[1st step]Map the Application

1.Explore Visible Content

Configure your browser to use your favorite integrated proxy/spidering tool.
    EX:Burp and WebScarab

if you find it useful, configure your browser to use an extension such as IEWatch to monitor and analyze the HTTP and HTML content being processed by the browser.

visiting every link and URL, submitting every form, and proceeding through all multistep functions to completion.

Try browsing with JavaScript enabled and disabled, and with cookies enabled and disabled.

If the application uses authentication, and you have or can create a login account.

2.Consult Public Resources:

Use Internet search engines and archives (such as the Wayback Machine) to identify what content they have indexed and stored for your target application.

3.Discover Hidden Content:

handles requests for nonexistent items.
Make some manual requests for known valid and invalid resources, and compare the server responses to establish an easy way to identify when an item does not exist.

4.Discover Default Content:

Run Nikto against the web server to detect any default or well-known content that is present.

Use Nikto options to maximize its effective-ness.
For example, you can use the â€“root option to specify a directory.

5.Enumerate Identifier-Specified Functions:

Identify any instances where specific application functions are accessed bypassing an identifier of the function

in a request parameter (for example,
/admin.jsp?action=editUser or /main.php?func=A21)

6.Test for Debug Parameters:

Choose one or more application pages or functions where hidden debug parameters (such as debug=true) may be implemented. These are most likely to appear in key functionality such as login, search, and file upload or download.

===================================

[2nd step]Test Client-Side Controls
===================================

1.Test Transmission of Data Via the Client:

Locate all instances within the application where hidden form fields, cookies, and URL parameters are apparently being used to transmit data via the client.

Modify the item's value in ways that are relevant to its role in the application's functionality.

If the application uses the ASP.NET ViewState , test to confirm whether this can be tampered with or whether it contains any sensitive information.

2.Test Client-Side Controls Over User Input:

Identify any cases where client-side controls such as length limits and JavaScript checks are used to validate user input before it is submitted to the server. These controls can be bypassed easily, because you can send arbitrary requests to the server.
example:

```
<form action="order.asp" onsubmit="return Validate(this)">
<input maxlength="3" name="quantity">
```

3.Test each affected input field in turn by submitting input

HTML form to identify any disabled elements, such as grayed-out submit buttons. For example:

```
<input disabled="true" name="product">
```


4.Test Browser Extension Components:

Understand the Client Application's Operation

Decompile the Client
Attach a Debugger
Test ActiveX controls

===

[3rd step]Test the Authentication Mechanism

===

1.Test Password Quality:
Review the application for any description of the minimum quality rules enforced on user passwords

Attempt to set various kinds of weak passwords, using any self-registration or password change functions to establish the rules actually enforced.

Establish the authentication technologies in use (for example, forms, certificates, or multifactor).

Attempt to log in using different variations on this password, by removing the last character, by changing a character’s case, and by removing any special characters. If any of these login attempts is successful, continue experimenting systematically to identify what validation is actually being performed.

2.Test for Username Enumeration:

Review every detail of the server’s responses to each pair of requests, including the HTTP status code, any redirects, information displayed on-screen, any differences hidden in the HTML page source, and the time taken for the server to respond.

Locate any subsidiary authentication that accepts a username, and determine whether it can be used for username enumeration.

3.Test Any Account Recovery Function:
the application contains any facility for users to regain control of their account if they have forgotten their credentials.

If the function uses a password hint, perform the same exercise to harvest a list of password hints, and identify any that appear to be easily guessable.

4.Test Any Remember Me Function
Test Any Impersonation Function
Test Username Uniqueness
Test Predictability of Autogenerated Credentials

Check for Unsafe Transmission of Credential
Check for Unsafe Distribution of Credentials
Test for Insecure Storage

5.Test for Logic Flaws
Test for Fail-Open Conditions
Test Any Multistage Mechanisms
Exploit Any Vulnerabilities to Gain
Unauthorized Access

===

[4th step]Test the Session Management Mechanism

===

1.Test Tokens for Meaning:
Log in as several different users at different times, and record the tokens
received from the server. If self-registration is available and you can choose
your username.
Analyze the tokens you receive for any correlations that appear to be
related to the username and other user-controllable data.

2.Test Tokens for Predictability:
If you identify any patterns, capture a second sample of tokens using
a different IP address and a different username. This will help you
identify whether the same pattern is detected and whether tokens
received in the first exercise could be extrapolated to guess tokens
received in the second.

3.Check for Insecure Transmission of Tokens:

starting with unauthenticated
content at the start URL, proceeding through the login process, and
then going through all the application’s functionality.

4.If HTTP cookies are being used as the transmission mechanism for session tokens,
verify whether the secure flag is set, preventing them from
ever being transmitted over HTTP connections.

If the HTTPS area of the application contains any links to HTTP URLs,
follow these and verify whether the session token is submitted. If it is,
determine whether it continues to be valid or is immediately terminated
by the server.

5.Check for Disclosure of Tokens in Logs:

If they are intended for administrators only, determine whether any other vulnerabilities exist that could enable a lower-privileged user to access them.
Identify any instances where session tokens are transmitted within the URL.If so, these may be transmitted in the Referer header when users follow any off-site links.

6.Check Mapping of Tokens to Sessions:

Log in and log out several times using the same user account, either from different browser processes or from different computers. Determine whether a new session token is issued each time, or whether the same token is issued every time the same account logs in.
In this situation,there is no way to protect against concurrent logins or properly enforce session timeout.

7.Check for CSRF:

If the application relies solely on HTTP cookies as its method of transmitting session tokens, it may be vulnerable to cross-site request forgery attacks.

8.Check Cookie Scope:

If the application uses HTTP cookies to transmit session tokens (or any other sensitive data), review the relevant Set-Cookie headers, and check for any domain or path attributes used to control the scope of the cookies.

===

[5th step]Test Access Controls & Understand the Access Control Requirements

===

1.Test with Multiple Accounts:

use a less-privileged account and attempt to access each item of this functionality.
Using Burp, browse all the application’s content within one user context.

2.Test with Limited Access:
Many common vulnerabilities will
be much harder to locate, because you do not know the names of the URLs,
identifiers, and parameters that are needed to exploit the weaknesses.

Decompile all compiled clients that are present, and extract any references to server-side functionality.

3.Test for Insecure Access Control Methods:

Some applications implement access controls based on request
parameters in an inherently unsafe way.

Some applications base access control decisions on the HTTP Referer
header.

If HEAD is an allowed method on the site, test for insecure container-managed access control to URLs. Make a request using the HEAD method
to determine whether the application permits it.

===

[6th step]Test for Input-Based Vulnerabilities & Fuzz All Request Parameters

===

1.Test for SQL Injection:

If any database error messages were returned, investigate their meaning.
Use the section “SQL Syntax and Error Reference” in to help
interpret error messages on some common database platforms.

If submitting a single quotation mark in the parameter causes an error
or other anomalous behavior, submit two single quotation marks

Try using common SQL string concatenator functions to construct a string
that is equivalent to some benign input

‘||’FOO
‘+’FOO
‘ ‘FOO
If the application’s logic can be systematically manipulated in this
way,
it is almost certainly vulnerable to SQL injection.

2.Test for XSS and Other Response Injection:

Identify Reflected Request Parameters

Test for Reflected XSS
Test for HTTP Header Injection
Test for Stored Attacks

3.Test for Path Traversal:

The application may be checking the file extension being requested and allowing access to only certain kinds of files. Try using a null byte or newline attack together with a known accepted file extension in an attempt to bypass the filter. For example:

```
../../../../../boot.ini%00.jpg
../../../../../etc/passwd%0a.jpg
```

4.Test for Script Injection:

If the application appears to be vulnerable, verify this by injecting further commands specific to the scripting platform in use. For example, you can use attack payloads similar to those used when fuzzing for OS command injection:

```
system('ping%20127.0.0.1')
```

5.Test for File Inclusion:

If you received any incoming HTTP connections from the target application's infrastructure during your fuzzing, the application is almost certainly vulnerable to remote file inclusion. Repeat the relevant tests in a single-threaded and time-throttled way to determine exactly which parameters are causing the application to issue the HTTP requests.

[7th step]Test for Function-Specific Input Vulnerabilities

1.Test for SMTP Injection:

For each request employed in e-mail-related functionality, submit each of the following test strings as each parameter in turn, inserting your own e-mail address at the relevant position.

You can use Burp Intruder to automate this, as described for general fuzzing. These test strings already have special characters URL-encoded, so do not apply any additional encoding to them.

```
<youremail>%0aCc:<youremail>
<youremail>%0d%0aCc:<youremail>
%0aDATA%0afoo%0a%2e%0aMAIL+FROM:+<youremail>%0aRCPT+TO:+<youremail>
```

Review the results to identify any error messages the application returns.

Test for Native Software Vulnerabilities:
1.Test for Buffer Overflows
2.Test for Integer Vulnerabilities
3.Test for Format String Vulnerabilities

2.Test for SOAP Injection:

Target each parameter in turn that you suspect is being processed via a SOAP message. Submit a rogue XML closing tag, such as </foo> . If no error occurs, your input is probably not being inserted into a SOAP message or is being sanitized in some way.

SOAP message, which may change the application’s logic or result in a different error condition that may divulge information.

3.Test for LDAP Injection:

Submit the * character. If a large number of results are returned, this is a good indicator that you are dealing with an LDAP query.

Try entering a number of closing parentheses:

```
))))))))))
```

Having extracted the name of the parent node, use a series of conditions with the following form to extract all the data within the XML tree:

```
substring(//parentnodename[position()=1]/child::node()[position()=1]
/text(),1,1)=’a
```

4.Test for Back-End Request Injection:

Target a request parameter that returns a specific page for a specific value, and try to append a new injected parameter using various syntax, including the following:

```
%26foo%3dbar (URL-encoded &foo=bar )
%3bfoo%3dbar (URL-encoded ;foo=bar )
%2526foo%253dbar (Double URL-encoded &foo=bar )
```

If the application behaves as if the original parameter wee unmodified, there is a chance of HTTP parameter injection vulnerabilities.

5.Test for XXE Injection:

If users are submitting XML to the server, an external entity injection attack may be possible. If a field is known that is returned to the user, attempt to specify an external entity, as in the following example:

```
POST /search/128/AjaxSearch.ashx HTTP/1.1
Host: mdsec.net
Content-Type: text/xml; charset=UTF-8
Content-Length: 115
<!DOCTYPE foo [ <!ENTITY xxe SYSTEM “file:///windows/win.ini” > ]>
<Search><SearchTerm>&xxe;</SearchTerm></Search>
```

“http://192.168.1.1:25” and monitor the page response time.

==============================
[8th step]Test for Logic Flaws
==============================

1.Identify the Key Attack Surface

Logic flaws can take a huge variety of forms and exist within any aspect of the application’s functionality. To ensure that probing for logic flaws is feasible, you should first narrow down the attack surface to a reasonable area for manual testing.

Identify any instances of the following features:
1. Multistage processes
2.Critical security functions, such as login
3.Transitions across trust boundaries (for example, moving from being
4.anonymous to being self-registered to being logged in)
5.Context-based functionality presented to a user
6.Checks and adjustments made to transaction prices or quantities

---

2.Test Multistage Processes:

The sequence of stages may be accessed via a series of GET or POST requests for distinct URLs, or they may involve submitting different sets of parameters to the same URL. You may specify the stage being requested by submitting a function name or index within a request parameter.

3.Test Handling of Incomplete Input:

For critical security functions within the application, which involve processing several items of user input and making a decision based on these, test the application’s resilience to requests containing incomplete input.

4.Test Transaction Logic:

the application imposes transaction limits, test the effects of submitting negative values. If these are accepted, it may be possible to beat the limits by making large transactions in the opposite direction.

[9th step]Test for Shared Hosting Vulnerabilities

1.Test Segregation in Shared Infrastructures:

If the application is hosted in a shared infrastructure, examine the access mechanisms provided for customers of the shared environment to update and manage their content and functionality.

If you can achieve command execution, SQL injection, or arbitrary file access within one application, investigate carefully whether this provides any way to escalate your attack to target other applications

----

2.Test Segregation Between ASP-Hosted Applications:

If the application belongs to an ASP-hosted service composed of a mix of shared and customized components, identify any shared components such as logging mechanisms, administrative functions, and database code components.

[10th step]Test for Application Server Vulnerabilities

1.Test for Default Credentials:

Review the results of your application mapping exercises to identify the web server and other technologies in use that may contain accessible administrative interfaces.

If you gain access to an administrative interface, review the available functionality and determine whether it can be used to further compromise the host and attack the main application

----

2.Test for Default Content:

Review the results of your Nikto scan to identify any default content that may be present on the server but that is not an integral part of the application.

Use search engines and other resources such as www.exploit-db.com and www.osvdb.org to identify default content and functionality included within the technologies you know to be in use.

3.Test for Dangerous HTTP Methods:
Use the OPTIONS method to list the HTTP methods that the server states are available. Note that different methods may be enabled in different directories. You can perform a vulnerability scan in Paros to perform this check.

Try each reported method manually to confirm whether it can in fact be used.

4.Test for Proxy Functionality:

Using both GET and CONNECT requests, try to use the web server as a proxy to connect to other servers on the Internet and retrieve content from them.

Using both GET and CONNECT requests, attempt to connect to common port numbers on the web server itself by specifying 127.0.0.1 as the target host in the request.

5.Test for Virtual Hosting Misconfiguration:
Submit GET requests to the root directory using the following:
The correct Host header
A bogus Host header

The server's IP address in the Host header
No Host header (use HTTP/1.0 only)

Nikto scan using the -vhost option to identify any default content that may have been overlooked during initial application mapping.

6.Test for Web Server Software Bugs:
Review resources such as Security Focus, Bugtraq, and Full Disclosure to find details of any recently discovered vulnerabilities that may not have been fixed on your target.

If the application was developed by a third party, investigate whether it ships with its own web server (often an open source server).

If it does,investigate this for any vulnerabilities. Be aware that in this case, the server's standard banner may have been modified.

7.Test for Web Application Firewalling:

For all fuzzing strings and requests, use payload strings that are unlikely to exist in a standard signature database. Although giving examples of these is by definition impossible, avoid using /etc/passwd or /windows/system32/config/sam as payloads for file retrieval. Also avoid using terms such as <script> in an XSS attack and using alert() or xss as XSS payloads.

On ASP.NET, also try submitting the parameter as a cookie. The API Request.Params["foo"] will retrieve the value of a cookie named foo if the parameter foo is not found in the query string or message body.

===============================

[11th step]Miscellaneous Checks

===============================

1.Check for DOM-Based Attacks:
Perform a brief code review of every piece of JavaScript received from the application. Identify any XSS or redirection vulnerabilities that can be triggered by using a crafted URL to introduce malicious data into the DOM of the relevant page. Include all standalone JavaScript files and scripts contained within HTML pages

----

2.Identify all uses of the following APIs, which may be used to access DOM data that can be controlled via a crafted URL:

```
document.location
document.URL
document.URLUnencoded
```

3.APIs, the application may be vulnerable to XSS:

```
document.write()
document.writeln()
document.body.innerHtml
eval()
window.execScript()
window.setInterval()
```

4.Check for Local Privacy Vulnerabilities:
logs created by your intercepting proxy to identify all the Set-Cookie directives received from the application during your testing. if any of these contains an expires attribute with a date that is in the future, the cookie will be stored by users' browsers until that date.
Review the contents of any persistent cookies for sensitive data.

Identify any instances within the application in which sensitive data is transmitted via a URL parameter. If any cases exist, examine the browser history to verify that this data has been stored there.

5.Check for Weak SSL Ciphers:

If the application uses SSL for any of its communications, use the tool THCSSLCheck to list the ciphers and protocols supported.

Opera browser to attempt to perform a complete handshake using specified weak protocols to confirm whether these can actually be used to access the application.

6.Check Same-Origin Policy Configuration:

Test an application's handling of cross-domain requests using XMLHttpRequest by adding an Origin header specifying a different domain and examining any Access-Control headers that are returned. The security implications of allowing two-way access from any domain, or from specified other domains, are the same as those described for the Flash cross-domain policy.

[12th step]Follow Up Any Information Leakage

If you receive any unusual error messages, investigate these using standard search engines. You can use various advanced search features to narrow down your results.

example:

```
"unable to retrieve" filetype:php
```

Use Google code search to locate any publicly available code that may be responsible for a particular error message.

Search for snippets of error messages that may be hard-coded into the application's source code. You can also use various advanced search features to specify the code language and other details, if these are known.

```
unable\ to\ retrieve lang:php package:mail
```
