esse

ESSE Einführung in Security – VO 04: Sicherheit in der Softwareentwicklung

Florian Fankhauser, Christian Schanes

INSO – Industrial Software
Institut für Information Systems Engineering | Fakultät für Informatik | Technische Universität Wien

Agenda

Grundlagen

Sicherheit in der Analysephase

Sicherheit in der Entwurfsphase

Sicherheit in der Implementierung

Sicherheit in der Testphase

Sicherheit in der Betriebsphase

Referenzen, weiterführende Literatur

Zusammenfassung

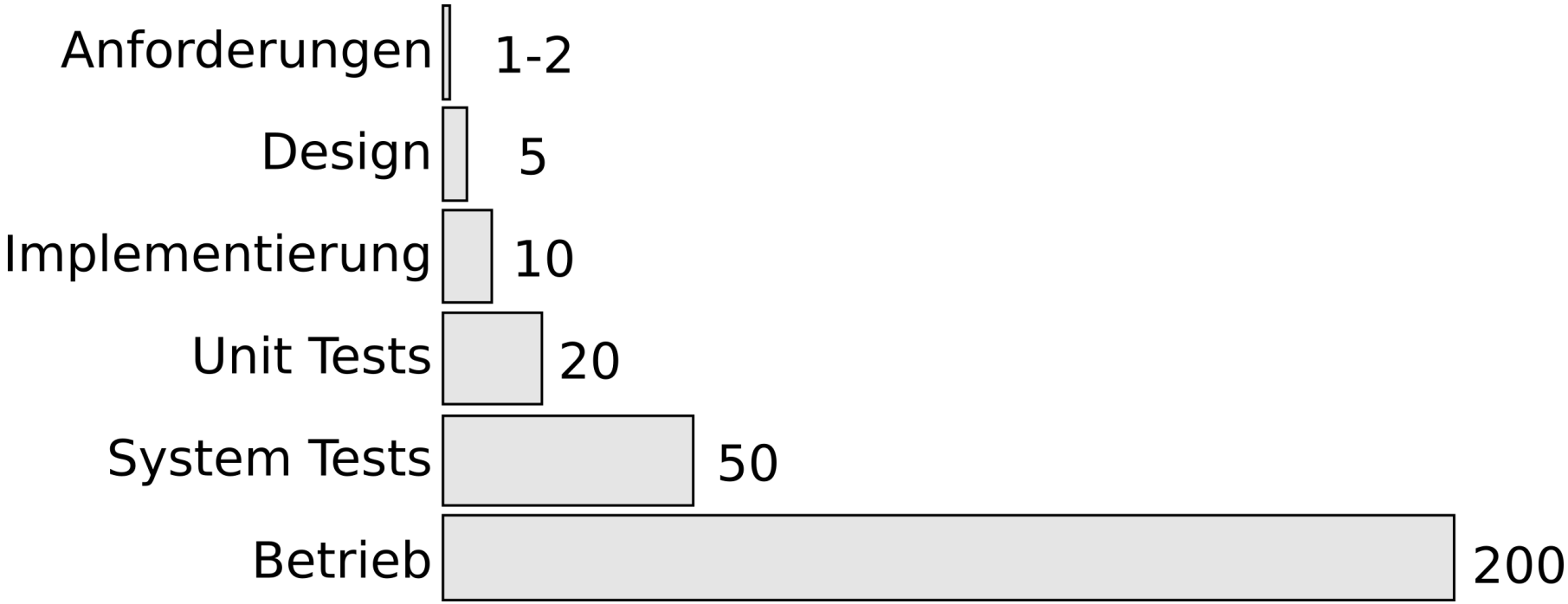
Softwareentwicklungsprozess

Als Software Development Lifecycle wird der gesamte Prozess der Softwareentwicklung bezeichnet. Die weitere Betrachtung erfolgt auf Basis folgender Entwicklungsphasen:

- Analyse/Anforderungen
- Entwurf
- Implementierung
- Test
- Betrieb

Sicherheit ist kein Feature – „Sicherheit ist ein Prozess“ (Bruce Schneier)
→ es ist nicht ausreichend Sicherheit in nur einem der Schritte zu betrachten (z.B. Verwendung einer Firewall)

Relative Kosten von Softwarefehlern



Z.B. wenn die Fehlerbehebung in der Phase der Anforderungen 1 EUR kostet, kann das 50 EUR in der Phase der System Tests kosten

(Vergleiche Stuart R. Faulk, 1995)

Softwareentwicklungs-Modelle für sichere Software

- Vorgehensmodell mit Berücksichtigung von Sicherheit
- Definition von Aktivitäten und Verantwortlichkeiten (z.B. Bedrohungsanalysen, Schulungen, . . .)
- Integration in vorhandene Vorgehensmodelle erforderlich
- Vertreter:
  - Comprehensive Lightweight Application Security Process (CLASP)
  - Microsoft Security Development Lifecycle (SDL)
  - Software Assurance Maturity Model (SAMM)

Sicherheit in der Analysephase

Analyse

Als Analyse wird die Erhebung und Aufbereitung der Sicherheitsanforderungen für das zu entwickelnde System bezeichnet.

- Funktionale und nicht-funktionale Sicherheitsanforderungen
- Unterschiedliche/widersprüchliche Anforderungen von Stakeholdern (am System beteiligte Personen)
- Weitere Anforderungen durch Normen, Gesetze, Standards
- Bei sicherheitskritischen Systemen oft Evaluierungen:
  - Berücksichtigung der Evaluierungsanforderungen
  - Z.B. Evaluierungen anhand von Protection Profiles
- Sicherheit wird als selbstverständlich angesehen und nicht explizit von den Stakeholdern gefordert

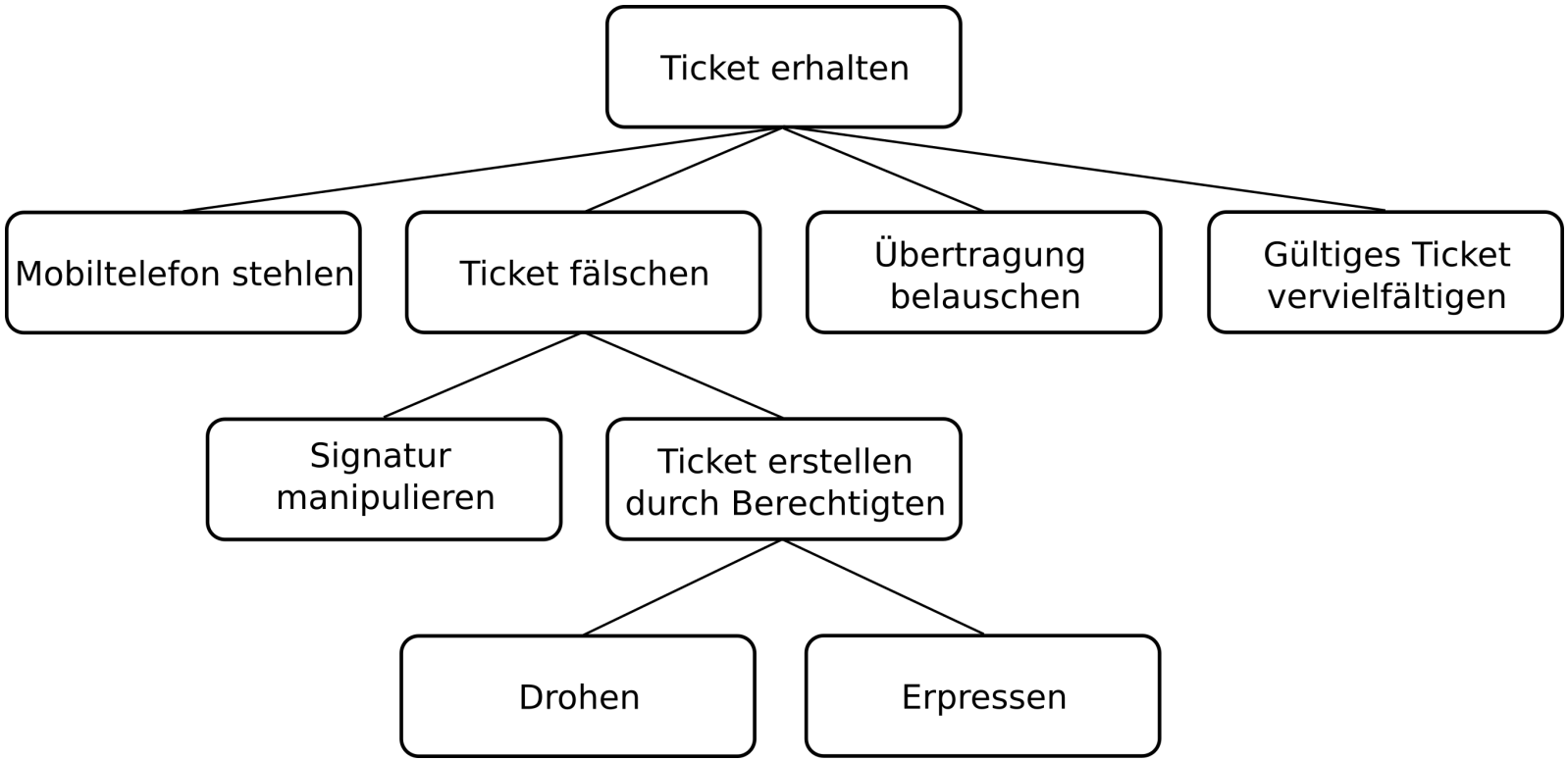
Durchführung der Analyse

- Ermittlung der Bedrohungen und Risiken
- Vollständige Erfassung der Angriffsoberfläche
- Festlegung der Schutzziele für die verarbeiteten Daten
- Systematisches Vorgehen zur vollständigen Erfassung erforderlich
- Werkzeuge zur Durchführung vorhanden, z.B. Angriffsbäume oder Threat Modelling

Angriffsbäume/Attack Trees

- Technik zur Dokumentation von Bedrohungen
- Identifizierung möglicher Bedrohungen mit iterativer Detaillierung
- Vorgestellt von Bruce Schneier
- Darstellung erfolgt in Baumstruktur
- Knoten/Kanten können zur Bewertung der Eintrittswahrscheinlichkeit einer Bedrohung verwendet werden
- Erstellung ist manuell oder automatisch möglich
- Bei größeren Systemen werden Angriffsbäume schnell unübersichtlich. Z.B. Angriffsbaum für Netzwerk

Beispiel für Angriffsbäume/Attack Trees



(Vergleiche Fankhauser, Schanes, Brem in Softwaretechnik, 2009)

Threat Modeling

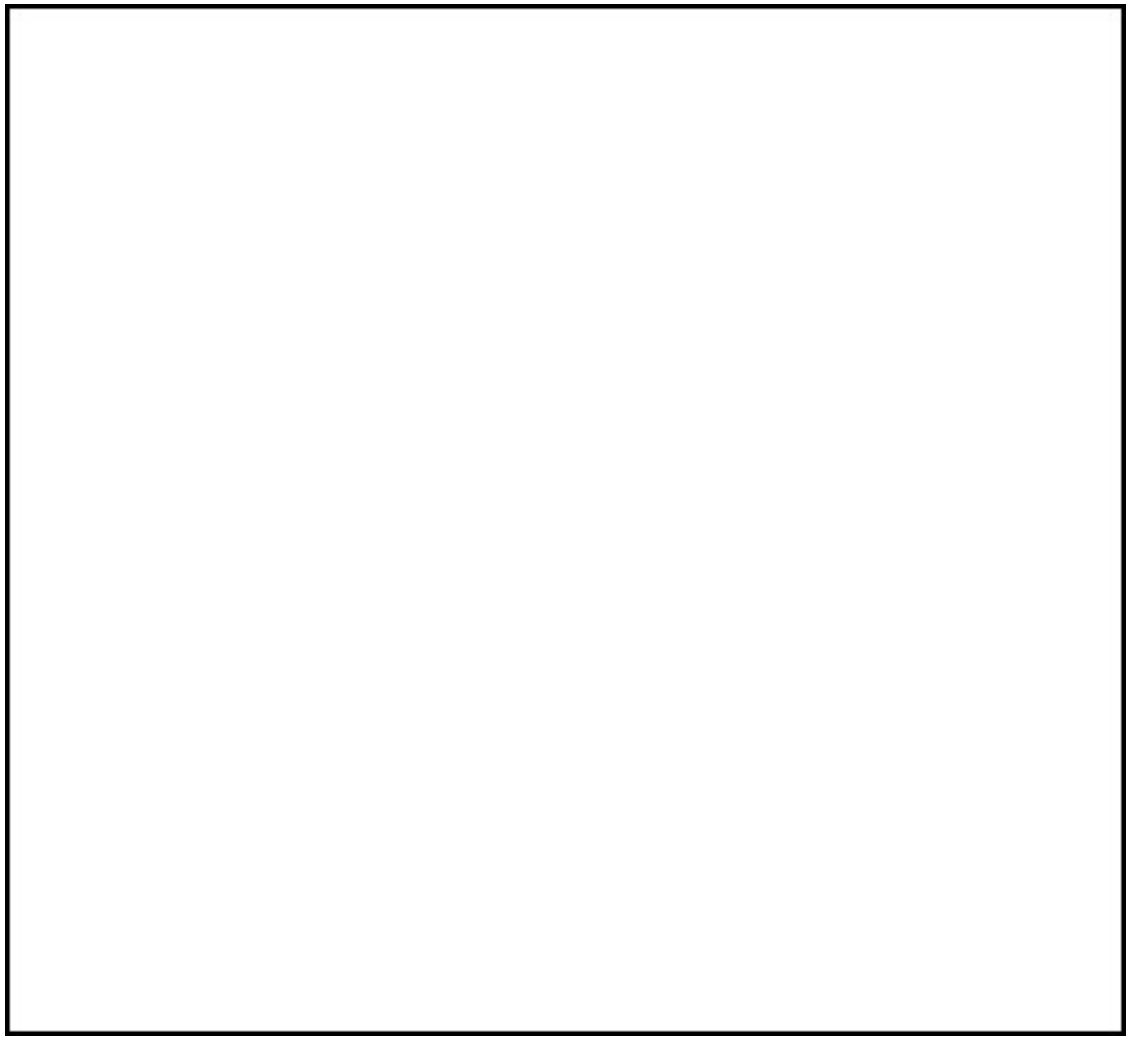
- Methode von Microsoft zur Modellierung von Bedrohungen in der Software
- Frühzeitige Beschäftigung mit den möglichen Bedrohungen
- Schrittweise Modellierung und Detaillierung
- Bedrohungen identifizieren – Denken wie ein Angreifer/eine Angreiferin – und nicht: „*But no one would ever do that!*“
- Wird derzeit von Microsoft eingesetzt (Teil des SDL – Security Development Lifecycle)

Sicherheit in der Entwurfsphase

Entwurf des Systems

- Abbildung von Sicherheitsanforderungen durch den Entwurf
- Robuste Architektur bildet die Basis für ein sicheres System
- Berücksichtigung von Best Practices
  - *Sicherheitsmuster (Security Patterns)* stellen für immer wieder auftretende Herausforderungen im Sicherheitsbereich Entwurfslösungen zur Verfügung
  - *Angriffsmuster (Attack Patterns)* Beschreibung von Angriffen inklusive der Vorbedingungen welche erfüllt werden müssen. Ableitung von Sicherheitsaspekten für das eigene System möglich
  - *UML* Spezielle Erweiterungen zur Modellierung von Sicherheitsanforderungen vorhanden z.B. UMLsec

Architektur



(Vergleiche CERT, https://www.securecoding.cert.org/)

8 Design Principles – (i)

- Einfachheit der Schutzmechanismen (Economy of Mechanism)
  - Keep it simple
  - Design komplex $\rightarrow$ Fehleranfällig
  - Kleine, einfache Codeteile können besser getestet werden
- Fail-Safe Defaults
  - In der Standard-Einstellung in einem sicheren Zustand sein und
  - Im Fehlerfall auf einen sicheren Zustand zurück gehen, z.B.
    - Zugriff standardmäßig ablehnen
    - Input-Validierung durch Allowlists (Whitelists)

(Vergleiche Jerome H. Saltzer und Michael D. Schroeder, 1975)

8 Design Principles – (ii)

- Vollständige Berechtigungsprüfung (Complete Mediation)
  - Sämtliche Zugriffe über einen Mediator prüfen
  - Java Security Manager prüft sämtliche Zugriffe auf potentiell gefährliche Funktionen wie zum Beispiel Dateizugriffe
- Offenes Design (Open Design)
  - Die Sicherheit eines Systems darf nicht auf der Geheimhaltung des Designs oder der Implementierung beruhen
  - Kerckhoffs' Prinzip: Sicherheit kryptographischer Verfahren basiert nur auf der Geheimhaltung des Schlüssels
  - Und nicht: Security by Obscurity

(Vergleiche Jerome H. Saltzer und Michael D. Schroeder, 1975)

8 Design Principles – (iii)

- 4-Augen-Prinzip (Separation of Privilege)
  - Zugriff auf Informationen durch z.B. 2 kryptographische Schlüssel geschützt
- Minimum an Rechten (Least Privilege)
  - Nur erforderliche Rechte verwenden/vergeben.
  - Beispiel: Kompromittierung eines Accounts mit niedrigen Rechten ermöglicht keinen Zugriff auf weitere Systemteile

(Vergleiche Jerome H. Saltzer und Michael D. Schroeder, 1975)

8 Design Principles – (iv)

- Minimum an gemeinsamen Mechanismen (Least Common Mechanism)
  - Abhängigkeiten minimieren
  - Gemeinsame Variablen/Dateien vermeiden
- Psychologische Akzeptanz (Psychological Acceptability)
  - Usability
  - Einsatz von Sicherheitsmechanismen kann durch einfach bedienbare Benutzerschnittstellen begünstigt werden
  - Reduzierung der Sicherheit, wenn Sicherheitssysteme nicht gut benutzbar.

(Vergleiche Jerome H. Saltzer und Michael D. Schroeder, 1975)

Sicherheit in der Implementierung

Sichere Architektur vs. Sichere Programmierung

- Eine sichere Softwarearchitektur kann durch Programmierfehler unsicher werden
  - Fehlerhafte Prüfung von Zertifikaten
- Eine Software ohne Schwachstellen durch Programmierfehler kann Schwächen durch die Architektur aufweisen
  - Fehlerfreie telnet Implementierung ist durch unverschlüsselte Kommunikation angreifbar

Sichere Programmierung

- Auflistungen der häufigsten Sicherheitsfehler, z.B.
  - OWASP: Top Ten Web Vulnerabilities
  - CWE/SANS: TOP 25 Most Dangerous Programming Errors
- Programmierrichtlinien
  - SEI CERT Secure Coding Standards
  - Secure Coding Guidelines for the Java Programming Language
- Sicherheitskonzepte von Programmiersprachen
  - Automatische Längenprüfung und Speicherverwaltung, ...
  - Kenntnis über vorhandene Sicherheitskonzepte
  - Korrekte Anwendung der vorhandenen Sicherheitskonzepte
- Code Review

Penetrate and Patch

Ansatz „Penetrate and Patch“ für sichere Software nicht ausreichend!

- Sicherheitsfehler werden nur im Betrieb gefunden und mit Patches behoben
- Wird von einigen Unternehmen noch immer betrieben
- Schwachstellen, welche von Angreifer:innen nicht gemeldet werden, können auch nicht behoben werden
- Patches beheben nicht immer die eigentliche Ursache sondern oft nur Symptome
- Zeitfenster für Angriffe bleibt vorhanden, bis alle Systeme gepatcht werden. Bei einigen Systemen werden Patches nie eingespielt.
- Aus Patches können Angreifer:innen Details über die Schwachstelle ableiten

(Vergleiche John Viega und Gary McGraw, 2002)

SQL Injection Angriff

(Vergleiche `https://xkcd.com/327/`)

SQL Injection – Fehlerhafter Code

Listing 1: action.php

```
$pw=$HTTP_GET_VARS["pw"];
...
$result=mysql_query("SELECT * FROM users
                     WHERE password='$pw'");
```

Listing 2: index.html

```
<form method="get" name="reg" action="action.php">
  Login: <input type="text" name="pw">
  <input type="submit" name="do" value="Login">
</form>
```

SQL Injection – Grundlagen

Eine Manipulation des SQL Statements durch eine/n Angreifer:in ist möglich.

- Einfaches Hochkomma: `te'st`. Fehlermeldung der Datenbank wird angezeigt. Schwachstelle erkennbar. Zusätzlich werden u.U. Informationen zur Datenbank und zum Aufbau des SQL Statements ausgegeben.
- Verwendung von Tautologien wie `' or '1'='1` was zu `SELECT * FROM users WHERE password='' or '1'='1'` führt
- Auslesen von Daten aus weiteren Tabellen mittels UNION oder Sub Queries
  - `' UNION SELECT * FROM system`

SQL Injection – GIS

SQL Injection – Gegenmaßnahmen

- *Jede Eingabe* muss *validiert werden.* Es ist nicht ausreichend nur Formularfelder zu validieren. Injection ist auch zum Beispiel durch HTTP Header-Felder möglich.
- Ausfiltern von speziellen Datenbankzeichen wie zum Beispiel einfache Hochkomma → Denylist (Blacklist) Filtering
- Zu bevorzugen ist eine Prüfung der Eingaben auf gültige Zeichen → Allowlist (Whitelist) Filtering
- Precompiled Statements verwenden. Das Statement wird in der Datenbank mit Platzhaltern kompiliert. Spätere Manipulation des Statements durch Eingaben nicht mehr möglich.
- Minimierung der erforderlichen Rechte in der Datenbank → *Principle of Least Privilege*. Dies verhindert SQL Injection nicht, verkleinert jedoch den Schaden durch einen Angriff.

Command Injection

Wie bei SQL Injection wird eine Eingabe ohne ausreichender Validierung für einen Systemaufruf verwendet. Dabei besteht die Möglichkeit den Aufruf zu manipulieren und beliebige Programme auszuführen.

site.php

```
$search = $HTTP_GET_VARS["prod"];
$cmd = "grep $search products.csv";
exec($cmd, $search_result, $errnr);
```

Durch `site.php?prod=* file; cat /etc/passwd; cat` wird das File `/etc/passwd` ausgegeben.

Command Injection – Gegenmaßnahmen

- Validierung der Eingaben!
- Verwendung der Funktionen von Programmiersprachen zum Lesen von Dateien
- Beschränkung der Zugriffsrechte auf Dateien durch Rechtevergabe oder Verwendung von chroot Umgebungen
- Minimierung der möglichen ausführbaren Dateien
- Berücksichtigung der System-Konfigurationen (z.B. sichere PHP Konfiguration)

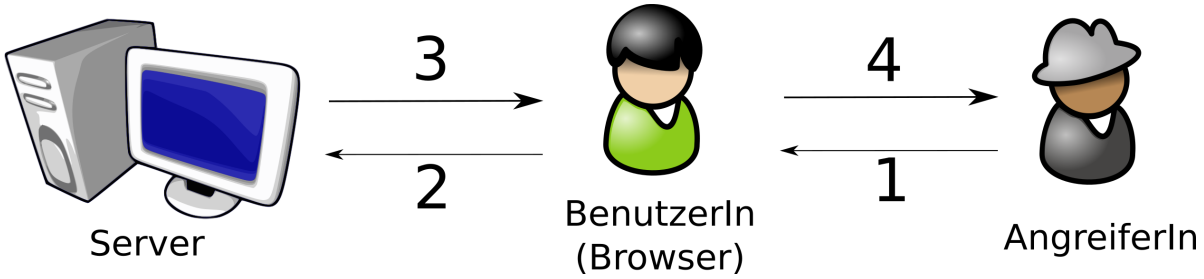
Cross-Site-Scripting (XSS) – Beschreibung

- Unvalidierte Benutzer:inneneingaben werden von der Software entgegen genommen und ausgegeben
  `<script>alert("xss");</script>`
- Ausführung des Codes durch vertrauenswürdigen Server; Zertifikate und Verschlüsselung der Kommunikation zwischen Server und Client bieten keinen Schutz
- Eingeschleuster Code wird direkt von der trusted Domäne ausgeführt
- Arten:
  - Reflected XSS
  - Stored XSS

Cross-Site-Scripting (XSS) – Gefahren und Maßnahmen

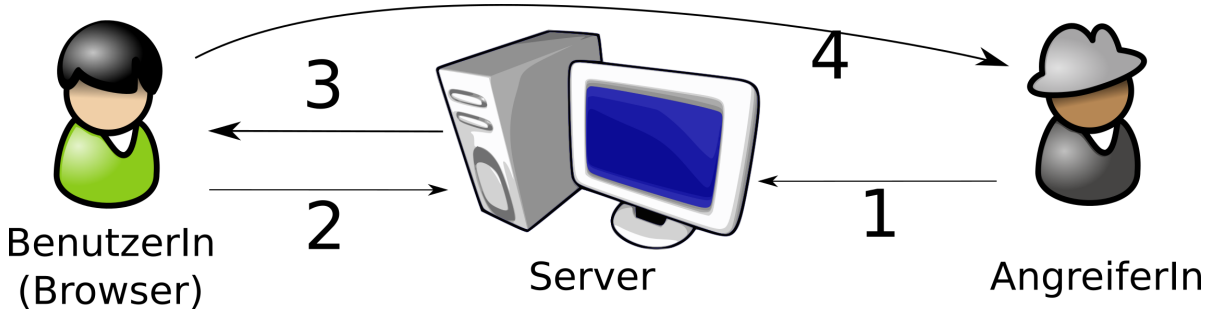
- Gefahr:
  - Session-Hijacking
  - Teile der Web-Seite können überschrieben werden
  - Weiterleitung von Benutzern zu einer anderen Seite
- Maßnahmen:
  - Alle Eingaben validieren vor dem erneuten Anzeigen in der Web-Seite
  - HTML-Encoding der Eingaben von Benutzer:innen

Cross-Site-Scripting (XSS) – „Reflected“



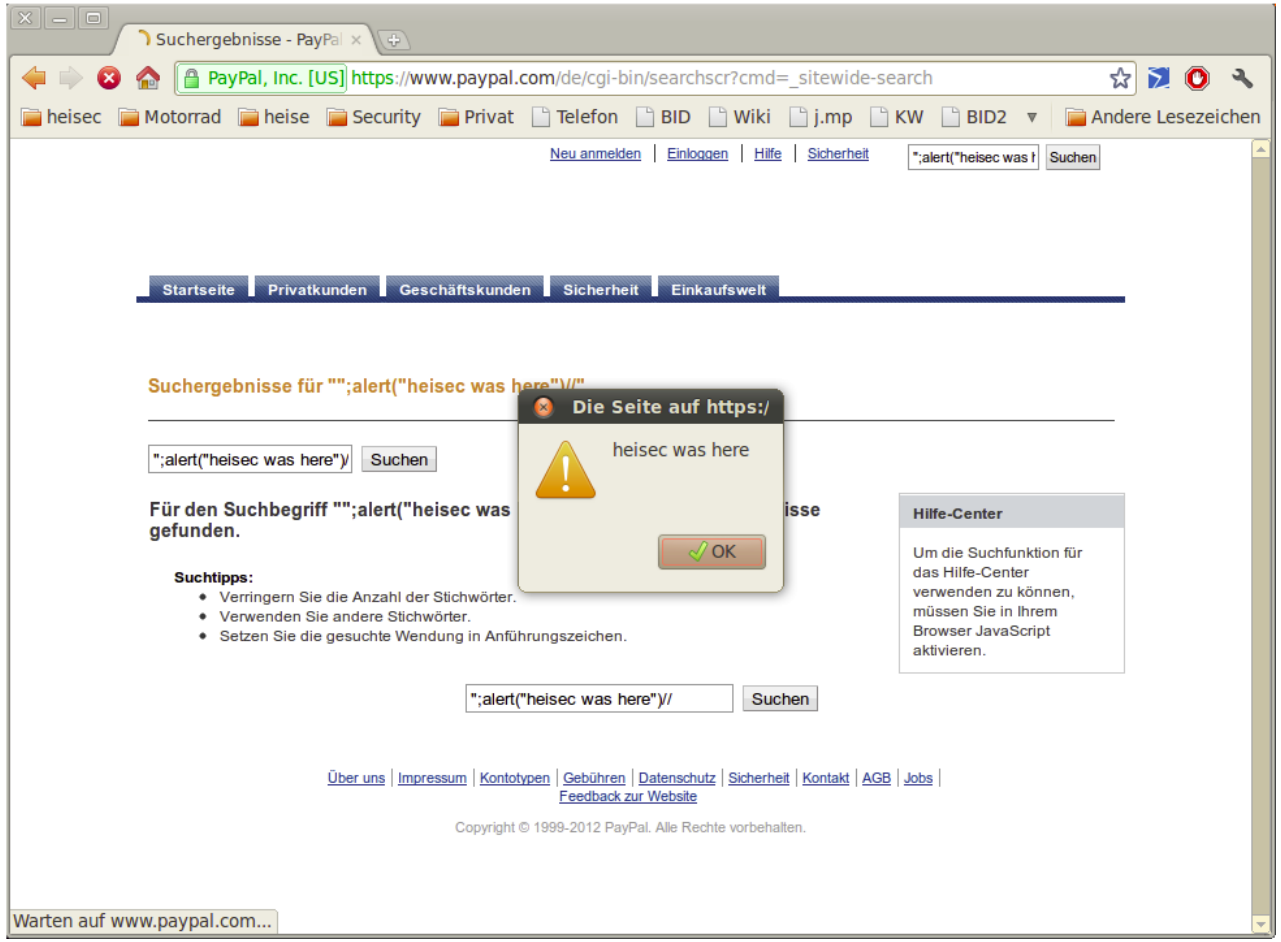
1. Angreifer:in sendet präparierte URL an Benutzer:in über weiteren Kanal (z.B. E-Mail)
2. Benutzer:in ruft Seite im Browser auf
3. Server liefert reguläre Seite mit zusätzlichen Inhalten aus präparierter URL zurück
4. Browser wertet Rückgabe von Server aus. Je nach Angriff sofort Übermittlung von Daten an Angreifer oder umleiten nach weiteren Eingaben des Benutzers/der Benutzerin zum Angreifer/zur Angreiferin

Cross-Site-Scripting (XSS) – „Stored“



1. Angreifer:in fügt präparierten Inhalt ein. Persistente Speicherung des Inhaltes durch Server
2. Benutzer:in ruft Seite im Browser auf
3. Server liefert reguläre Seite mit zusätzlichen präparierten Inhalten des Angreifers/der Angreiferin zurück
4. Browser wertet Rückgabe von Server aus. Je nach Angriff sofort Übermittlung von Daten an Angreifer:in oder umleiten nach weiteren Eingaben des Benutzers/der Benutzerin zum Angreifer/zur Angreiferin

Cross-Site-Scripting (XSS) – Beispiel



(Vergleiche `https://heise.de/-1477324`)

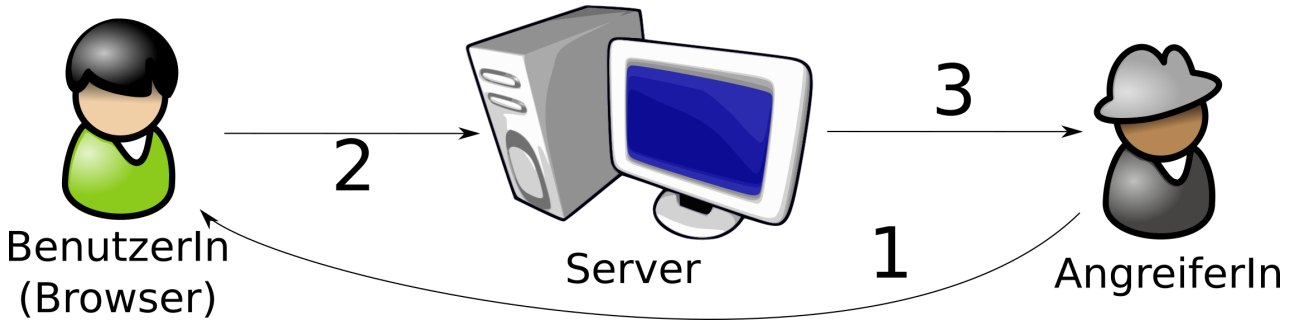
Cross-Site-Scripting (XSS) – Gegenmaßnahmen

- Benutzereingaben validieren (siehe auch SQL Injection)
- Kodieren (HTML Encoded, URL Encoded, . . .) von Benutzereingaben und von gespeicherten Inhalten (z.B. Datenbankeinträgen) vor einer Ausgabe
- Client-seitige Verhinderung durch Browser oder Client-Firewalls
  - Antworten nur an die ursprüngliche Domäne erlauben

Cross-Site-Request-Forgery (CSRF) – Grundlagen

- CSRF ist eine Design-Schwachstelle
- Durch untergeschobene präparierte URL kann ein Angreifer/eine Angreiferin einen Benutzer/eine Benutzerin zu einer Aktion am Server missbrauchen
- Aktionen wie Logout, Einträge erstellen, Passwörter abändern, ...
- `https://server/changePassword.php?newPWD=angreifer:in`
- Durch vorhandenes Authentifizierungs-Cookie beim Benutzer/bei der Benutzerin ist kein Login erforderlich
- Angreifer:in kann Änderungen über IP-Adresse des Benutzers/der Benutzerin durchführen (Verschleierung)

Cross-Site-Request-Forgery (CSRF) – Funktionsweise



1. Angreifer:in sendet präparierte URL an Benutzer:in durch zusätzlichen Kanal (z.B. E-Mail)
2. Benutzer:in ruft Seite im Browser auf – Server führt Aktion direkt aus
3. (optional) Weiter Zugriff auf den Server, z.B. nach Passwortänderungen. Beispielsweise bei einer Überweisung könnte der Schritt entfallen.

Cross-Site-Request-Forgery (CSRF) – Gegenmaßnahmen

- Hinzufügen eines Shared Secrets zu jeder internen URL und zu Formularen. Ein Shared Secret ist ein geheimer Token, den nur der Server und der Client kennen. Bei jeder Anfrage vom Client wird dieser Token dem Server übermittelt – und dort geprüft.
  - Bereits in einigen Web Frameworks enthalten
  - Bei unverschlüsselter Kommunikation kann ein Angreifer/eine Angreiferin diesen Token stehlen
- Verifizierung von Aktionen durch Benutzer:innen anfordern
  - „Möchten Sie wirklich Ihr...?“

Buffer Overflows – Grundlagen

Buffer-Overflows entstehen durch Programmierfehler, wenn keine oder falsche Längenüberprüfung beim Schreiben in Buffer durchgeführt wird

- Buffer Overflow überschreibt bei zu langen Eingaben nachfolgende Speicherbereiche
- Überschreiben kann unterschiedliche Auswirkungen haben
  - Absturz der Applikation mittels Exception/Segfault
  - Ungewöhnliches Systemverhalten
  - Ausführen von eingeschleusten Code
  - Keine Auswirkungen zum Beispiel wenn der zusätzlich überschriebene Speicher nicht mehr verwendet wird
- Verwendung von low-level Programmiersprachen ohne automatisches Bounds-Checking

Buffer Overflows – Beispiele für Gegenmaßnahmen

- Korrekte Input Validierung und Längenüberprüfung
- Verwendung von Programmiersprachen mit automatischer Längenüberprüfung
- Vermeidung von gefährdeten Funktionen (strcpy(), gets(), strcat(),. . .)
- Verwendung von Testmethoden zum Finden von Buffer Overflows im Quellcode (Statische Codeanalyse, Fuzzing,. . .)

Sicherheit in der Testphase

Testing

Siehe VU-Einheit „Testing“ am 17.11.2023

Sicherheit in der Betriebsphase

Betrieb

- Es zeigt sich, ob Sicherheitsanforderungen tatsächlich erfüllt werden
- Kenntnis von Sicherheitsanforderungen für den Betrieb
- Definition und Einhaltung des Service Levels (Service Level Agreement (SLA))
- Bemerken von Sicherheitsvorfällen
- Mögliche Reaktionen auf Sicherheitsvorfälle
- ITIL (Information Technology Infrastructure Library)
- Organisatorische Sicherheitsanforderungen

Literatur, Quellen, Weblinks 1/2

- Hinweis. Die Slides enthalten Teile von Slides von (früheren) ESSE-Kolleg:innen
- Gary McGraw. Software Security: Building Security In (Addison-Wesley Software Security Series). Februar 2006
- Stuart R. Faulk. Software Requirements: A Tutorial. Technischer Bericht, Center for High Assurance Computer Systems, US Navy, 1995. `https://web.archive.org/web/20040930125657/http://www.cs.umd.edu/class/spring2004/cmsc838p/Requirements/Faulk_Req_Tut.pdf`
- Jerome H. Saltzer und Michael D. Schroeder. The protection of information in computer systems. In *Proceedings of the IEEE*, Band 63, Seiten 1278–1308, 1975

Literatur, Quellen, Weblinks 2/2

- John Viega und Gary McGraw. *Building Secure Software: How to Avoid Security Problems the Right Way*. Addison-Wesley, 2002
- `https://www.owasp.org/`
- `https://www.schneier.com/paper-attacktrees-ddj-ft.html`
- Ron Ross, Mark Winstead, und Michael McEvilley. NIST Special Publication NIST SP 800-160v1r1: Engineering Trustworthy Secure Systems, November 2022. `https://doi.org/10.6028/NIST.SP.800-160v1r1`
- Florian Fankhauser, Christian Schanes, und Christian Brem. Sicherheit in der Softwareentwicklung. In *Softwaretechnik - Mit Fallbeispielen aus realen Entwicklungsprojekten*, Kapitel 13, Seiten 589–646. Pearson Studium, München, 1. Auflage, 2009

Zusammenfassung

- Sicherheit in allen Phasen der Entwicklung berücksichtigen
- Sicherheitsbewusstsein erzeugen
- Sicherheitsprobleme und Sicherheitsfeatures von Programmiersprachen berücksichtigen
- Security *immer* berücksichtigen – es werden immer wieder bereits Prototypen im Produktivbetrieb eingesetzt!
- Regelmäßige Analyse des Systems z.B. auf Grund neuer Angriffstechniken erforderlich $\rightarrow$ „Sicherheit ist ein Prozess“
- Für die Implementierung: *Input-Validierung!* (Ursache für viele Sicherheitsfehler)

Vielen Dank!

https://establishing-security.at/