IT-Sicherheit

Kapitel 5: Applikationssicherheit

Teil 1

- Motivation
- OWASP
- SQL-Injection
- Cross-Site-Scripting (XSS)
- Lösungsansätze für sichere Software

Prof. Dr. Reiner Hüttl, TH Rosenheim, © 2021, 15.03.2021

Worum geht es?

- Was sind die häufigsten Schwachstellen von (Web-)Applikationen?
- Wie funktionieren Angriffe mit SQL-Injection und Cross Site Scripting?
- Welche Maßnahmen gegen Attacken kann ich ergreifen?
- Wie kann ich Sicherheit von Anfang an in die Software Entwicklung integrieren?

Motivation

- Häufige Fehler
  - Applikation wird geplant, entwickelt und getestet ohne Sicherheit zu berücksichtigen
  - Viele Sicherheitslücken sind auf schlechte, fahrlässige und unsichere Programmierung zurückzuführen
  - Es werden Sicherheitstechnologien (als Alibi) eingebaut ohne die tatsächlichen Bedrohungen zu berücksichtigen
  - Sicherheit wird nachträglich als Anhängsel eingebaut

Beispiel: Apple‘s SSL bug: goto fail;

Fehlerhafte Methode:

```
SSLVerifySignedServerKeyExchange(SSLContext *ctx, bool isRsa, SSLBuffer signedParams,
                                 uint8_t *signature, UInt16 signatureLen)
{
```

Fehlerhafte Code-Ausschnitt:

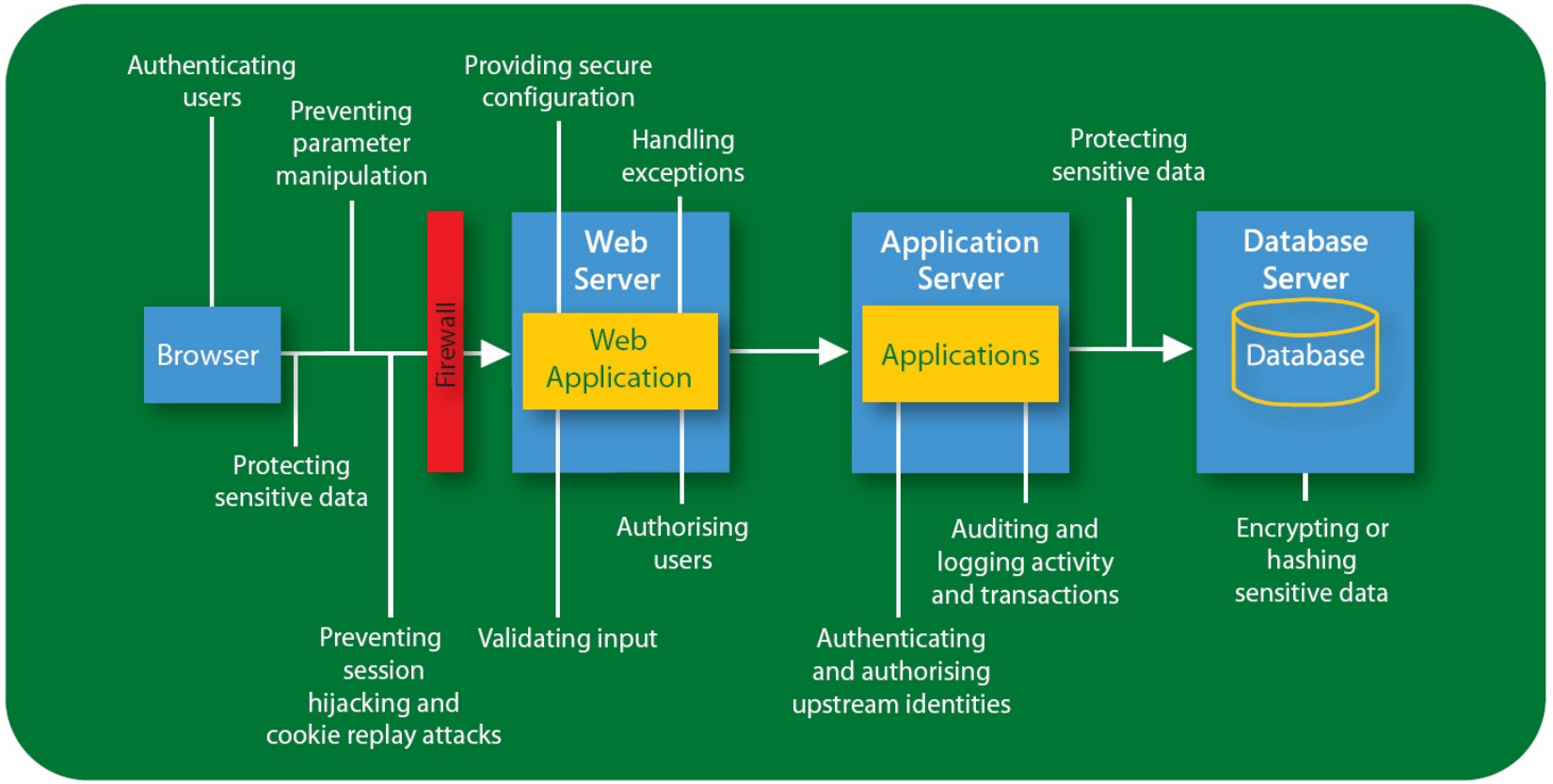
```
    if ((err = ReadyHash(&SSLHashSHA1, &hashCtx)) != 0)
        goto fail;
    if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)
        goto fail;
    if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
        goto fail;
    if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
        goto fail;
        goto fail;
    if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
        goto fail;

        err = sslRawVerify(ctx,
                           ctx->peerPubKey,
                           dataToSign,                  /* plaintext */
                           dataToSignLen,               /* plaintext length */
                           signature,
                           signatureLen);
        if(err) {
                sslErrorLog("SSLDecodeSignedServerKeyExchange: sslRawVerify "
                    "returned %d\n", (int)err);
                goto fail;
        }

fail:
    SSLFreeBuffer(&signedHashes);
    SSLFreeBuffer(&hashCtx);
    return err;
```

Immer **goto fail;**

Wird nie aufgerufen

Immer erfolgreich?
Nicht das was man erwartet!

Top Web Application Security Issues



Aus „Developer Highway Code (The drive für safer coding)“, Microsoft
http://download.microsoft.com/documents/uk/msdn/security/The%20Developer%20Highway%20Code.pdf

Top-Ten Verwundbarkeiten in Web Applikationen

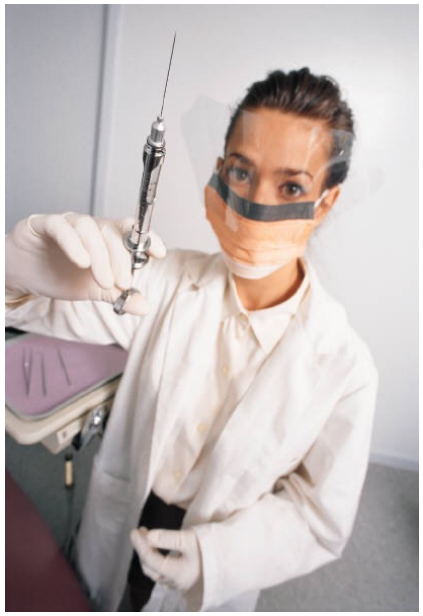
Liste vom Open Web Application Security Project (OWASP)

http://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project

OWASP
The Open Web Application Security Project

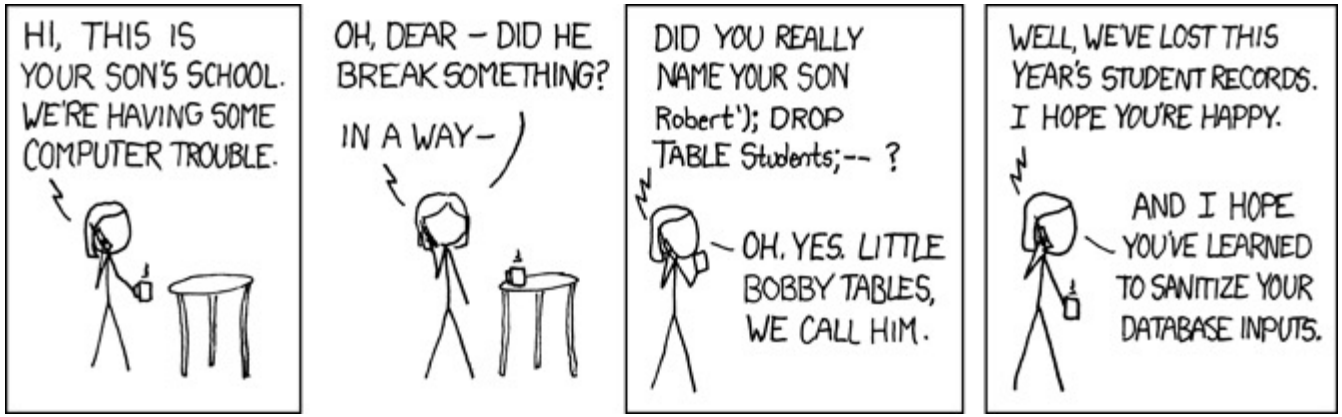
OWASP Top 10 - 2013		OWASP Top 10 - 2017
A1 – Injection	→	A1:2017-Injection
A2 – Broken Authentication and Session Management	→	A2:2017-Broken Authentication
A3 – Cross-Site Scripting (XSS)	↘	A3:2017-Sensitive Data Exposure
A4 – Insecure Direct Object References [Merged+A7]	U	A4:2017-XML External Entities (XXE) [NEW]
A5 – Security Misconfiguration	↘	A5:2017-Broken Access Control [Merged]
A6 – Sensitive Data Exposure	↗	A6:2017-Security Misconfiguration
A7 – Missing Function Level Access Contr [Merged+A4]	U	A7:2017-Cross-Site Scripting (XSS)
A8 – Cross-Site Request Forgery (CSRF)	☒	A8:2017-Insecure Deserialization [NEW, Community]
A9 – Using Components with Known Vulnerabilities	→	A9:2017-Using Components with Known Vulnerabilities
A10 – Unvalidated Redirects and Forwards	☒	A10:2017-Insufficient Logging&Monitoring [NEW,Comm.]

SQL-Injection



- Problem: Dynamisch erstellte Datenbankanfragen werden durch Benutzereingaben manipuliert
- Mögliche Schäden
  - Ausspionieren von Daten
  - Böswilliges Ändern und Löschen von Daten
  - Einschleusen von fremden Code, um Systeme zu schädigen
  - Umgehung von Passwortschutz von Web-Applikationen
- Tritt „nur“ bei Anwendungen auf, die mit SQL-basierten Datenbanken arbeiten

Beispiele für SQL-Injection



http://xkcd.com/327/

SELECT * FROM Usr WHERE UserName = ' **john' –** ' AND Password= ''

SELECT * FROM Usr WHERE UserName = ' **john' OR 'a' = 'b'** AND Password= ''

SELECT * FROM Customer WHERE CustId = **1; DELETE FROM Customer**

SELECT Id, Title, Abstract FROM News
WHERE Category= **1 UNION SELECT 1, UsrName, Passwd FROM Usr**

SQL-Injection: Fehlermeldungen provozieren

http://www.stock.example/fund.asp?id=1+**OR+qwe=1**

Fehlermeldung:
Invalid Column name qwe

SELECT * FROM news WHERE id = **1 HAVING 1=1**

Fehlermeldung:
Attribute news.id must be GROUPED or used in an aggregate function

Lösungen für SQL-Injection (1)

- Sichere APIs verwenden
  - Schirmen den Interpreter ab
  - Bieten parametrisierte Schnittstellen an
- Prepared Statements (Parametrisierte Queries)
  - Führen automatische Escaping von Metazeichen durch
  - Haben zusätzlich bessere Performance bei DB-Anfragen
- Beispiel mit JDBC

```
PreparedStatement ps = conn.prepareStatement("UPDATE news SET
title=? WHERE id = ?;
…
ps.setString(1, title);
ps.setInt(2, id);
int rowCount = ps.executeUpdate( );
```

Lösungen für SQL-Injection (2)

- Administrative Maßnahmen
  - Minimale Rechte für SQL-Nutzer
  - Keine Systemaccounts für Datenbanknutzer
  - Web Application Firewall installieren
  - White List Input Validation

- SQL Kommandos in Queries verwenden die große Datenmengen als Resultat verhindern
  - Z.B. LIMIT

Lösungen für SQL-Injection (3)

- Neutralisierung von SQL-Metazeichen („Daten waschen“)
  - Wichtig: alle Metazeichen identifizieren
  - Beispiele
    - einfache Anführungszeichen verdoppeln
    - Backslash verdoppeln
    - resultierende Zeichenketten in neue Anführungszeichen kapseln
    - Funktionen die numerische Eingaben überprüfen
  - Beispiel: PHP-Funktion zur Validierung

```
function SQLInteger($s) {
    return (int) (trim($s) + 0);
}
```

  - Beispiel: OWASP Enterprise Security API (http://www.owasp.org/index.php/ESAPI)
    - API mit Databank Encoder für Oracle und MySQL in verschiedenen Programmiersprachen

Ähnliche Probleme

- **Shell-Command Injection**
  - Shell Kommandos werden aus Web-Applikation mit dynamischen Parametern aufgerufen
  - Beispiel in Perl

```
$username = $form{“username”};
print ‘finger $username’;
```

Aufruf: finger qwe; rm –rf /

  - Lösungen: ohne Shell Kommandos auskommen
    - Externe Programme direkt aufrufen (system ,exec)
    - Programmierung der Funktionalität (z.B. mail)
    - Metazeichen behandeln
    - Benutzereingaben in Befehlsargumente verhindern

OWASP Nr. 1: Injection

- Anwendung benutzt zur Laufzeit interpretierten Code
  - Interpretierter Code wird dynamisch zur Laufzeit erstellt/modifiziert
  - Benutzereingaben fließen direkt in erstellten Code
  - Beispiele: SQL, Shell, LDAP, XML, …
- Schutzmaßnahmen gegen Attacken
  - **Input Sanitization**: escaping oder entfernen der Metazeichen
  - **Blacklisting** : Nur Unerwünschte Zeichen in Eingabe werden escaped / entfernt
  - **Whitelisting** : Nur Eingaben mit erlaubten Zeichen werden zugelassen
  - Einsatz von Framework die Schutz übernehmen
    z.B. LdapQueryBuilder in Spring-LDAP
  - Einsatz von **Web Application Firewalls** (**WAF**)

OWASP Nr. 7: Cross-Site-Scripting (XSS)

- Problem
  - Von einem Angreifer fabrizierte HTML-Konstrukte werden über eine Web-Anwendung an die Browser anderer Benutzer übergeben
  - Javascript wird ohne Rückfragen im Browser ausgeführt
  - Benutzereingaben einer Webseite werden ungefiltert als Querystring weitergereicht und können gefährliche Tags enthalten
  - XSS ist ein **Metazeichen- und ein Ausgabeproblem**
- Mögliche Folgen:
  - Ausführen von böswilligem Script im Browser eines Clients
  - Ausspionieren von Benutzern durch Stehlen von Cookies
- Tritt besonders bei Webanwendungen auf in denen User den Content liefern
  - Web-Frontends für E-Mail-Systeme und Newsgroups
  - Webbasierte Diskussionsforen
  - Social Networks
  - Kann kombiniert werden mit „Social Engineering“

Varianten von XSS

- Klassisch (2005)
  - **Persistent (Stored) XSS**: die Anwendung speichert ungeprüfte Benutzereingaben
  - **Reflected XSS**: ungeprüfte Benutzereingaben landen direkt wieder im HTML-Output
  - **DOM Based XSS**: Quelle der Daten ist im DOM, Daten landen wieder im DOM
- Modern (2012)

	Where untrusted data is used		
Data Persistence	**XSS**	**Server**	**Client**
	Stored	Stored Server XSS	Stored Client XSS
	Reflected	Reflected Server XSS	Reflected Client XSS

- ❑ **DOM-Based XSS is a subset of Client XSS (where the data source is from the client only)**
- ❑ **Stored vs. Reflected only affects the likelihood of successful attack, not nature of vulnerability or defense**

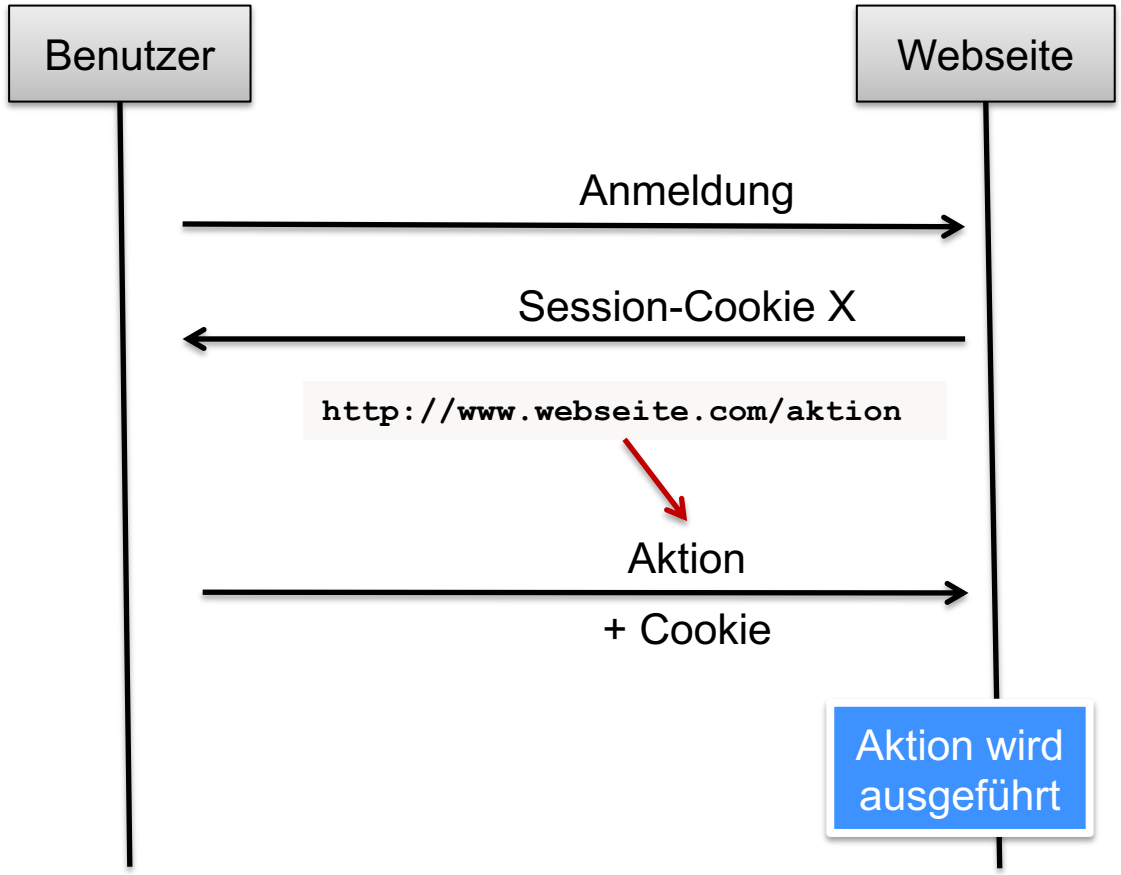
https://owasp.org/www-community/Types_of_Cross-Site_Scripting

Maßnahmen für XSS

- Server XSS Defense
  - **Kontext-Sensitive Ausgabe-Kodierung auf dem Server**
  - **HTML-Kodierung** der nicht vertrauenswürdiger Daten vor dem Einfügen in den HTML Output (HTML-Body, Attribute, JavaScript, CSS, URL)
  - Aber reines **Encoding** reicht meist nicht aus, da Daten zwischen <script> Tags, in event-Handler, in CSS oder in einer URL immer noch XSS Attacken beinhalten können
  - Deshalb Einsatz von Security Encoding Libraries (z.B. Microsoft Anti-Cross Site Scripting Library, OWASP Java Encoder Project)
  - Verwendung von Frameworks die XSS by Design maskieren (z.B. Ruby on Rails, React JS)
  - Trennung von nicht vertrauenswürdigen Daten von aktiven Browserinhalten
- Client XSS Defense
  - Verwendung von **sicheren JavaScript APIs**

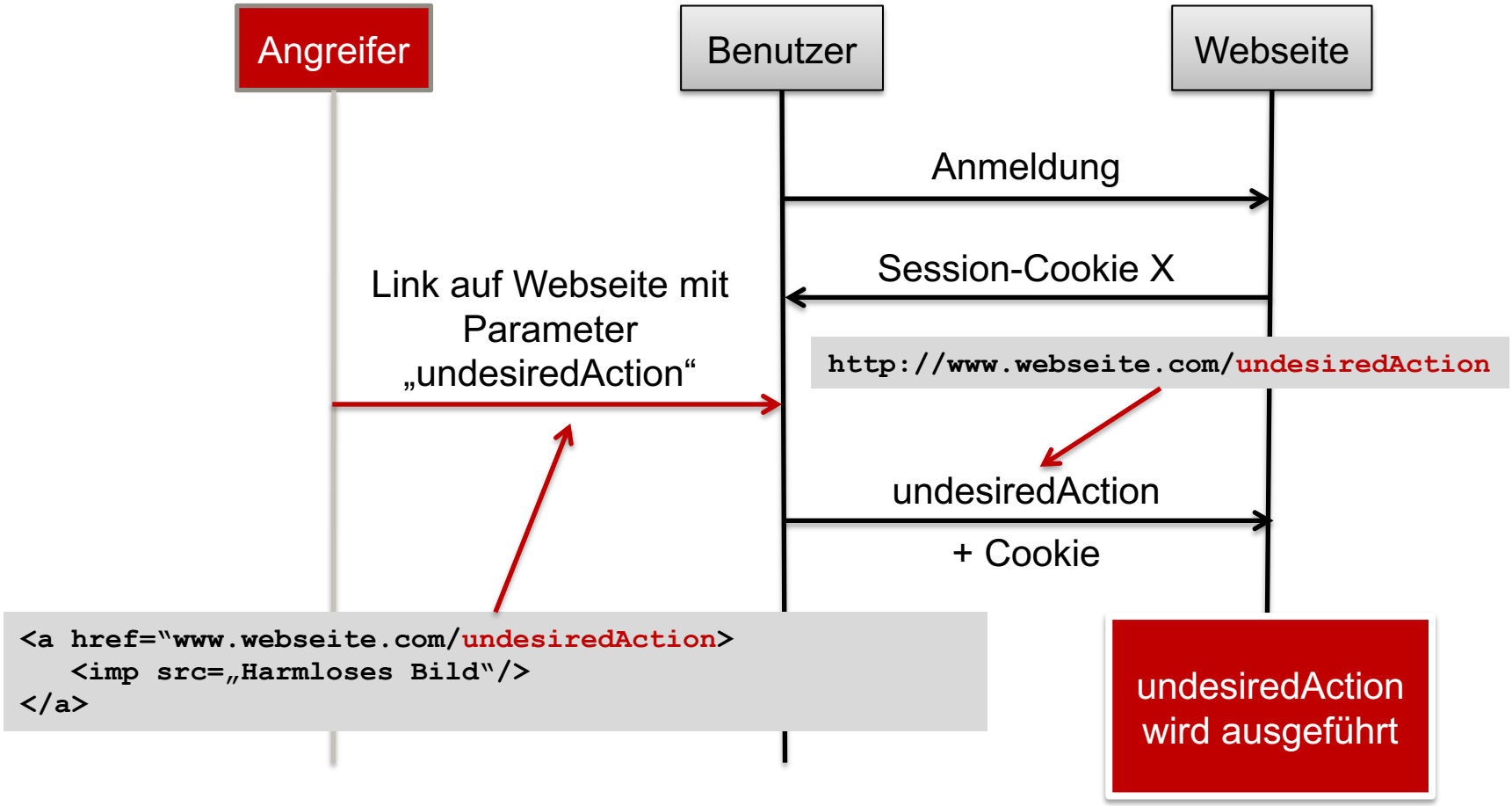
OWASP 2013 Nr. 8: Cross Site Request Forgery (XSRF)

Schritt 1: Der Benutzer meldet sich an einer Webseite an



Cross Site Request Forgery (XSRF)

Schritt 2: Der Angreifer führt auf der Webseite mit der Identität des Opfers eine bösartige Aktion aus



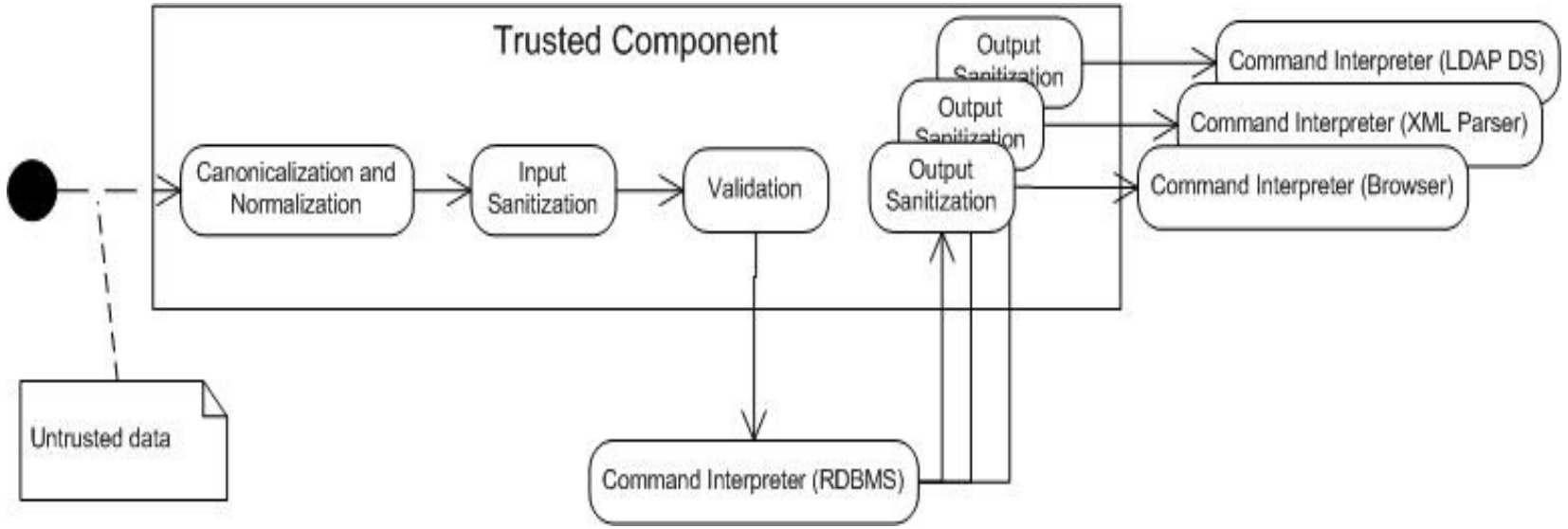
Cross Site Request Forgery (XSRF) - Schutzmaßnahmen

- Empfohlene Schutzmaßnahme: **Synchronizer Token Pattern**
  - Auf dem Server wird ein „Challenge“ Token generiert
  - Dieses wird in die ausgelieferte Seite eingebettet
  - Request von sensitiven Aktionen muss „Challenge“ Token enthalten
  - „Challenge“ Token im Request muss auf dem Server vor Ausführen der Aktion geprüft werden

```
<form action="/transfer.do" method="post">
        <input type="hidden" name="CSRFToken"
          value="OWY4NmQwODE4ODRjN2Q2NWEwYzTVhWQwMTVhjJiMjMTViMGYwMGEwOA==">
        ...
</form>
```

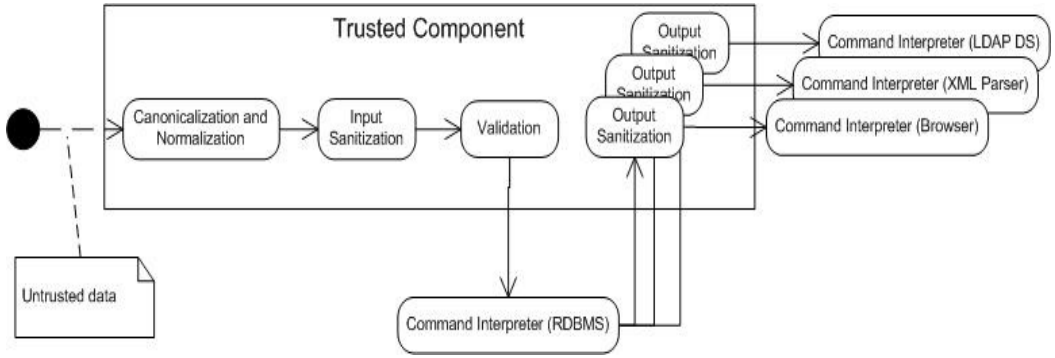
Sichere Programmierung mit Komponenten

- Sichere Programmierung hilft gegen die Schwachstellen der OWASP und all die anderen Verwundbarkeiten von Applikationen
- Eine durchdachte Zerlegung in Komponenten unter Berücksichtigung der Sicherheit erleichtert die sichere Programmierung
- Sichere Komponenten haben Trust Boundaries.



Vertrauenswürdige Komponenten

- Canonicalization & Normalization
  - Reduktion auf die einfachste, standardisierte Form der Darstellung



- Sanitization
  - Sicherstellen das übergebene Daten den Anforderungen der Drittkomponente entsprechen
  - Verhindern von Information Disclosure and Leakage

- Validation
  - Überprüfung das Eingaben den erwarteten Muster entsprechen
  - Erwartete Typen
  - Anforderungen an numerische Eingaben

Kanonisierung

- Problem:
  - Beim Vergeben eines Namens gibt es oft mehrere Möglichkeiten
  - Angreifer können Code ausnutzen, der Entscheidungen anhand von Dateinamen oder URLs trifft

- Maßnahme: Kanonisierung von Dateinamen (nur lange Dateinamen, Punkt am Ende weg, absoluter Pfad) und URLs

- Beispiele für Probleme bei URLs:
  - ROSE == R%6fse ???
  - http://www%2emicrosoft%2ecom%2ftechnet%2fsecurity
  - http://172.43.122.12 == http://2888530444
    Dotless-IP: a.b.c.d = (a * 256^3) + (b* 256^2) + (c * 256^1) + (d * 256^0)

- Beispiele für potentielle Probleme bei Dateinamen:
  - Alte Dateinamensformate (Fiscal04Budget.xls == FISCAL~1.xls)
  - Punkt am Ende vom Dateiname zulässig, wird aber automatische entfernt
  - Absolute und relative Dateinamen
  - Groß- und Kleinschreibung

Validierung von Benutzereingaben

- Motivation: „Falsche“ Eingaben können Programme schadhaften Verhalten verleiten
- Identifizieren alle Quellen von Eingaben einer Web-Anwendung
  - Alle URL-Parameter
  - Mit POST übermittelte Daten aus Texteingaben, Kontrollfelder, Optionsfelder, Auswahllisten, Submit-Button, versteckte Felder
    ACHTUNG: auch vordefinierte Werte können manipuliert werden
  - Daten aus http-Header und Cookies, Auswahllisten, Buttons
  - Eingaben aus anderen Quellen (DB-Tabellen, Dateien)
- Ziel der Validierung
  - Sicherstellung, dass Daten das erwartete Format aufweisen

Validierung (1)

- Domänentypen (fachliche Datentypen) bilden
  (Bsp: E-Mail, Konto, Datum, Kunden-ID)
- Achte darauf, dass alle Eingaben identifiziert und validiert werden
  - Validierung vor jeder anderen Aufgabe
  - Autorisierung zusammen mit Validierung durchführen
- Schreibe Funktionen die Validierung durchführen
- Prüfe Länge, Bereich (z.B. numerisch und nicht negativ) , Format und Bereich
- Whitelisting statt Blacklisting zum Filtern benutzen
  - **Whitelisting** lässt nur Daten zu von denen man glaubt ,dass sie harmlos sind
  - **Blacklisting** lässt alles zu was nicht explizit verboten ist
- Clientseitige Validierung nie als einzige Basis zur Entscheidung verwenden

Validierung (2)

- Verwendung von Daten-Indirektion
  - Wichtige Daten so weit wie möglich am Server halten
  - Ankommende Daten sind nicht die Zieldaten sondern nur zur Suche der Zieldaten geeignet
  - Bsp: Kontonummer, Preis einer Ware über Kunden-Nr oder Artikel-Nr referenzieren
- Überprüfe auch Servererzeugte Eingaben an Subsysteme
  - Identifiziere Zeichen die in einem Subsystem als Metazeichen gelten (s. SQL-Injection, Shell-Command-Injection)
  - Behandle die Metazeichen bevor Daten an Subsysteme weitergegeben werden
  - Schütze Eingaben an Subsysteme durch kryptografische Hash-Funktionen (MAC) oder Verschlüsselung

Maßnahmen für Metazeichenprobleme

- Metazeichen
  - Zeichen, die innerhalb eines bestimmten Kontext nicht für sich selbst stehen, sondern eine besondere Bedeutung haben
  - Bei der Übergabe von Daten an ein Subsystem wandeln sie sich von einem Textzeichen in ein Steuerzeichen
- Maßnahmen
  - Escape Zeichen einfügen z.B /, wenn Metazeichen auch als normales Zeichen Sinn ergibt
  - Sonst: Meta-Zeichen entfernen
  - Subsysteme zur Interpretation von Metazeichen verwenden (z.B. Prepared Statements, DOM)
  - Kapselung der Kommunikation mit anderen Systemen
  - Berechtigungen in Subsystemen minimal halten
  - Eingabevalidierung
  - Gestaffelte Abwehr: Wenn ein Sicherheitsmechanismus versagt sollte ein anderer das Problem behandeln
    Bsp: Restriktive Administration einer DB

Verwende Reguläre Ausdrücke zur Validierung

- Ein regulärer Ausdruck ist eine Zeichenkette zur Beschreibung einer Menge von Zeichenketten (Details siehe Anhang)
- Einsatz in der IT-Sicherheit: Validierung von Eingabe und Ausgabe
- Beispiel: regulärer Ausdruck für Dateinamen:
  - ^[cd] (\\ \w+) + \\ \w {1,32} \. (txt | jpg | gif)$
  - Zulässige Datei: d:\mydir\a\myfile.jpg
- Viele Implementierungen in verschiedenen Programmiersprachen verfügbar
  - Java: Klassen Pattern, Matcher
  - PHP: Funktionen ereg, eregi
  - .Net: Klassen Regex, Match
- Reguläre Ausdrücke sind nützlich aber auch fehleranfällig
  → wenn vorhanden ist es besser bewährte APIs zu verwenden

HTML-Kodierung

- Bestimmte HTML-Metazeichen werden auf äquivalente Zeichen abgebildet
  - Jedes & auf &amp
  - Jedes „ auf &quot
  - Jedes < auf &lt
  - Jedes > auf &gt
  - Einfache Anführungszeichen auf &#39
- Es gibt Implementierungen in verschiedenen Web-Programmiersprachen
  - htmlspecialcharacter in PHP
  - HttpServerUtility.HTMLEncode in ASP.Net
- HTML-Kodierung bewirkt, dass der Browser Daten anzeigt wie sie geschrieben wurden und nicht als Tag-Kennzeichen interpretiert