|  |
| --- |
| 4aWI |
| DVWA |
| Beer Martin |

|  |
| --- |
| Beer Martin  10.1.2025 |

Inhalt

[1.1 Brute Force 0](#_Toc195200154)

[1.1 Information 0](#_Toc195200155)

[1.2 Graphische Darstellung 1](#_Toc195200156)

[1.3 Code 2](#_Toc195200157)

[1.4 Tools zum Hacken und Testen 4](#_Toc195200158)

[2 Command Injection 5](#_Toc195200159)

[2.1 Information 5](#_Toc195200160)

[2.2 Graphische Darstellung 6](#_Toc195200161)

[2.3 Code 7](#_Toc195200162)

[2.4 Tools zum Hacken und Testen 8](#_Toc195200163)

[3 CSRF 9](#_Toc195200164)

[3.1 Information 9](#_Toc195200165)

[3.2 Graphische Darstellung 9](#_Toc195200166)

[3.3 Code 10](#_Toc195200167)

[3.4 Tools zum Hacken und Testen 12](#_Toc195200168)

[4 File Inclusion 13](#_Toc195200169)

[4.1 Information 13](#_Toc195200170)

[4.2 Graphische Darstellung 15](#_Toc195200171)

[4.3 Code 16](#_Toc195200172)

[4.4 Tools zum Hacken und Testen 18](#_Toc195200173)

[5 File Upload 19](#_Toc195200174)

[5.1 Information 19](#_Toc195200175)

[5.2 Graphische Darstellung 20](#_Toc195200176)

[5.3 Code 21](#_Toc195200177)

[5.4 Tools zum Hacken und Testen 23](#_Toc195200178)

[6 Insecure CAPTCHA 24](#_Toc195200179)

[6.1 Information 24](#_Toc195200180)

[6.2 Graphische Darstellung 25](#_Toc195200181)

[6.3 Code 26](#_Toc195200182)

[6.4 Tools zum Hacken und Testen 28](#_Toc195200183)

[7 SQL Injection 29](#_Toc195200184)

[7.1 Information 29](#_Toc195200185)

[7.2 Graphische Darstellung 30](#_Toc195200186)

[7.3 Code 30](#_Toc195200187)

[7.3 Tools zum Hacken und Testen 32](#_Toc195200188)

[8 SQL Injection (blind) 33](#_Toc195200189)

[8.1 Information 33](#_Toc195200190)

[8.2 Graphische Darstellung 35](#_Toc195200191)

[8.3 Code 36](#_Toc195200192)

[8.4 Tools zum Hacken und Testen 38](#_Toc195200193)

[9 Weak Session IDs 39](#_Toc195200194)

[10 XSS (DOM) 39](#_Toc195200195)

[11 XSS (Reflected) 39](#_Toc195200196)

[12 XSS (Stored) 39](#_Toc195200197)

[13 CSP Byepass 39](#_Toc195200198)

[14 JavaScript 39](#_Toc195200199)

[15 Authorisation Byepass 39](#_Toc195200200)

[16 Open HTTP Redirect 39](#_Toc195200201)

[17 Cryptography 39](#_Toc195200202)

[18 TOOLS 39](#_Toc195200203)

[Abbildungsverzeichnis 40](#_Toc195200204)

# Brute Force

## 1.1 Information

Ein Brute Force Angriff (Brute Force Attack auf Englisch) ist eine Methode, die Cyberkriminelle anwenden, um Passwörter und andere Zugangsdaten zu knacken. Bei einem Brute Force Angriff greift ein Angreifer auf eine Liste an häufigen Wörtern zurück und probiert sie der Reihe nach durch, bis eines funktioniert. Wenn diese Liste zu keinem Ergebnis führt, versucht er es mit verschiedenen Buchstabenkombinationen. Es braucht manchmal tausende von Versuchen, bis ein Passwort geknackt ist. Deshalb kommen für Brute Force, Automatisierungstools zum Einsatz, die es einem Angreifer ermöglichen, sehr viele Versuche möglichst schnell durchzuführen.[[1]](#footnote-1)

Durch einige Rechenbeispiele soll das Zusammenwirken von Länge und verwendeter Zeichen für die Sicherheit eines Passworts veranschaulicht werden. In den Rechenbeispielen wird mit einer Generierung von 2 Milliarden Schlüsseln pro Sekunde gerechnet, da dies ungefähr der Geschwindigkeit eines sehr starken Einzelrechners entspricht.

Beim Erstellen eines Passworts stehen Ihnen in der Regel folgende Zeichen zur Verfügung:

* Zahlen (10 verschiedene: 0-9)
* Buchstaben (52 verschiedene: A-Z und a-z)
* Sonderzeichen (32 verschiedene).[[2]](#footnote-2)

## 1.2 Graphische Darstellung

[](https://www.researchgate.net/publication/379624465/figure/fig2/AS:11431281234681580@1712414089963/Work-architecture-of-Brute-Force-Attack7.ppm)

Abbildung 1BruteForce Attack https://www.researchgate.net/publication/379624465/figure/fig2/AS:11431281234681580@1712414089963/Work-architecture-of-Brute-Force-Attack7.ppm

## 1.3 Code

**Low Brute Force Code:**

<?php

if( isset( $\_GET[ 'Login' ] ) ) {

// Get username

$user = $\_GET[ 'username' ];

// Get password

$pass = $\_GET[ 'password' ];

$pass = md5( $pass );

// Check the database

$query = "SELECT \* FROM `users` WHERE user = '$user' AND password = '$pass';";

$result = mysqli\_query($GLOBALS["\_\_\_mysqli\_ston"], $query ) or die( '<pre>' . ((is\_object($GLOBALS["\_\_\_mysqli\_ston"])) ? mysqli\_error($GLOBALS["\_\_\_mysqli\_ston"]) : (($\_\_\_mysqli\_res = mysqli\_connect\_error()) ? $\_\_\_mysqli\_res : false)) . '</pre>' );

if( $result && mysqli\_num\_rows( $result ) == 1 ) {

// Get users details

$row = mysqli\_fetch\_assoc( $result );

$avatar = $row["avatar"];

// Login successful

echo "<p>Welcome to the password protected area {$user}</p>";

echo "<img src=\"{$avatar}\" />";

}

else {

// Login failed

echo "<pre><br />Username and/or password incorrect.</pre>";

}

((is\_null($\_\_\_mysqli\_res = mysqli\_close($GLOBALS["\_\_\_mysqli\_ston"]))) ? false : $\_\_\_mysqli\_res);

}

?>

**Unbegrenzte Login-Versuche**: Das Skript blockiert den Angreifer nicht nach einer bestimmten Anzahl falscher Passworteingaben.

**Schnelles Durchprobieren**: Da es kein künstliches Delay (z. B. sleep) gibt, können automatisierte Tools sehr schnell alle möglichen Kombinationen testen.

**Keine Account-Sperre**: Selbst nach vielen Fehleingaben wird der Login nicht für eine bestimmte Zeit gesperrt, sodass ein automatisiertes Ausprobieren nicht gestoppt wird.

**Impossible Brute Force Code:**

<?php

if (isset($\_POST['Login'], $\_POST['username'], $\_POST['password'])) {

// 1) Anti-CSRF prüfen

checkToken($\_REQUEST['user\_token'], $\_SESSION['session\_token'], 'index.php');

// 2) Eingaben validieren/escapen

$user = mysqli\_real\_escape\_string($GLOBALS["\_\_\_mysqli\_ston"], stripslashes($\_POST['username']));

$pass = md5(mysqli\_real\_escape\_string($GLOBALS["\_\_\_mysqli\_ston"], stripslashes($\_POST['password'])));

// 3) Lockout-Einstellungen

$failed\_limit = 3;

$lockout\_time = 15; // Minuten

$account\_locked = false;

// 4) Ist der Account schon gesperrt?

$data = $db->prepare('SELECT failed\_login, last\_login FROM users WHERE user = :user LIMIT 1;');

$data->bindParam(':user', $user, PDO::PARAM\_STR);

$data->execute();

$row = $data->fetch();

if ($data->rowCount() && $row['failed\_login'] >= $failed\_limit) {

if (time() < strtotime($row['last\_login']) + ($lockout\_time \* 60)) {

$account\_locked = true;

}

}

// 5) Prüfung der Logindaten

$data = $db->prepare('SELECT \* FROM users WHERE user = :user AND password = :pass LIMIT 1;');

$data->bindParam(':user', $user, PDO::PARAM\_STR);

$data->bindParam(':pass', $pass, PDO::PARAM\_STR);

$data->execute();

// 6) Erfolgreicher Login?

if ($data->rowCount() && !$account\_locked) {

// a) Fehlversuche zurücksetzen

$upd = $db->prepare('UPDATE users SET failed\_login = 0 WHERE user = :user LIMIT 1;');

$upd->bindParam(':user', $user, PDO::PARAM\_STR);

$upd->execute();

echo "Willkommen, $user";

} else {

// a) Verzögerung bei falschem Login

sleep(rand(2, 4));

// b) Fehlversuche hochzählen

$upd = $db->prepare('UPDATE users SET failed\_login = failed\_login + 1 WHERE user = :user LIMIT 1;');

$upd->bindParam(':user', $user, PDO::PARAM\_STR);

$upd->execute();

echo "Fehler: Ungültige Login-Daten oder Account gesperrt.";

}

// 7) Letzte Loginzeit aktualisieren

$upd = $db->prepare('UPDATE users SET last\_login = NOW() WHERE user = :user LIMIT 1;');

$upd->bindParam(':user', $user, PDO::PARAM\_STR);

$upd->execute();

}

// 8) Anti-CSRF-Token erzeugen

generateSessionToken();

?>

**Entscheidende Punkte für sicheren Code sind:**

**Anti-CSRF-Token**: Verhindert, dass Angreifer Logins von fremden Seiten auslösen.

**Eingabesanitierung**: Schützt vor SQL-Injections.

**Lockout**: Nach totalfailedlogintotal\_failed\_logintotalf​ailedl​ogin Fehlversuchen wird der Account für lockouttimelockout\_timelockoutt​ime Minuten gesperrt.

**Zeitverzögerung**: sleep(rand(2, 4)) erschwert automatisierte Angriffe.

**Zurücksetzen und Inkrementieren der**: Erlaubt dem System, erfolgreiche Logins wieder zu normalisieren und bei Fehlern konsequent zu erhöhen.

**Update der letzten Login-Zeit**: Dient zur Berechnung der Sperrzeit und zur Protokollierung.

## 1.4 Tools zum Hacken und Testen

* Hydra
* Burpsuite
* OWASP ZAP
* Klassische Programmierung: Java, JavaScript, node
* Sqlmap (Testing-Tool)

# Command Injection

## Information

Command Injection ist ein Angriff, bei dem das Ziel darin besteht, beliebige Befehle auf dem Host-Betriebssystem über eine verwundbare Anwendung auszuführen. Command Injection-Angriffe sind möglich, wenn eine Anwendung unsichere, vom Benutzer bereitgestellte Daten (wie Formulare, Cookies, HTTP-Header etc.) an eine System-Shell übergibt. Bei diesem Angriff werden die vom Angreifer gelieferten Betriebssystembefehle in der Regel mit den Privilegien der verwundbaren Anwendung ausgeführt. Command Injection-Angriffe sind vor allem auf unzureichende Eingabevalidierung zurückzuführen.

Dieser Angriff unterscheidet sich von Code Injection, bei der es dem Angreifer ermöglicht wird, eigenen Code hinzuzufügen, der dann von der Anwendung ausgeführt wird. Bei Command Injection erweitert der Angreifer die Standardfunktionalität der Anwendung, die Systembefehle ausführt, ohne dass es notwendig ist, eigenen Code einzuspritzen.[[3]](#footnote-3)

**Wie man OS-Command-Injection-Angriffe verhindert**

Der effektivste Weg, OS-Command-Injection-Schwachstellen zu verhindern, besteht darin, niemals von Anwendungscode aus Betriebssystembefehle aufzurufen. In fast allen Fällen gibt es alternative Möglichkeiten, die erforderliche Funktionalität mithilfe sichererer Plattform-APIs zu implementieren.

Falls es unumgänglich ist, Betriebssystembefehle mit benutzerdefinierten Eingaben auszuführen, muss eine strenge Eingabevalidierung erfolgen. Einige Beispiele für eine effektive Validierung sind:

* Validierung anhand einer Positivliste (Whitelist) zugelassener Werte.
* Überprüfung, ob die Eingabe eine Zahl ist.
* Validierung, dass die Eingabe ausschließlich alphanumerische Zeichen enthält, ohne weitere Syntax oder Leerzeichen.[[4]](#footnote-4)

## Graphische Darstellung

Ein Bild, das Text, Diagramm, Entwurf, Plan enthält.

KI-generierte Inhalte können fehlerhaft sein.

Abbildung 2 Command Injection https://encrypted-tbn0.gstatic.com/images?q=tbn:ANd9GcT5MDilLllu\_4qBsOfOnh94\_TBFumhVfExXnpK-cXRSaTsYrXnr

Ein Bild, das Text, Diagramm, Screenshot, parallel enthält.

KI-generierte Inhalte können fehlerhaft sein.

Figure 1 Command Injection Codesteigerung Quelle: Schwärzler

## Code

**Low Command Injection Code:**

<?php

if( isset( $\_POST[ 'Submit' ] ) ) {

// Get input

$target = $\_REQUEST[ 'ip' ];

// Determine OS and execute the ping command.

if( stristr( php\_uname( 's' ), 'Windows NT' ) ) {

// Windows

$cmd = shell\_exec( 'ping ' . $target );

}

else {

// \*nix

$cmd = shell\_exec( 'ping -c 4 ' . $target );

}

// Feedback for the end user

echo "<pre>{$cmd}</pre>";

}

?>

Der "Low Command Injection Code" ist unsicher, weil er den vom Benutzer eingegebenen Wert direkt in den Shell-Befehl einfügt, ohne diesen zu validieren oder zu bereinigen. Dadurch kann ein Angreifer zusätzliche Befehle einschleusen (z. B. durch das Einfügen von Semikolons oder anderen Shell-Metazeichen), was zu einer Ausführung beliebiger Systembefehle führen kann.

**Impossible Command Injection Code:**

<?php

if( isset( $\_POST[ 'Submit' ] ) ) {

// Check Anti-CSRF token

checkToken( $\_REQUEST[ 'user\_token' ], $\_SESSION[ 'session\_token' ], 'index.php' );

// Get input

$target = $\_REQUEST[ 'ip' ];

$target = stripslashes( $target );

// Split the IP into 4 octects

$octet = explode( ".", $target );

// Check IF each octet is an integer

if( ( is\_numeric( $octet[0] ) ) && ( is\_numeric( $octet[1] ) ) && ( is\_numeric( $octet[2] ) ) && ( is\_numeric( $octet[3] ) ) && ( sizeof( $octet ) == 4 ) ) {

// If all 4 octets are int's put the IP back together.

$target = $octet[0] . '.' . $octet[1] . '.' . $octet[2] . '.' . $octet[3];

// Determine OS and execute the ping command.

if( stristr( php\_uname( 's' ), 'Windows NT' ) ) {

// Windows

$cmd = shell\_exec( 'ping ' . $target );

}

else {

// \*nix

$cmd = shell\_exec( 'ping -c 4 ' . $target );

}

// Feedback for the end user

echo "<pre>{$cmd}</pre>";

}

else {

// Ops. Let the user name theres a mistake

echo '<pre>ERROR: You have entered an invalid IP.</pre>';

}

}

// Generate Anti-CSRF token

generateSessionToken();

?>

Der Impossible Command Injection Code hingegen ergreift mehrere Sicherheitsmaßnahmen:

* **Strenge Validierung:** Der eingegebene IP-Wert wird mittels explode in vier Oktette aufgeteilt. Anschließend wird überprüft, ob jedes Oktett numerisch ist und ob tatsächlich genau vier Teile vorliegen. Das stellt sicher, dass nur gültige IPv4-Adressen akzeptiert werden.
* **Anti-CSRF Token:** Durch den Einsatz eines Anti-CSRF Tokens wird zusätzlich verhindert, dass Angreifer über Cross-Site-Request-Forgery-Angriffe schädliche Anfragen stellen können.

## Tools zum Hacken und Testen

* Commix (COMMand Injection eXploiter)
* Metasploit Framework
* Burpsuit
* Nuclei
* Custom Scripts (Python, Bash, …)

# CSRF

## Information

CSRF ist ein Angriff, der das Opfer dazu bringt, eine bösartige Anfrage abzusenden. Dabei übernimmt der Angriff die Identität und Privilegien des Opfers, um in dessen Namen eine unerwünschte Funktion auszuführen (wobei zu beachten ist, dass dies bei Login-CSRF, einer speziellen Form des Angriffs, wie im Folgenden beschrieben, nicht zutrifft). Bei den meisten Webseiten beinhalten Browseranfragen automatisch alle mit der Seite verbundenen Zugangsdaten, wie das Sitzungscookie des Benutzers, die IP-Adresse, Windows-Domänenanmeldeinformationen und so weiter. Daher hat die Webseite, wenn der Benutzer gerade authentifiziert ist, keine Möglichkeit zu unterscheiden, ob es sich um eine gefälschte Anfrage des Opfers oder um eine legitime Anfrage handelt.[[5]](#footnote-5)

## Graphische Darstellung

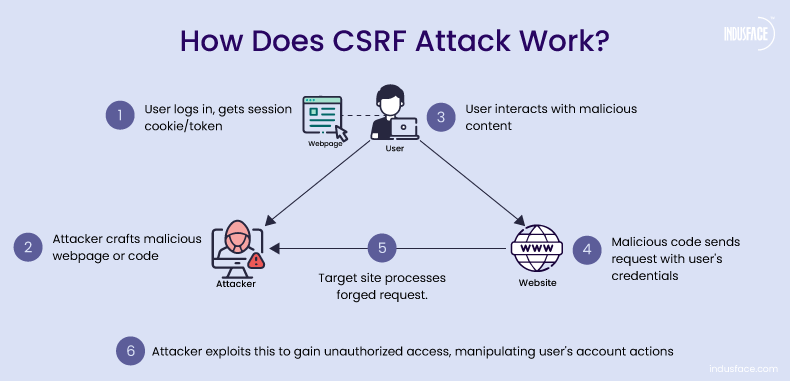


Abbildung 3 CSRF https://encrypted-tbn0.gstatic.com/images?q=tbn:ANd9GcSNaq9pJtukr0lln4EroZ6BOW8noYucd2H4pc1KNhkS1MXflmpP

## Code

**Low CSRF Code:**

<?php

if( isset( $\_GET[ 'Change' ] ) ) {

// Get input

$pass\_new = $\_GET[ 'password\_new' ];

$pass\_conf = $\_GET[ 'password\_conf' ];

// Do the passwords match?

if( $pass\_new == $pass\_conf ) {

// They do!

$pass\_new = ((isset($GLOBALS["\_\_\_mysqli\_ston"]) && is\_object($GLOBALS["\_\_\_mysqli\_ston"])) ? mysqli\_real\_escape\_string($GLOBALS["\_\_\_mysqli\_ston"], $pass\_new ) : ((trigger\_error("[MySQLConverterToo] Fix the mysql\_escape\_string() call! This code does not work.", E\_USER\_ERROR)) ? "" : ""));

$pass\_new = md5( $pass\_new );

// Update the database

$current\_user = dvwaCurrentUser();

$insert = "UPDATE `users` SET password = '$pass\_new' WHERE user = '" . $current\_user . "';";

$result = mysqli\_query($GLOBALS["\_\_\_mysqli\_ston"], $insert ) or die( '<pre>' . ((is\_object($GLOBALS["\_\_\_mysqli\_ston"])) ? mysqli\_error($GLOBALS["\_\_\_mysqli\_ston"]) : (($\_\_\_mysqli\_res = mysqli\_connect\_error()) ? $\_\_\_mysqli\_res : false)) . '</pre>' );

// Feedback for the user

echo "<pre>Password Changed.</pre>";

}

else {

// Issue with passwords matching

echo "<pre>Passwords did not match.</pre>";

}

((is\_null($\_\_\_mysqli\_res = mysqli\_close($GLOBALS["\_\_\_mysqli\_ston"]))) ? false : $\_\_\_mysqli\_res);

}

?>

**Übertragung sensibler Daten über GET:**  
Die neuen Passwörter werden per GET übermittelt, was dazu führt, dass diese Informationen in Browser-Historien und Server-Logs landen können.

**Fehlender CSRF-Schutz:**  
Es wird kein Anti-CSRF-Token verwendet, sodass ein Angreifer das Opfer dazu bringen könnte, ungewollt das Passwort zu ändern.

**Unzureichende SQL-Sicherheit:**  
Obwohl bei $pass\_new eine Escape-Funktion genutzt wird, wird der Benutzername direkt in die SQL-Anfrage eingebettet. Dadurch entsteht ein potenzielles Risiko für SQL-Injection, da keine Prepared Statements verwendet werden.

**Veraltetes Hashing-Verfahren:**  
Die Verwendung von MD5 zum Hashen der Passwörter gilt als unsicher, da MD5 anfällig für Kollisionen und Brute-Force-Angriffe ist.

**Mangelhafte Fehlerbehandlung:**  
Durch den direkten Einsatz von die() können im Fehlerfall interne Details an den Angreifer preisgegeben werden.

**Impossible CSRF Code:**

<?php

if (isset($\_GET['Change'])) {

// Anti-CSRF-Überprüfung

checkToken($\_REQUEST['user\_token'], $\_SESSION['session\_token'], 'index.php');

// Eingaben abrufen und sanitizen

$pass\_curr = md5(mysqli\_real\_escape\_string($db\_connection, stripslashes($\_GET['password\_current'])));

$pass\_new = $\_GET['password\_new'];

$pass\_conf = $\_GET['password\_conf'];

// Aktuelles Passwort mit Prepared Statement prüfen

$stmt = $db->prepare('SELECT password FROM users WHERE user = :user AND password = :password LIMIT 1;');

$current\_user = dvwaCurrentUser();

$stmt->bindParam(':user', $current\_user, PDO::PARAM\_STR);

$stmt->bindParam(':password', $pass\_curr, PDO::PARAM\_STR);

$stmt->execute();

// Prüfung: Ist das aktuelle Passwort korrekt und stimmen die neuen Passwörter überein?

if ($stmt->rowCount() === 1 && $pass\_new === $pass\_conf) {

$pass\_new = md5(mysqli\_real\_escape\_string($db\_connection, stripslashes($pass\_new)));

$update = $db->prepare('UPDATE users SET password = :password WHERE user = :user;');

$update->bindParam(':password', $pass\_new, PDO::PARAM\_STR);

$update->bindParam(':user', $current\_user, PDO::PARAM\_STR);

$update->execute();

echo "<pre>Password Changed.</pre>";

} else {

echo "<pre>Passwords did not match or current password incorrect.</pre>";

}

}

// Neuer Anti-CSRF Token für die Session

generateSessionToken();

?>

**Wesentliche Sicherheitsaspekte:**

* **Anti-CSRF-Überprüfung:** Mit checkToken() wird sichergestellt, dass die Anfrage von einem berechtigten Benutzer stammt.
* **Input-Sanitization:** Durch stripslashes() und mysqli\_real\_escape\_string() werden Eingaben bereinigt, um SQL-Injection zu verhindern.
* **Prepared Statements:** Die Verwendung von Prepared Statements mit bindParam() verhindert SQL-Injection, da Benutzereingaben nicht direkt in die SQL-Abfrage eingebettet werden.
* **Passwort-Hashing:** Die Passwörter werden mittels md5() gehasht, um sie nicht im Klartext zu speichern (obwohl md5 heutzutage als veraltet gilt).
* **Token-Erneuerung:** Mit generateSessionToken() wird nach der Anfrage ein neuer Anti-CSRF-Token erstellt, um die Sicherheit der Session zu gewährleisten.

## 3.4 Tools zum Hacken und Testen

* BurpSuite
* OWASP ZAP (Zed Attack Proxy)
* Postman
* NoCSRF
* CSRF PoC Generator
* WebGoat
* BeEF

# File Inclusion

## Information

Eine der häufigsten Arten von Angriffen auf Webserver resultiert aus File-Inclusion-Schwachstellen. Diese Schwachstellen finden sich hauptsächlich in Webanwendungen, die eine Scripting-Laufzeit verwenden. Sie ermöglichen es Angreifern, auf sensible Dateien auf den Webservern zuzugreifen oder die Include-Funktion zu nutzen, um schädliche Dateien auf den Servern auszuführen. Durch den Zugriff auf unautorisierte Dateien können Angreifer sensible Informationen erlangen oder die Netzwerke des Opfers weiter kompromittieren.

Es gibt zwei Typen von File-Inclusion-Schwachstellen, die Sie bei der Vorbereitung auf einen Angriff kennen sollten: lokale File-Inclusion (LFI) und Remote-File-Inclusion (RFI).

Der Hauptunterschied zwischen den beiden liegt darin, wo sich die kompromittierte Datei ursprünglich befindet. LFI-Schwachstellen werden durch eine auf dem Zielserver gespeicherte Datei ausgenutzt, während RFI eine Datei aus einer Drittquelle verwendet.

**LFI involves:**

Wenn Eingaben auf einer Website oder in einer Webanwendung nicht ordnungsgemäß bereinigt werden, können lokale Dateien auf einem Server anfällig für Angriffe werden. Benutzer-Eingaben, die Pfade zu Dateien enthalten und falsch validiert werden, ermöglichen es Angreifern, auf diese Dateien zuzugreifen und sensible Dateien in anderen Verzeichnissen abzurufen.

Das folgende INFOSEC-Beispiel veranschaulicht, wie ein lokaler File-Inclusion-Angriff erfolgen kann:

http://victim\_site/abc.php?file=userinput.txt

<?php

…

include $\_REQUEST['file'];

…

?>

Angreifer können eine bösartige Eingabe einfügen, um sensible Dateien im aktuellen Verzeichnis abzurufen oder in andere Verzeichnisse zu wechseln, um Ihr System weiter zu kompromittieren.

**RFI**

Obwohl ähnlich, nutzen RFI-Schwachstellen eine externe Quelle, anstatt Dateien vom lokalen Server zuzugreifen, um einen Angriff auszuführen. Angreifer verwenden den Befehl „dynamic file include“, um schädliche externe Dateien oder Skripte einzufügen. Ohne ordnungsgemäße Bereinigung von Dateien können Angreifer Webanwendungen ausnutzen, um externe Dateien mit schädlichen Skripten einzufügen.

Hier ist ein Beispiel dafür, wie eine RFI-Schwachstelle auftreten könnte:

Bash:

[www.victim\_site.com/abc.php?testfile=example](http://www.victim_site.com/abc.php?testfile=example)

Der anfällige PHP-Code:

Shell:

$test = $\_REQUEST["testfile"];

include($test . ".php");

Der Parameter „testfile“ in diesem Beispiel wird vom Benutzer bereitgestellt, und der Code fügt den Wert von „testfile“ in die PHP-Datei ein.

In beiden Fällen resultieren die Schwachstellen direkt aus mangelhafter Eingabevalidierung, was bedeutet, dass eine der wenigen Möglichkeiten zur Vermeidung von File-Inclusion-Schwachstellen in der konsequenten Anwendung von Bereinigungspraktiken liegt.[[6]](#footnote-6)

## Graphische Darstellung

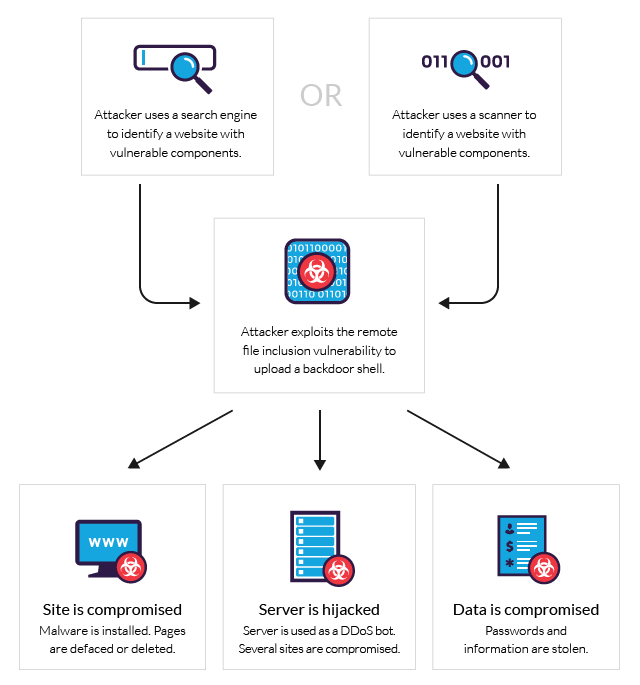


Abbildung 4 File-Inclusion <https://encrypted-tbn2.gstatic.com/images?q=tbn:ANd9GcSD-W7x1jQSSMl8jkIsfJl3qipeEgxk2fTBA_D8lkzumw3OlH8w>

## Code

**Low File-Inclusion Code:**

<?php

// The page we wish to display

$file = $\_GET[ 'page' ];

?>

**Warum der "Low Security" Code unsicher ist:**

Im "Low Security" Code wird der Dateiname direkt aus der GET-Variable übernommen, ohne jegliche Überprüfung oder Validierung. Das bedeutet, dass ein Angreifer beliebige Dateinamen übergeben kann. Dadurch kann er:

* **Lokale File Inclusion (LFI)** ausnutzen, um sensible Dateien auf dem Server zu lesen.
* **Remote File Inclusion (RFI)**, sofern die Serverkonfiguration es erlaubt, ausführen, wodurch schädlicher Code von extern geladen wird.
* **Directory Traversal**-Angriffe durchführen, indem er Pfadmanipulationen (z. B. ../) einsetzt, um Dateien außerhalb des erlaubten Verzeichnisses abzurufen.

Da keinerlei Sicherheitsmechanismen wie eine Whitelist oder eine gründliche Eingabevalidierung vorhanden sind, ist der Code sehr anfällig für Angriffe.

**Impossible File-Inclusion Code:**

<?php

// The page we wish to display

$file = $\_GET[ 'page' ];

// Only allow include.php or file{1..3}.php

$configFileNames = [

'include.php',

'file1.php',

'file2.php',

'file3.php',

];

if( !in\_array($file, $configFileNames) ) {

// This isn't the page we want!

echo "ERROR: File not found!";

exit;

}

?>

**Warum der "Impossible Security" Code so sicher ist:**

Im "Impossible Security" Code wird ebenfalls der Dateiname aus der GET-Variable entnommen, aber anschließend wird eine Whitelist verwendet. Nur wenn der übergebene Dateiname exakt mit einem der erlaubten Einträge (include.php, file1.php, file2.php, file3.php) übereinstimmt, wird der Code ausgeführt. Andernfalls wird ein Fehler ausgegeben und der Prozess beendet. Dadurch wird:

* **Arbiträre Dateiauswahl verhindert:** Ein Angreifer kann nur eine der explizit erlaubten Dateien anfordern.
* **File Inclusion Angriffe verhindert:** Da keine nicht erlaubten Dateien einbezogen werden können, wird die Gefahr von LFI oder RFI effektiv eliminiert.
* **Eingabevalidierung gewährleistet:** Die Anwendung überprüft die Eingabe gegen eine fest definierte Liste und stellt so sicher, dass nur legitime und vordefinierte Dateien geladen werden.

## 4.4 Tools zum Hacken und Testen

**DVWA (Damn Vulnerable Web Application):**

Eine bewusst unsichere Webanwendung, die verschiedene Sicherheitslücken, darunter File Inclusion, simuliert. DVWA ermöglicht es, Sicherheitslücken zu erkennen und zu beheben.

**bWAPP (Buggy Web Application):**

Ein weiteres Open-Source-Tool, das eine breite Palette an Schwachstellen bietet. bWAPP enthält Module, um File Inclusion-Fehler gezielt zu testen.

**OWASP WebGoat:**

Eine von OWASP bereitgestellte Lehrumgebung, in der verschiedene Angriffe und Schwachstellen, einschließlich File Inclusion, simuliert werden. Es ist ideal für interaktive Lernszenarien.

**Mutillidae II:**

Eine kostenlose, leicht zu installierende Webanwendung, die gezielt zum Testen von Web-Angriffen und Sicherheitstools entwickelt wurde. Mutillidae II enthält Szenarien zu File Inclusion und anderen typischen Angriffen.

**Google Gruyere:**

Eine webbasierte Übungsumgebung, die speziell für das Erlernen von Web-Sicherheit konzipiert wurde. Gruyere bietet praktische Übungen zu diversen Schwachstellen, darunter File Inclusion.

# File Upload

## 5.1 Information

File-Upload-Schwachstellen treten auf, wenn ein Webserver es Benutzern erlaubt, Dateien in sein Dateisystem hochzuladen, ohne ausreichend Aspekte wie den Namen, Typ, Inhalt oder die Größe zu validieren. Werden diese Einschränkungen nicht ordnungsgemäß durchgesetzt, kann bereits eine einfache Image-Upload-Funktion dazu genutzt werden, beliebige und potenziell gefährliche Dateien hochzuladen. Dies könnte sogar serverseitige Skriptdateien einschließen, die eine Remote-Code-Ausführung ermöglichen.

In einigen Fällen reicht der Akt des Hochladens der Datei allein aus, um Schaden anzurichten. Andere Angriffe können eine nachfolgende HTTP-Anfrage für die Datei beinhalten, um deren Ausführung durch den Server zu triggern.

Die Auswirkungen von Datei-Upload-Schwachstellen hängen im Allgemeinen von zwei wesentlichen Faktoren ab:

* **Überprüfung der Dateieigenschaften:**  
  Entscheidend ist, welche Eigenschaften der hochgeladenen Datei – wie Größe, Typ oder Inhalt – von der Webanwendung nicht korrekt validiert werden.
* **Angewandte Beschränkungen:**  
  Ebenso wichtig ist, welche Einschränkungen nach dem Hochladen greifen, etwa wo und wie die Datei gespeichert oder ausgeführt wird.

Im schlimmsten Fall wird der Dateityp nicht ordnungsgemäß geprüft und der Server erlaubt, dass bestimmte Dateiformate (wie .php oder .jsp) als Code ausgeführt werden. Dadurch könnte ein Angreifer eine serverseitige Code-Datei hochladen, die als Webshell dient und ihm somit vollständige Kontrolle über den Server verschafft.

Wenn der Dateiname nicht korrekt validiert wird, besteht zudem die Gefahr, dass ein Angreifer kritische Dateien überschreibt, indem er eine Datei mit demselben Namen hochlädt. Sollte der Server außerdem anfällig für Directory-Traversal sein, kann dies dazu führen, dass Dateien an unvorhergesehenen Orten gespeichert werden.

Fehlt zudem die Prüfung, ob die Dateigröße innerhalb zulässiger Grenzen liegt, kann ein Angreifer den verfügbaren Speicherplatz füllen und so einen Denial-of-Service (DoS)-Angriff auslösen.[[7]](#footnote-7)

## 5.2 Graphische Darstellung

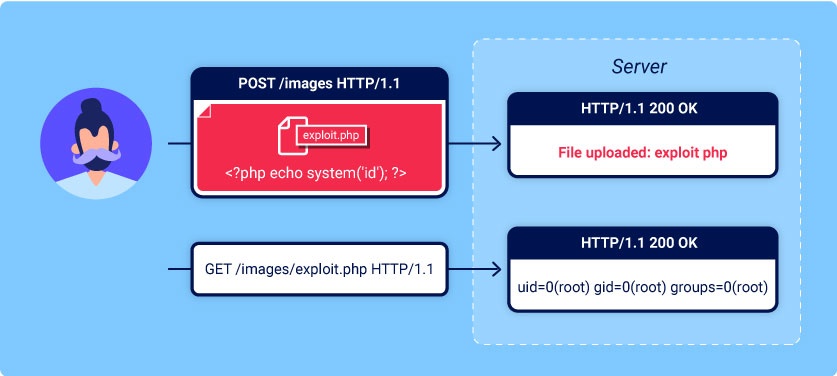


Abbildung 5 File Upload https://portswigger.net/web-security/file-upload/images/file-upload-vulnerabilities.jpg

## 5.3 Code

**Low File Upload Code:**

<?php

if( isset( $\_POST[ 'Upload' ] ) ) {

// Where are we going to be writing to?

$target\_path = DVWA\_WEB\_PAGE\_TO\_ROOT . "hackable/uploads/";

$target\_path .= basename( $\_FILES[ 'uploaded' ][ 'name' ] );

// Can we move the file to the upload folder?

if( !move\_uploaded\_file( $\_FILES[ 'uploaded' ][ 'tmp\_name' ], $target\_path ) ) {

// No

echo '<pre>Your image was not uploaded.</pre>';

}

else {

// Yes!

echo "<pre>{$target\_path} succesfully uploaded!</pre>";

}

}

?>

**Warum der „Low Security Code“ unsicher ist:**

* **Fehlende Validierung:** Es gibt keine Überprüfung des Dateityps, der Dateigröße oder des Inhalts. Dadurch können beliebige Dateien, z. B. PHP-Skripte, hochgeladen werden.
* **Keine Anti-CSRF-Maßnahmen:** Es fehlt ein Schutzmechanismus gegen Cross-Site Request Forgery, was Manipulationen ermöglicht.
* **Keine Metadaten-Bereinigung:** Es wird keine Überprüfung oder Entfernung von potenziell schädlichen Metadaten vorgenommen.

**Impossible File Upload Code:**

if(isset($\_POST['Upload'])) {

// Anti-CSRF-Prüfung

checkToken($\_REQUEST['user\_token'], $\_SESSION['session\_token'], 'index.php');

// Dateiinformationen

$uploaded\_ext = strtolower(pathinfo($\_FILES['uploaded']['name'], PATHINFO\_EXTENSION));

$uploaded\_size = $\_FILES['uploaded']['size'];

$uploaded\_type = $\_FILES['uploaded']['type'];

$uploaded\_tmp = $\_FILES['uploaded']['tmp\_name'];

// Überprüfungen

if(($uploaded\_ext == 'jpg' || $uploaded\_ext == 'jpeg' || $uploaded\_ext == 'png') &&

($uploaded\_size < 100000) &&

($uploaded\_type == 'image/jpeg' || $uploaded\_type == 'image/png') &&

getimagesize($uploaded\_tmp)) {

// Re-Encoding zur Metadaten-Entfernung

$img = ($uploaded\_type == 'image/jpeg') ? imagecreatefromjpeg($uploaded\_tmp) : imagecreatefrompng($uploaded\_tmp);

$temp\_file = sys\_get\_temp\_dir() . DIRECTORY\_SEPARATOR . md5(uniqid() . $\_FILES['uploaded']['name']) . '.' . $uploaded\_ext;

($uploaded\_type == 'image/jpeg') ? imagejpeg($img, $temp\_file, 100) : imagepng($img, $temp\_file, 9);

imagedestroy($img);

// Sicherer Dateiname

$target\_file = md5(uniqid() . $\_FILES['uploaded']['name']) . '.' . $uploaded\_ext;

$target\_path = DVWA\_WEB\_PAGE\_TO\_ROOT . 'hackable/uploads/' . $target\_file;

if(rename($temp\_file, $target\_path)) {

echo "<a href='{$target\_path}'>{$target\_file}</a> succesfully uploaded!";

} else {

echo 'Your image was not uploaded.';

}

} else {

echo 'Your image was not uploaded. We only accept JPEG or PNG images.';

}

}

generateSessionToken();

**Warum der „Impossible Security Code“ so sicher ist:**

* **Anti-CSRF-Token:** Durch die Überprüfung des Anti-CSRF-Tokens wird sichergestellt, dass nur authentische Upload-Anfragen verarbeitet werden.
* **Umfassende Dateiprüfung:**
  + **Erweiterungs- und MIME-Typ-Check:** Nur JPEG- oder PNG-Dateien werden akzeptiert.
  + **Größenprüfung:** Dateien, die zu groß sind, werden abgelehnt.
  + **Validierung durch getimagesize:** Dies stellt sicher, dass die Datei tatsächlich ein Bild ist.
* **Metadaten-Entfernung:** Durch das Re-Encoding der Bilder werden schädliche Metadaten entfernt.
* **Sicherer Dateiname:** Der Dateiname wird zufällig generiert (mit md5 und uniqid), um Manipulationen zu verhindern.

## 5.4 Tools zum Hacken und Testen

**OWASP ZAP:** Open-Source-Security-Tool zum Auffinden von Schwachstellen.

**Burp Suite (Community Edition):** Ermöglicht das Abfangen und Manipulieren von Upload-Anfragen.

**DVWA:** Speziell entwickelte Übungsumgebung für File-Upload-Angriffe.

**OWASP Juice Shop:** Realistische Web-App mit diversen Sicherheitslücken zum Üben.

# Insecure CAPTCHA

## 6.1 Information

Ein CAPTCHA ist ein Programm, das feststellen kann, ob sein Nutzer ein Mensch oder ein Computer ist. Du hast solche CAPTCHAs wahrscheinlich schon gesehen – farbige Bilder mit verzerrtem Text, die häufig am Ende von Web-Registrierungsformularen auftauchen. Viele Websites setzen CAPTCHAs ein, um sich vor Missbrauch durch „Bots“ zu schützen. Dabei handelt es sich um automatisierte Programme, die meistens entwickelt wurden, um Spam zu erzeugen. Kein Computerprogramm kann verzerrten Text so gut lesen wie ein Mensch, weshalb Bots Webseiten, die mit CAPTCHAs geschützt sind, nicht erfolgreich nutzen können.

CAPTCHAs werden oft eingesetzt, um sensible Funktionen vor automatisierten Bots zu schützen. Solche Funktionen umfassen typischerweise die Registrierung und Änderungen von Nutzerkonten, Passwortänderungen sowie das Veröffentlichen von Inhalten. In diesem Beispiel schützt das CAPTCHA die Funktion zur Passwortänderung eines Benutzerkontos. Dadurch bietet es einen gewissen Schutz vor CSRF-Angriffen (Cross-Site Request Forgery) sowie vor automatisierten Versuchen durch Bots, Passwörter zu erraten.[[8]](#footnote-8)

## 6.2 Graphische Darstellung

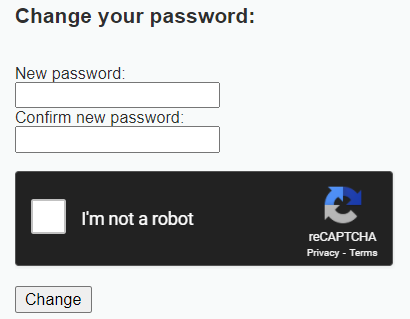


Figure 2 Insecure Captcha Anwendung [https://miro.medium.com/v2/resize:fit:820/1\*Si7zAJhlCEF2BxHBaT3eQg.png](https://miro.medium.com/v2/resize:fit:820/1*Si7zAJhlCEF2BxHBaT3eQg.png)

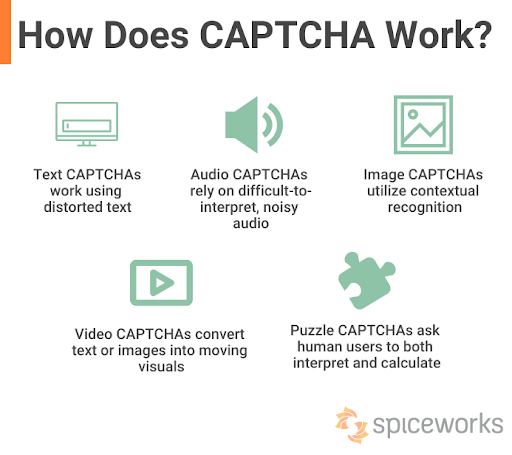


Figure 3 Insecure Captcha Funktion <https://zd-brightspot.s3.us-east-1.amazonaws.com/wp-content/uploads/2023/03/13170658/How-CAPTCHA-Works.png>

## 6.3 Code

**Low Insecure Captcha Code:**

<?php

if(isset($\_POST['Change'])) {

$step = $\_POST['step'];

$pass\_new = $\_POST['password\_new'];

$pass\_conf = $\_POST['password\_conf'];

if($step == '1') {

// Prüfe CAPTCHA

if(!recaptcha\_check\_answer($\_DVWA['recaptcha\_private\_key'], $\_POST['g-recaptcha-response'])) {

echo "<pre>CAPTCHA falsch.</pre>";

} elseif($pass\_new !== $pass\_conf) {

echo "<pre>Passwörter stimmen nicht überein.</pre>";

} else {

// Nächster Schritt

echo "<pre>CAPTCHA korrekt. Bestätige deine Änderungen.</pre>

<form method=\"POST\">

<input type=\"hidden\" name=\"step\" value=\"2\" />

<input type=\"hidden\" name=\"password\_new\" value=\"{$pass\_new}\" />

<input type=\"hidden\" name=\"password\_conf\" value=\"{$pass\_conf}\" />

<input type=\"submit\" name=\"Change\" value=\"Change\" />

</form>";

}

} elseif($step == '2') {

if($pass\_new === $pass\_conf) {

// Unsichere Passwortspeicherung (MD5)!

$hashed\_pass = md5(mysqli\_real\_escape\_string($GLOBALS["\_\_\_mysqli\_ston"], $pass\_new));

mysqli\_query($GLOBALS["\_\_\_mysqli\_ston"], "UPDATE users SET password='$hashed\_pass' WHERE user='".dvwaCurrentUser()."'")

or die("<pre>Datenbankfehler.</pre>");

echo "<pre>Passwort geändert.</pre>";

} else {

echo "<pre>Passwörter stimmen nicht überein.</pre>";

}

}

}

?>

**Warum der Low Security Code unsicher ist:**

**MD5-Verschlüsselung**:

Der Code verwendet MD5 zur Speicherung von Passwörtern. MD5 gilt heute als unsicher, da es mit wenig Aufwand mittels Rainbow Tables oder Brute-Force geknackt werden kann.

**SQL-Injection-Risiko**:

Obwohl hier mysqli\_real\_escape\_string genutzt wird, ist es grundsätzlich sicherer, Prepared Statements (Prepared Queries) zu verwenden. Ein Fehler bei der Nutzung von Escape-Funktionen kann leicht zu SQL-Injection führen.

**Unsichere Logik**:

Das Passwort wird über ein verstecktes Formularfeld (Hidden Input) vom Client an den Server übertragen. Dadurch könnte das Passwort durch Man-in-the-Middle-Attacken oder Manipulation abgefangen werden.

Es gibt keine serverseitige Validierung auf Passwortkomplexität.

**Keine CSRF-Schutzmaßnahmen**:

Der CAPTCHA bietet nur begrenzten Schutz vor Cross-Site Request Forgery (CSRF). Angriffe könnten dennoch möglich sein, insbesondere bei schlecht implementierten Sessions.

**Impossible Insecure Captcha Code:**

<?php

if(isset($\_POST['Change'])) {

checkToken($\_REQUEST['user\_token'], $\_SESSION['session\_token'], 'index.php');

$pass\_curr = $\_POST['password\_current'];

$pass\_new = $\_POST['password\_new'];

$pass\_conf = $\_POST['password\_conf'];

if(!recaptcha\_check\_answer($\_DVWA['recaptcha\_private\_key'], $\_POST['g-recaptcha-response'])) {

echo "<pre>CAPTCHA falsch. Versuche es erneut.</pre>";

} else {

$stmt = $db->prepare('SELECT password FROM users WHERE user=:user LIMIT 1;');

$stmt->bindParam(':user', dvwaCurrentUser());

$stmt->execute();

if($stmt->rowCount() === 1) {

$user = $stmt->fetch();

// Sichere Passwortprüfung (z.B. bcrypt)

if(password\_verify($pass\_curr, $user['password']) && ($pass\_new === $pass\_conf)) {

$pass\_hashed = password\_hash($pass\_new, PASSWORD\_DEFAULT);

$update = $db->prepare('UPDATE users SET password=:pass WHERE user=:user;');

$update->bindParam(':pass', $pass\_hashed);

$update->bindParam(':user', dvwaCurrentUser());

$update->execute();

echo "<pre>Passwort erfolgreich geändert.</pre>";

} else {

echo "<pre>Aktuelles Passwort falsch oder neue Passwörter stimmen nicht überein.</pre>";

}

}

}

}

// Neues CSRF-Token erzeugen

generateSessionToken();

?>

**1. Verwendung sicherer Passwort-Hashes:**

* password\_hash() und password\_verify() nutzen standardmäßig starke Algorithmen (bcrypt/Argon2).
* Dies verhindert effizientes Cracken durch Rainbow-Tables oder Brute-Force-Angriffe.

**2. Prepared Statements mit PDO:**

* Prepared Statements verhindern zuverlässig SQL-Injections.
* Nutzereingaben werden klar getrennt von der SQL-Logik verarbeitet.

**3. CSRF-Token Validierung:**

* Schutz gegen Cross-Site Request Forgery Angriffe durch checkToken() und regelmäßiges Erneuern der Tokens.

**4. CAPTCHA als Bot-Schutz:**

* ReCAPTCHA schützt zuverlässig vor automatisierten Angriffen.

## 6.4 Tools zum Hacken und Testen

**Tesseract OCR** – Open-Source-Tool zur Texterkennung, ideal zum Knacken einfacher Text-CAPTCHAs.

**Sentry MBA** – Automatisierungstool für Angriffe auf Login-Formulare mit schwachem CAPTCHA-Schutz.

**GSA CAPTCHA Breaker** – Kommerzielles Tool zur automatischen Erkennung und Lösung von CAPTCHAs.

**Death by Captcha** – Dienst, der CAPTCHAs durch echte Personen oder KI lösen lässt.

**OpenCV + Python** – Bibliothek zur Bildverarbeitung, mit der sich eigene CAPTCHA-Lösungen bauen lassen.

**Burp Suite** – Penetration-Testing-Tool zum Testen von Webanwendungen, inklusive CAPTCHA-Manipulation.

# SQL Injection

## Information

**SQL-Injection (SQLi)** ist eine Sicherheitslücke bei Webanwendungen, die es einem Angreifer ermöglicht, in die Datenbankabfragen der Anwendung einzugreifen. Dadurch kann der Angreifer Daten einsehen, auf die er normalerweise keinen Zugriff hätte – zum Beispiel Informationen anderer Benutzer oder andere sensible Daten, auf die die Anwendung Zugriff hat.

In vielen Fällen kann ein Angreifer diese Daten auch verändern oder löschen und damit dauerhafte Änderungen am Inhalt oder Verhalten der Anwendung verursachen.

Unter bestimmten Umständen kann ein Angreifer eine SQL-Injection sogar ausweiten, um den zugrunde liegenden Server oder andere Teile der Backend-Infrastruktur zu kompromittieren. Zudem kann er dadurch auch Denial-of-Service-Angriffe (DoS) durchführen.

Ein erfolgreicher SQL-Injection-Angriff kann zu unbefugtem Zugriff auf sensible Daten führen, wie zum Beispiel:

* Passwörter
* Kreditkartendaten
* Persönliche Informationen von Nutzern

SQL-Injection-Angriffe wurden in der Vergangenheit bei vielen aufsehenerregenden Datenlecks eingesetzt. Diese Vorfälle führten häufig zu erheblichem Reputationsverlust und hohen Geldstrafen durch Aufsichtsbehörden.  
In manchen Fällen gelingt es einem Angreifer sogar, eine dauerhafte Hintertür in den Systemen eines Unternehmens zu installieren – was zu einer langfristigen Kompromittierung führt, die über einen längeren Zeitraum unbemerkt bleiben kann.[[9]](#footnote-9)

## Graphische Darstellung

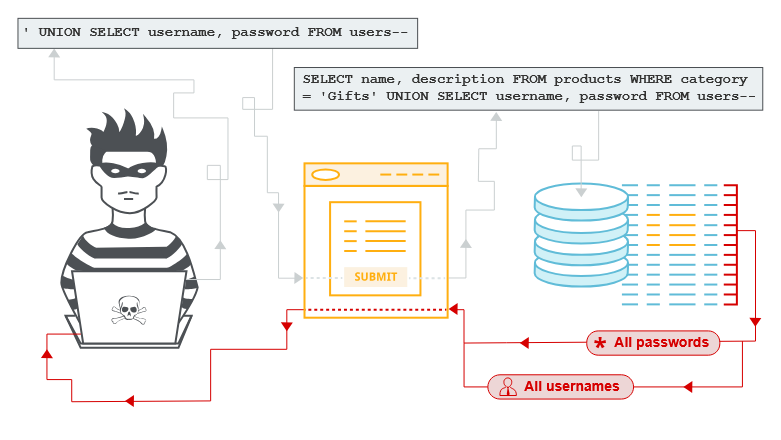


Figure 4 SQL-Injection Funktion <https://portswigger.net/web-security/images/sql-injection.svg>

## 7.3 Code

**Low SQL-Injection Code**

<?php

if (isset($\_REQUEST['Submit'])) {

$id = $\_REQUEST['id']; ***//Unsicherer Input direkt aus der Anfrage***

***// Unsichere SQL-Abfrage (anfällig für SQL Injection)***

$query = "SELECT first\_name, last\_name FROM users WHERE user\_id = '$id';"; ***//SQL Injection möglich***

$result = mysqli\_query($GLOBALS["\_\_\_mysqli\_ston"], $query); ***//Unsicher ausgeführte Query***

while ($row = mysqli\_fetch\_assoc($result)) {

echo "<pre>ID: {$id}<br />First name: {$row['first\_name']}<br />Surname: {$row['last\_name']}</pre>";

}

mysqli\_close($GLOBALS["\_\_\_mysqli\_ston"]);

}

?>

**Warum dieser Code Low Security ist:**

1. **Direkte Eingabe aus dem Nutzerinput ($\_REQUEST['id'])**  
   → Keine Filterung, Validierung oder Desinfektion.
2. **SQL-Abfrage wird direkt zusammengesetzt**  
   → Angreifer kann SQL-Befehle einschleusen wie z. B.:

' OR 1=1 –

1. **Keine Prepared Statements oder Parameterbindung**  
   → Moderne Schutzmethoden fehlen vollständig.

**Impossible SQL Injection Source**

<?php

if (isset($\_GET['Submit'])) {

checkToken($\_REQUEST['user\_token'], $\_SESSION['session\_token'], 'index.php');

$id = $\_GET['id'];

if (is\_numeric($id)) {

$id = intval($id);

// Sichere PDO-Abfrage mit Prepared Statements (MySQL)

$data = $db->prepare('SELECT first\_name, last\_name FROM users WHERE user\_id = (:id) LIMIT 1;');

$data->bindParam(':id', $id, PDO::PARAM\_INT);

$data->execute();

if ($data->rowCount() == 1) {

$row = $data->fetch();

echo "<pre>ID: {$id}<br />First name: {$row['first\_name']}<br />Surname: {$row['last\_name']}</pre>";

}

}

}

generateSessionToken();

?>

**Warum dieser Code sicher ist (SQL Injection Impossible):**

**1. Input-Validierung mit** is\_numeric() und intval()

* Verhindert, dass Angreifer SQL-Code oder Sonderzeichen einschleusen können.
* Nur Zahlen werden akzeptiert → schützt direkt vor manipulativen Inputs.

**2. Prepared Statements mit** bindParam() (PDO)

* Keine direkte Einbindung von Nutzereingaben in SQL-Abfragen.
* Variablenbindung schützt vollständig vor SQL-Injection.

**3. Anti-CSRF Token Prüfung**

* Schutz gegen Cross-Site Request Forgery – zusätzlicher Layer für Formularsicherheit.

## Tools zum Hacken und Testen

 DVWA (Damn Vulnerable Web Application)

 bWAPP (Buggy Web Application)

 SQLMap

 WebGoat (OWASP)

 Burp Suite (Community Edition)

 Hackazon

# SQL Injection (blind)

## Information

Eine **Blind SQL Injection** liegt vor, wenn eine Anwendung anfällig für SQL-Injection ist, aber die HTTP-Antworten keine Ergebnisse der ausgeführten SQL-Abfrage oder detaillierte Fehlermeldungen der Datenbank enthalten.

Viele Techniken, wie zum Beispiel UNION-Angriffe, funktionieren bei **Blind SQL Injection** nicht, weil diese darauf angewiesen sind, die Ergebnisse der eingeschleusten SQL-Abfrage direkt in der Antwort der Anwendung zu sehen. Trotz dieser Einschränkungen lässt sich die **Blind SQL Injection** dennoch ausnutzen, allerdings müssen andere Methoden verwendet werden.

**Blind SQL Injection durch das Auslösen von bedingten Antworten ausnutzen:**

Angenommen, eine Anwendung nutzt Tracking-Cookies, um Analysedaten über die Nutzung zu sammeln. HTTP-Anfragen an die Anwendung enthalten dabei einen Cookie-Header wie diesen:

Cookie: TrackingId=u5YD3PapBcR4lN3e7Tj4

Wenn eine Anfrage mit einem TrackingId-Cookie verarbeitet wird, nutzt die Anwendung eine SQL-Abfrage, um festzustellen, ob es sich dabei um einen bekannten Nutzer handelt:

SELECT TrackingId FROM TrackedUsers WHERE TrackingId = 'u5YD3PapBcR4lN3e7Tj4'

Diese Abfrage ist für SQL-Injection anfällig, doch die Ergebnisse der Abfrage werden nicht an den Benutzer zurückgegeben. Allerdings verhält sich die Anwendung unterschiedlich, je nachdem, ob die Abfrage Ergebnisse liefert oder nicht. Wird eine gültige TrackingId übermittelt, liefert die Abfrage Ergebnisse und es erscheint in der Antwort die Meldung „Welcome back“.

Dieses Verhalten genügt bereits, um die Blind SQL Injection erfolgreich auszunutzen. Man kann Informationen gewinnen, indem man bedingte Reaktionen auslöst, abhängig von einer eingeschleusten SQL-Bedingung.

Um zu verstehen, wie dieser Angriff funktioniert, betrachten wir zwei Anfragen, die jeweils einen der folgenden TrackingId-Cookie-Werte enthalten:

…xyz' AND '1'='1

…xyz' AND '1'='2

* Der erste Wert sorgt dafür, dass die Abfrage Ergebnisse liefert, da die eingeschleuste Bedingung (AND '1'='1) wahr ist. Als Ergebnis wird die Meldung „Welcome back“ angezeigt.
* Der zweite Wert sorgt dafür, dass die Abfrage keine Ergebnisse liefert, da die eingeschleuste Bedingung falsch ist. Die „Welcome back“-Meldung wird deshalb nicht angezeigt.

Auf diese Weise lässt sich die Antwort auf jede einzelne SQL-Bedingung ableiten und Stück für Stück Daten extrahieren.

Angenommen, es existiert eine Tabelle namens Users mit den Spalten Username und Password, und darin ein Nutzer namens Administrator. Das Passwort dieses Nutzers lässt sich durch wiederholte Anfragen, bei denen das Passwort Zeichen für Zeichen geprüft wird, herausfinden.

Dazu startet man mit folgender Eingabe:

xyz' AND SUBSTRING((SELECT Password FROM Users WHERE Username = 'Administrator'), 1, 1) > 'm

Erscheint jetzt die „Welcome back“-Meldung, bedeutet dies, dass die eingeschleuste Bedingung zutrifft – der erste Buchstabe des Passworts liegt im Alphabet hinter „m“.

Anschließend sendet man z. B. folgende Eingabe:

xyz' AND SUBSTRING((SELECT Password FROM Users WHERE Username = 'Administrator'), 1, 1) > 't

Bleibt diesmal die Meldung „Welcome back“ aus, ist klar, dass der erste Buchstabe des Passworts nicht hinter „t“ liegt.

Am Ende sendet man eine Eingabe wie diese, die dann die Meldung „Welcome back“ zurückliefert, wodurch bestätigt wird, dass der erste Buchstabe des Passworts „s“ lautet:

xyz' AND SUBSTRING((SELECT Password FROM Users WHERE Username = 'Administrator'), 1, 1) = 's

Durch systematisches Vorgehen lässt sich so das komplette Passwort für den Benutzer „Administrator“ schrittweise ermitteln.[[10]](#footnote-10)

## Graphische Darstellung

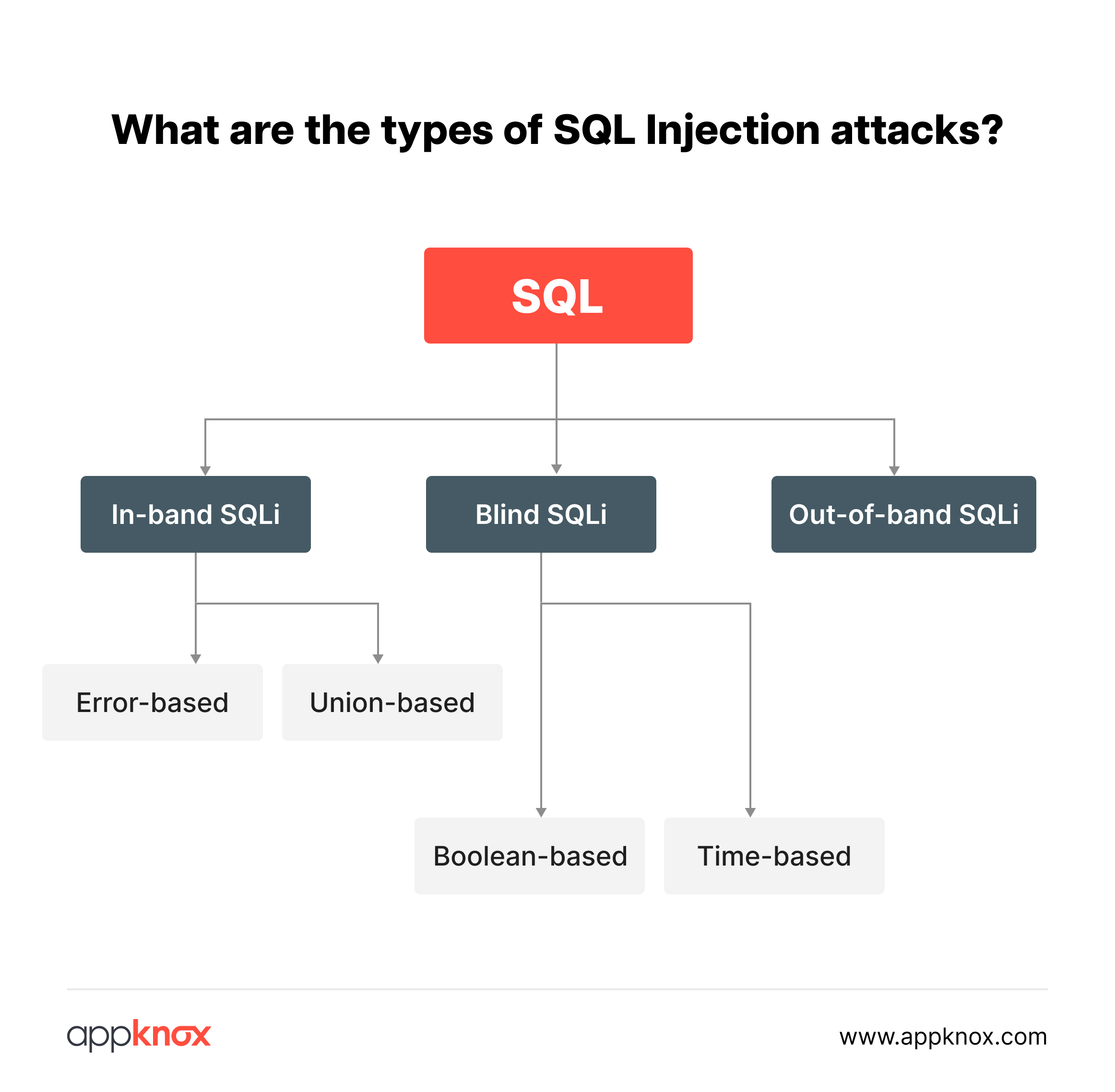


Abbildung 6 Arten von SQL-Injection <https://www.appknox.com/hs-fs/hubfs/What%20are%20the%20types%20of%20SQL%20Injection%20attacks_.png?width=2324&height=2276&name=What%20are%20the%20types%20of%20SQL%20Injection%20attacks_.png>

## Code

**Low SQL Injection (Blind) Source**

<?php

if (isset($\_GET['Submit'])) {

$id = $\_GET['id'];

// Unsichere SQL-Abfrage mit Nutzereingabe

$query = "SELECT first\_name, last\_name FROM users WHERE user\_id = '$id';";

$result = mysqli\_query($GLOBALS["\_\_\_mysqli\_ston"], $query);

// Prüft nur, ob die Abfrage Ergebnisse liefert (Blind SQLi)

if ($result && mysqli\_num\_rows($result) > 0) {

echo '<pre>User ID exists in the database.</pre>';

} else {

header($\_SERVER['SERVER\_PROTOCOL'] . ' 404 Not Found');

echo '<pre>User ID is MISSING from the database.</pre>';

}

mysqli\_close($GLOBALS["\_\_\_mysqli\_ston"]);

}

?>

**Warum dieser Blind SQL Injection-Code unsicher ist:**

**Keine Eingabevalidierung:**

* Direkte Nutzung von $\_GET['id'] ohne Validierung oder Sanitizing erlaubt Angreifern SQL-Code einzuschleusen.

**Direkte SQL-Query-Konkatenation:**

* Die SQL-Abfrage wird direkt durch Verkettung der Nutzereingabe erstellt, was eine Manipulation der SQL-Syntax ermöglicht.

**Blind SQL Injection:**

* Das unterschiedliche Verhalten der Seite („exists“ vs. „missing“) verrät dem Angreifer, ob seine eingeschleuste Bedingung wahr oder falsch ist.

**Impossible SQL Injection (Blind) Source**

<?php

if (isset($\_GET['Submit'])) {

checkToken($\_REQUEST['user\_token'], $\_SESSION['session\_token'], 'index.php');

$id = $\_GET['id'];

$exists = false;

if (is\_numeric($id)) {

$id = intval($id);

// Sichere SQL-Abfrage mit Prepared Statements

$data = $db->prepare('SELECT first\_name FROM users WHERE user\_id = :id LIMIT 1;');

$data->bindParam(':id', $id, PDO::PARAM\_INT);

$data->execute();

$exists = ($data->rowCount() === 1);

}

if ($exists) {

echo '<pre>User ID exists in the database.</pre>';

} else {

header($\_SERVER['SERVER\_PROTOCOL'] . ' 404 Not Found');

echo '<pre>User ID is MISSING from the database.</pre>';

}

}

generateSessionToken();

?>

**Warum dieser Code "Impossible Security" hat (besonders sicher ist):**

**1. Strenge Eingabevalidierung (**is\_numeric + intval**):**

* Nur Zahlen als Eingabe erlaubt.
* Verhindert Einschleusen von SQL-Code zuverlässig.

**2. Prepared Statements mit PDO (bindParam):**

* Nutzer-Eingaben gelangen niemals direkt in die SQL-Abfrage.
* Komplett immun gegen SQL Injection.

**3. CSRF-Schutz durch Token:**

* checkToken() und generateSessionToken() verhindern Cross-Site-Request-Forgery-Angriffe.

## 8.4 Tools zum Hacken und Testen

**SQLMap**  
Automatisiertes, spezialisiertes Tool, besonders gut für Blind SQL Injection geeignet.

**Burp Suite (Community/Pro)**  
Ermöglicht das manuelle und halbautomatische Testen von Blind SQL Injection durch Conditional Responses.

**Havij**  
Einfaches Tool mit GUI für Blind- und Time-Based SQL Injections (Windows).

**SQLNinja**  
Tool speziell für Blind SQL Injection gegen Microsoft SQL Server.

**BBQSQL**  
Open-Source-Framework, speziell für komplexe Blind SQL-Injection-Angriffe.

**DVWA (Damn Vulnerable Web Application)**  
Ideale Übungsplattform, um Blind SQL Injection praktisch zu demonstrieren.

**bWAPP (Buggy Web Application)**  
Lehranwendung mit integrierten Blind-SQL-Injection-Herausforderungen.

# 9 Weak Session IDs

# 10 XSS (DOM)

# 11 XSS (Reflected)

# 12 XSS (Stored)

# 13 CSP Byepass

# 14 JavaScript

# 15 Authorisation Byepass

# 16 Open HTTP Redirect

# 17 Cryptography

# 18 TOOLS

# Abbildungsverzeichnis

[Abbildung 1BruteForce Attack https://www.researchgate.net/publication/379624465/figure/fig2/AS:11431281234681580@1712414089963/Work-architecture-of-Brute-Force-Attack7.ppm 1](#_Toc195200137)

[Abbildung 2 Command Injection https://encrypted-tbn0.gstatic.com/images?q=tbn:ANd9GcT5MDilLllu\_4qBsOfOnh94\_TBFumhVfExXnpK-cXRSaTsYrXnr 6](#_Toc195200138)

[Abbildung 3 CSRF https://encrypted-tbn0.gstatic.com/images?q=tbn:ANd9GcSNaq9pJtukr0lln4EroZ6BOW8noYucd2H4pc1KNhkS1MXflmpP 9](#_Toc195200139)

[Abbildung 4 File-Inclusion https://encrypted-tbn2.gstatic.com/images?q=tbn:ANd9GcSD-W7x1jQSSMl8jkIsfJl3qipeEgxk2fTBA\_D8lkzumw3OlH8w 15](#_Toc195200140)

[Abbildung 5 File Upload https://portswigger.net/web-security/file-upload/images/file-upload-vulnerabilities.jpg 20](#_Toc195200141)

[Abbildung 6 Arten von SQL-Injection https://www.appknox.com/hs-fs/hubfs/What%20are%20the%20types%20of%20SQL%20Injection%20attacks\_.png?width=2324&height=2276&name=What%20are%20the%20types%20of%20SQL%20Injection%20attacks\_.png 35](#_Toc195200142)

1. <https://www.proofpoint.com/de/threat-reference/brute-force-attack> [↑](#footnote-ref-1)
2. <https://www.password-depot.de/de/know-how/brute-force-angriffe.htm> [↑](#footnote-ref-2)
3. <https://owasp.org/www-community/attacks/Command_Injection> [↑](#footnote-ref-3)
4. <https://portswigger.net/web-security/os-command-injection> [↑](#footnote-ref-4)
5. <https://owasp.org/www-community/attacks/csrf> [↑](#footnote-ref-5)
6. <https://www.sumologic.com/glossary/file-inclusion/> [↑](#footnote-ref-6)
7. <https://portswigger.net/web-security/file-upload> [↑](#footnote-ref-7)
8. <https://cspanias.github.io/posts/DVWA-Insecure-CAPTCHA/> [↑](#footnote-ref-8)
9. <https://portswigger.net/web-security/sql-injection> [↑](#footnote-ref-9)
10. <https://portswigger.net/web-security/sql-injection/blind> [↑](#footnote-ref-10)