EXXETA
CONSULTING AND TECHNOLOGIES

WIR VERBINDEN WELTEN

BUSINESS. IT. INNOVATION.

SECURE CODING WORKBOOK

Die Inhalte des Workbooks basieren auf den OWASP Top 10,

ein Projekt der OWASP Foundation.

Versionskontrolle			
Version	Datum	Autor	Änderungen
1.0	Januar 2020	Bernhard Hirschmann	Initialdokument
1.1	April 2021	Marketing	Kontaktdaten

1 A1:2017 Injection

Bei **Injection** versucht ein Angreifer, Befehle in eine Anwendung zu schmuggeln (zu injizieren). Dies kann passieren, wenn eine Anwendung Daten entgegennimmt, diese allerdings als Befehle interpretiert. Ermöglicht wird dies durch eine fehlende bzw. nicht richtig funktionierende Filterung der Daten.

Injection-Arten:

- SQL-Injection — Dabei werden SQL-Befehle an eine DB gesendet. Dies ist die bekannteste und am weitesten verbreitete Art von Injection.
- OS-Injection — Hier werden Befehle in das Betriebssystem (Operating System) injiziert.
- LDAP-Injection — Das Senden von Befehlen an ein Active Directory über das LDAP-Protokoll.

Da die SQL-Injection eine der häufigsten genutzten Schwachstellen ist, wird im Folgenden noch einmal gesondert darauf eingegangen.

SQL-Injection	
 **Bedrohungsquelle**	Anwendungsspezifisch Externe und interne Nutzer sowie Administratoren. Jeder, der Daten an das System übermitteln kann, die nicht ausreichend geprüft werden.
 **Angriffsvektor**	Ausnutzbarkeit: **EINFACH** Der Angreifer sendet einfache textbasierte Angriffe, die die Syntax des Zielinterpreters missbrauchen. Fast jede Datenquelle kann einen Injection-Vektor darstellen, einschließlich interner Quellen.
 **Schwachstellen**	Verbreitung: **HÄUFIG** Auffindbarkeit: **DURCHSCHNITTLICH** Injection-Schwachstellen tauchen auf, wenn eine Anwendung nicht vertrauenswürdige Daten an einen Interpreter weiterleitet. Sie sind weit verbreitet, besonders in veraltetem Code. Sie finden sich in SQL-, LDAP-, XPath- und NoSQL-Anfragen, in Betriebssystembefehlen sowie in XML, SMTP-Headern, Parametern etc. Injection-Schwachstellen lassen sich durch Code-Prüfungen einfach entdecken, schwerer durch externe Tests. Angreifer setzen Scanner und Fuzzer ein, um Schwachstellen zu erkennen.
 **Auswirkungen**	Technische Auswirkung: **SCHWERWIEGEND** Injection kann zu Datenverlust oder -verfälschung, Fehlen von Zurechenbarkeit oder Zugangssperre führen. Unter Umständen kann es zu einer vollständigen Systemübernahme kommen. Geschäftliche Auswirkung: Der Wert betroffener Daten für das Unternehmen sowie die Laufzeitumgebung des Interpreters sind zu berücksichtigen. Daten können entwendet, verändert oder sogar gelöscht werden. Kann ein Image-Schaden entstehen?

1.1 Mögliche Angriffsszenarien

Szenario 1: Die Anwendung nutzt ungeprüfte Eingabedaten bei der Konstruktion der **verwundbaren** SQL-Abfrage:

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

Szenario 2: Blindes Vertrauen in den Einsatz eines Frameworks (hier z. B. Hibernate Query Language (HQL)):

```
Query unsafeHQLQuery = session.createQuery("from accounts where
custID='"+request.getParameter("id")+"'");
```

Der Angreifer verändert in beiden Fällen den 'id'-Parameter im Browser und sendet: **' or '1'='1**.

```
http://example.com/app/accountView?id=' or '1'='1
```

Das ändert die Logik der Anfrage, sodass in diesem Fall alle Datensätze der Tabelle 'accounts' zurückgegeben werden.

Schlimmstenfalls werden durch Injections Daten verändert oder sogar Stored Procedures (in den DB fest abgelegte Abfrage-Prozeduren) gestartet, um Daten zu manipulieren.

1.2 Prävention

Um Injection zu verhindern, müssen Eingabedaten und Befehle konsequent getrennt werden.

1. Der bevorzugte Ansatz ist die Nutzung einer sicheren API, die den Aufruf von Interpretern vermeidet oder eine typgebundene Schnittstelle bereitstellt. Seien Sie vorsichtig bei APIs, z. B. Stored Procedures, die trotz Parametrisierung anfällig für Injection sein können.
2. Wenn eine typsichere API nicht verfügbar ist, sollten Sie Metazeichen unter Berücksichtigung der jeweiligen Syntax sorgfältig entschärfen. ESAPI, die frei verfügbare Bibliothek von OWASP, stellt hierfür viele spezielle Methoden bereit.
3. Auch die Eingabeprüfung gegen Positivlisten ("white list") wird empfohlen, ist aber kein vollständiger Schutz, da viele Anwendungen Metazeichen in den Eingaben erfordern. Ist der Einsatz von Sonderzeichen notwendig, können diese nur unter Beachtung von Punkt 1 und 2 (s. o.) sicher verwendet werden.
4. Alle Maßnahmen unbedingt serverseitig umsetzen.

1.2.1 Java

1.2.1.1 Prepared Statements verwenden

Java – Standard

```
String custname = request.getParameter("customerName");

// perform input validation to detect attacks
String query = "SELECT account_balance FROM user_data WHERE user_name = ? ";
PreparedStatement pstmt = connection.prepareStatement( query );
pstmt.setString( 1, custname);
ResultSet results = pstmt.executeQuery( );
```

Java – Hibernate

Beispiel mit Hibernate Query Language (HQL): Empfehlung des Projekts 'Top 10 für Entwickler' für komplexe Abfragen, vgl. auch OWASP Query_Parameterization Cheat Sheet:

```
@Entity // declare as entity;
@NamedQuery(
 name="findByDescription",
 query="FROM Inventory i WHERE i.productDescription = :productDescription"
)
public class Inventory implements Serializable {
 @Id
 private long id;
 private String productDescription;
}

// use case
String userSuppliedParameter = request.getParameter("Product-Description"); //
This should REALLY be validated too

// perform input validation to detect attacks
List<Inventory> list =
 session.getNamedQuery("findByDescription")
  .setParameter("productDescription", userSuppliedParameter).list();
```

Weitere Beispiele: SANS: fix-sql-injection-in-java-hibernate

Beispiel mit Hibernate Criteria Queries (Empfehlung des Projekts 'Top 10 für Entwickler'):

```
String userSuppliedParameter = request.getParameter("Product-Description"); //
This should REALLY be validated too
// perform input validation to detect attacks
Inventory inv =
(Inventory)
session.createCriteria(Inventory.class).add(Restrictions.eq("productDescription"
, userSuppliedParameter)).uniqueResult();
```

☑ Benutzereingaben sorgfältig prüfen und entschärfen

Die Metazeichen sind je Backend-Typ und meist auch je Backend-Hersteller unterschiedlich (z. B. Datenbank-Hersteller), vgl. OWASP's ESAPI.

```
Codec ORACLE_CODEC = new OracleCodec(); //added
String query = "SELECT user_id FROM user_data WHERE user_name = '" +
      ESAPI.encoder().encodeForSQL( ORACLE_CODEC, //added
      req.getParameter("userID") ) + "' and user_password = '" +
      ESAPI.encoder().encodeForSQL( ORACLE_CODEC, //added
      req.getParameter("pwd") ) +"'";
```

☑ Benutzereingaben mittels Positivlisten prüfen

Diese Technik sollte bei SQL nur als Zusatzmaßnahme oder als Notlösung für alte Software in Betracht gezogen werden. Für die anderen Typen ist sie die einzige bekannte Maßnahme. Die Benutzereingaben sind zunächst zu normalisieren (= kanonisieren). APIs, wie OWASP's ESAPI, erledigen dies automatisch.

Beschränken Sie bei den Regeln für die Postivlisten die erlaubten Zeichen, ggf. die erlaubte Zeichenfolge und den gültigen Wertebereich bzw. die Länge der erwarteten Eingabe. Seien Sie besonders vorsichtig, wenn Sie Zeichen erlauben möchten, die das Backend als Metazeichen benutzt, z. B. ' und " (vgl. „Potenziell

gefährliche Zeichen für Interpreter“ (BSI)). Wandeln Sie diese jeweils in ungefährliche Zeichen, bspw. mittels Kodierung, vor der weiteren Verarbeitung um (Encoding, z. B. in ' und ").

```
//performing input validation
ESAPI.validator().getValidInput
...
String query = "SELECT user_id FROM user_data WHERE user_name = '" + ...
...
```

1.2.2 PHP

☑ Prepared Statements (Parameterized Queries)

PHP – PDO

```
$stmt = $dbh->prepare("INSERT INTO REGISTRY (name, value) VALUES
(:name, :value)");

$stmt->bindParam(':name', $name);

$stmt->bindParam(':value', $value);
```

vgl. OWASP Query_Parameterization Cheat Sheet

PHP – mysqli

```
$stmt = $mysqli->prepare( 'SELECT * FROM foo WHERE bar = ?' );

$stmt->bind_param( 's', $value );
```

☑ Benutzereingaben sorgfältig prüfen und entschärfen

Diese Technik sollte bei SQL nur als Zusatzmaßnahme oder als Notlösung für alte Software in Betracht gezogen werden. Für andere Injection-Typen, außer SQL-Injection, ist sie die einzige bekannte Maßnahme. Die Metazeichen sind je Backend-Typ und meist auch je Backend-Hersteller unterschiedlich (z. B. Datenbank-Hersteller), vgl. OWASP's ESAPI für Java.

Hier besteht die Gefahr, dass das Escapen an einer Stelle vergessen wird, was die Sicherung komplett aushebeln würde.

Manueller String Filter:

```
$stmt = $mysqli->query( 'SELECT * FROM foo WHERE bar = ' . $mysqli-
>real_escape_string($input) );
```

1.2.3 .NET

☑ StoredProcedure verwenden

```
var conn = new SqlConnection(connString);
using (var command = new SqlCommand("GetProducts", conn))
{
 command.CommandType = CommandType.StoredProcedure;
 command.Parameters.Add("@CategoryID", SqlDbType.Int).Value = catID;
 command.Connection.Open();
 grdProducts.DataSource = command.ExecuteReader();
 grdProducts.DataBind();
}
```

☑ Benutzereingaben prüfen und entschärfen

Diese Technik sollte bei SQL nur als Zusatzmaßnahme oder als Notlösung für alte Software in Betracht gezogen werden. Für andere Injection-Typen, außer SQL-Injection, ist sie die einzige bekannte Maßnahme. Die Metazeichen sind je Backend-Typ und meist auch je Backend-Hersteller unterschiedlich (z. B. Datenbank-Hersteller), vgl. OWASP's ESAPI.

Manueller Filter für eine Zahl:

```
var catID = Request.QueryString["CategoryID"];
var positiveIntRegex = new Regex(@"^0*[1-9][0-9]*$");
if(!positiveIntRegex.IsMatch(catID))
{
 lblResults.Text = "An invalid CategoryID has been specified.";
 return;
}
```

1.3 Checkliste zu A1:2017

	SQL-Injection
☐	**Werden SQL-Statements ohne String-Konkatenation aus ungeprüften Parametern erstellt?**
	Die Zusammensetzung durch ungeprüfte Parameter ist ein großes Risiko.
☐	**Werden ausschließlich sichere APIs (z. B. PreparedStatement) verwendet, um per SQL DB-Abfragen zu tätigen?**
	SQL-Statements sollten ausschließlich durch sichere APIs erstellt werden, ohne dass Benutzereingaben manuell hinzugefügt werden.
☐	**Falls sichere API nicht verfügbar: Werden Benutzereingaben und Parameter sorgfältig per Whitelisting vor der Konkatenation mit der SQL-Anweisung geprüft?**
	Für jede verbliebene dynamische Query müssen Sonder- und Metazeichen für den jeweiligen Interpreter mit der richtigen Escape-Syntax entschärft werden.
☐	**Wird die Ergebnismenge eingeschränkt?**
	SQL-Querys sollten LIMIT oder andere SQL-Controls verwenden, um den möglichen Massenabfluss von Daten zu verhindern.
	Injection allgemein
☐	**Werden Benutzereingaben sorgfältig per Whitelisting vor Konkatenation mit der SQL-Anweisung geprüft?**
	Außer bei SQL-Injection gibt es kaum eine andere Möglichkeit sich gegen Injection zu wehren, da die Interpreter für LDAP und für OS- und Shell-Kommandos in der Regel keine sichere API anbieten.

1.4 Referenzen

OWASP

- OWASP Proactive Controls: Parameterize Queries
- OWASP ASVS: V5 Input Validation and Encoding
- OWASP Testing Guide: SQL Injection, Command Injection, ORM injection
- OWASP Cheat Sheet: Injection Prevention
- OWASP Cheat Sheet: SQL Injection Prevention
- OWASP Cheat Sheet: Injection Prevention in Java
- OWASP Cheat Sheet: Query Parameterization
- OWASP Automated Threats to Web Applications – OAT-014

Weitere

- CWE-77: Command Injection
- CWE-89: SQL Injection
- CWE-564: Hibernate Injection
- CWE-917: Expression Language Injection
- PortSwigger: Server-side template injection

EXXETA AG
Albert-Nestler-Straße 19, 76131 Karlsruhe
t +49 721 50994-5000 security@EXXETA.com