Definition and scope of ethical hacking

Definition of Ethical Hacking:

Ethical hacking, also known as penetration testing or white hat hacking, refers to the practice of using hacking techniques and methodologies to identify vulnerabilities and weaknesses in computer systems, networks, applications, and other digital assets. The purpose of ethical hacking is to help organizations identify and fix security flaws before malicious hackers can exploit them for nefarious purposes.

Scope of Ethical Hacking:

The scope of ethical hacking is broad and encompasses various aspects of information technology security. Here are some key areas within the scope of ethical hacking:

1. **Network Security**:
   - Ethical hackers assess the security of network infrastructure, including routers, switches, firewalls, and other devices. They look for vulnerabilities that could allow unauthorized access or data interception.
2. **Web Application Security**:
   - This involves testing the security of websites and web applications to identify vulnerabilities such as SQL injection, cross-site scripting (XSS), and other common web application flaws.
3. **Wireless Network Security**:
   - Ethical hackers evaluate the security of wireless networks, looking for weak encryption, unauthorized access points, and other vulnerabilities that could lead to unauthorized access.
4. **Operating System Security**:
   - This involves examining the security configurations and settings of operating systems (e.g., Windows, Linux, macOS) to ensure they are properly hardened against potential threats.
5. **Database Security**:

- Ethical hackers assess the security of databases to identify vulnerabilities that could lead to unauthorized access, data leakage, or other data-related risks.

6. **Social Engineering**:
    - Ethical hackers may use social engineering techniques to test the human factor in security. This can involve techniques like phishing, pretexting, and tailgating to assess an organization's susceptibility to manipulation.
7. **Physical Security**:
    - Ethical hackers evaluate the physical security measures in place, such as access controls, surveillance, and environmental controls, to identify weaknesses that could lead to unauthorized physical access.
8. **Application Security**:
    - This includes testing the security of software applications, both desktop and mobile, to identify vulnerabilities that could be exploited to compromise the confidentiality, integrity, or availability of data.
9. **Cloud Security**:
    - Ethical hackers assess the security of cloud-based services, including cloud infrastructure, platforms, and applications, to identify vulnerabilities and misconfigurations.
10. **IoT (Internet of Things) Security**:
    - Ethical hackers evaluate the security of IoT devices and networks to identify vulnerabilities that could be exploited to gain unauthorized access or control over IoT assets.
11. **Incident Response and Forensics**:
    - Ethical hackers may assist in incident response activities, helping organizations investigate and mitigate security incidents, as well as conduct digital forensics to gather evidence.

The goal of ethical hacking is to provide organizations with actionable insights and recommendations to strengthen their security posture and protect against cyber threats. It is a proactive approach to cybersecurity that helps organizations stay one step ahead of potential attackers.

Top 5 OWASP CWE

AO1 2021 Broken Acess Control

Description

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.

CWE-284: Improper Access Control

Description

Access control involves the use of several protection mechanisms such as:

- Authentication (proving the identity of an actor)

- Authorization (ensuring that a given actor can access a resource), and
- Accountability (tracking of activities that were performed)

When any mechanism is not applied or otherwise fails, attackers can compromise the security of the product by gaining privileges, reading sensitive information, executing commands, evading detection, etc.

There are two distinct behaviors that can introduce access control weaknesses:

- Specification: incorrect privileges, permissions, ownership, etc. are explicitly specified for either the user or the resource (for example, setting a password file to be world-writable, or giving administrator capabilities to a guest user). This action could be performed by the program or the administrator.
- Enforcement: the mechanism contains errors that prevent it from properly enforcing the specified access control requirements (e.g., allowing the user to specify their own privileges, or allowing a syntactically-incorrect ACL to produce insecure settings). This problem occurs within the program itself, in that it does not actually enforce the intended security policy that the administrator specifies.

A02:2021 – Cryptographic Failures

Description

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.

CWE-259: Use of Hard-coded Password

Description

A hard-coded password typically leads to a significant authentication failure that can be difficult for the system administrator to detect. Once detected, it can be difficult to fix, so the administrator may be forced into disabling the product entirely. There are two main variations:

Inbound: the product contains an authentication mechanism that checks for a hard-coded password.

Outbound: the product connects to another system or component, and it contains hard-coded password for connecting to that component.

In the Inbound variant, a default administration account is created, and a simple password is hard-coded into the product and associated with that account. This hard-coded password is the same for each installation of the product, and it usually cannot be changed or disabled by system administrators without manually modifying the program, or otherwise patching the product. If the password is ever discovered or published (a common occurrence on the Internet), then anybody with knowledge of this password can access the product. Finally, since all installations of the product will have the same password, even across different organizations, this enables massive attacks such as worms to take place.

The Outbound variant applies to front-end systems that authenticate with a back-end service. The back-end service may require a fixed password which can be easily discovered. The programmer may simply hard-code those back-end credentials into the front-end product. Any user of that program may be able to extract the password. Client-side systems with hard-coded passwords pose even more of a threat, since the extraction of a password from a binary is usually very simple.

1. **A03:2021 – Injection**

Description

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.

CWE-20: Improper Input Validation
Description
Input validation is a frequently-used technique for checking potentially dangerous inputs in order to ensure that the inputs are safe for processing within the code, or when communicating with other components. When software does not validate input properly, an attacker is able to craft the input in a form that is not expected by the rest of the application. This will lead to parts of the system receiving unintended input, which may result in altered control flow, arbitrary control of a resource, or arbitrary code execution. Input validation is not the only technique for processing input, however. Other

techniques attempt to transform potentially-dangerous input into something safe, such as filtering (CWE-790) - which attempts to remove dangerous inputs - or encoding/escaping (CWE-116), which attempts to ensure that the input is not misinterpreted when it is included in output to another component. Other techniques exist as well (see CWE-138 for more examples.)

Input validation can be applied to:

• raw data - strings, numbers, parameters, file contents, etc.

• metadata - information about the raw data, such as headers or size

Data can be simple or structured. Structured data can be composed of many nested layers, composed of combinations of metadata and raw data, with other simple or structured data.

Many properties of raw data or metadata may need to be validated upon entry into the code, such as:

• specified quantities such as size, length, frequency, price, rate, number of operations, time, etc.

• implied or derived quantities, such as the actual size of a file instead of a specified size

• indexes, offsets, or positions into more complex data structures

• symbolic keys or other elements into hash tables, associative arrays, etc.

• well-formedness, i.e. syntactic correctness - compliance with expected syntax

• lexical token correctness - compliance with rules for what is treated as a token

• specified or derived type - the actual type of the input (or what the input appears to be)

• consistency - between individual data elements, between raw data and metadata, between references, etc.

• conformance to domain-specific rules, e.g. business logic

• equivalence - ensuring that equivalent inputs are treated the same

• authenticity, ownership, or other attestations about the input, e.g. a cryptographic signature to prove the source of the data

Implied or derived properties of data must often be calculated or inferred by the code itself. Errors in deriving properties may be considered a contributing factor to improper input validation.

Note that "input validation" has very different meanings to different people, or within different classification schemes. Caution must be used when referencing this CWE entry or mapping to it. For example, some weaknesses might involve inadvertently giving control to an attacker over an input when they should not be able to provide an input at all, but sometimes this is referred to as input validation.

Finally, it is important to emphasize that the distinctions between input validation and output escaping are often blurred, and developers must be careful to understand the difference, including how input validation is not always sufficient to prevent vulnerabilities, especially when less stringent data types must be supported, such as free-form text. Consider a SQL injection scenario in which a person's last name is inserted into a query. The name "O'Reilly" would likely pass the validation step since it is a common last name in the English language. However, this valid name cannot be directly inserted into the database because it contains the "'" apostrophe character, which would need to be escaped or otherwise transformed. In this case, removing the apostrophe might reduce the risk of SQL injection, but it would produce incorrect behavior because the wrong name would be recorded.

1. **A04:2021 – Insecure Design**

Description

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.

CWE-256: Plaintext Storage of a Password

Description

Password management issues occur when a password is stored in plaintext in an application's properties, configuration file, or memory. Storing a plaintext password in a configuration file allows anyone who can read the file access to the password-protected resource. In some contexts, even storage of a plaintext password in memory is considered a security risk if the password is not cleared immediately after it is used.

1. **A05:2021 – Security Misconfiguration**

Description

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).

CWE-520: .NET Misconfiguration: Use of Impersonation

Description

.NET server applications can optionally execute using the identity of the user authenticated to the client. The intention of this functionality is to bypass authentication and access control checks within the .NET application code. Authentication is done by the underlying web server (Microsoft Internet Information Service IIS), which passes the authenticated token, or unauthenticated anonymous token, to the .NET application. Using the token to impersonate the client, the application then relies on the settings within the NTFS directories and files to control access. Impersonation enables the application, on the server running the .NET application, to both execute code and access resources in the context of the authenticated and authorized user.

web application vulnerabilities

Cross Site Scripting (XSS)

Cross-Site Scripting (XSS) attacks are a type of injection, in which malicious scripts are injected
into otherwise benign and trusted websites. XSS attacks occur when an attacker uses a web
application to send malicious code, generally in the form of a browser side script, to a different
end user. Flaws that allow these attacks to succeed are quite widespread and occur anywhere a
web application uses input from a user within the output it generates without validating or
encoding it.

An attacker can use XSS to send a malicious script to an unsuspecting user. The end user’s
browser has no way to know that the script should not be trusted, and will execute the script.
Because it thinks the script came from a trusted source, the malicious script can access any
cookies, session tokens, or other sensitive information retained by the browser and used with
that site. These scripts can even rewrite the content of the HTML page.

Description

Cross-Site Scripting (XSS) attacks occur when:
1. Data enters a Web application through an untrusted source, most frequently a web request.
2. The data is included in dynamic content that is sent to a web user without being

validated for malicious content.
The malicious content sent to the web browser often takes the form of a segment of JavaScript,
but may also include HTML, Flash, or any other type of code that the browser may execute. The
variety of attacks based on XSS is almost limitless, but they commonly include transmitting
private data, like cookies or other session information, to the attacker, redirecting the victim to
web content controlled by the attacker, or performing other malicious operations on the user's
machine under the guise of the vulnerable site.

SQL Injection

A SQL injection attack consists of insertion or "injection" of a SQL query via the input data from
the client to the application. A successful SQL injection exploit can read sensitive data from the
database, modify database data (Insert/Update/Delete), execute administration operations on
the database (such as shutdown the DBMS), recover the content of a given file present on the
DBMS file system and in some cases issue commands to the operating system. SQL injection
attacks are a type of injection attack, in which SQL commands are injected into data-plane input
in order to affect the execution of predefined SQL commands.

Description

SQL injection attack occurs when:

1. An unintended data enters a program from an untrusted source.
2. The data is used to dynamically construct a SQL query

The main consequences are:

- **Confidentiality**: Since SQL databases generally hold sensitive data, loss of confidentiality is a frequent problem with SQL Injection vulnerabilities.
- **Authentication**: If poor SQL commands are used to check user names and passwords,
  it may be possible to connect to a system as another user with no previous knowledge of
  the password.
- **Authorization**: If authorization information is held in a SQL database, it may be possible
  to change this information through the successful exploitation of a SQL Injection vulnerability.
- **Integrity**: Just as it may be possible to read sensitive information, it is also possible to
  make changes or even delete this information with a SQL Injection attack.

Cross Site Request Forgery (CSRF)

Cross-Site Request Forgery (CSRF) is an attack that forces an end user to execute unwanted
actions on a web application in which they're currently authenticated. With a little help of social
engineering (such as sending a link via email or chat), an attacker may trick the users of a web
application into executing actions of the attacker's choosing. If the victim is a normal user, a
successful CSRF attack can force the user to perform state changing requests like transferring
funds, changing their email address, and so forth. If the victim is an administrative account,
CSRF can compromise the entire web application.

Description

CSRF is an attack that tricks the victim into submitting a malicious request. It inherits the identityand privileges of the victim to perform an undesired function on the victim's behalf (though note that this is not true of login CSRF, a special form of the attack described below). For most sites, browser requests automatically include any credentials associated with the site, such as the user’s session cookie, IP address, Windows domain credentials, and so forth. Therefore, if the user is currently authenticated to the site, the site will have no way to distinguish between the forged request sent by the victim and a legitimate request sent by the victim. CSRF attacks target functionality that causes a state change on the server, such as changing the victim’s email address or password, or purchasing something. Forcing the victim to retrieve data doesn’t benefit an attacker because the attacker doesn’t receive the response, the victim does. As such, CSRF attacks target state-changing requests.

An attacker can use CSRF to obtain the victim’s private data via a special form of the attack,
known as login CSRF. The attacker forces a non-authenticated user to log in to an account the attacker controls. If the victim does not realize this, they may add personal data—such as credit card information—to the account. The attacker can then log back into the account to view this data, along with the victim’s activity history on the web application.

It’s sometimes possible to store the CSRF attack on the vulnerable site itself. Such vulnerabilities are called “stored CSRF flaws". This can be accomplished by simply storing an
IMG or IFRAME tag in a field that accepts HTML, or by a more complex cross-site scripting attack. If the attacker can store a CSRF attack in the site, the severity of the attack is amplified. In particular, the likelihood is increased because the victim is more likely to view the page containing the attack than some random page on the Internet. The likelihood is also increased because the victim is sure to be authenticated to the site already.

Server Side Request Forgery

In a Server-Side Request Forgery (SSRF) attack, the attacker can abuse functionality on the
server to read or update internal resources. The attacker can supply or modify a URL which the
code running on the server will read or submit data to, and by carefully selecting the URLs, the
attacker may be able to read server configuration such as AWS metadata, connect to

internal
services like http enabled databases or perform post requests towards internal services which
are not intended to be exposed.

Description

The target application may have functionality for importing data from a URL, publishing data to a
URL or otherwise reading data from a URL that can be tampered with. The attacker modifies the
calls to this functionality by supplying a completely different URL or by manipulating how URLs
are built (path traversal etc.).

When the manipulated request goes to the server, the server-side code picks up the manipulated URL and tries to read data to the manipulated URL. By selecting target URLs the
attacker may be able to read data from services that are not directly exposed on the internet:

- Cloud server meta-data - Cloud services such as AWS provide a REST interface on http://169.254.169.254/ where important configuration and sometimes even authentication keys can be extracted.
- Database HTTP interfaces - NoSQL databases such as MongoDB provide REST interfaces on HTTP ports. If the database is expected to only be available internally, authentication may be disabled and the attacker can extract data.
- Internal REST interfaces.
- Files - The attacker may be able to read files using file:// URIs.

  The attacker may also use this functionality to import untrusted data into code that expects to only read data from trusted sources, and as such circumvent input validation

XML External Entity (XXE) Processing

Description

An *XML External Entity* attack is a type of attack against an application that parses XML input. This attack occurs when XML input containing a reference to an external entity is processed by a weakly configured XML parser. This attack may lead to the disclosure of confidential data, denial of service, server side request forgery, port scanning from the perspective of the machine where the parser is located, and other system impacts. The XML 1.0 standard defines the structure of an XML document. The standard defines a concept called an entity, which is a storage unit of some type. There are a few different types of
entities, external general/parameter parsed entities often shortened to **external entities**, that
can access local or remote content via a declared system identifier. The system identifier is
assumed to be a URI that can be dereferenced (accessed) by the XML processor when
processing the entity. The XML processor then replaces occurrences of the named external
entity with the contents dereferenced by the system identifier. If the system identifier contain stainted data and the XML processor dereferences this tainted data, the XML processor may
disclose confidential information normally not accessible by the application. Similar attack
vectors apply the usage of external DTDs, external stylesheets, external schemas, etc. which,
when included, allow similar external resource inclusion style attacks. Attacks can include disclosing local files, which may contain sensitive data such as passwords or private user data, using file: schemes or relative paths in the system identifier. Since the attack occurs relative to the application processing the XML document, an attacker may use this
trusted application to pivot to other internal systems, possibly disclosing other internal content via http(s) requests or launching a CSRF attack to any unprotected internal services. In some situations, an XML processor library that is vulnerable to client-side memory corruption issues may be exploited by dereferencing a malicious URI, possibly allowing arbitrary code execution under the application account. Other attacks can access local resources that may not stop returning data, possibly impacting application availability if too many threads or processes are not released.

Note that the application does not need to explicitly return the response to the attacker for it to
be vulnerable to information disclosures. An attacker can leverage DNS information to exfiltrate
data through subdomain names to a DNS server that they control.

Insecure Direct Object References

Insecure Direct Object References (IDOR) occur when an application provides direct access to
objects based on user-supplied input. As a result of this vulnerability attackers can bypass
authorization and access resources in the system directly, for example database records or
files. Insecure Direct Object References allow attackers to bypass authorization and access
resources directly by modifying the value of a parameter used to directly point to an object.
Such resources can be database entries belonging to other users, files in the system, and more.
This is caused by the fact that the application takes user supplied input and uses it to retrieve an
object without performing sufficient authorization checks.

Directory traversal

Directory traversal (also known as file path traversal) is a web security vulnerability that allows
an attacker to read arbitrary files on the server that is running an application. This might include
application code and data, credentials for back-end systems, and sensitive operating system
files. In some cases, an attacker might be able to write to arbitrary files on the server, allowing
them to modify application data or behavior, and ultimately take full control of the server.

Static application security testing

Static application security testing (**SAST**) is used to secure software by reviewing the source code of the software to identify sources of vulnerabilities. Although the process of statically analyzing the source code has existed as long as computers have existed, the technique spread to security in the late 90s and the first public discussion of SQL injection in 1998 when Web applications integrated new technologies like JavaScript and Flash.

Unlike dynamic application security testing (DAST) tools for black-box testing of application functionality, SAST tools focus on the code content of the application, white-box testing. A SAST tool scans the source code of applications and its components to identify potential security vulnerabilities in their software and architecture. Static analysis tools can detect an estimated 50% of existing security vulnerabilities.

In the software development life cycle (SDLC), SAST is performed early in the development process and at code level, and also when all pieces of code and components are put together in a consistent testing environment. SAST is also used for software quality assurance. even if the many resulting false-positive impede its adoption by developers

SAST tools are integrated into the development process to help development teams as they are primarily focusing on developing and delivering software respecting requested specifications.
SAST tools, like other security tools, focus on reducing the risk of downtime of applications or that private information stored in applications will not be compromised.

For the year of 2018, the Privacy Rights Clearinghouse database shows that more than 612 million records have been compromised by hacking.

Session fixation

Description

Session Fixation is an attack that permits an attacker to hijack a valid user session. The attack
explores a limitation in the way the web application manages the session ID, more specifically
the vulnerable web application. When authenticating a user, it doesn't assign a new session ID,
making it possible to use an existent session ID. The attack consists of obtaining a valid session
ID (e.g. by connecting to the application), inducing a user to authenticate himself with that
session ID, and then hijacking the user-validated session by the knowledge of the used session
ID. The attacker has to provide a legitimate Web application session ID and try to make the
victim's browser use it.

The session fixation attack is not a class of Session Hijacking, which steals the established
session between the client and the Web Server after the user logs in. Instead, the Session
Fixation attack fixes an established session on the victim's browser, so the attack starts before
the user logs in.

There are several techniques to execute the attack; it depends on how the Web application
deals with session tokens. Below are some of the most common techniques:
• Session token in the URL argument: The Session ID is sent to the victim in a hyperlink and the
victim accesses the site through the malicious URL.

- Session token in a hidden form field: In this method, the victim must be tricked to authenticate
  in the target Web Server, using a login form developed for the attacker. The form could be
  hosted in the evil web server or directly in html formatted e-mail.
  • Session ID in a cookie:

o Client-side script
Most browsers support the execution of client-side scripting. In this case, the aggressor could
use attacks of code injection as the XSS (Cross-site scripting) attack to insert a malicious code
in the hyperlink sent to the victim and fix a Session ID in its cookie. Using the function
document.cookie, the browser which executes the command becomes capable of fixing values
inside of the cookie that it will use to keep a session between the client and the Web Application.

<META>

tag

<META>

tag also is considered a code injection attack, however, different from the XSS attack where
undesirable scripts can be disabled, or the execution can be denied. The attack using this
method becomes much more efficient because it’s impossible to disable the processing of these
tags in the browsers.

o HTTP header response
This method explores the server response to fix the Session ID in the victim’s browser. Including
the parameter Set-Cookie in the HTTP header response, the attacker is able to insert the value
of Session ID in the cookie and sends it to the victim’s browser.

Interactive Application Security Testing

IAST (interactive application security testing) is an application security testing method that tests
the application while the app is run by an automated test, human tester, or any activity "interacting" with the application functionality.
The core of an IAST tool is sensor modules, software libraries included in the application code.
These sensor modules keep track of application behavior while the interactive tests are running.
If a vulnerability is detected, an alert will be sent.

The process and feedback are done in real time in your integrated development environment
(IDE), continuous integration (CI) environment, or quality assurance, or while in production. The
sensors have access to:

- Entire code
- Dataflow and control flow
- System configuration data
- Web components
- Back-end connection data

  Examples of such vulnerabilities could be hardcoding API keys in cleartext, not sanitizing your
  users inputs, or using connections without SSL encryption.

By: Pavan Kumar.S

21BAI1527