Einführung in Web-Security

Alexander »alech« Klink

Gulaschprogrammiernacht 2013

Agenda

Cross-Site-Scripting (XSS)
Authentifizierung und Sessions
Cross-Site-Request-Forgery ([XC]SRF)
SQL-Injections
Autorisierungsprobleme
Code-/Command-Injections

… aus der Sicht von Angreifer, Entwickler und User

Cross-Site-Scripting (XSS)

<html>
[…]

ungefilterte Nutzereingaben

[…]

reflexiv vs. permanent
(in der URL vs. in der Datenbank)

Cross-Site-Scripting (XSS) Angreifersicht

<script>alert("XSS")</script> als Minimalbeispiel

Session-Cookie auslesen und weiterleiten

beliebige Aktionen in der Anwendung durchführen

Informationen/Dateien exfiltrieren
(ggf. von interner Anwendung ins Internet)

Cross-Site-Scripting (XSS) Entwicklersicht

Eingabevalidierung/Ausgabeencodierung
aber: kontextabhängig (innerhalb eines Tags und eines Attributs unterschiedlich)
Gute Frameworks/Templatingsysteme verwenden
ggf. Content-Security-Policy

Cross-Site-Scripting (XSS) Entwicklersicht

Eingabevalidierung/Ausgabeencodierung
aber: kontextabhängig (innerhalb eines Tags und eines Attributs unterschiedlich)
Gute Frameworks/Templatingsysteme verwenden
ggf. Content-Security-Policy

Cross-Site-Scripting (XSS)
Nutzersicht

Javascript abschalten (?)
NoScript o.ä.
Browsertrennung
Medienkompetenz
Backups von „in der Cloud“ gespeicherten Daten

Sessions und Authentifizierung

HTTP hat keinen State
daher Session-IDs mitübertragen
meistens als Session-Cookie

Authentifizierung mit Username & Passwort
wie sinnvoll speichern?
salted hashes, Runden, ...

Sessions und Authentifizierung
Angreifersicht

Session-Cookie erlangen (z.B. durch XSS) heißt komplette (und einfache!) Kontrolle über die Anwendung im Nutzerkontext

Session Fixation

Passwörter raten

Hashes bruteforcen

Angriffe auf Passwortspeicherung im Browser

Passwort-Wiederbenutzung

Sessions und Authentifizierung
Entwicklersicht

Session-Cookies wirklich zufällig?
secure und *httpOnly*-Flags

Muss ich wirklich eigene Accounts haben?
Passwortspeicherung überdenken
Verschlüsselung ≠ Hashing

Sessions und Authentifizierung Nutzersicht

Unterschiedliche Passwörter für unterschiedliche Dienste

Aber: wo speichern?

Cross-Site-Request-Forgery (CSRF)

Aktionen in Webapplikationen sind erstmal nur GET/POST-Requests

Authentisierung mittels Session-Cookie

Dieser wird immer übermittelt

Cross-Site-Request-Forgery (CSRF)
Angreifersicht

Angemeldeten Nutzer einen Request unterschieben

z.B. <img src=
“http://www.webapp.example.com/delete/all“>

POST via IFrame und JS-Submit

ggf. müssen IDs erraten/durchprobiert werden

Social Engineering-Aspekt

Cross-Site-Request-Forgery (CSRF) Entwicklersicht

Zufällige Tokens in Formulare einbinden
… und auch überprüfen!
<input type="hidden" name="csrf_token"
value="deafbeefdeadbeefdeadbeef"/>
auch hier hilft ein geeignetes Framework

Cross-Site-Request-Forgery (CSRF) Nutzersicht

Browsertrennung
z.B. „Onlinebanking“ vs.
„Random links from the interwebs“

SQL-Injections

Datenbank als Storage-Backend für Webapps
oft ein technischer User mit Vollzugriff auf alle Applikationsdaten

ist als Vektor mittlerweile recht gut erkannt und wird gefühlt seltener

SQL-Injections
Angreifersicht

$sql_statement = „SELECT * FROM privmsgs WHERE subject LIKE '%“ + $req['query'] + „%' AND user_id = '“ + $session['user_id'] + „'“;

/searchmsgs?query=' OR 1=1 –-
Query: „SELECT * FROM privmsgs WHERE subject LIKE '%' OR 1=1 --“
ggf. Statement Stacking, UNIONs, blind, …

SQL-Injections
Entwicklersicht

Bind parameters:
$sql_statement = „SELECT * FROM privmsgs
WHERE subject LIKE '%?%' AND user_id = '?'“
$result = execute_bind_params($req['query'],
$session['user_id'])

alternativ auch gerne Object Relational Mapper
(dann muss man auch kein SQL können ;-))

SQL-Injections
Entwicklersicht

Bind parameters:
$sql_statement = „SELECT * FROM privmsgs
WHERE subject LIKE '%?%' AND user_id = '?'“
$result = execute_bind_params($req['query'],
$session['user_id'])

alternativ auch gerne Object Relational Mapper
(dann muss man auch kein SQL können ;-))

SQL-Injections
Nutzersicht

ähmja.

Autorisierungsprobleme

Authentifizierung und Autorisierung sind nicht dasselbe!
Nur weil ich eingeloggt bin, muss ich nicht alles tun dürfen.
Dämlicher aber typischer Fehler: fortlaufende IDs, kein Check ob ID zu User gehört:
/invoices?customer_id=2342

Autorisierungsprobleme Angreifersicht

Numerische IDs immer mal probeweise modifizieren
Wenn Zugriff auf höher priviligierten Account vorhanden: Requests mitschneiden und mit niedrig priviligierten Account ausprobieren

Autorisierungsprobleme Entwicklersicht

Konsequente Autorisierungsprüfung: gehört das Objekt wirklich dem User, „darf der das?“

Frameworks nutzen

Quick-and-dirty: ausreichend lange zufällige IDs

Autorisierungsprobleme Nutzersicht

ähmja.

Code-/Command-Injections

Nice to have, aber an die Daten kommt man ja auch anders (s.o.)
Schön für die Botnetzbetreiber

Code-/Command-Injections Angreifersicht

eval „require Foo::Bar::$request['module'];“
/foo?module=Baz; `touch /tmp/pwned`

$output = `git log $request['reflike']`
/gitlog?reflike=HEAD; touch /tmp/pwned

Code-Injections
Entwicklersicht

Inputvalidierung. Inputvalidierung. Inputvalidierung.

Vorsicht mit „eval“ und ähnlichen Konstrukten.

Bei Aufrufen von externen Programmen mal auf die Shell verzichten
Achtung: auch beliebige Parameter können Probleme machen (ssh -oProxyCommand)

Code-Injections Nutzersicht

ähmja.

… und sonst so?

Logische Fehler
File Disclosure
uralte Serverversionen
XML-Parsing (XML Entity Expansion, etc.)
eigene/statische Krypto
Debugfunktionen
etc. pp.
Selber ausprobieren: Altaro Mutual
(http://demo.testfire.net)

Q & A

Fragen?
Antworten?
Diskussionsbedarf?