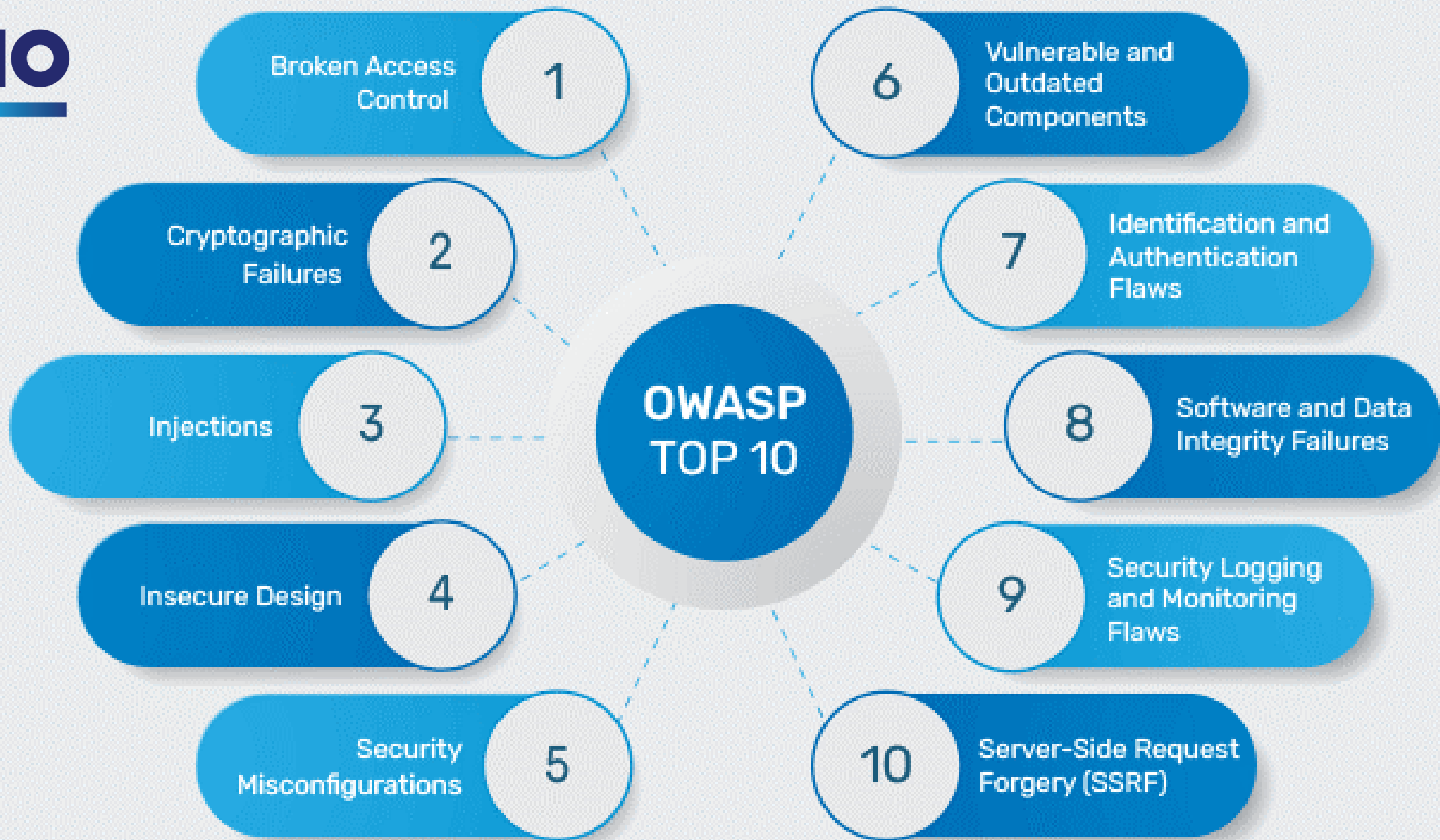
OWASP Top 10

- A standard awareness document for developers and web application security
- Represents a broad consensus about the most critical security risks to web applications

Bare minimal!

Not a compliance

If you does not meet all the category does not mean your application is secure



A01:2021 – Broken Access Control

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.

How to Prevent

Access control is only effective in trusted server-side code or server-less API, where the attacker cannot modify the access control check or metadata.

- **Except for public resources, deny by default.**
- **Implement access control mechanisms once and re-use them throughout the application, including minimizing Cross-Origin Resource Sharing (CORS) usage.**
- **Model access controls should enforce record ownership rather than accepting that the user can create, read, update, or delete any record.**
- **Unique application business limit requirements should be enforced by domain models.**
- **Disable web server directory listing and ensure file metadata (e.g., .git) and backup files are not present within web roots.**
- **Log access control failures, alert admins when appropriate (e.g., repeated failures).**
- **Rate limit API and controller access to minimize the harm from automated attack tooling.**
- **Stateful session identifiers should be invalidated on the server after logout. Stateless JWT tokens should rather be short-lived so that the window of opportunity for an attacker is minimized. For longer lived JWTs it's highly recommended to follow the OAuth standards to revoke access.**

Example Attack Scenarios

Scenario #1: The application uses unverified data in a SQL call that is accessing account information:

```
pstmt.setString(1, request.getParameter("acct"));
```

```
ResultSet results = pstmt.executeQuery( );
```

An attacker simply modifies the browser's 'acct' parameter to send whatever account number they want. If not correctly verified, the attacker can access any user's account.

<https://example.com/app/accountInfo?acct=notmyacct>

Scenario #2: An attacker simply forces browses to target URLs. Admin rights are required for access to the admin page.

<https://example.com/app/getappInfo>

https://example.com/app/admin_getappInfo

If an unauthenticated user can access either page, it's a flaw. If a non-admin can access the admin page, this is a flaw.

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? Use secure protocol
- 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?

A02:2021 – Cryptographic Failures

- 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?

How to Prevent

Do the following, at a minimum, and consult the references:

- **Classify data processed, stored, or transmitted by an application. Identify which data is sensitive according to privacy laws, regulatory requirements, or business needs.**
- **Don't store sensitive data unnecessarily. Discard it as soon as possible or use PCI DSS compliant tokenization or even truncation. Data that is not retained cannot be stolen.**
- **Make sure to encrypt all sensitive data at rest.**
- **Ensure up-to-date and strong standard algorithms, protocols, and keys are in place; use proper key management.**
- **Encrypt all data in transit with secure protocols such as TLS with forward secrecy (FS) ciphers, cipher prioritization by the server, and secure parameters. Enforce encryption using directives like HTTP Strict Transport Security (HSTS).**
- **Disable caching for response that contain sensitive data.**
- **Apply required security controls as per the data classification.**

- Do not use legacy protocols such as FTP and SMTP for transporting sensitive data.
- Store passwords using strong adaptive and salted hashing functions with a work factor (delay factor), such as Argon2, scrypt, bcrypt or PBKDF2.
- Initialization vectors must be chosen appropriate for the mode of operation. For many modes, this means using a CSPRNG (cryptographically secure pseudo random number generator). For modes that require a nonce, then the initialization vector (IV) does not need a CSPRNG. In all cases, the IV should never be used twice for a fixed key.
- Always use authenticated encryption instead of just encryption.
- Keys should be generated cryptographically randomly and stored in memory as byte arrays. If a password is used, then it must be converted to a key via an appropriate password base key derivation function.
- Ensure that cryptographic randomness is used where appropriate, and that it has not been seeded in a predictable way or with low entropy. Most modern APIs do not require the developer to seed the CSPRNG to get security.
- Avoid deprecated cryptographic functions and padding schemes, such as MD5, SHA1, PKCS number 1 v1.5.
- Verify independently the effectiveness of configuration and settings.

Example Attack Scenarios

Scenario #1: An application encrypts credit card numbers in a database using automatic database encryption. However, this data is automatically decrypted when retrieved, allowing a SQL injection flaw to retrieve credit card numbers in clear text.

Scenario #2: A site doesn't use or enforce TLS for all pages or supports weak encryption. An attacker monitors network traffic (e.g., at an insecure wireless network), downgrades connections from HTTPS to HTTP, intercepts requests, and steals the user's session cookie. The attacker then replays this cookie and hijacks the user's (authenticated) session, accessing or modifying the user's private data. Instead of the above they could alter all transported data, e.g., the recipient of a money transfer.

Scenario #3: The password database uses unsalted or simple hashes to store everyone's passwords. A file upload flaw allows an attacker to retrieve the password database. All the unsalted hashes can be exposed with a rainbow table of pre-calculated hashes. Hashes generated by simple or fast hash functions may be cracked by GPUs, even if they were salted.

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.

How to Prevent

Preventing injection requires keeping data separate from commands and queries:

- The preferred option is to use a safe API, which avoids using the interpreter entirely, provides a parameterized interface, or migrates to Object Relational Mapping Tools (ORMs).
- Note: Even when parameterized, stored procedures can still introduce SQL injection if PL/SQL or T-SQL concatenates queries and data or executes hostile data with `EXECUTE IMMEDIATE` or `exec()`.
- Use positive server-side input validation. This is not a complete defense as many applications require special characters, such as text areas or APIs for mobile applications.
- For any residual dynamic queries, escape special characters using the specific escape syntax for that interpreter.
- Note: SQL structures such as table names, column names, and so on cannot be escaped, and thus user-supplied structure names are dangerous. This is a common issue in report-writing software.
- Use `LIMIT` and other SQL controls within queries to prevent mass disclosure of records in case of SQL injection.

Example Attack Scenarios

Scenario #1: An application uses untrusted data in the construction of the following vulnerable SQL call:

```
String query = "SELECT \* FROM accounts WHERE custID=" + request.getParameter("id") + "";
```

Scenario #2: Similarly, an application's blind trust in frameworks may result in queries that are still vulnerable, (e.g., Hibernate Query Language (HQL)):

```
Query HQLQuery = session.createQuery("FROM accounts WHERE custID=" + request.getParameter("id") + "");
```

In both cases, the attacker modifies the 'id' parameter value in their browser to send: ' UNION SLEEP(10);--. For example:

```
http://example.com/app/accountView?id=' UNION SELECT SLEEP(10);--
```

This changes the meaning of both queries to return all the records from the accounts table. More dangerous attacks could modify or delete data or even invoke stored procedures.

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.

Requirements and Resource Management

Collect and negotiate the business requirements for an application with the business, including the protection requirements concerning confidentiality, integrity, availability, and authenticity of all data assets and the expected business logic. Take into account how exposed your application will be and if you need segregation of tenants (additionally to access control). Compile the technical requirements, including functional and non-functional security requirements. Plan and negotiate the budget covering all design, build, testing, and operation, including security activities.

Secure Design

Secure design is a culture and methodology that constantly evaluates threats and ensures that code is robustly designed and tested to prevent known attack methods. Threat modeling should be integrated into refinement sessions (or similar activities); look for changes in data flows and access control or other security controls. In the user story development determine the correct flow and failure states, ensure they are well understood and agreed upon by responsible and impacted parties. Analyze assumptions and conditions for expected and failure flows, ensure they are still accurate and desirable. Determine how to validate the assumptions and enforce conditions needed for proper behaviors. Ensure the results are documented in the user story. Learn from mistakes and offer positive incentives to promote improvements. Secure design is neither an add-on nor a tool that you can add to software.

Secure Development Lifecycle

Secure software requires a secure development lifecycle, some form of secure design pattern, paved road methodology, secured component library, tooling, and threat modeling. Reach out for your security specialists at the beginning of a software project throughout the whole project and maintenance of your software.

How to Prevent

- Establish and use a secure development lifecycle with AppSec professionals to help evaluate and design security and privacy-related controls
- Establish and use a library of secure design patterns or paved road ready to use components
- Use threat modeling for critical authentication, access control, business logic, and key flows
- Integrate security language and controls into user stories
- Integrate plausibility checks at each tier of your application (from frontend to backend)
- Write unit and integration tests to validate that all critical flows are resistant to the threat model. Compile use-cases and misuse-cases for each tier of your application.
- Segregate tier layers on the system and network layers depending on the exposure and protection needs
- Segregate tenants robustly by design throughout all tiers
- Limit resource consumption by user or service

Example Attack Scenarios

Scenario #1: A credential recovery workflow might include “questions and answers,” which is prohibited by NIST 800-63b, the OWASP ASVS, and the OWASP Top 10. Questions and answers cannot be trusted as evidence of identity as more than one person can know the answers, which is why they are prohibited. Such code should be removed and replaced with a more secure design.

Scenario #2: A cinema chain allows group booking discounts and has a maximum of fifteen attendees before requiring a deposit. Attackers could threat model this flow and test if they could book six hundred seats and all cinemas at once in a few requests, causing a massive loss of income.

Scenario #3: A retail chain’s e-commerce website does not have protection against bots run by scalpers buying high-end video cards to resell auction websites. This creates terrible publicity for the video card makers and retail chain owners and enduring bad blood with enthusiasts who cannot obtain these cards at any price. Careful anti-bot design and domain logic rules, such as purchases made within a few seconds of availability, might identify inauthentic purchases and rejected such transactions.

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.**
- **Without a concerted, repeatable application security configuration process, systems are at a higher risk.**

How to Prevent

Secure installation processes should be implemented, including:

- **A repeatable hardening process makes it fast and easy to deploy another environment that is appropriately locked down. Development, QA, and production environments should all be configured identically, with different credentials used in each environment. This process should be automated to minimize the effort required to set up a new secure environment.**
- **A minimal platform without any unnecessary features, components, documentation, and samples. Remove or do not install unused features and frameworks.**
- **A task to review and update the configurations appropriate to all security notes, updates, and patches as part of the patch management process. Review cloud storage permissions (e.g., S3 bucket permissions).**
- **A segmented application architecture provides effective and secure separation between components or tenants, with segmentation, containerization, or cloud security groups (ACLs).**
- **Sending security directives to clients, e.g., Security Headers.**
- **An automated process to verify the effectiveness of the configurations and settings in all environments.**

Example Attack Scenarios

Scenario #1: The application server comes with sample applications not removed from the production server. These sample applications have known security flaws attackers use to compromise the server. Suppose one of these applications is the admin console, and default accounts weren't changed. In that case, the attacker logs in with default passwords and takes over.

Scenario #2: Directory listing is not disabled on the server. An attacker discovers they can simply list directories. The attacker finds and downloads the compiled Java classes, which they decompile and reverse engineer to view the code. The attacker then finds a severe access control flaw in the application.

Scenario #3: The application server's configuration allows detailed error messages, e.g., stack traces, to be returned to users. This potentially exposes sensitive information or underlying flaws such as component versions that are known to be vulnerable.

Scenario #4: A cloud service provider (CSP) has default sharing permissions open to the Internet by other CSP users. This allows sensitive data stored within cloud storage to be accessed.

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**

A06:2021 – Vulnerable and Outdated Components

How to Prevent

There should be a patch management process in place to:

- Remove unused dependencies, unnecessary features, components, files, and documentation.
- Continuously inventory the versions of both client-side and server-side components (e.g., frameworks, libraries) and their dependencies using tools like versions, OWASP Dependency Check, retire.js, etc. Continuously monitor sources like Common Vulnerability and Exposures (CVE) and National Vulnerability Database (NVD) for vulnerabilities in the components. Use software composition analysis tools to automate the process. Subscribe to email alerts for security vulnerabilities related to components you use.
- Only obtain components from official sources over secure links. Prefer signed packages to reduce the chance of including a modified, malicious component.
- Monitor for libraries and components that are unmaintained or do not create security patches for older versions. If patching is not possible, consider deploying a virtual patch to monitor, detect, or protect against the discovered issue.
- Every organization must ensure an ongoing plan for monitoring, triaging, and applying updates or configuration changes for the lifetime of the application or portfolio.

A06:2021 – Vulnerable and Outdated Components

Example Attack Scenarios

Scenario #1: Components typically run with the same privileges as the application itself, so flaws in any component can result in serious impact. Such flaws can be accidental (e.g., coding error) or intentional (e.g., a backdoor in a component). Some example exploitable component vulnerabilities discovered are:

CVE-2017-5638, a Struts 2 remote code execution vulnerability that enables the execution of arbitrary code on the server, has been blamed for significant breaches.

While the internet of things (IoT) is frequently difficult or impossible to patch, the importance of patching them can be great (e.g., biomedical devices).

There are automated tools to help attackers find unpatched or misconfigured systems. For example, the Shodan IoT search engine can help you find devices that still suffer from Heartbleed vulnerability patched in April 2014.

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.
- 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.

A07:2021 – Identification and Authentication Failures

- Where possible, implement multi-factor authentication to prevent automated credential stuffing, brute force, and stolen credential reuse attacks.
- Do not ship or deploy with any default credentials, particularly for admin users.
- Implement weak password checks, such as testing new or changed passwords against the top 10,000 worst passwords list.
- Align password length, complexity, and rotation policies with National Institute of Standards and Technology (NIST) 800-63b's guidelines in section 5.1.1 for Memorized Secrets or other modern, evidence-based password policies.
- Ensure registration, credential recovery, and API pathways are hardened against account enumeration attacks by using the same messages for all outcomes.
- Limit or increasingly delay failed login attempts, but be careful not to create a denial of service scenario. Log all failures and alert administrators when credential stuffing, brute force, or other attacks are detected.
- Use a server-side, secure, built-in session manager that generates a new random session ID with high entropy after login. Session identifier should not be in the URL, be securely stored, and invalidated after logout, idle, and absolute timeouts.

A07:2021 – Identification and Authentication Failures

Example Attack Scenarios

Scenario #1: Credential stuffing, the use of lists of known passwords, is a common attack. Suppose an application does not implement automated threat or credential stuffing protection. In that case, the application can be used as a password oracle to determine if the credentials are valid.

Scenario #2: Most authentication attacks occur due to the continued use of passwords as a sole factor. Once considered best practices, password rotation and complexity requirements encourage users to use and reuse weak passwords. Organizations are recommended to stop these practices per NIST 800-63 and use multi-factor authentication.

Scenario #3: Application session timeouts aren't set correctly. A user uses a public computer to access an application. Instead of selecting "logout," the user simply closes the browser tab and walks away. An attacker uses the same browser an hour later, and the user is still authenticated.

TOP10 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.

TOP10 A08:2021 – Software and Data Integrity Failures

How to Prevent

- Use digital signatures or similar mechanisms to verify the software or data is from the expected source and has not been altered.
- Ensure libraries and dependencies, such as npm or Maven, are consuming trusted repositories. If you have a higher risk profile, consider hosting an internal known-good repository that's vetted.
- Ensure that a software supply chain security tool to verify that components do not contain known vulnerabilities
- Ensure that there is a review process for code and configuration changes to minimize the chance that malicious code or configuration could be introduced into your software pipeline.
- Ensure that your CI/CD pipeline has proper segregation, configuration, and access control to ensure the integrity of the code flowing through the build and deploy processes.
- Ensure that unsigned or unencrypted serialized data is not sent to untrusted clients without some form of integrity check or digital signature to detect tampering or replay of the serialized data

TOP10 A08:2021 – Software and Data Integrity Failures

Example Attack Scenarios

Scenario #1 Update without signing: Many home routers, set-top boxes, device firmware, and others do not verify updates via signed firmware. Unsigned firmware is a growing target for attackers and is expected to only get worse. This is a major concern as many times there is no mechanism to remediate other than to fix in a future version and wait for previous versions to age out.

Scenario #2 SolarWinds malicious update: Nation-states have been known to attack update mechanisms, with a recent notable attack being the SolarWinds Orion attack. The company that develops the software had secure build and update integrity processes. Still, these were able to be subverted, and for several months, the firm distributed a highly targeted malicious update to more than 18,000 organizations, of which around 100 or so were affected. This is one of the most far-reaching and most significant breaches of this nature in history.

Scenario #3 Insecure Deserialization: A React application calls a set of Spring Boot microservices. Being functional programmers, they tried to ensure that their code is immutable. The solution they came up with is serializing the user state and passing it back and forth with each request. An attacker notices the "rO0" Java object signature (in base64) and uses the Java Serial Killer tool to gain remote code execution on the application server.

A09:2021 – Security Logging and Monitoring Failures

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**
- **The application cannot detect, escalate, or alert for active attacks in real-time or near real-time.**

A09:2021 – Security Logging and Monitoring Failures

Developers should implement some or all the following controls, depending on the risk of the application:

- **Ensure all login, access control, and server-side input validation failures can be logged with sufficient user context to identify suspicious or malicious accounts and held for enough time to allow delayed forensic analysis.**
- **Ensure that logs are generated in a format that log management solutions can easily consume.**
- **Ensure log data is encoded correctly to prevent injections or attacks on the logging or monitoring systems.**
- **Ensure high-value transactions have an audit trail with integrity controls to prevent tampering or deletion, such as append-only database tables or similar.**
- **DevSecOps teams should establish effective monitoring and alerting such that suspicious activities are detected and responded to quickly.**
- **Establish or adopt an incident response and recovery plan, such as National Institute of Standards and Technology (NIST) 800-61r2 or later.**

Example Attack Scenarios

Scenario #1: A children's health plan provider's website operator couldn't detect a breach due to a lack of monitoring and logging. An external party informed the health plan provider that an attacker had accessed and modified thousands of sensitive health records of more than 3.5 million children. A post-incident review found that the website developers had not addressed significant vulnerabilities. As there was no logging or monitoring of the system, the data breach could have been in progress since 2013, a period of more than seven years.

Scenario #2: A major Indian airline had a data breach involving more than ten years' worth of personal data of millions of passengers, including passport and credit card data. The data breach occurred at a third-party cloud hosting provider, who notified the airline of the breach after some time.

Scenario #3: A major European airline suffered a GDPR reportable breach. The breach was reportedly caused by payment application security vulnerabilities exploited by attackers, who harvested more than 400,000 customer payment records. The airline was fined 20 million pounds as a result by the privacy regulator.

TOP10 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.

TOP10 A10:2021 – Server-Side Request Forgery (SSRF)

How to Prevent

Developers can prevent SSRF by implementing some or all the following defense in depth controls:

From Network layer

- Segment remote resource access functionality in separate networks to reduce the impact of SSRF
- Enforce “deny by default” firewall policies or network access control rules to block all but essential intranet traffic.

Hints:

- ~ Establish an ownership and a lifecycle for firewall rules based on applications.
- ~ Log all accepted and blocked network flows on firewalls (see A09:2021-Security Logging and Monitoring Failures).

TOP10 A10:2021 – Server-Side Request Forgery (SSRF)

From Application layer:

- Sanitize and validate all client-supplied input data
- Enforce the URL schema, port, and destination with a positive allow list
- Do not send raw responses to clients
- Disable HTTP redirections
- Be aware of the URL consistency to avoid attacks such as DNS rebinding and “time of check, time of use” (TOCTOU) race conditions
- Do not mitigate SSRF via the use of a deny list or regular expression. Attackers have payload lists, tools, and skills to bypass deny lists.

Additional Measures to consider:

- Don't deploy other security relevant services on front systems (e.g. OpenID). Control local traffic on these systems (e.g. localhost)
- For frontends with dedicated and manageable user groups use network encryption (e.g. VPNs) on independent systems to consider very high protection needs

TOP10 A10:2021 – Server-Side Request Forgery (SSRF)

Example Attack Scenarios

Attackers can use SSRF to attack systems protected behind web application firewalls, firewalls, or network ACLs, using scenarios such as:

Scenario #1: Port scan internal servers – If the network architecture is unsegmented, attackers can map out internal networks and determine if ports are open or closed on internal servers from connection results or elapsed time to connect or reject SSRF payload connections.

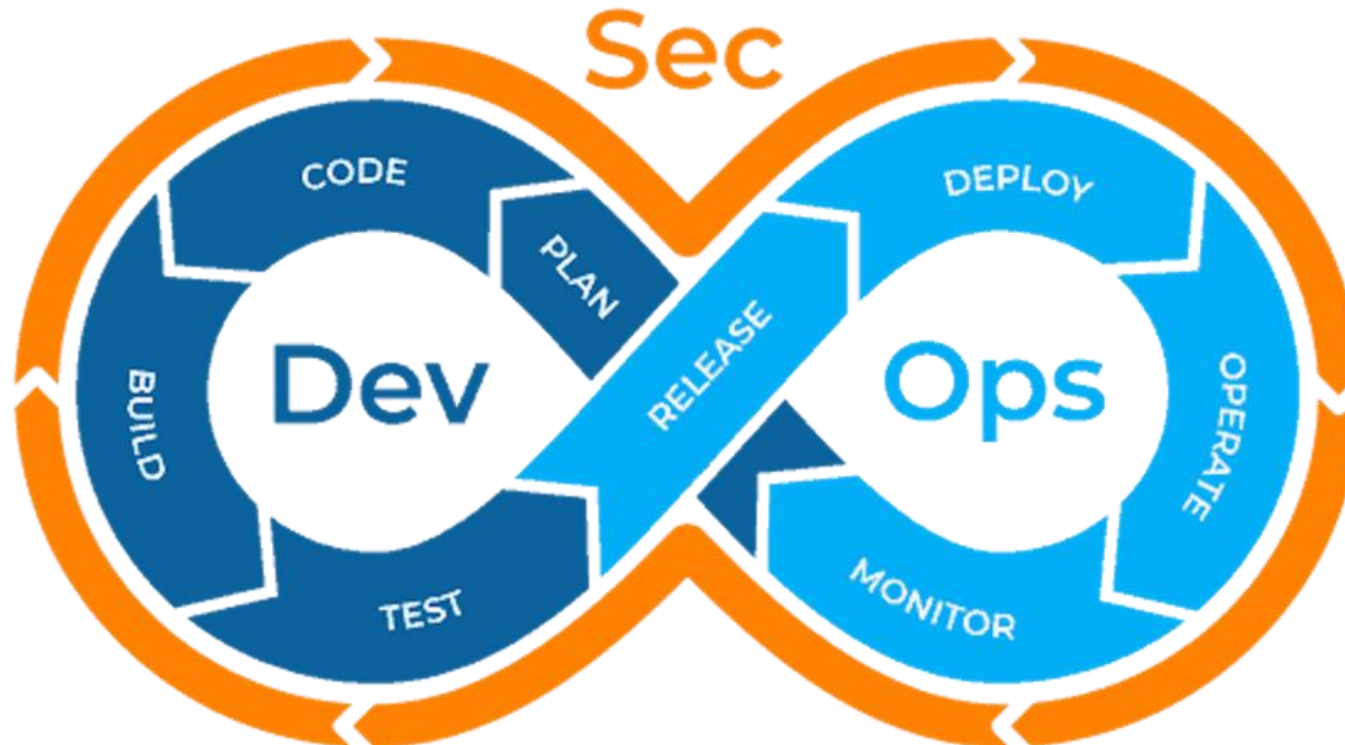
Scenario #2: Sensitive data exposure – Attackers can access local files or internal services to gain sensitive information such as `file:///etc/passwd` and `http://localhost:28017/`.

Scenario #3: Access metadata storage of cloud services – Most cloud providers have metadata storage such as `http://169.254.169.254/`. An attacker can read the metadata to gain sensitive information.

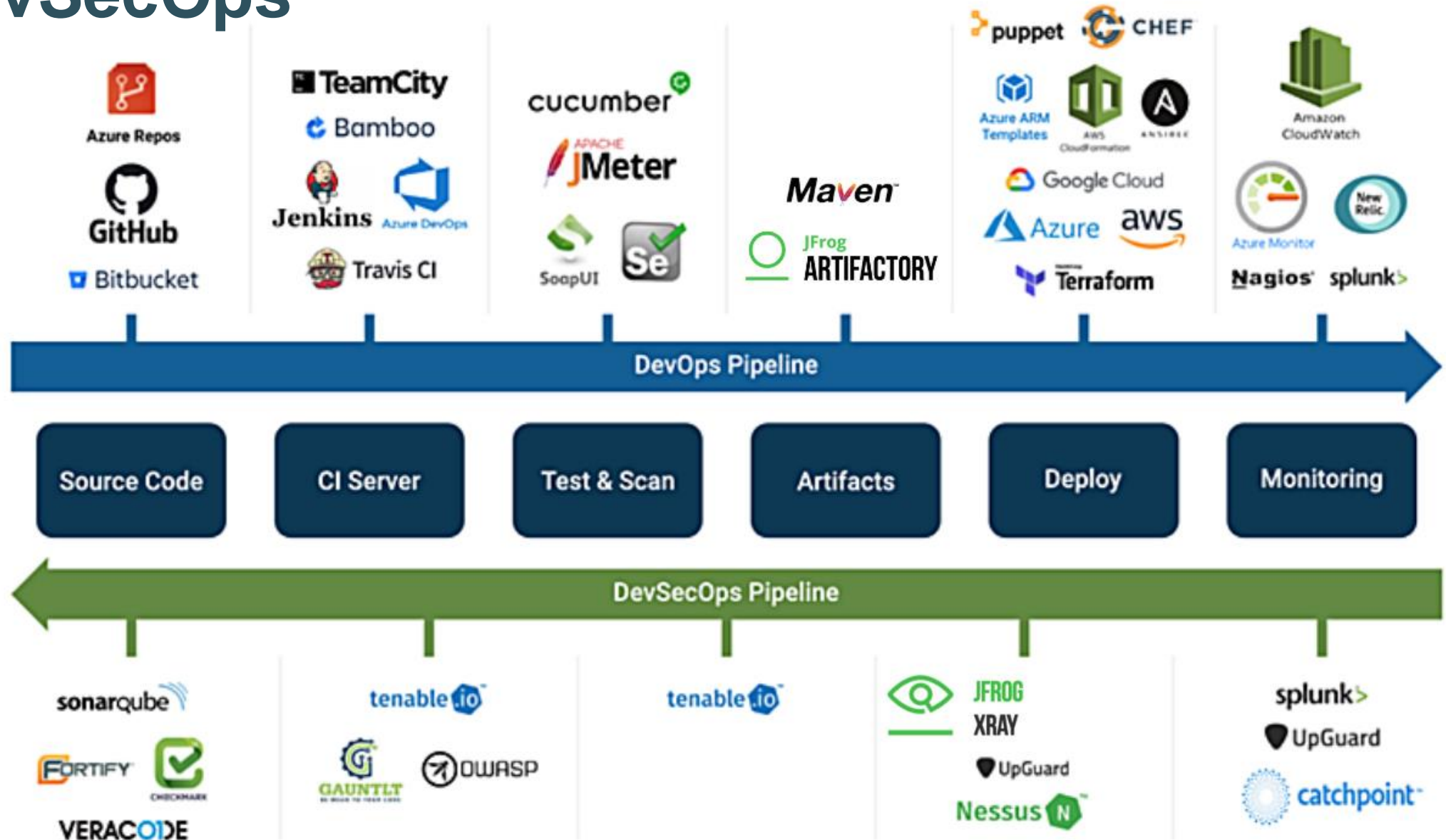
Scenario #4: Compromise internal services – The attacker can abuse internal services to conduct further attacks such as Remote Code Execution (RCE) or Denial of Service (DoS).

DevSecOps

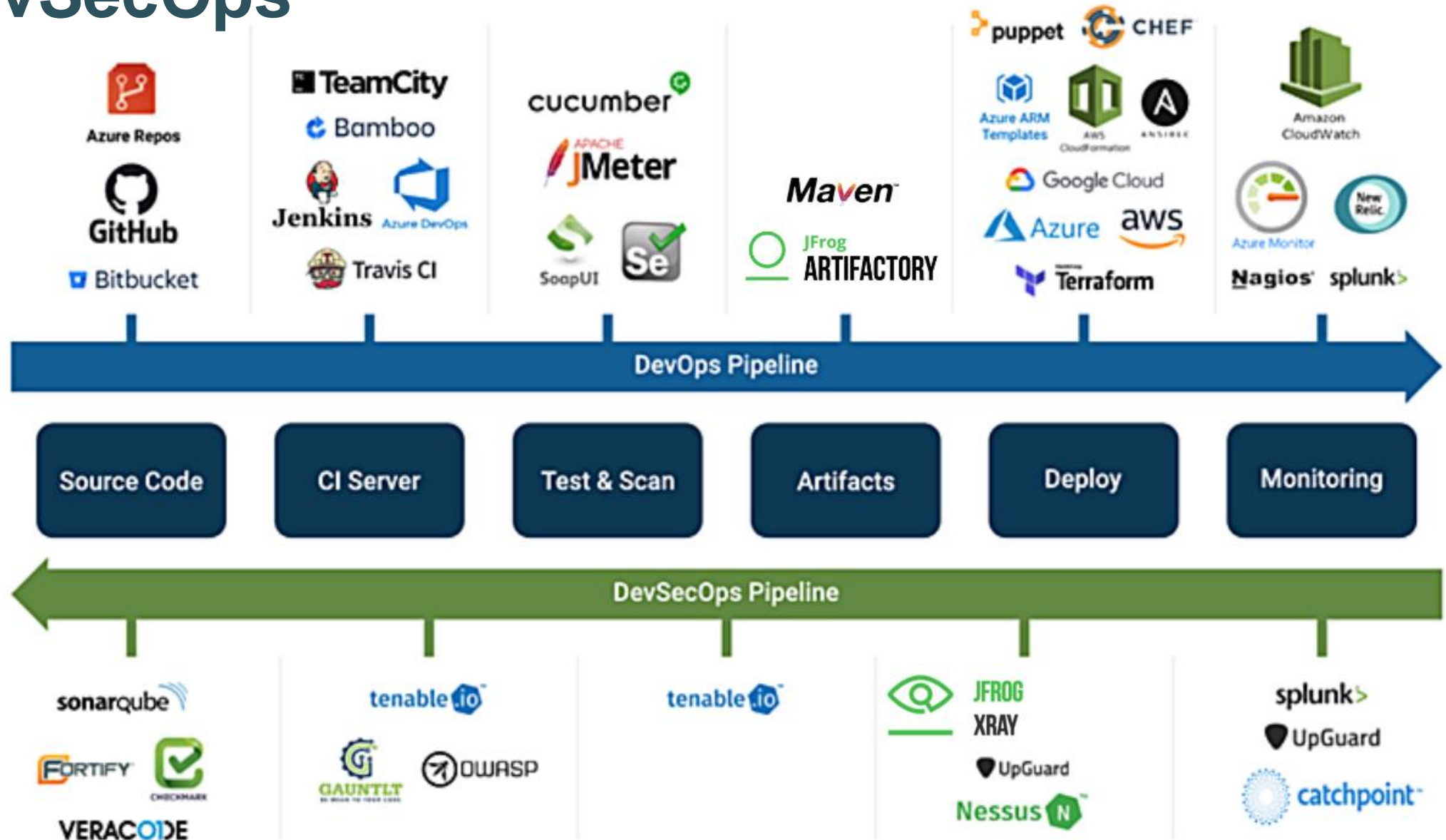
DevSecOps exists at the crossroads of **people**, **processes**, and **technology**. It represents a culture of application development and operations that speeds up the delivery of capabilities by **embedding security throughout the entire application lifecycle**.



DevSecOps



DevSecOps



Our Internal Security Architecture

SAKTTI

Standar Arsitektur Keamanan Tingkat Tinggi Informasi

High Grade Information Security Architecture Standard

Balance between deterrent, preventive, detective and corrective controls of people, process, technology and physical in all information life cycle (create, distribute, use, maintain, archive, destroy) by implementing security principles.

SAKTTI Security Principles

Holistic and Integrated
Information Security

Effective Monitoring and
Incident Management

High Availability

- Good Performance
- Effective Backup and Fail Over

High Confidentiality

- End-to-end encryption
- Full network and storage encryption

High Integrity

- White List Approach
- Effective Change Management
- Clear Separation of Duties
- Effective Defense in Depth
- Enforce Least Privilege
- Effective Traffic Control
- Effective Data Integrity Assurance
- Secure key management

Technological Implementation

Source code
repositories

Automatic
builder

Automatic
deployment

Image
scanner

Registry

Virtualization

Certificate
Authority

Validation
Authority

Registration
Authority

IDS

SIEM

Multilayer
firewalls

Container
hardening

OS
hardening

Vulnerability
assessment

Traffic rules
enforcement

Dark network

Availability
monitoring

Incident
management

Key Takeaways

Follow best practices

Continuous learning

Choose technology wisely

No software is perfect, do continuous improvements!

Design is cheaper than implementation

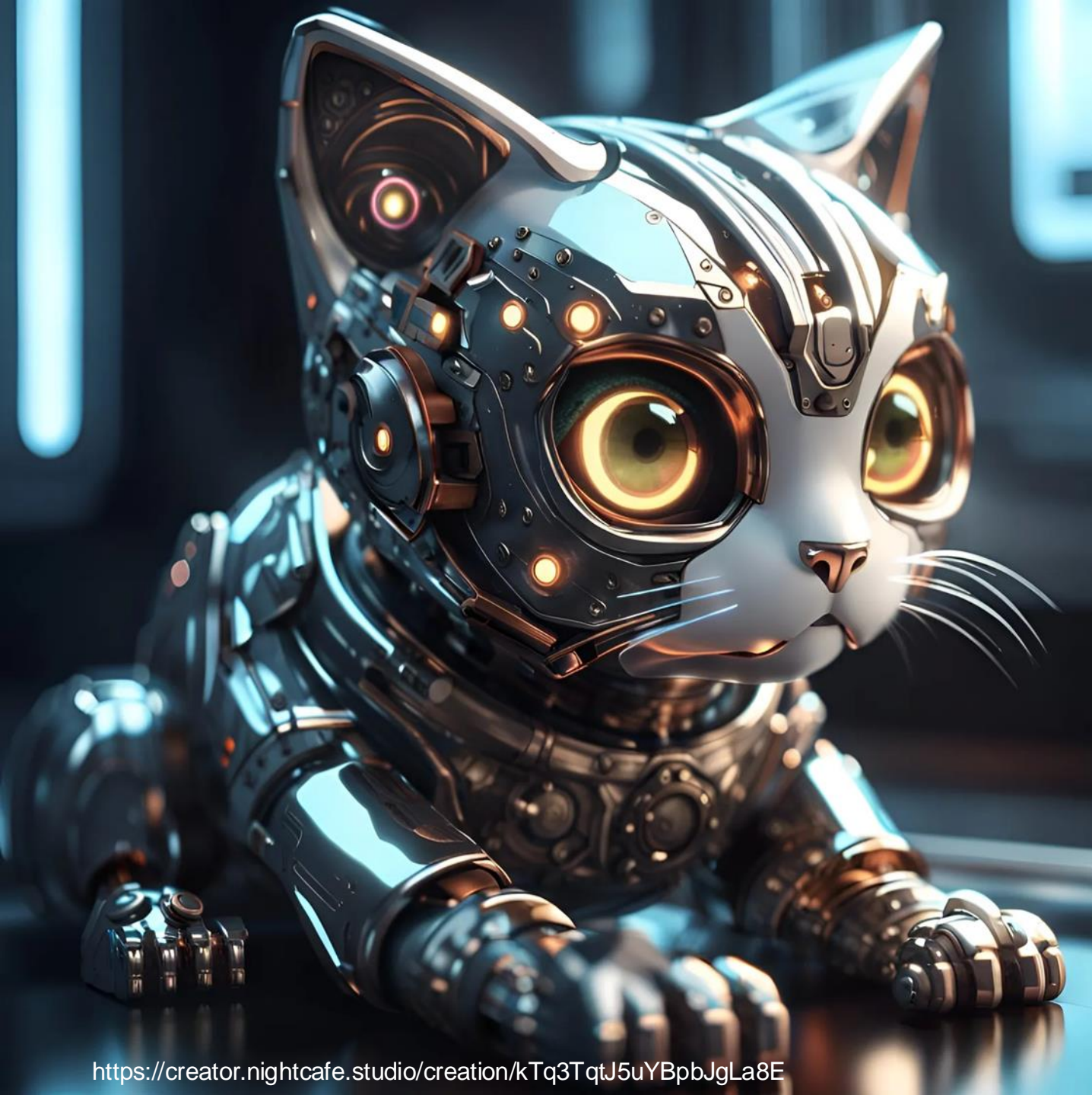
Management supports is very important

Know what you design and make

Secure software is in high demand!

Use automation tools as much as possible

Embraces DevSecOps!



<https://creator.nightcafe.studio/creation/kTq3TqtJ5uYBpbJgLa8E>

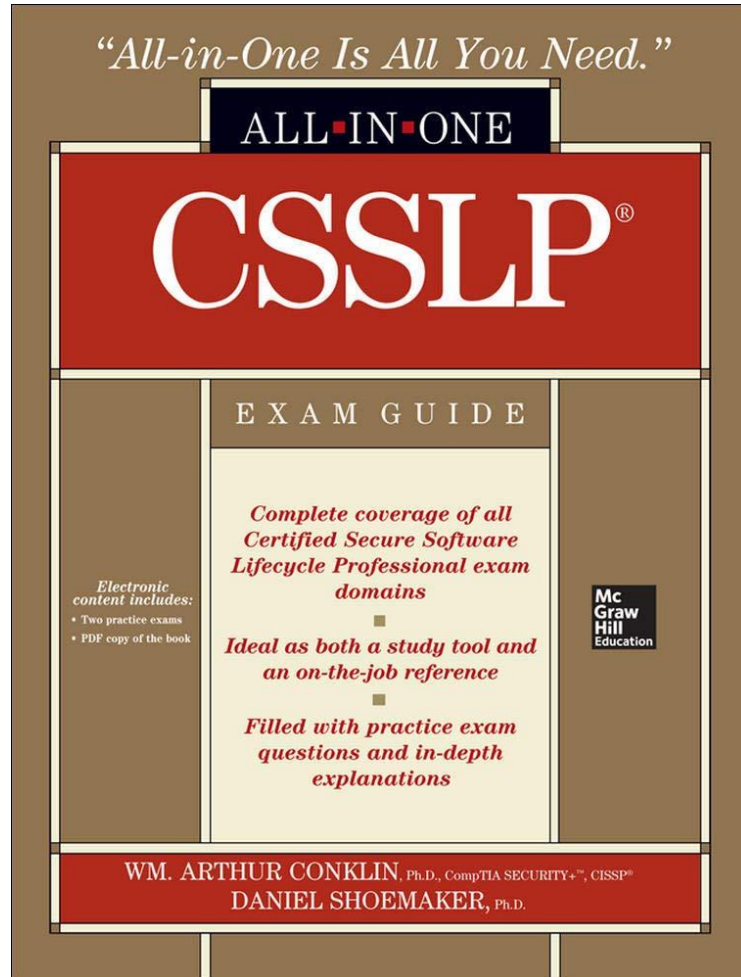


PyGoat
<http://oren.mojok.org:8081>

Vulnerable Flask Application

<http://oren.mojok.org:8082>

References



- CSSLP Exam Guide
- OWASP Top 10 2021
- Dod DevSecOps
- SAKTTI



Stop learning and you are burying your IT career!

Prof. Yudho Giri Sucahyo, S.Kom., M.Kom., Ph.D. Lecturer Fasilkom UI