Sample HTTP Requests

GET request

```
GET /search.jsp?
name=blah&type=1 HTTP/1.0
User-Agent: Mozilla/4.0
Host: www.mywebsite.com
Cookie:
SESSIONID=2KDSU72H9GSA289
<CRLF>
```

POST request

```
POST /search.jsp HTTP/1.0
User-Agent: Mozilla/4.0
Host: www.mywebsite.com
Content-Length: 16
Cookie:
SESSIONID=2KDSU72H9GSA289
<CRLF>
name=blah&type=1
<CRLF>
```

Context for application security

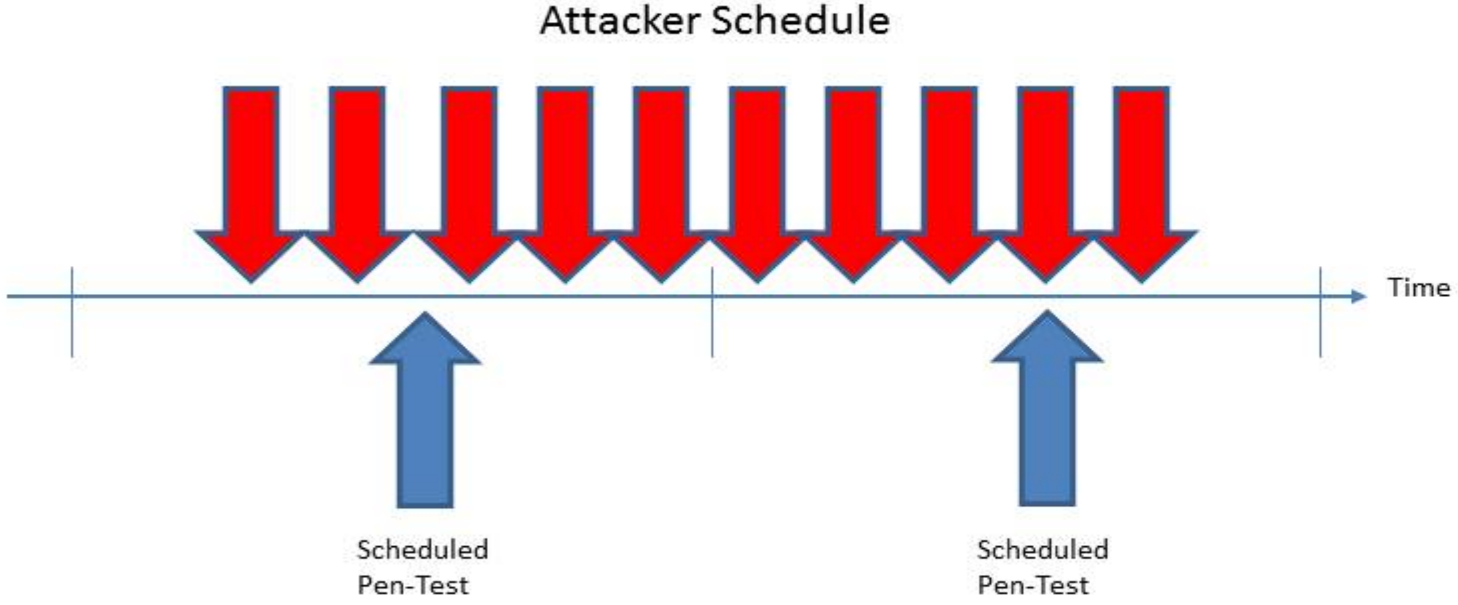
Web applications are the technology of choice for business applications

- Platform independent
- Thin client
- Vendors deploying online portals, web application interfaces
- Customer demand for increased convenience
- Mobile application risk is primarily from webservices on the backend

Majority of web applications are developed internally or outsourced to a third party system integrator

Pressure is common to deliver functionality on time and within budget – security can be an afterthought

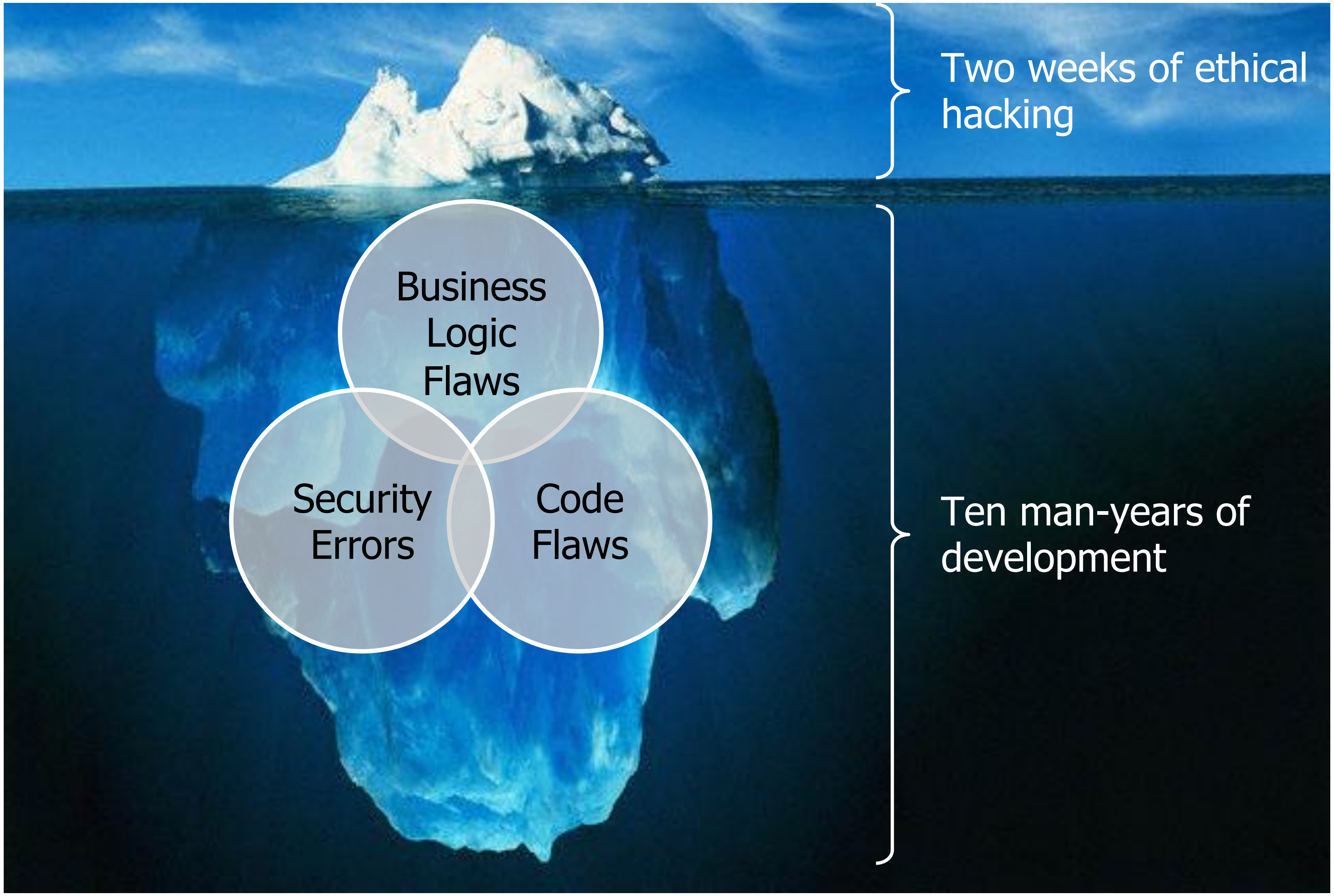
An Attacker has 24x7x365 to Attack



The Defender has 20 man days per year to detect and defend

Who has the edge?

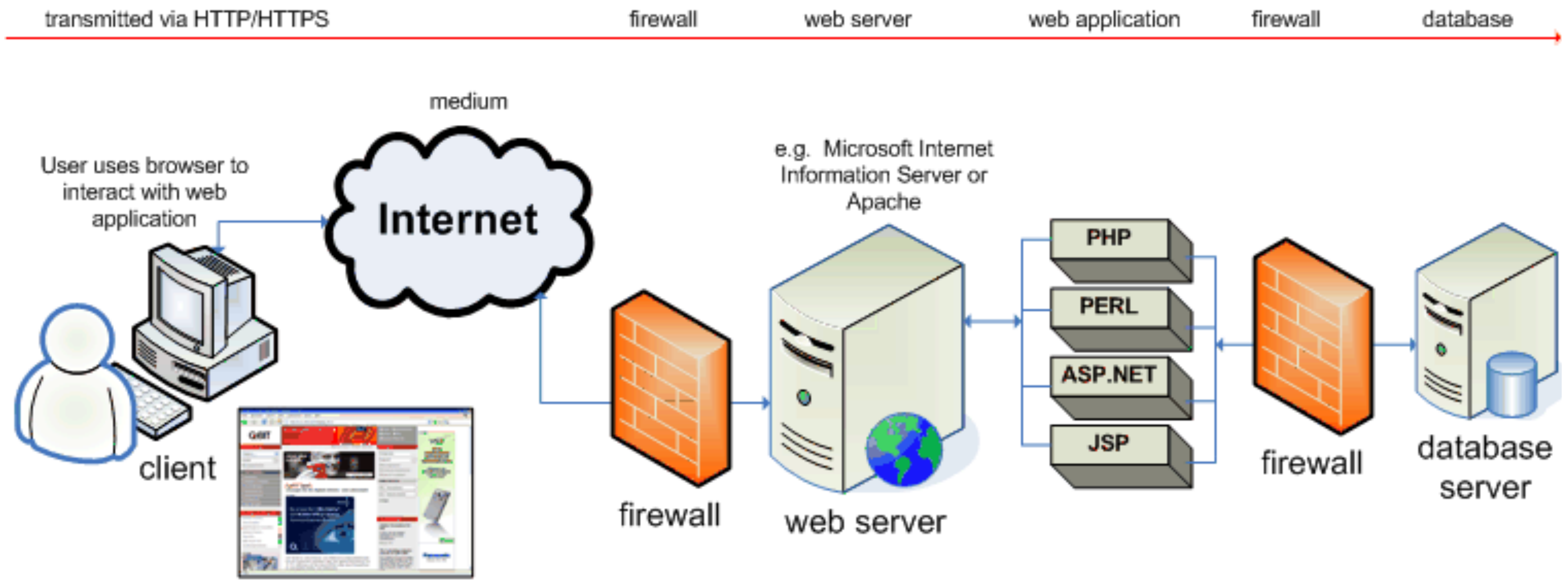
An inconvenient truth



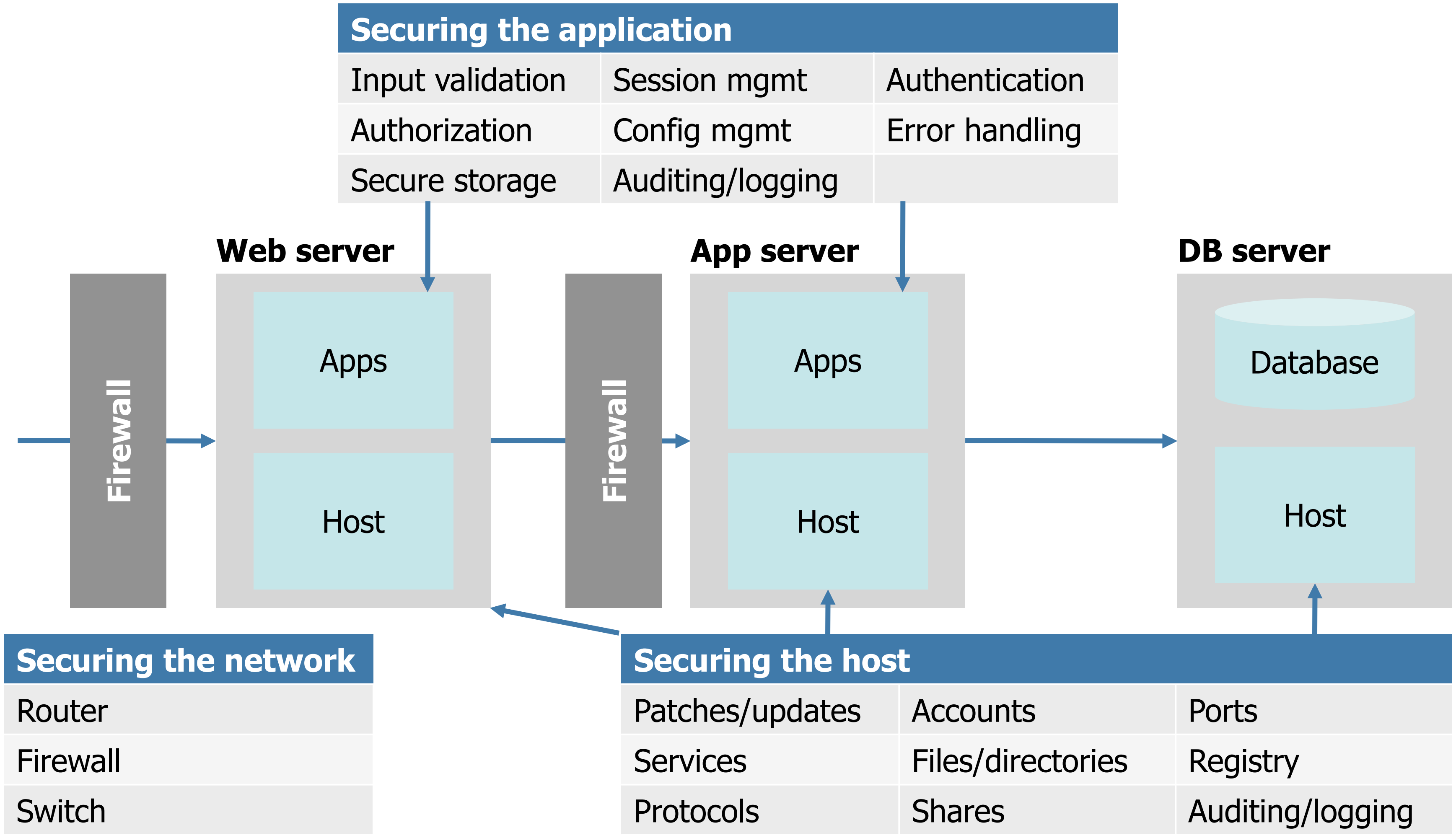
Web application security risks

Blurring traditional boundaries	Organizations are exposing internal data and critical functionality to the public Internet through web application deployments
Data privacy	Weak security controls may be exploited by skilled attackers to access sensitive information or perform unauthorized activities on your organizations' systems
Impact of a security breach	Loss of customer confidence and reputational damage via the negative publicity associated with a security breach

Web application components


transmitted via HTTP/HTTPS
firewall
web server
web application
firewall
database
medium
User uses browser to interact with web application
Internet
e.g. Microsoft Internet Information Server or Apache
PHP
PERL
ASP.NET
JSP
client
firewall
web server
firewall
database server

Web Application Security



Securing the application		
Input validation	Session mgmt	Authentication
Authorization	Config mgmt	Error handling
Secure storage	Auditing/logging	

Securing the network
Router
Firewall
Switch

Securing the host		
Patches/updates	Accounts	Ports
Services	Files/directories	Registry
Protocols	Shares	Auditing/logging

Web application behaviour

HTTP is stateless and hence requests and responses to communicate between browser and server have no memory.

Most typical HTTP requests utilise either GET or POST methods

Scripting can occur on:

- Server-Side (e.g. perl, asp, jsp)
- Client-Side (javascript, flash, applets)

Web server file mappings allow the web server to handle certain file types using specific handlers (ASP, ASP.NET, Java, JSP,CFM etc)

Data is posted to the application through HTTP methods, this data is processed by the relevant script and result returned to the user's browser

POST and GET methods

- "GET" exposes sensitive authentication information in the URL
  - In Web Server and Proxy Server logs
  - In the http referer [sic] header
  - In Bookmarks/Favorites - often sent to others
- "POST" places information in the body of the request and not the URL

Keep sensitive data out of all GET requests

Only submit sensitive data over HTTPS POST

' ;

Anatomy of a SQL Injection Attack

```
$NEW_EMAIL = Request[‘new_email’];
$USER_ID = Request[‘user_id’];


update users set email=‘$NEW_EMAIL’
where id=$USER_ID;
```

Anatomy of a SQL Injection Attack

```
$NEW_EMAIL = Request['new_email'];
$USER_ID = Request['user_id'];

update users set email='$NEW_EMAIL'
where id=$USER_ID;
```

SUPER AWESOME HACK: `$NEW_EMAIL = ';`

```
update users set email='';
```

Best Practice: Parameterized Queries

Parameterized Queries ensure that an attacker is not able to change the intent of a query, even if SQL commands are inserted by an attacker.

Java EE – use PreparedStatement()

```
String query = "SELECT account_balance FROM user_data WHERE user_name = ? ";
PreparedStatement pstmt = connection.prepareStatement( query );
pstmt.setString( 1, custname); ResultSet results = pstmt.executeQuery( );
```

Safe C# .NET Prepared Statement Example

```
String query = "SELECT account_balance FROM user_data WHERE user_name = ?";
try
{
OleDbCommand command = new OleDbCommand(query, connection);
command.Parameters.Add(new OleDbParameter("customerName", CustomerName Name.Text));
OleDbDataReader reader = command.ExecuteReader(); // …
}
catch (OleDbException se) {
// error handling
}
```

.NET Parameterized Query

Dynamic SQL: (Not so Good)

```
string sql = "SELECT * FROM User WHERE Name = '" + NameTextBox.Text + "' AND
Password = '" + PasswordTextBox.Text + "'";
```

Parameterized Query: (Nice, Nice!)

```
SqlConnection objConnection = new SqlConnection(_ConnectionString);
objConnection.Open();
SqlCommand objCommand = new SqlCommand(
    "SELECT * FROM User WHERE Name = @Name AND Password =
    @Password", objConnection);
objCommand.Parameters.Add("@Name", NameTextBox.Text);
objCommand.Parameters.Add("@Password", PasswordTextBox.Text);
SqlDataReader objReader = objCommand.ExecuteReader();
if (objReader.Read()) { ...
```

Java Prepared Statement

Dynamic SQL: (Injectable)

```
String sqlQuery = “UPDATE EMPLOYEES SET SALARY = ‘ +
        request.getParameter(“newSalary”) + ‘ WHERE ID = ‘ +
        request.getParameter(“id”) + ‘”;
```

PreparedStatement: (Not Injectable)

```
double newSalary = request.getParameter(“newSalary”) ;
int id = request.getParameter(“id”);
PreparedStatement pstmt = con.prepareStatement("UPDATE EMPLOYEES
            SET SALARY = ? WHERE ID = ?");
pstmt.setDouble(1, newSalary);
pstmt.setInt(2, id);
```

Best Practice: Parameterized Queries

Unsafe HQL Statement Query (Hibernate)

```
unsafeHQLQuery = session.createQuery("from Inventory where
productID='"+userSuppliedParameter+"'");
```

Safe version of the same query using named parameters

```
Query safeHQLQuery = session.createQuery("from Inventory where productID=:productid");


safeHQLQuery.setParameter("productid", userSuppliedParameter);
```

Language specific recommendations:

- Java EE – use PreparedStatement() with bind variables
- .NET – use parameterized queries like SqlCommand() or OleDbCommand() with bind variables
- PHP – use PDO with strongly typed parameterized queries (using bindParam())
- Hibernate - use createQuery() with bind variables (called named parameters in Hibernate)

Best Practice: Parameterized Queries

ASP.NET

```
string sql = "SELECT * FROM Customers WHERE CustomerId = @CustomerId";
SqlCommand command = new SqlCommand(sql); command.Parameters.Add(new
SqlParameter("@CustomerId",
System.Data.SqlDbType.Int));
command.Parameters["@CustomerId"].Value = 1;
```

RUBY – Active Record

```
# Create
Project.create!(:name => 'owasp')
# Read
Project.all(:conditions => "name = ?", name)
Project.all(:conditions => { :name => name })
Project.where("name = :name", :name => name)
# Update
project.update_attributes(:name => 'owasp')
# Delete
Project.delete(:name => 'name')
```

Best Practice: Parameterized Queries

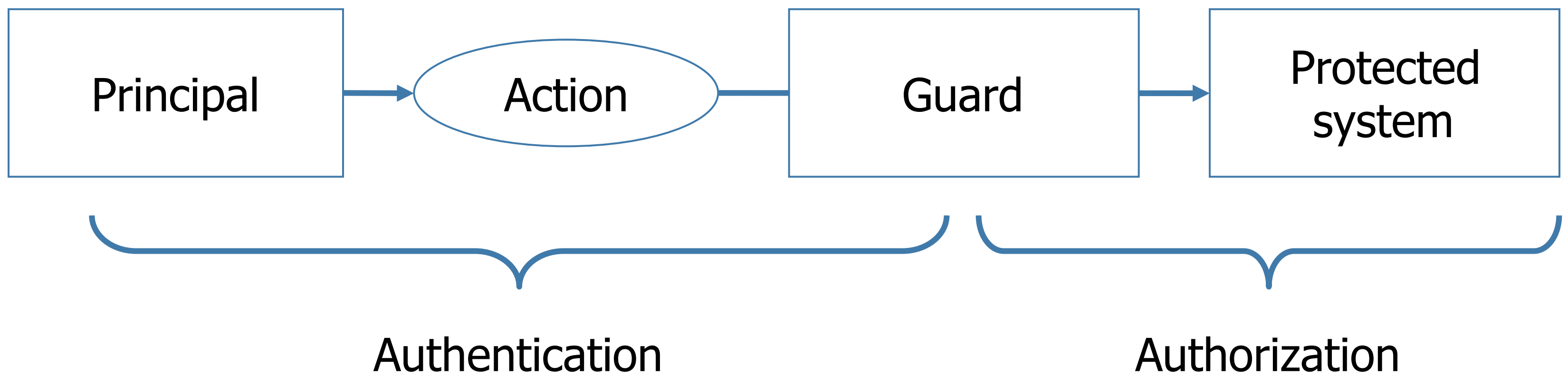
Cold Fusion

```
<cfquery name = "getFirst" dataSource = "cfsnippets">
                    SELECT * FROM #strDatabasePrefix#_courses WHERE intCourseID =
                    <cfqueryparam value = #intCourseID# CFSQLType = "CF_SQL_INTEGER">
</cfquery>
```

Perl - DBI

```
my $sql = "INSERT INTO foo (bar, baz) VALUES ( ?, ? )";
my $sth = $dbh->prepare( $sql );
$sth->execute( $bar, $baz );
```

General Access Control Model



Hard Coded Roles

```
if ((user.isManager() ||
          user.isAdministrator() ||
          user.isEditor()) &&
          user.id() != 1132))
{
   //execute action
}
```

How do you change the policy of this code?

Best Practice: Code to the Activity

```
if (AC.hasAccess("article:edit:12"))
{
  //execute activity
}
```

- Code it once, never needs to change again
- Implies policy is centralized in some way
- Implies policy is persisted in some way
- Requires more design/work up front to get right

Best Practice: Centralized ACL Controller

Define a centralized access controller

- ACLService.isAuthorized(ACTION_CONSTANT)
- ACLService.assertAuthorized(ACTION_CONSTANT) throws AccessControlException()

Access control decisions go through these simple API’s

Centralized logic to drive policy behavior and persistence

May contain data-driven access control policy information

Using a Centralized Access Controller

In Presentation Layer	See code below
In Controller	See code below

In Presentation Layer

```
if (isAuthorized(VIEW_LOG_PANEL))
{
    <h2>Here are the logs</h2>
    <%=getLogs();%/>
}
```

In Controller

```
try (assertAuthorized(DELETE_USER))
{
    deleteUser();
}
```

Best Practice: Verifying policy server-side

Keep user identity verification in session

Load entitlements server side from trusted sources

Force authorization checks on ALL requests

- JS file, image, AJAX and FLASH requests as well!
- Force this check using a filter if possible

SQL Integrated Access Control

Example Feature

- http://mail.example.com/viewMessage?msgid=2356342

This SQL would be vulnerable to tampering

- Select * from messages where messageid = 2356342

Ensure the owner is referenced in the query!

- Select * from messages where messageid = 2356342 AND messages.message_owner = <userid_from_session>

Data Contextual Access Control

Data Contextual / Horizontal Access Control API examples:

- ACLService.isAuthorized(“car:view:321”)
- ACLService.assertAuthorized(“car:edit:321”)

Long form:

- is Authorized(user, Perm.EDIT_CAR, Car.class, 14)

Check if the user has the right role in the context of a specific object

Protecting data a the lowest level!

What is XSS besides being a misnomer?

- Cross-site Scripting (XSS)
- Attacker driven JavaScript
  - Most common web vulnerability
  - Easy vulnerability to find via auditing
  - Easy vulnerability to exploit
  - Certain types of XSS are very complex to fix
  - Difficult to fix all XSS for a large app
  - Easy to re-introduce XSS in development
  - Significant business and technical impact potential

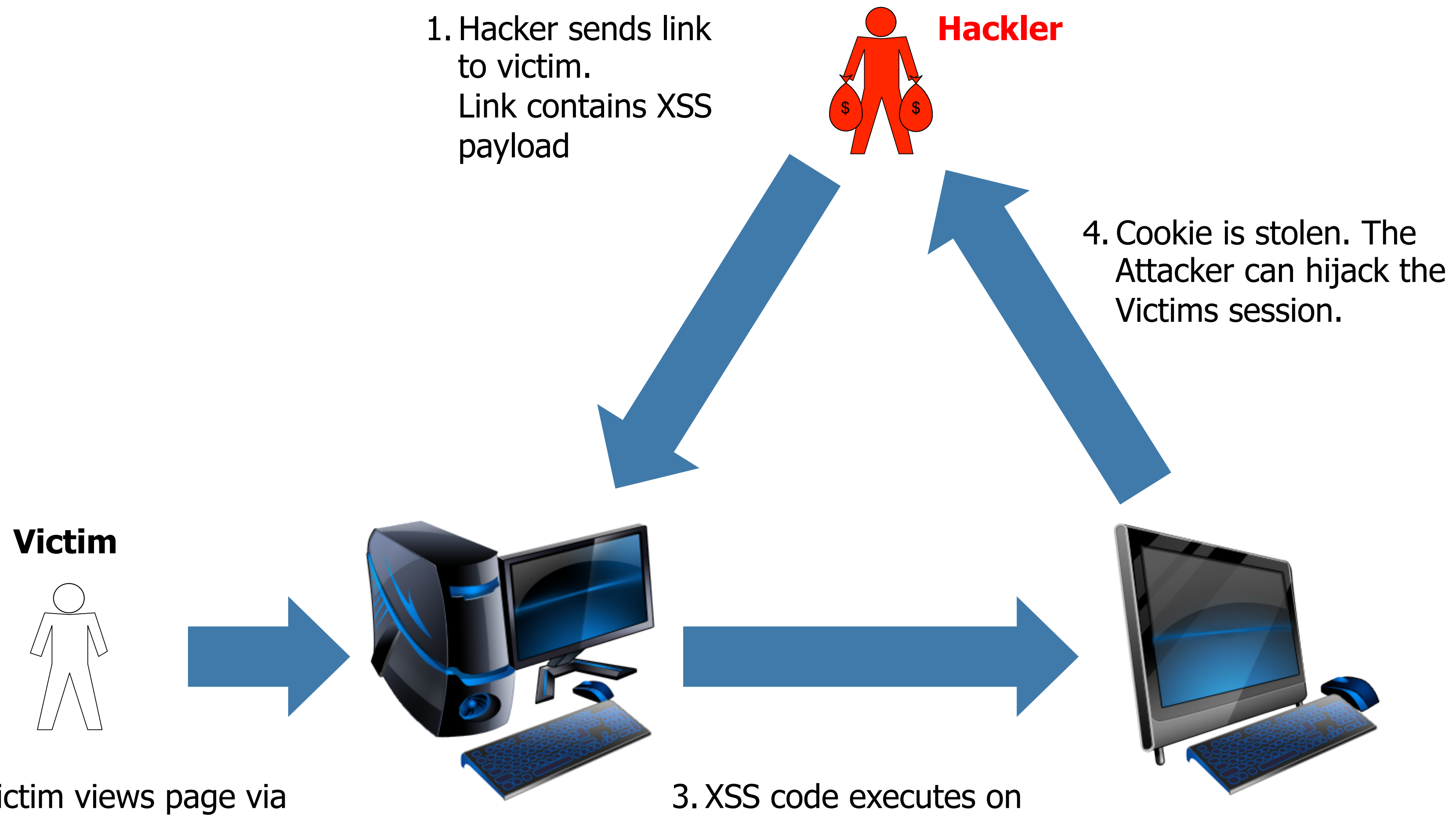
XSS Attack Payload Types

- Session hijacking
- Site defacement potential
- Network scanning
- Undermining CSRF defenses
- Site redirection/phishing
- Data theft
- Keystroke logging
- Loading of remotely hosted scripts

XSS Variants

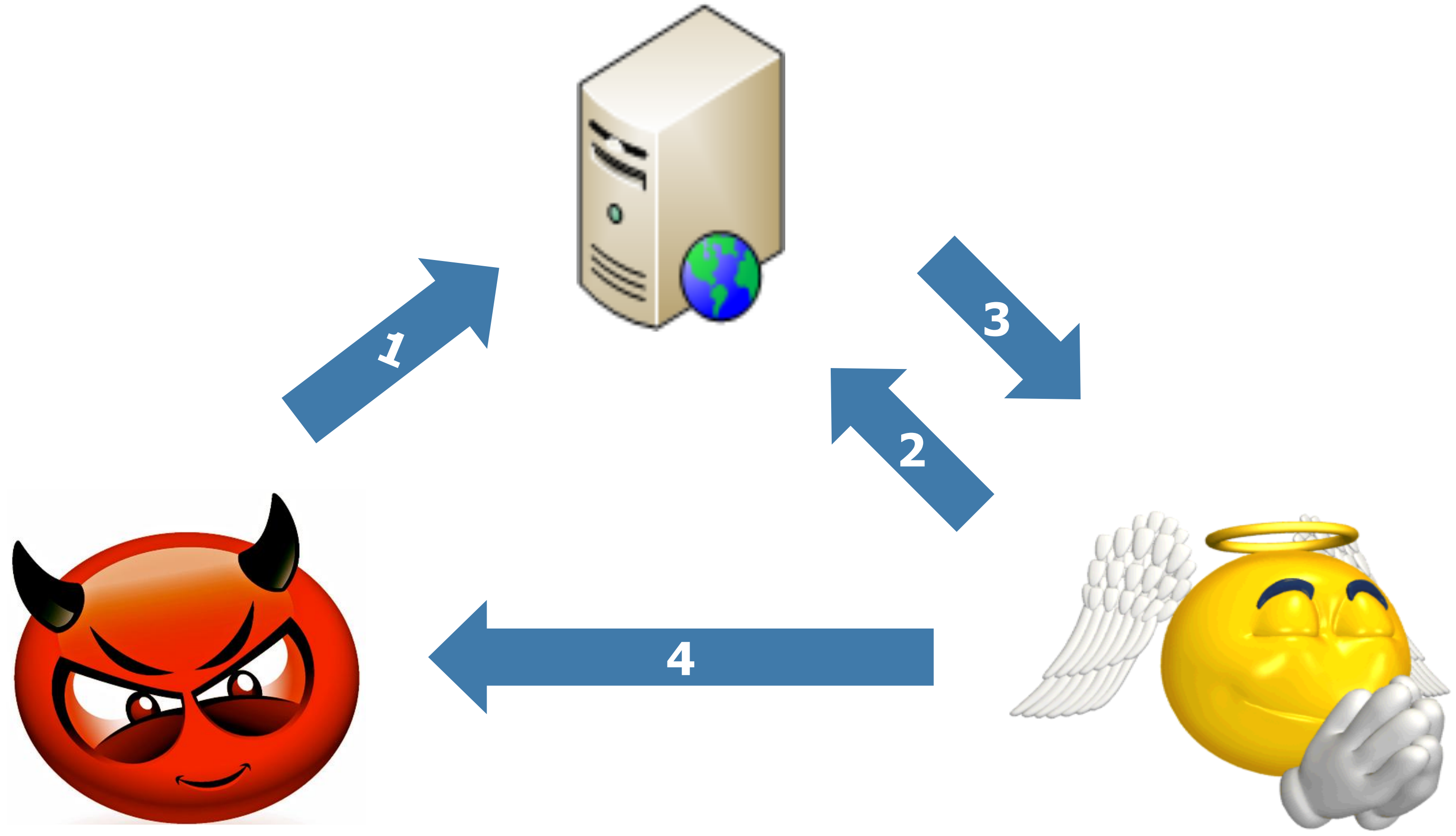
Reflected/ Transient	■ Data provided by a client is immediately used by server-side scripts to generate a page of results for that user. ■ Search engines
Stored/ Persistent	■ Data provided by a client is first stored persistently on the server (e.g., in a database, filesystem), and later displayed to users ■ Bulletin Boards, Forums, Blog Comments
DOM based XSS	■ A page's client-side script itself accesses a URL request parameter and uses this information to dynamically write some HTML to its own page ■ DOM XSS is triggered when a victim interacts with a web page directly without causing the page to reload. ■ Difficult to test with scanners and proxy tools – why?

Reflected XSS

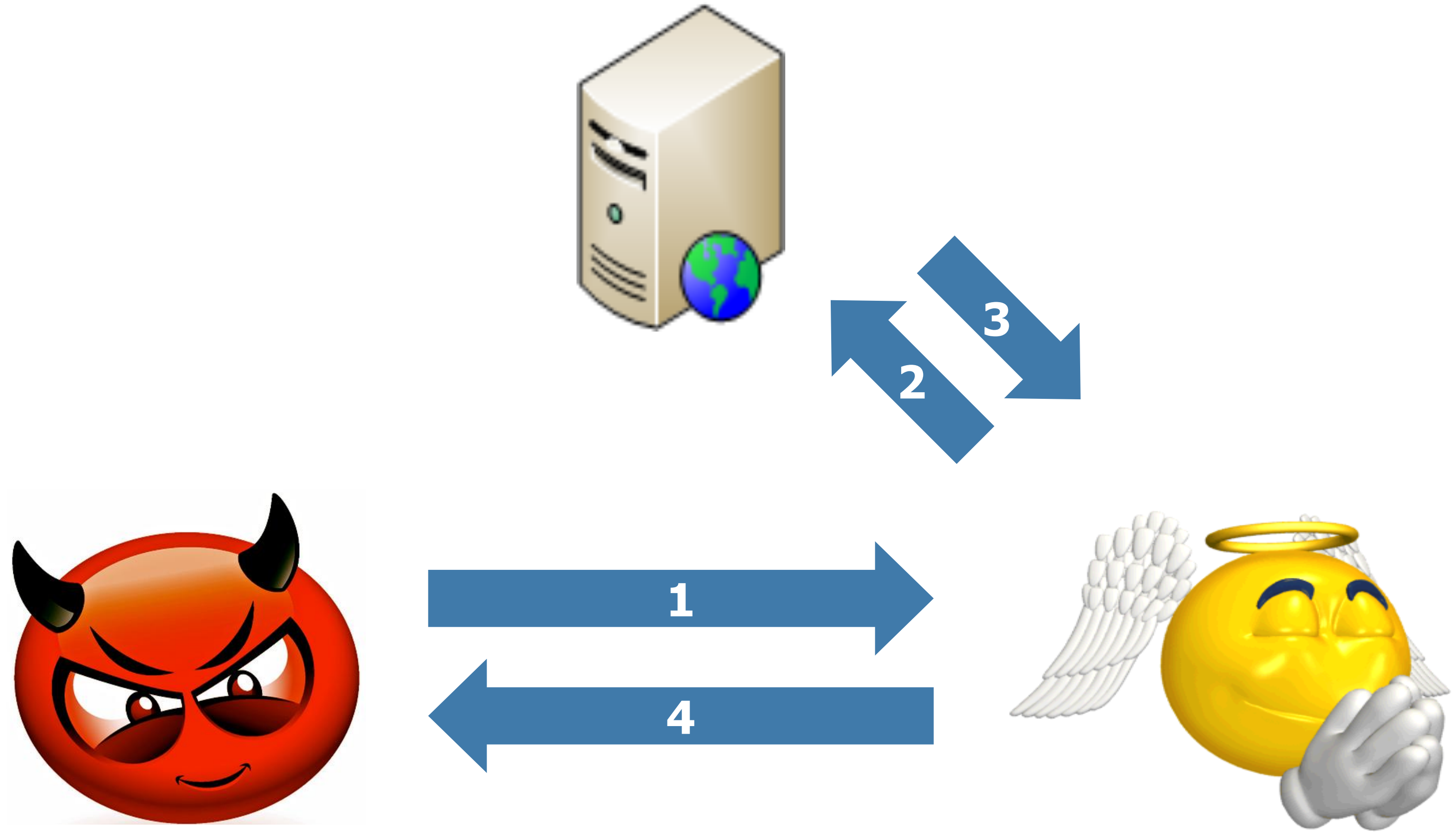


1. Hacker sends link to victim.
   Link contains XSS payload
2. Victim views page via XSS link supplied by attacker.
3. XSS code executes on victims browser and sends cookie to evil server
4. Cookie is stolen. The Attacker can hijack the Victims session.

Persistent/Stored XSS



DOM-Based XSS



XSS Defense by Data Type and Context

Data Type	Context	Defense
String	HTML Body	HTML Entity Encode
String	HTML Attribute	Minimal Attribute Encoding
String	GET Parameter	URL Encoding
String	Untrusted URL	URL Validation, avoid javascript: URL's, Attribute encoding, safe URL verification
String	CSS	Strict structural validation, CSS Hex encoding, good design
HTML	HTML Body	HTML Validation (JSoup, AntiSamy, HTML Sanitizer)
Any	DOM	DOM XSS Cheat sheet
Untrusted JavaScript	Any	Sandboxing
JSON	Client parse time	JSON.parse() or json2.js

Safe HTML Attributes include: align, alink, alt, bgcolor, border, cellpadding, cellspacing, class, color, cols, colspan, coords, dir, face, height, hspace, ismap, lang, marginheight, marginwidth, multiple, nohref, noresize, noshade, nowrap, ref, rel, rev, rows, rowspan, scrolling, shape, span, summary, tabindex, title, usemap, valign, value, vlink, vspace, width

Best Practice: Validate and Encode User Input

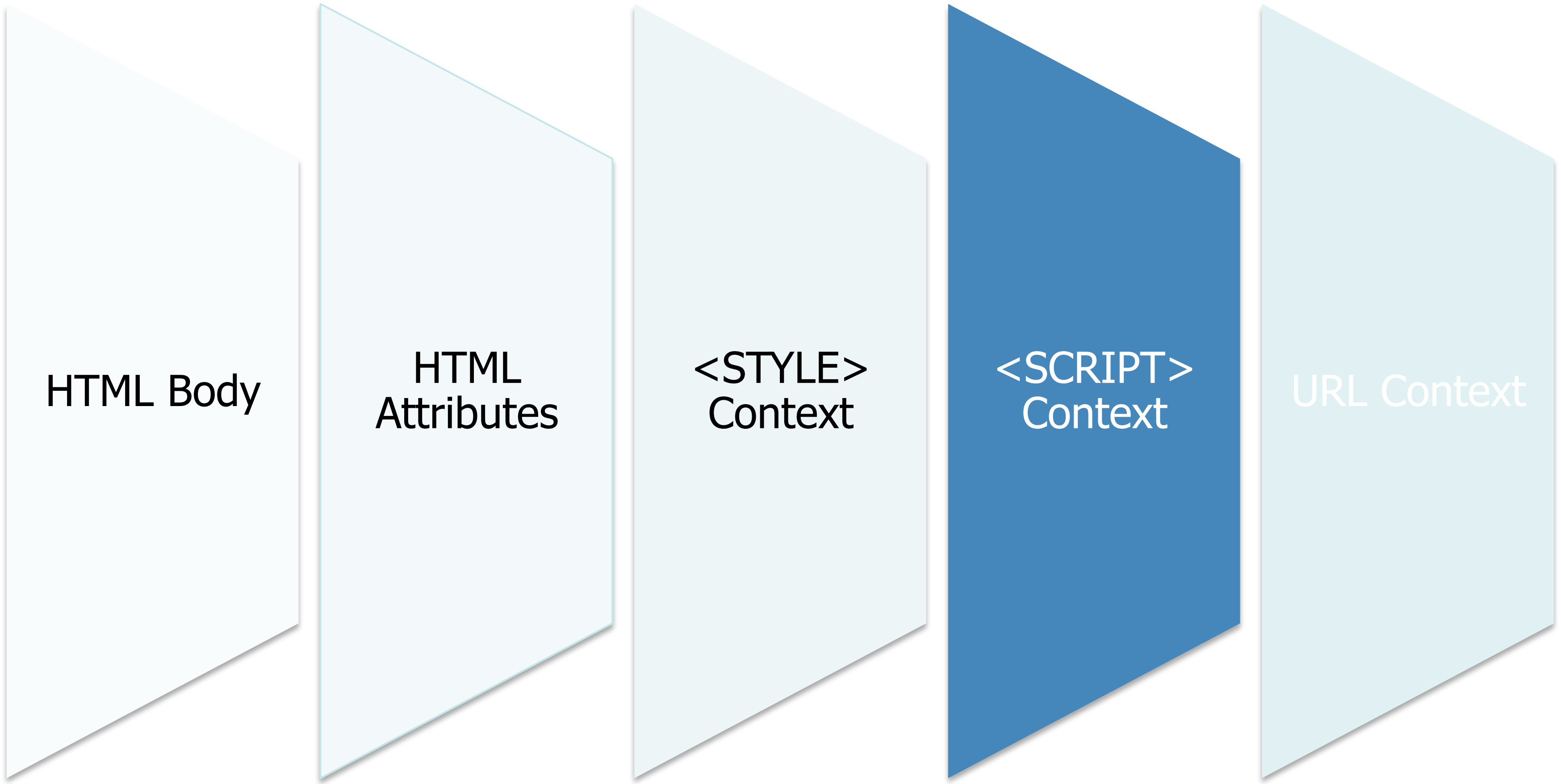
```
String email = request.getParameter("email");
out.println("Your email address is: " + email);
```

```
String email = request.getParameter("email");
String  expression =
"^\w+((-\w+)|(\.\w+))*\@[A-Za-z0-9]+((\.|-)[A-Za-z0-9]+)*\.[A-Za-z0-9]+$";

Pattern pattern = Pattern.compile(expression,Pattern.CASE_INSENSITIVE);
Matcher matcher = pattern.matcher(email);
if (macher.maches())
{
        out.println("Your email address is: " +  StringEscapeUtils.escapeHtml(email));
}
else
{
        //log & throw a specific validation exception and fail safely
}
```

Danger: Multiple Contexts

Different encoding and validation techniques needed for different contexts!



HTML Input

- Clients side widgets like TinyMCE and CKEditor
- Users can edit content beyond plain text
  - Bold, Bullet Points, Color, etc
- These class of widgets submit HTML via request parameters, simple validation not enough
- Validate user-driven HTML on the server with an HTML policy engine
  - Java OWASP AntiSamy or HTML Sanitizer
  - PHP HTML Purifier
  - Java JSoup

XSS in HTML Body

- Reflective XSS attack example:
  - example.com/error?error_msg=You cannot access that file.
- Untrusted data may land in a UI snippet like the following:
  - <div><%= request.getParameter("error_msg") %></div>
- Sample test attack payload:
  - example.com/error?
    error_msg=<script>alert(document.cookie)</script>
- HTML Encoding stops XSS in this context!

HTML Attribute Context

- Aggressive escaping is needed when placing untrusted data into typical attribute values like width, name, value, etc.
- This rule is NOT ok for complex attributes likes href, src, style, or any event handlers like onblur or onclick.
- Escape all non alpha-num characters with the &#xHH; format
- This rule is so aggressive because developers frequently leave attributes unquoted
- <div id=DATA></div>

Javascript Context

Escape **all** non alpha-num characters with the \xHH format

<script>var x=’<%= encodeForJS(DATA) %>';</script>

You're now protected from XSS at the time data is assigned

What happens to x after you assign it?

URL Parameter Escaping

Escape **all** non alpha-num characters with the %HH format

<a href="/search?data=<%=DATA %>">

Be careful not to allow untrusted data to drive entire URL's or URL fragments

This encoding only protects you from XSS at the time of rendering the link

Treat DATA as untrusted after submitted

CSS Pwnage Test Case

```
<div style="width: <%=temp3%>;"> Mouse over </div>
```

```
temp3 =
  ESAPI.encoder().encodeForCSS("expression(alert(String.fromChar
  Code (88,88,88)))");
```

```
<div style="width: expression\28 alert\28 String\2e
  fromCharCode\20 \28 88\2c 88\2c 88\29 \29 \29 ;"> Mouse over
  </div>
```

- Pops in at least **IE6** and **IE7**.

lists.owasp.org/pipermail/owasp-esapi/2009-February/000405.html

CSS Context: XSS Defense

Escape **all** non alpha-num characters with the \HH format

<span style=bgcolor:DATA;>text</style>

Do not use any escaping shortcuts like \"

Strong positive structural validation is also required

If possible, design around this "feature"

- Use trusted CSS files that users can choose from
- Use client-side only CSS modification (font size)

Dangerous Contexts

There are just certain places in HTML documents where you cannot place untrusted data

- Danger: <a $DATA>

There are just certain JavaScript functions that cannot safely handle untrusted data for input

- Danger: <script>eval($DATA);</script>

OWASP ESAPI for Java

ESAPI library provides powerful encoding via ESAPI.encoder():

- String encodeForHTML(String input)
- String encodeForHTMLAttribute(String input)
- String encodeForJavaScript(String input)
- String encodeForURL(String input)
- String encodeForCSS(String input)
- And more! (LDAP, XML, OS, etc)

ESAPI Output Encoding

```
<style>
  bgcolor: <%=ESAPI.encoder().encodeForCSS(data) %>;
</style>
Hello, <%=ESAPI.encoder().encodeForHTML(data) %>!
<script>
  var uName='<%=ESAPI.encoder().encodeForJavaScript(data) %>';
</script>
<div id='<%=ESAPI.encoder().encodeForHTMLAttribute(data) %>'>
<a href="/mysite.com/editUser.do?userName=<%=
ESAPI.encoder().encodeForURL(data) %>">Please click me!</a>
```

Basic XSS Defense Summary

Cross-Site Scripting (XSS)

- Harmful JavaScript artificially introduced into your web app

All user input must be validated!

All user input must be encoded or sanitized specific to each context before being displayed back to the browser!

Plenty of Web 2.0 vectors to consider such as JSON parsing and DOM XSS (See Advanced XSS Defense Module)

DOM Based XSS Defense

DOM Based XSS is a complex risk

Suppose that x landed in …
<script>setInterval(x, 2000);</script>

For some Javascript functions, even JavaScript encoded untrusted data will still execute!

Dangerous JavaScript Sinks

Direct execution	■ eval() ■ window.execScript()/function()/setInterval()/setTimeout(), requestAnimationFrame() ■ script.src(), iframe.src()
Build HTML/ JavaScript	■ document.write(), document.writeln() ■ elem.innerHTML = danger, elem.outerHTML = danger ■ elem.setAttribute("dangerous attribute", danger) – attributes like: href, src, onclick, onload, onblur, etc.
Within execution context	■ onclick() ■ onload() ■ onblur(), etc

Source: https://www.owasp.org/index.php/DOM_based_XSS_Prevention_Cheat_Sheet

Some Safe JavaScript Sinks

Setting a value	▪ elem.innerText(danger) ▪ formfield.val(danger)
Safe JSON parsing	▪ JSON.parse() (rather than eval())

jQuery API's

Dangerous jQuery 1.7.2 Data Types	
CSS	Some Attribute Settings
HTML	URL (Potential Redirect)

jQuery methods that directly update DOM or can execute JavaScript	
$() or jQuery()	.attr()
.add()	.css()
.after()	.html()
.animate()	.insertAfter()
.append()	.insertBefore()
.appendTo()	Note: .text() updates DOM, but is safe.

jQuery methods that accept URLs to potentially unsafe content	
jQuery.ajax()	jQuery.post()
jQuery.get()	load()
jQuery.getScript()	

Don't send untrusted data to these methods, or properly escape the data before doing so

jQuery – But there's more…

More danger

- jQuery(danger) or $(danger)
  - This immediately evaluates the input!!
  - E.g., $("<img src=x onerror=alert(1)>")
- jQuery.globalEval()
- All event handlers: .bind(events), .bind(type, [,data], handler()), .on(), .add(html)

Same safe examples

- .text(danger), .val(danger)

Some serious research needs to be done to identify all the safe vs. unsafe methods

- There are about 300 methods in jQuery

Source: http://code.google.com/p/domxsswiki/wiki/jQuery

Client Side Context Sensitive Output Escaping

Context	Escaping Scheme	Example

HTML Element	(&, <, >, ") → &entity;
	(', /) → &#xHH;
		$ESAPI.encoder().
		encodeForHTML()

HTML Attribute	All non-alphanumeric < 256 → &#xHH	$ESAPI.encoder(). encodeForHTMLAttribute()
JavaScript	All non-alphanumeric < 256 → \xHH	$ESAPI.encoder(). encodeForJavaScript()
HTML Style	All non-alphanumeric < 256 → \HH	$ESAPI.encoder(). encodeForCSS()
URI Attribute	All non-alphanumeric < 256 → %HH	$ESAPI.encoder(). encodeForURL()

Encoding methods built into a jquery-encoder:
https://github.com/chrisisbeef/jquery-encoder

Note: Nested contexts like HTML within JavaScript, and decoding before encoding to prevent double encoding are other issues not specifically addressed here.

JQuery Encoding with JQencoder

Contextual encoding is a crucial technique needed to stop all types of XSS

jqencoder is a jQuery plugin that allows developers to do contextual encoding in JavaScript to stop DOM-based XSS

- http://plugins.jquery.com/plugin-tags/security
- $('#element').encode('html', UNTRUSTED-DATA);

“Secure” DOM XSS/AJAX Workflow

Initial loaded page should only be static content

Load JSON data via AJAX or via embedded JSON

Only use the following methods to populate the DOM

- Node.textContent
- document.createTextNode
- Element.setAttribute

Caution: Element.setAttribute is one of the most dangerous JS methods

Caution: If the first element to setAttribute is any of the JavaScript event handlers or a URL context based attribute ("src", "href", "backgroundImage", "backgound", etc.) then XSS will pop

Best Practice: DOM Based XSS Defense I

- Untrusted data should only be treated as displayable text
- JavaScript encode and delimit untrusted data as quoted strings
- Use document.createElement("…"), element.setAttribute("…","value"), element.appendChild(…), etc. to build dynamic interfaces
- Avoid use of HTML rendering methods
- Make sure that any untrusted data passed to eval() methods is delimited with string delimiters and enclosed within a closure or JavaScript encoded to
N-levels based on usage and wrapped in a custom function

Best Practice: DOM Based XSS Defense II

Limit the usage of dynamic untrusted data to right side operations. And be aware of data which may be passed to the application which look like code (eg. location, eval()).

Limit access to properties objects when using object[x] access functions

Don't eval() JSON to convert it to native JavaScript objects. Instead use JSON.toJSON() and JSON.parse()

Run untrusted script in a sandbox (ECMAScript object sealing, HTML 5 frame sandbox, etc)

Best Practice: Sandboxing

JavaScript Sandboxing (ECMAScript 5)

- Object.seal(obj)
- Object.isSealed(obj)
- Sealing an object prevents other code from deleting, or changing the descriptors of, any of the object's properties

iFrame Sandboxing (HTML5)

- <iframe src="demo_iframe_sandbox.htm" sandbox="”></iframe>
- Allow-same-origin, allow-top-navigation, allow-forms, allow-scripts

Best Practice: Content Security Policy

Anti-XSS W3C standard

CSP 1.1 Draft 19 published August 2012

- https://dvcs.w3.org/hg/content-security-policy/raw-file/tip/csp-specification.dev.html

Must move all inline script and style into external scripts

Add the X-Content-Security-Policy response header to instruct the browser that CSP is in use

- Firefox/IE10PR: X-Content-Security-Policy
- Chrome Experimental: X-WebKit-CSP
- Content-Security-Policy-Report-Only

Define a policy for the site regarding loading of content

Best Practice: Content Security Policy

Externalize all Java-Script within web pages	■ No inline script tag ■ No inline JavaScript for onclick or other handling events ■ Push all JavaScript to formal .js files using event binding
Define Content Security Policy	■ Developers define which scripts are valid ■ Browser will only execute supported scripts ■ Inline JavaScript code is ignored

Content Security Policy by Example

Source: http://www.html5rocks.com/en/tutorials/security/content-security-policy/

Site that loads resources from a content delivery network and does not need framed content or any plugins

X-Content-Security-Policy: default-src https://cdn.example.net; frame-src 'none'; object-src 'none'

Attacking Sensitive Transactions

Once authenticated, users are trusted throughout the lifetime of their session

Applications do not require users to re-authenticate when executing sensitive transactions

Cross-Site Request Forgery (XSRF/CSRF)

- Attacks the trust a web application has for authenticated users
- Browser instances share cookies
- Users typically browse multiple sites simultaneously
- Attackers can abuse the shared cookie jar to send requests as the authenticated user

Anatomy of an CSRF Attack

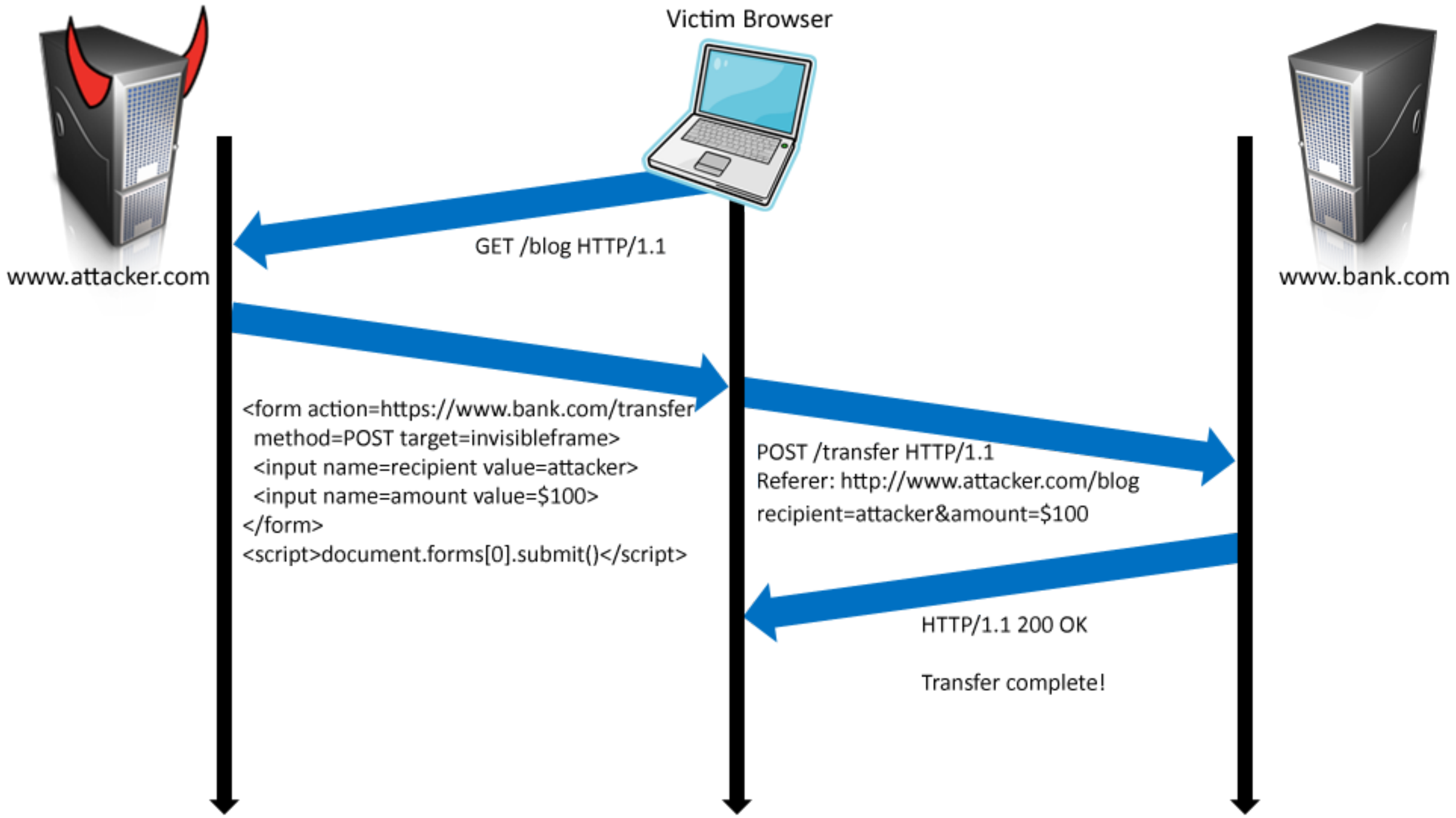
Consider a consumer banking application that contains the following form

```
<form action="http://site.com/Transfer.asp" method="POST" id="form1">
  <p>Account Num: <input type="text" name="acct        value="2345"/
></p>
  <p>Transfer Amt: <input type="text" name="amount"   value="10000"/
></p>
</form>
<script>document.getElementById("form1").submit();</script>
```

This form will generate requests that resemble the following

GET http://www.example.com/Transfer.asp?acct=##&amount=##

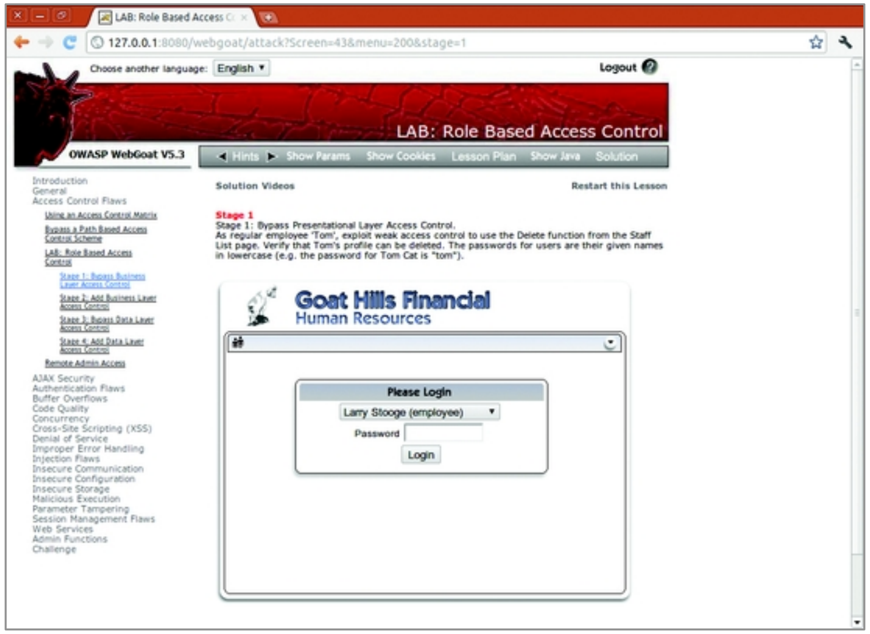
Cross-Site Request Forgery



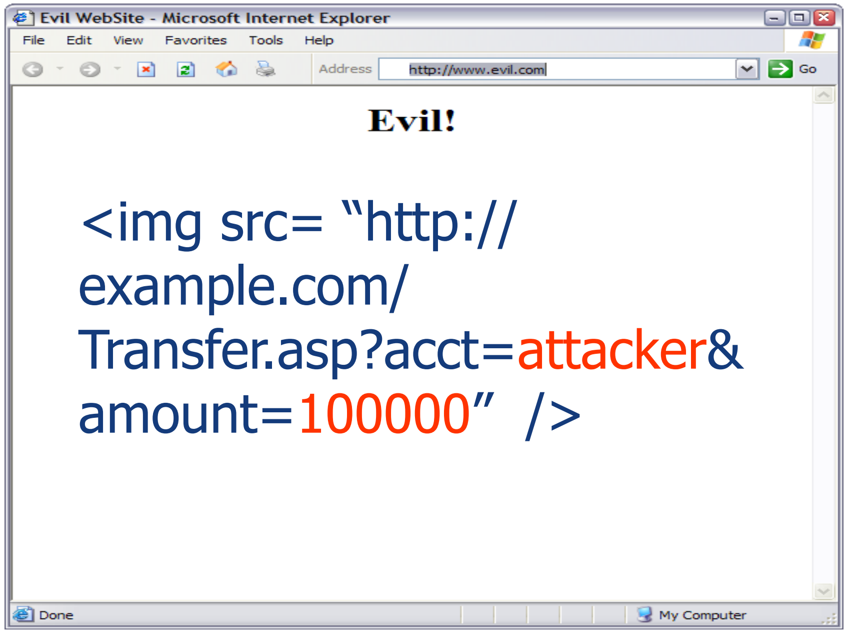
```
<form action=https://www.bank.com/transfer
  method=POST target=invisibleframe>
  <input name=recipient value=attacker>
  <input name=amount value=$100>
</form>
<script>document.forms[0].submit()</script>
```

```
POST /transfer HTTP/1.1
Referer: http://www.attacker.com/blog
recipient=attacker&amount=$100
```

Anatomy of an CSRF Attack



Stage 1
Stage 1: Bypass Presentational Layer Access Control.
As regular employee 'Tom', exploit weak access control to use the Delete function from the Staff List page. Verify that Tom's profile can be deleted. The passwords for users are their given names in lowercase (e.g. the password for Tom Cat is "tom").



Evil!

<img src= "http://example.com/Transfer.asp?acct=attacker&amount=100000" />

What is the Result?

When the <img> tag loads, the attacker’s web site will send a request to the consumer banking application

The user’s browser will attach the appropriate cookie to the attacker’s forged request, thus “authenticating” it

The banking application will verify that the cookie is valid and process the request

The attacker cannot see the resultant response from the forged request

- Does that matter?

Defenses

Request that cause side effects should use the POST method

- Alone, this is not sufficient

Validation of HTTP REFERER header (not recommended)

- Tracking valid refererring pages may be problematic
- Easy to spoof

Require users to re-authenticate

Cryptographic Tokens

Challenge-Response

Challenge-Response is another defense option for CSRF

The following are some examples of challenge-response options.

- CAPTCHA
- Re-Authentication (password)
- One-time Token

While challenge-response is a very strong defense to CSRF (assuming proper implementation), it does impact user experience.

For applications in need of high security, tokens (transparent) and challenge-response should be used on high risk functions.

Synchronizer Token Pattern

"Hidden" token in HTML	Value defined by server when page is rendered. Value is stored in session. Consider leveraging the java.security.SecureRandom class for Java applications.
	Upon Submit, token is sent with form.
	Token value must match with value in session.
	Attacker would not have token value. (XSS attack could get token is page was vulnerable to XSS)

```
<form action="/transfer.do" method="post"> <input type="hidden" name="CSRFToken"
value="OWY4NmQwODE4ODRjN2Q2NTlhMmZlYWEwYzU1YWQwMTVhM2JmNGYxYjJiMGI
4MjJjZDE1ZDZjMTVi MGYwMGEwOA=="> … </form>
```

See also

https://www.owasp.org/index.php/Category:OWASP_CSRFGuard_Project

https://www.owasp.org/index.php/PHP_CSRF_Guard

https://www.owasp.org/index.php/.Net_CSRF_Guard

Other CSRF Defenses

Require users to re-authenticate	Amazon.com does this *really* well
Double-cookie submit defense	Decent defense, but not based on randomness; based on SOP

Basic Cryptographic Mechanisms

- Cryptographic Hash Functions (aka, Message Digests)
  - What they are
  - Desired properties
  - Common uses
- Ciphers
  - Symmetric ciphers
  - Asymmetric ciphers
- Digital signatures

Basic Cryptographic Mechanisms

VERSION 2

Cryptographic Hash Functions (aka, Message Digests)	What they are
	Desired properties
	Common uses
Ciphers	Symmetric ciphers
	Asymmetric ciphers
Digital signatures	…

Uses for Cryptographic Mechanisms

Hash Functions	Integrity, authentication, digital signatures
Symmetric Ciphers	Confidentiality, authentication protocols
Asymmetric ciphers	Confidentiality (especially key exchange), digital signatures
Digital signatures	Integrity, authentication, non-repudiation, attestation

Uses for Cryptographic Mechanisms

VERSION 2

Crypto-graphic Mechanisms

Hash Functions	Integrity, authentication, digital signatures
Symmetric Ciphers	Confidentiality, authentication protocols
Asymmetric ciphers	Confidentiality (especially key exchange), digital signatures
Digital signatures	Integrity, authentication, non-repudiation, attestation

Cryptographic Hash Functions: What Are They?

- **Hash function** is a computationally efficient function mapping arbitrary length bit strings to some fixed-length binary string called **hash-values or message digests.**
- These are many-to-one mapping functions; may be thought of as compression functions.

Cryptographic Hash Functions: Common Uses and Types

- To provide data integrity (detection of tampering)
- To provide digital signatures in a more efficient manner
- To store passwords or pass phrases
- Two types: Unkeyed (AKA, Message Integrity Code) and keyed (Message Authentication Code)

HMACs

Defined in RFC 2104, which defines HMAC for various message digest algorithms.

Defined as:

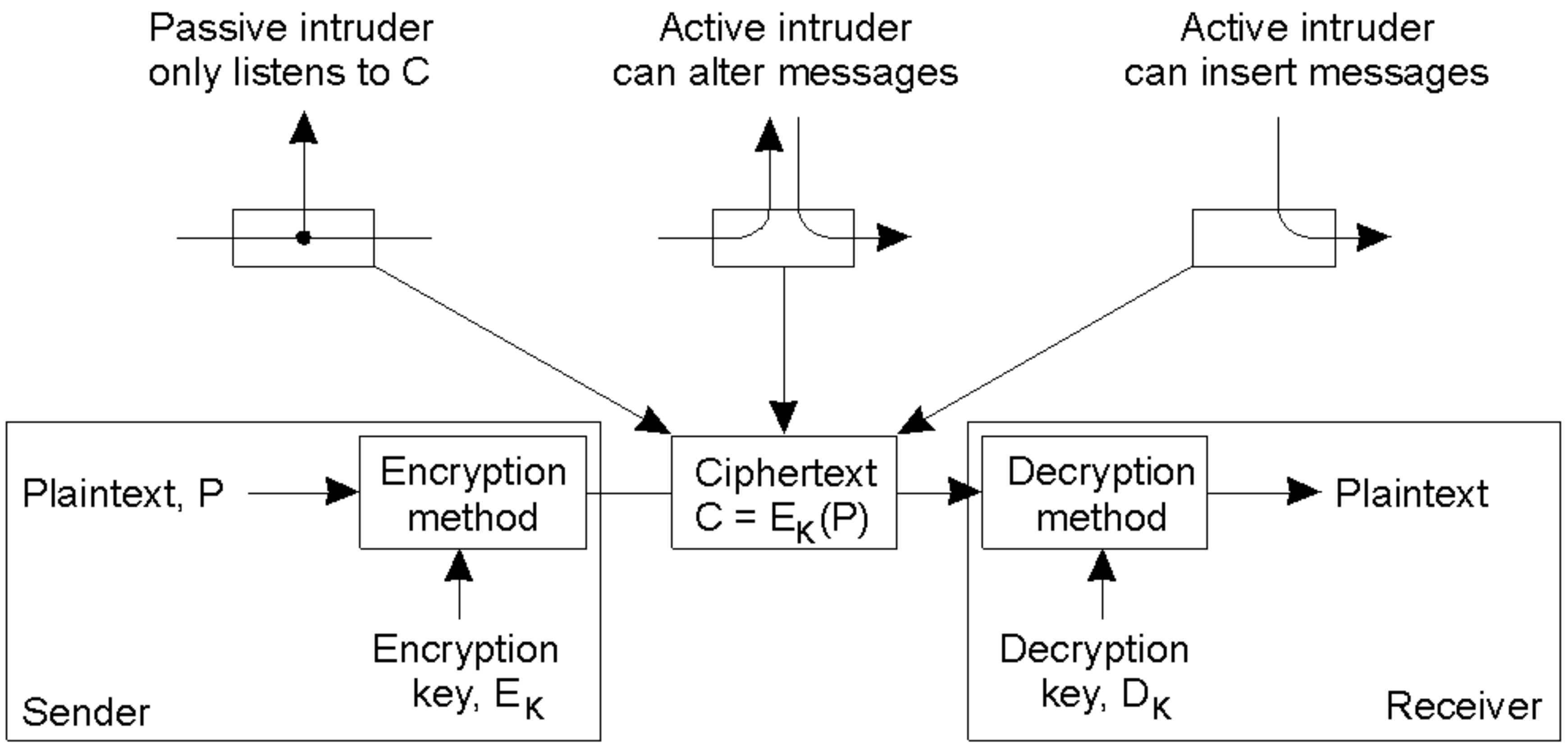
- H(K XOR opad, H(K XOR ipad, M))

where,

- ipad = the byte 0x36 repeated B times
- opad = the byte 0x5C repeated B times
- for a B byte message M.

If optimized, HMAC-**md_alg** can be computed almost efficiently as **md_alg**.

Encryption / Decryption: Threat Sources



Intruders and eavesdroppers in communication

Types of Ciphers

Symmetric Ciphers

- Use same key for encryption and decryption.
- Come in two main types:
  - Block ciphers: Operate on block (several characters, typically 8 or 16 octets) at once.
  - Stream ciphers: Operate on a single bit (or occasionally byte) at a time.
  - NOTE: Block ciphers may operate as stream ciphers, depending on the "cipher mode".

Asymmetric Ciphers (AKA, public/private key)

- Two keys; one for encryption, one for decryption (or more commonly, one private, one public).

Basic Steps for Encryption

- Choose a cipher algorithm
- Choose a key size
- Choose a cipher mode
- Choose a padding scheme
- Key management
  - Key Generation
  - Handling crypto keys

Recommended Cipher Algorithms

Symmetric Block Cipher	■ AES (a no-brainer)
Symmetric Streaming Cipher	■ A protocol: TLS 1.2 or SSHv2 ■ AES in streaming cipher mode (e.g., CTR, OFB)
Asymmetri c Cipher	■ Encryption: RSA or ECC ■ Signing: RSA or DSA

A Word About Key Size

AES encrypts and decrypts w/ 3 standard key lengths:

- 128-bit key length corresponding to approx. 3.4 x 10^{38} keys
- 192-bit key length corresponding to approx 6.2 x 10^{57} keys
- 256-bit key length corresponding to approx. 1.1 x 10^{77} keys

Compare to DES with approx. 7.2 x 10^{16} keys

Recommendations for Cipher Key Size

AES:

- 128-bit is more than adequate for 95% of the choices
- Use 256-bit if you must or if you are ultra-paranoid

RC4 (if you must)

- 128-bit should suffice and often is max supported

RSA or DSA

- At least 1024-bit modulus recommended
- 2048-bit now preferred by NIST

ECC

- It's much more complicated; lots of nuances.
  - At least 224-bits
  - Follow NIST recommendations in FIPS 186-3

Good resource: http://www.keylength.com/

Details on Cipher Modes

What is a cipher mode?

Cover strengths and weaknesses of common cipher modes.

Cover cryptographic attacks specific to cipher modes.

Details on Cipher Modes

An operation, generally performed on block ciphers, that is used to strengthen the security of the resulting ciphertext

Using the cipher “as-is” with no additional block operations is known as “Electronic Code Book” or ECB mode

For the next few slides discussing cipher modes, (+) will be used to indicate an advantage and (-) will be used to indicate a disadvantage

Cipher Modes – A Summary

Cipher Mode	Supports Authenticity?	Acts as Stream Cipher?	Notes
ECB	No	No	Very weak; avoid if possible; FIPS 140-1
CBC	No	No	ESAPI default. FIPS 140-1
CFB	No	Yes	FIPS 140-1
OFB	No	Yes	FIPS 140-1
CTR	No	Yes	
CCM	Yes	Yes	Only defined for ciphers with 128-bit block size; slow(er): 2 cipher operations required. NIST & FIPS approved.
GCM	Yes	Yes	Only defined for ciphers with 128-bit block size; efficient: 1 cipher operation plus 128-bit multiply required. NIST & FIPS approved.
CWC	Yes	Yes	Never reuse {key, IV} pair. Probably more efficient than CCM.
EAX	Yes	Yes	Mostly used w/ Full Disk Encryption (AEAD)
CMAC	Yes	No	Authenticity only. No encryption. Block cipher equivalent of HMAC.

When To Use Padding

Use padding when:

- You are programming to use block cipher in a non-streaming mode (e.g., CBC, GCM) and must accept input of a variable length.

You can skip padding when:

- You are using a block cipher in streaming mode (e.g., OFB, CFB) or using a streaming cipher (e.g., RC4).
- Your plaintext data is always a fixed length or an even multiple of the cipher block size.

When in doubt, use padding.

Recommended Padding Schemes

Use “no padding” option only when applicable.

Symmetric block ciphers

- PKCS7 (RFC 5652)
  - PKCS5 padding is the same as PKCS7, but technically limited to ciphers with 64-bit block size.
- SSL3Padding

Asymmetric block ciphers

- OAEPWithSHA-256AndMGF1Padding
- Note: PKCS1 padding *should be avoided!*

Some Final Recommendations

Here are some good overall recommendations:

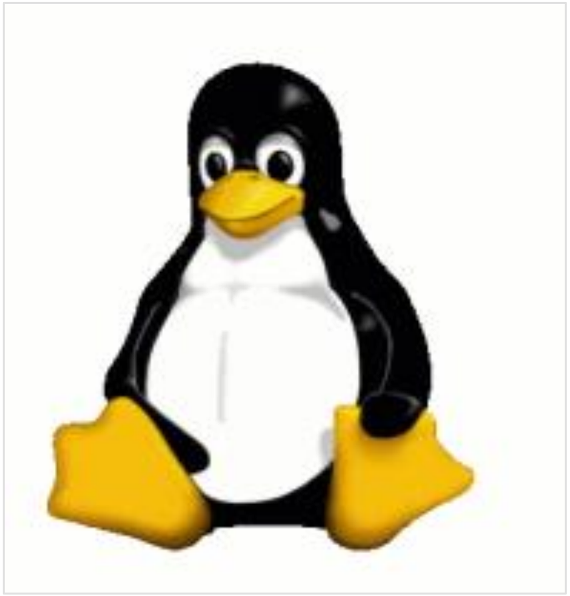
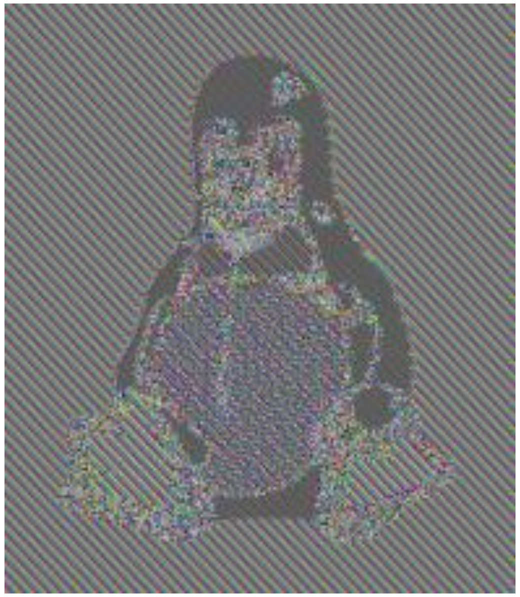
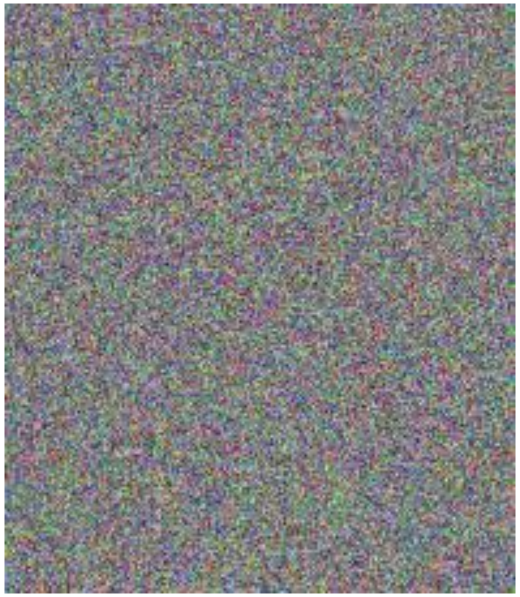
Symmetric block ciphers

- Use random IV for **each** encryption.
- 128-bit AES in GCM mode and no padding (AES/GCM/NoPadding, available in Bouncy Castle)
- 128-bit AES in CBC mode and PKCS5 padding (AES/CBC/PKCS5Padding) and HMAC over IV+ciphertext
  - ESAPI 2.0 default, with HMAC added to ensure authenticity

Asymmetric block ciphers

- RSA/ECB/OAEPWithSHA-256AndMGF1Padding with > 1024-bit modulus.

What's Wrong with ECB Mode?

Original Tux image	Tux image encrypted with ECB mode	Tux image encrypted with any other cipher mode
		

Aside: Padding Oracle Attack (1/2)

What is it?

- First described in 2002 in context of IPSec by Serge Vaudenay
- Attack on CBC mode of operation where “oracle” leaks info whether or not padding of the ciphertext is correct.
- “Oracle” typically is either different error messages / exceptions being returned or timing side-channel attack.

Aside: Padding Oracle Attack (2/2)

So what's the harm?

- Allows adversary to decrypt (and encrypt) data without knowledge of the secret key.
- Is efficient: Works without a large "work factor"

Reference: Brian Holyfield's NYC OWASP presentation:

http://blog.gdssecurity.com/storage/presentations/Padding_Oracle_OWASP_NYC.pdf

Asymmetric Ciphers (1/3)

Different key to encrypt and decrypt

- Generally can encrypt with either key and decrypt with either key.
- Choose one key to be *private* and the other to be *public*

Partly addresses the key distribution problem

Efficiency

- *Much slower* than symmetric ciphers, so frequently used with symmetric ciphers.
- Longer the key, the slower the algorithm works

Chosen & adaptive plaintext attacks always feasible, since public key always available

Asymmetric Ciphers (2/3)

Algorithms generally based on concept of “trap door” functions found
in number theory

- Problem difficult (time-consuming) to solve if key piece of the puzzle missing.
- Often based on difficult-to-compute inverses

Can provide non-repudiation, which isn’t easily provided by secret key algorithms

Asymmetric Ciphers (3/3)

Research driven by problem with key distribution.

- Secret key distribution not scalable. For N parties, O(N^2) keys needed for point-to-point or O(N) for KDC.
- Secret key distribution prohibitive for spontaneous secure communications between two parties who have previously not met

Original contributors: Ellis and Cocks (GCHQ), and Merkel, Diffie, Hellman

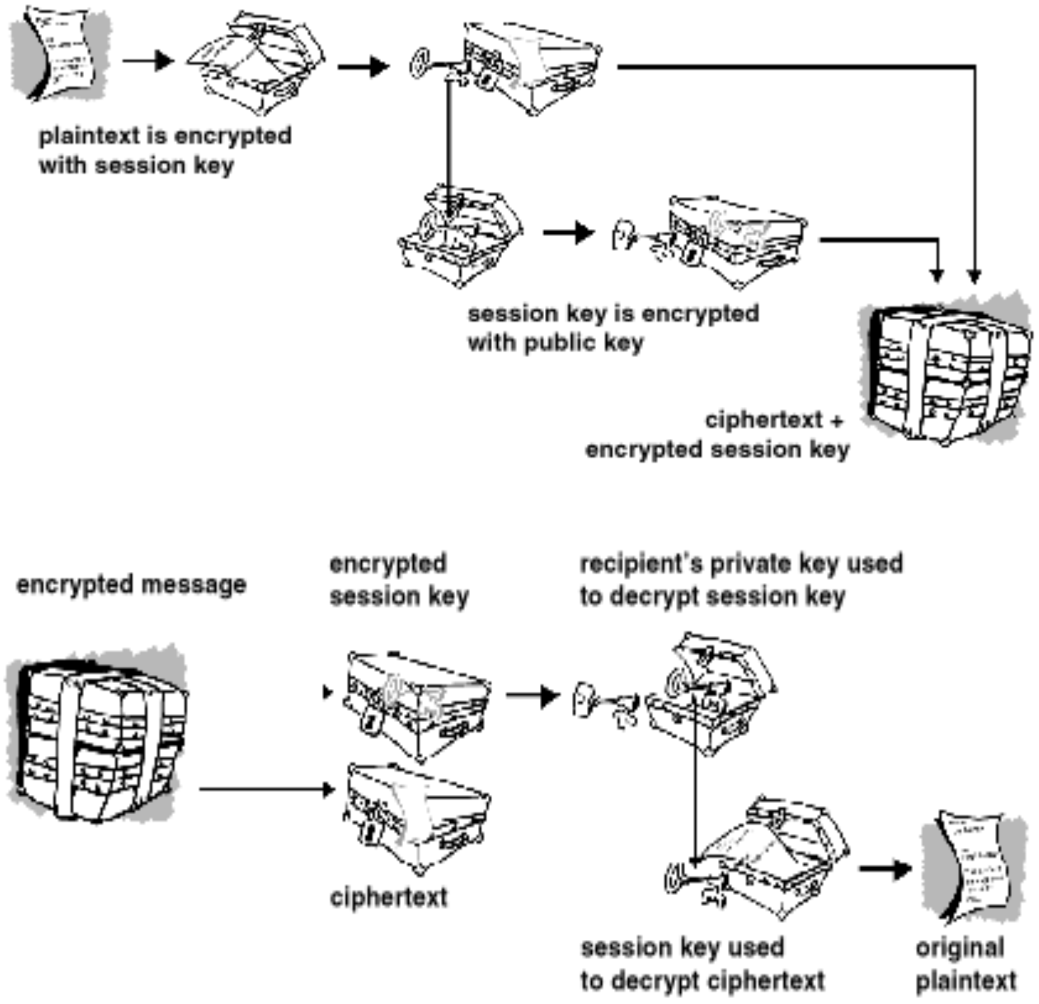
Later contributors: Rivest, Shamir, Adleman

Asymmetric Ciphers: Notation

Because multiple keys, generally involving multiple parties, are used with asymmetric ciphers, cryptographers often use the following notation

Alice's **public** key:	$K^{+}A$
Alice's **private** key:	$K^{-}A$
Bob's **public** key:	$K^{+}B$
Bob's **private** key:	$K^{-}B$

Digital Envelopes


plaintext is encrypted with session key
session key is encrypted with public key
ciphertext + encrypted session key
encrypted message
encrypted session key
recipient's private key used to decrypt session key
ciphertext
session key used to decrypt ciphertext
original plaintext

Any key exchange protocol using a public-key cryptosystem to encrypt a secret (session) key for a secret-key cryptosystem.

Asymmetric Ciphers: Key Generation

Dynamic asymmetric key pair generation rare

- Public key must be exchanged with other participants or signed by some trusted third party.
- Lots of nuances because of relation between public and private keys. Thus use something like openssl or Java’s keytool

Store private key encrypted with passphrase in key store (PKCS#12 recommended)

Asymmetric Ciphers: Encryption (1/2)

Assumption	Alice and Bob both have a public/private key pairs and have securely provided each other their public keys.
Goal	Alice wants to encrypt a symmetric session key, K, to send to Bob.
How?	Alice encrypt's session key K using Bob's public key, K^{+}_{B} Bob decrypt's the ciphertext with his private key, $K\text{-}_{B}$

Asymmetric Ciphers: Encryption (2/2)

Use some form of OAEP padding. Do not use PKCS#1 (v1.5 or earlier) padding	Note that with OAEP padding, there is a length limit to what you may encrypt. Length depends on which secure hash algorithm you use.
You almost always will be encrypting a symmetric session key and only using asymmetric encryption for key exchange	Therefore, ECB mode is the proper cipher mode.

Asymmetric Ciphers: Signing

Assumption	Alice and Bob both have a public/private key pairs and have securely provided each other their public keys.
Goal	Alice wants to sign a message M and send it to Bob.
How?	Alice signs message M using Alice's private key, $K^{-}A$ Bob validates the signature on M using Alice's public key, $K^{+}A$

Asymmetric Ciphers: Summary

You encrypt using the *recipient's public key*

Recipients decrypt using *their private keys*

You sign using your *own private key*

You validate using the *sender's public key*

Mess this up, and it's game over.

Digital Signatures

Digital Signatures: Basic Definitions

Digital signature: A cryptographically unique data string that associates a message with some originating entity.

A **Key Generation Algorithm** creates a public and private key for each User or Entity.

Digital signature generation algorithm: A method for producing a digital signature when given the senders private key and a message.

Digital signature verification algorithm: A method for verifying the authenticity of a message when given the senders public key, the message, and the digital signature.

Digital signature scheme: A signature generation algorithm and an associated verification algorithm.

Digital Signature Issues

There are standard attacks and specialized attacks on digital signatures in general and on specific digital signature schemes in particular. See *Handbook of Applied Cryptography* if interested.

Biggest problem is one of impersonation.

- How can Alice verify that Bob's public key actually belongs to Bob and vice-versa.
- Several easy attacks (MITM, social engineering, etc.)

Digital Signatures: Dealing with the Impersonation Issue (1/3)

Possibilities	
	Alice and Bob exchange public keys on floppy or other device in face-to-face meeting.
	Alice and Bob use trusted courier service.
	Alice and Bob belong to same company and trust their IT dept to make their keys available as part of PKI.
	Alice and Bob exchange over insecure channel (e-mail?), and then confirm fingerprint out-of-band, over more trusted medium (phone?).

Digital Signatures: Dealing with the Impersonation Issue (2/3)

More possibilities

Alice and Bob list their public keys on public places:

- Well-known public key servers
- Usenet news groups
- Archived mailing lists or with every e-mail
- Newspaper classified section
- PGP-like “web of trust” (bottom-up approach)

Digital Signatures: Dealing with the Impersonation Issue (3/3)

Problems with these “solutions”:	■ Not scalable ■ Not suitable for dynamic interaction between two parties who have never previously met. ■ Not fool-proof; still quite susceptible to social engineering attacks
Other problems	■ What if Alice’s or Bob’s *private* key is compromised? What if Alice or Bob just *claim* that it was?

Digital Certificates (1/4)

Digital certificates along with some mutually trusted third party (TTP; often referred to as *Trent* in crypto literature) addresses many of the deficiencies in digital signatures.

In its essence, a digital certificate is an entity's identity (a DN) and it's public key that has been signed by the private key of a TTP (known as Certificate Authority) trusted by anyone willing to use that certificate. It is a mechanism for managing risk by transferring liability.

Digital Certificates: What's in a Certificate? (2/4)

- Entity's identifier (called "distinguished name"). E.g., for a computer, may be fully qualified host name. For person, might be something like:

  cn=Kevin W. Wall,ou=IT,o=Qwest,s=oh,c=us

- Entity's public key.
- Expiration date
- Issuer (Certificate Authority [CA])
- Serial #

Digital Certificates: What Else? (3/4)

X.509 format	
	Version (of X.509)
	Serial # (unique id to prevent reuse of duplicate X.500 names)
	Signature algorithm
	Issuer
	Validity (i.e., expiration)
	Subject's public key info
	Signature

Digital Certificates: But Wait, There's More! (4/4)

Standard format, X.509v3 added a certificate extensions field and defined some standard extensions (critical and non-critical):

- Key and policy information
- Subject and issuer attributes
- Certification path constraints
- Extensions related to certificate revocation lists (CRLs)

Digital Certificates: But Wait, There's More! (4/4)

VERSION 2

Standard format, X.509v3 added a certificate extensions field and defined some standard **extensions** (critical and non-critical):

- Key and policy information
- Subject and issuer attributes
- Certification path constraints
- Extensions related to certificate revocation lists (CRLs)

General Steps To Obtaining A Certificate

- Generate public / private key pair. (Optional; but worry if it’s not!)
- Create CSR (info required varies by CA) and submit it to CA of your choice.
- Install CA’s root certificate if not already in trusted key store.
- Install your certificate. Protect it with a password if that is an option!!!

References (1/3)

Bruce Schneier, **Applied Cryptography: Protocols, Algorithms, and Source Code in C,** 2nd ed., 1996, John Wiley & Sons, ISBN 0-471-11709-9.

Alfred Menezes, Paul van Oorschot, Scott Vanstone, **Handbook of Applied Cryptography,** 1997, CRC Press, ISBN 0-8493-8523-7. (Online: http://cacr.math.uwaterloo.ca/hac/)

F. L. Bauer, **Decrypted Secrets: Methods and Maxims of Cryptology,** 2nd ed., 2000, Springer-Verlag, ISBN 3-540-66871-3.

References (2/3)

Various Wikipedia articles on cryptography

C. Ellison and B. Schneier, Ten Risks of PKI: What You're Not Being Told About Public Key Infrastructure, Computer Security Journal, v 16, n 1, 2000, pp. 1-7.

http://www.counterpane.com/pki-risks.html

References (3/3)

Cryptography Snake-oil FAQ (Matt Curtin, maintainer):

http://www.interhack.net/people/cmcurtin/snake-oil-faq.html

Bruce Schneier, **Why Cryptography Is Harder Than It Looks:**

http://www.counterpane.com/whycrypto.html

Bruce Schneier's Crypto-Gram newsletter:

http://www.counterpane.com/crypto-gram.html