وَقُل رَّبِّ زِدْنِي عِلْمًا

Chapter 21: A Web Application Hacker’s Methodology by @sl4x0

بِسْمِ اللَّهِ الرَّحْمَنِ الرَّحِيمِ

- Information gathered in one stage may enable you to return to an earlier stage and formulate more focused attacks. For example, an access control bug that enables you to obtain a **listing of all users** may enable you to perform a more effective **password-guessing** attack against the authentication function.
- Discovering a key vulnerability in one area of the application may enable you to shortcut some of the work in other areas. For example, a fi le disclosure vulnerability may enable to you perform a **code review of key application functions** rather than probing them in a solely black-box manner.
- The results of your testing in some areas may highlight patterns of recurring vulnerabilities that you can immediately probe for in other areas. For example, a generic defect in the application’s input validation filters may enable you to quickly find a **bypass** of its defenses against several different categories of attack.

General Guidelines

- Remember that several characters have special meaning in different parts of the HTTP request. When you are modifying the data within requests, you should **URL-encode** these characters to ensure that they are interpreted in the way you intend:
- & is used to **separate parameters** in the URL query string and message body. To insert a literal & character, you should encode this as %26.
- = is used to **separate the name and value** of each parameter in the URL query string and message body. To insert a literal = character, you should encode this as %3d.
- ? is used to mark the beginning of the **URL query string**. To insert a literal ? character, you should encode this as %3f.
- A space is used to mark **the end of the URL in the first line of requests** and can indicate the end of a cookie value in the Cookie header. To insert a literal space, you should encode this as %20 or +.
- Because + represents an encoded space, to insert a **literal +** character, you should encode this as %2b.
- ; is used to separate individual cookies in the Cookie header. To insert a **literal ;** character, you should encode this as %3b.
- # is used to mark the **fragment identifier within the URL**. If you enter this character into the URL within your browser, it effectively truncates the URL that is sent to the server. To insert a literal # character, you should encode this as %23.
- % is used as the **prefix in the URL-encoding scheme**. To insert a literal % character, you should encode this as %25.
- Any nonprinting characters such as null bytes and newlines must, of course, be URL-encoded using their ASCII character code — in this case, as **%00** and **%0a**, respectively.

- Furthermore, note that entering URL-encoded data into a form usually causes your browser to perform another layer of encoding. For example, submitting %00 in a form will probably result in a value of %2500 being sent to the server. For this reason it is normally best to observe the **final request within an intercepting proxy.**

- Many tests for common web application vulnerabilities involve sending various crafted input strings and monitoring the application's responses for anomalies, which indicate that a vulnerability is present. In any case where specific crafted input results in behavior associated with a vulnerability (such as a particular
error message), you should **double-check** whether submitting benign input in the relevant parameter also causes the same behavior. If it does, your tentative finding is probably a false positive.
- Applications typically accumulate an amount of state from previous requests, which affects how they respond to further requests. To do so, it is usually sufficient to begin
a **fresh session with a new browser process**, navigate to the location of the observed anomaly using only benign requests, and then resubmit your crafted input. Or you can user Burp Repeater.
- Some applications use a **load-balanced configuration** in which consecutive HTTP requests may be handled by different back-end servers at the web, presentation, data, or other tiers. Different servers may have small differences in configuration that affect your results. To isolate the effects of particular actions, it may be necessary to perform **several identical requests in succession**, testing the result of
each until your request is handled by the relevant server.

`1 Map the Application's Content`

1.1 Explore Visible Content

1.1.1 Both `Burp` and `WebScarab` can be used to passively spider the site by monitoring and parsing web content processed by the proxy.

1.1.2 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.

1.1.3 Browse the entire application in the normal way, 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.

1.1.4 Make an `Account` to test Protected Functions

1.1.5 As you browse, monitor the requests and responses passing through your intercepting proxy to gain an `understanding of the kinds of data being submitted`.

1.1.6 From the spider results, establish where each item was discovered (for example, in Burp Spider, check the `Linked From details`). Access each item using your browser so that the spider parses the response from the server to identify any further content.

1.1.7 Before doing an `automated crawl`, first identify any URLs that are dangerous or likely to
break the application session, and then configure the spider to exclude these from its scope.

1.2 Consult Public Resources

1.2.1 Use the `Wayback Machine` to identify the indexed content

1.2.2 Use `Google Dorking` to discover more content

1.2.3 Review any `published WSDL` fi les to generate a list of function names and parameter values potentially employed by the application.

1.3 Discover Hidden Content

1.3.1 Make some manual requests for known `valid and invalid resources`, and compare the server's responses to establish an easy way to identify when an item does not exist.

1.3.2 Try to understand the naming conventions used by application developers. For example, if
there are pages called `AddDocument.jsp` and `ViewDocument.jsp`, there may also be pages called `EditDocument.jsp` and `RemoveDocument.jsp`

1.3.3 Review the `client side code` and HTML comments

1.3.4 Using the automation techniques described in `Chapter 14`, make large numbers of requests based on your directory, filename, and file extension lists.

1.3.5 Perform these `content-discovery` exercises recursively, using new enumerated content and patterns as the basis for further user-directed spidering and further automated discovery.

1.4 Discover Default Content

1.4.1 Run `Nikto` against the web server to detect any default or well-known content that is present. For example, you can use the `–root` option to specify a directory to check for default content, or `-404` to specify a string that identifies a custom File Not Found page.

1.4.2 Verify any potentially interesting findings `manually` to eliminate any false positives within the results.

1.4.3 Request the server's `root directory`, specifying the IP address in the Host header, and determine if the application responds with any different content. If so, run a `Nikto` scan against the IP address as well as the server name.

1.4.4 Make a request to the server's root directory, specifying a range of `User-Agent headers`, as shown at `www.useragentstring.com/pages/useragentstring.php`.

1.5 Enumerate Identifier-Specified Functions

1.5.1 Identify any instances where specific application functions are accessed by `passing an identifier` of the function in a request parameter (for example, `/admin.jsp?action=editUser` or `/main.php?func=A21`).

1.5.2 Apply the `content discovery` techniques used in step 1.3 to the mechanism being used to access individual functions.

1.5.3 If applicable, `compile a map of application content based on functional paths`, rather than URLs, showing all the enumerated functions and the logical paths and dependencies between them.

1.6 Test for Debug Parameters

1.6.1 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.

1.6.2 Use listings of common debug parameter names (such as debug, test, hide, and source) and common values (such as true, yes, on, and 1). Iterate through all permutations of these, submitting each name/value pair to each targeted function. For POST requests, supply the parameter
in both the `URL query string` and the `request body`. Use the techniques described in Chapter 14 to automate this exercise. For example, you can use the cluster bomb attack type in Burp Intruder to combine all permutations of `two payload lists`.

1.6.3 Review the application's responses for any `anomalies` that may indicate that the added parameter has had an effect on the application's processing.

`2 Analyze the Application`

2.1 Identify Functionality

2.1.1 Identify the core functionality that the application was `created for` and the actions that each function is designed to perform when used as intended.

2.1.2 Identify the core `security mechanisms` employed by the application and how they work. In particular, understand the key mechanisms that handle authentication, session management, and access control, and the functions that support them, such as user registration and account recovery.

2.1.3 Identify all the more `peripheral functions` and behavior, such as the use of redirects, off-site links, error messages, and administrative and logging functions.

2.1.4 Identify any functionality that `diverges from the standard GUI appearance`, parameter naming, or navigation mechanism used elsewhere in the application, and single it out for in-depth testing

2.2 Identify Data Entry Points

2.2.1 Identify all the different entry points that exist for introducing user input into the application's processing, including URLs, query string parameters, POST data, cookies, and other HTTP headers processed by the application.

2.2.2 Examine any `customized data transmission or encoding mechanisms` used by the application, such as a nonstandard query string format. Understand whether the data being submitted encapsulates parameter names and values, or whether an alternative means of representation is being used.

2.2.3 Identify any `out-of-band channels` via which user-controllable or other third-party data is being introduced into the application's processing. An example is a web mail application that processes and renders messages received via `SMTP`.

2.3 Identify the Technologies Used

2.3.1 Identify each of the different technologies used on the client side, such as forms, scripts, cookies, Java applets, ActiveX controls, and Flash objects

2.3.2 As far as possible, establish which technologies are being used on the `server side`, including scripting languages, application platforms, and interaction with back-end components such as databases and e-mail systems.

2.3.3 Check the HTTP `Server` header returned in application responses, and also check for any other software identifiers contained within custom HTTP headers or HTML source code comments.

2.3.4 Run the `Httprint` tool to fingerprint the web server.

2.3.5 Review the results of your content-mapping exercises to identify any interesting-looking file extensions, directories, or other URL subsequences that may provide clues about the technologies in use on the server.

2.3.6 Identify any interesting-looking script names and query string parameters that may belong to third-party code components. Search for these on Google using the `inurl:` qualifier to find any other applications using the same scripts and parameters and that therefore may be using the same third-party components.

2.4 Map the Attack Surface

2.4.1 Try to ascertain the likely internal structure and functionality of the server-side application and the mechanisms it uses behind the scenes to deliver the behavior that is visible from the client perspective. For example, a function to `retrieve customer orders is likely to be interacting with a database.`

2.4.2 For each item of functionality, identify the kinds of common vulnerabilities that are often associated with it. For example, file upload functions may be vulnerable to `path traversal`, inter-user messaging may be vulnerable to XSS, and Contact Us functions may be vulnerable to SMTP injection.

2.4.3 Formulate a plan of attack, prioritizing the most interesting-looking functionality and the most serious of the potential vulnerabilities associated with it.

`3 Test Client-Side Controls`

3.1 Test Transmission of Data Via the Client

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

3.1.2 Attempt to determine the purpose that the item plays in the application's logic, based on the context in which it appears and on its name and value.

3.1.3 Modify the item's value in ways that are relevant to its role in the application's functionality. Determine whether the application processes arbitrary values submitted in the field and whether this fact can be exploited to interfere with the application's logic or subvert
any security controls.

3.1.4 If the application transmits opaque data via the client, you can attack this in various ways. If the item is obfuscated, you may be able to decipher the obfuscation algorithm and therefore submit arbitrary data within the opaque item. Even if it is securely encrypted, you may be able to replay the item in other contexts to interfere with the application's logic.

3.1.5 If the application uses the ASP.NET `ViewState`, test to confirm whether this can be tampered with or whether it contains any sensitive information. Note that the `ViewState` may be used differently on different application pages.

3.1.5.1 Use the `ViewState analyzer` in Burp Suite to confirm whether the `EnableViewStateMac` option has been enabled, meaning that the `ViewState’s contents` cannot be modified.
3.1.5.2 Review the decoded `ViewState` to identify any sensitive data it contains.
3.1.5.3 Modify one of the decoded parameter values and reencode and submit the `ViewState.` If the application accepts the modified value, you should treat the `ViewState` as an input channel for introducing arbitrary data into the application’s processing.
Perform the same testing on the data it contains as you would for any other request parameters.

3.2 Test Client-Side Controls Over User Input

3.2.1 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. For example:

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

3.2.2 Test each `affected input field` in turn by submitting input that would ordinarily be blocked by the client-side controls to verify whether these are replicated on the server.

3.2.3 The ability to bypass client-side validation does not necessarily represent any vulnerability. Nevertheless, you should review closely what validation is being performed. Confirm whether the application is relying on the client-side controls to protect itself from malformed input. Also confirm whether any exploitable conditions exist that can be triggered by such input.

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

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

If you find any, submit these to the server, along with the form's other parameters. See whether the parameter has any effect on the server's processing that you can leverage in an attack. Alternatively, use an automated proxy rule to automatically enable disabled fields, such as
Burp Proxy's "HTML Modification" rules.

3.3 Test Browser Extension Components

3.3.1 Understand the Client Application's Operation

3.3.1.1 Set up a local intercepting proxy for the client technology under review, and monitor all traffic passing between the client and server. If data is serialized, use a deserialization tool such as Burp's built-in AMF support or the `DSer` Burp plug-in for Java.

3.3.1.2 Step through the functionality presented in the client. Determine any potentially sensitive or powerful functions, using standard tools within the intercepting proxy to replay key requests or modify server responses.

3.3.2 Decompile the Client

3.3.2.1 Identify any applets employed by the application. Look for any of the following file types being requested via your intercepting proxy:

- `.class`, `.jar` : Java
- `.swf` : Flash
- `.xap` : Silverlight

You can also look for `applet tags` within the HTML source code of application pages. For example:

```
<applet code="input.class" id="TheApplet" codebase="/scripts/"></
applet>
```

3.3.2.2 Review all calls made to the `applet's methods` from within the invoking HTML, and determine whether data returned from the applet is being submitted to the server. If this data is opaque (that is, obfuscated or encrypted), to modify it you will probably need to `decompile` the applet to obtain its source code.

3.3.2.3 Download the applet bytecode by entering the URL into your browser, and save the file locally. The name of the bytecode fi le is specified in the code attribute of the applet tag. The file will be located in the directory specified in the codebase attribute if this is present. Otherwise, it
will be located in the same directory as the page in which the applet tag appears.

3.3.2.4 Use a suitable tool to decompile the bytecode into source code. For example:
`C:\>jad.exe input.class`

`Parsing input.class... Generating input.jad`

Here are some suitable tools for decompiling different browser extension components:

- Java — Jad
- Flash — SWFScan, Flasm/Flare
- Silverlight — .NET Reflector

3.3.2.6 Determine whether the applet contains any public methods that can be used to perform the relevant obfuscation on arbitrary input.

3.3.2.7 If it doesn't, modify the applet's source to neutralize any validation it performs or to allow you to obfuscate arbitrary input. You can then recompile the source into its original fi le format using the compilation tools provided by the vendor.

3.3.3 Attach a Debugger

3.3.3.1 For large client-side applications, it is often prohibitively difficult to decompile the whole application. For these applications it is generally quicker to attach a runtime debugger to the process. `JavaSnoop` does this very well for Java. `Silverlight Spy` is a freely available tool that allows runtime monitoring of Silverlight clients

3.3.3.2 Locate the key functions and values the application employs to drive security-related business logic, and place breakpoints when the targeted function is called. Modify the arguments or return value as needed to affect the security bypass.

3.3.4 Test ActiveX controls

3.3.4.1 Identify any ActiveX controls employed by the application. Look for any .cab fi le types being requested via your intercepting proxy, or look for object tags within the HTML source code of application pages. For example:

```
<OBJECT
 classid="CLSID:4F878398-E58A-11D3-BEE9-00C04FA0D6BA"
 codebase="https://wahh app.com/scripts/input.cab"
 id="TheAxControl">
</OBJECT>
```

3.3.4.2 It is usually possible to subvert any input validation performed within an ActiveX control by attaching a debugger to the process and directly modifying data being processed or altering the program's execution path.

3.3.4.3 It is often possible to guess the purpose of different methods that an ActiveX control exports based on their names and the parameters passed to them. Use the `COMRaider` tool to enumerate the methods exported by the control. Test whether any of these can be manipulated to affect the control's behavior and defeat any validation tests it implements.

3.3.4.4 If the control's purpose is to gather or verify certain information about the client computer, use the `Filemon` and `Regmon` tools to monitor the information the control gathers.

3.3.4.5 Test any ActiveX controls for vulnerabilities that could be exploited to attack other users of the application. You can modify the HTML used to invoke a control to pass arbitrary data to its methods and monitor the results. Look for methods with dangerous-sounding names, such as `LaunchExe`. You can also use `COMRaider` to perform some basic fuzz testing of ActiveX controls to identify flaws such as buffer overflows.

`4 Test the Authentication Mechanism`

4.1 Understand the Mechanism

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

4.1.2 Locate all the authentication-related functionality (including login, registration, account recovery, and so on).

4.1.3 If the application does not implement an automated self-registration mechanism, determine whether any other means exists of obtaining several user accounts.

4.2 Test Password Quality

4.2.1 Review the application for any description of the minimum quality rules enforced on user passwords.

4.2.2 Attempt to set various kinds of weak passwords, using any self-registration or password change functions to establish the rules actually enforced. Try short passwords, alphabetic characters only, single-case characters only, dictionary words, and the current username.

4.2.3 Test for incomplete validation of credentials. Set a strong and complex password (for example, 12 characters with mixed-case letters, numerals, and typographic characters). 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.

4.2.4 Having established the minimum password quality rules, and the extent of password validation, identify the range of values that a `passwordguessing` attack would need to employ to have a good probability of success.

4.3 Test for Username Enumeration

4.3.1 Identify every location within the various authentication functions where a `username` is submitted, including via an on-screen input field, a hidden form field, or a cookie. Common locations include the primary login, self-registration, password change, logout, and account recovery.

4.3.2 For each location, submit two requests, containing a valid and an invalid username. 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.

4.3.3 If you observe any differences between the responses where a valid and invalid username is submitted, repeat the test with a different pair of values and confirm that a systematic difference exists that can provide a basis for automated username enumeration.

4.3.4 Check for any other sources of information leakage within the application that may enable you to compile a list of valid usernames. Examples are `logging functionality`, actual listings of `registered users`, and direct mention of names or e-mail addresses in source code comments.

4.3.5 Locate any subsidiary authentication that accepts a username, and determine whether it can be used for username enumeration. Pay specific attention to a `registration page` that allows specification of a username.

4.4 Test Resilience to Password Guessing

4.4.1 Identify every location within the application where `user credentials` are submitted. The two main instances typically are the main login function and the password change function.

4.4.2 At each location, using an account that you control, manually send several requests containing the valid username but other invalid credentials. Monitor the application's responses to identify any differences. After about 10 failed logins, if the application has not returned a message about account lockout, submit a request containing valid credentials. If this request succeeds, an account lockout policy probably is not in force.

4.4.3 If you do not control any accounts, attempt to enumerate or guess a valid username, and make several invalid requests using this guess, monitoring for any error messages about account lockout.

4.5 Test Any Account Recovery Function

4.5.1 Identify whether the application contains any facility for users to regain control of their account if they have forgotten their credentials.

4.5.2 Establish how the account recovery function works by doing a `complete walk-through` of the recovery process using an account you control.

4.5.3 If the function uses a challenge such as a secret question, determine whether users can set or select their own challenge during registration.
If so, use a list of enumerated or common usernames to `harvest a list of challenges`, and review this for any that appear to be easily guessable.

4.5.4 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.

4.5.5 Perform the same tests on any account-recovery challenges that you performed at the main login function to assess vulnerability to automated guessing attacks.

4.5.6 If the function involves sending an e-mail to the user to complete the recovery process, look for any weaknesses that may enable you to take control of other users' accounts. Determine whether it is possible to control the address to which the e-mail is sent. If the message contains a unique recovery URL, obtain a number of messages using an e-mail address you control, and attempt to identify any patterns that may enable you to predict the URLs issued to other users. Apply the methodology described in step 5.3 to identify any predictable sequences.

4.6 Test Any Remember Me Function

4.6.1 If the main login function or its supporting logic contains a `Remember Me function`, activate this and review its effects. If this function allows the user to log in on subsequent occasions without entering any credentials, you should review it closely for any vulnerabilities.

4.6.2 Closely inspect all persistent `cookies` that are set when the Remember Me function is activated. Look for any data that identifies the user explicitly or appears to contain some predictable identifier of the user.

4.6.3 Even where the data stored appears to be heavily encoded or obfuscated, review this closely, and compare the results of remembering several very `similar usernames and/or passwords` to identify any opportunities to reverse-engineer the original data. Apply the methodology described in step 5.2 to identify any meaningful data.

4.6.4 Depending on your results, modify the contents of your cookie in suitable ways in an attempt to masquerade as other users of the application.

4.7 Test Any Impersonation Function

4.7.1 If the application contains any explicit functionality that allows one user to impersonate another, review this closely for any vulnerabilities that may enable you to impersonate arbitrary users without proper authorization.

4.7.2 Look for any user-supplied data that is used to determine the target of the impersonation. Attempt to manipulate this to impersonate other users, particularly administrative users, which may enable you escalate privileges.

4.7.3 If you perform any automated password-guessing attacks against other user accounts, look for any accounts that appear to have more than one valid password, or multiple accounts that appear to have the same password. This may indicate the presence of a backdoor password, which administrators can use to access the application as any user.

4.8 Test Username Uniqueness

4.8.1 If the application has a self-registration function that lets you specify a desired username, attempt to register the same username twice with different passwords.

4.8.2 If the application blocks the second registration attempt, you can exploit this behavior to enumerate registered usernames.

4.8.3 If the application registers both accounts, probe further to determine its behavior when a collision of username and password occurs. Attempt to change the password of one of the accounts to match that of the other. Also, attempt to register two accounts with identical usernames and passwords.

4.8.4 If the application alerts you or generates an error when a collision of username and password occurs, you can probably exploit this to perform an automated guessing attack to discover another user's password. Target an enumerated or guessed username, and attempt to create accounts that have this username and different passwords. When the application rejects a specific password, you have probably found the existing password for the targeted account.

4.8.5 If the application appears to tolerate a collision of username and password without an error, log in using the colliding credentials. Determine what happens and whether the application's behavior can be leveraged to gain unauthorized access to other users' accounts.

4.9 Test Predictability of Autogenerated Credentials

4.9.1 If the application automatically generates usernames or passwords, try to obtain several values in quick succession and identify any detectable sequences or patterns.

4.9.2 If usernames are generated in a predictable way, extrapolate backwards to obtain a list of possible valid usernames. You can use this as the basis for automated password-guessing and other attacks.

4.9.3 If passwords are generated in a predictable way, extrapolate the pattern to obtain a list of possible passwords issued to other application users.

4.10 Check for Unsafe Transmission of Credentials

4.10.1 Walk through all authentication-related functions that involve transmission of credentials, including the main login, account registration, password change, and any page that allows viewing or updating of user profile information. Monitor all traffic passing in both directions
between the client and server using your intercepting proxy.

4.10.2 Identify every case in which the credentials are transmitted in either direction. You can set interception rules in your proxy to flag messages containing specific strings.

4.10.3 If credentials are ever transmitted in the URL query string, these are potentially vulnerable to disclosure in the browser history, on-screen,
in server logs, and in the `Referer` header when third-party links are followed.

4.10.4 If credentials are ever stored in a cookie, these are potentially vulnerable to disclosure via `XSS attacks` or local privacy attacks.

4.10.5 If credentials are ever transmitted from the server to the client, these may be compromised via any vulnerabilities in `session management` or `access controls`, or in an `XSS` attack.

4.10.6 If credentials are ever transmitted over an unencrypted connection, these are vulnerable to interception by an `eavesdropper`.

4.10.7 If credentials are submitted using HTTPS but the login form itself is loaded using HTTP, the application is vulnerable to a `man-in-the-middle` attack that may be used to capture credentials.

4.11 Check for Unsafe Distribution of Credentials

4.11.1 If accounts are created via some out-of-band channel, or the application has a self-registration function that does not itself determine all of a user's initial credentials, establish the means by which credentials are distributed to new users. Common methods include sending a message to an e-mail or postal address.

4.11.2 If the application generates account activation URLs that are distributed out-of-band, try to register several new accounts in close succession, and identify any sequence in the URLs you receive. If a pattern can be determined, try to predict the URLs sent to recent and forthcoming users, and attempt to use these URLs to take ownership of their accounts.

4.11.3 Try to reuse a single activation URL multiple times, and see if the application allows this. If it doesn't, try locking out the target account before reusing the URL, and see if the URL still works. Determine whether this enables you to set a new password on an active account.

4.12 Test for Insecure Storage

4.12.1 If you gain access to hashed passwords, check for accounts that share the same hashed password value. Try to log in with common passwords for the most common hashed value.

4.12.2 Use an offline rainbow table for the hashing algorithm in question to recover the cleartext value.

4.13 Test for Logic Flaws

4.13.1 Test for Fail-Open Conditions

4.13.1.1 For each function in which the application checks a user's credentials, including the login and password change functions, walk through the process in the normal way, using an account you control. Note every request parameter submitted to the application.

4.13.1.2 Repeat the process numerous times, modifying each parameter in turn in various unexpected ways designed to interfere with the application's logic. For each parameter, include the following changes:

- Submit an empty string as the value.
- Remove the name/value pair.
- Submit very long and very short values

- Submit strings instead of numbers, and vice versa.
- Submit the same named parameter multiple times, with the same and different values.

4.13.1.3 Review closely the application's responses to the preceding requests. If any unexpected divergences from the base case occur, feed this observation back into your framing of further test cases. If one modification causes a change in behavior, try to combine this with other changes to push the application's logic to its limits.

4.13.2 Test Any Multistage Mechanisms

4.13.2.1 If any authentication-related function involves submitting credentials in a series of different requests, identify the apparent purpose of each distinct stage, and note the parameters submitted at each stage.

4.13.2.2 Repeat the process numerous times, modifying the sequence of requests in ways designed to interfere with the application's logic, including the following tests:

- Proceed through all stages, but in a different sequence than the one intended.
- Proceed directly to each stage in turn, and continue the normal sequence from there.
- Proceed through the normal sequence several times, skipping each stage in turn, and continuing the normal sequence from the next stage.
- On the basis of your observations and the apparent purpose of each stage of the mechanism, try to think of further ways to modify the sequence and to access the different stages that the developers may not have anticipated.

4.13.2.3 Determine whether any single piece of information (such as the username) is submitted at more than one stage, either because it is capture more than once from the user or because it is transmitted via the client in a hidden form field, cookie, or preset query string parameter. If so, try
submitting different values at different stages (both valid and invalid) and observing the effect. Try to determine whether the submitted item is sometimes superfluous, or is validated at one stage and then trusted subsequently, or is validated at different stages against different checks.

Try to exploit the application's behavior to gain unauthorized access or reduce the effectiveness of the controls imposed by the mechanism.

4.13.2.4 Look for any data that is transmitted via the client that has not been captured from the user at any point. If hidden parameters are used to track the state of the process across successive stages, it may be possible to interfere with the application's logic by modifying these parameters in crafted ways.

4.13.2.5 If any part of the process involves the application's presenting a randomly varying challenge, test for two common defects:

- If a parameter specifying the challenge is submitted along with the user's response, determine whether you can effectively choose your own challenge by modifying this value.
- Try proceeding as far as the varying challenge several times with the same username, and determine whether a different challenge is presented. If it is, you can effectively choose your own challenge by proceeding to this stage repeatedly until your desired challenge is presented.

4.14 Exploit Any Vulnerabilities to Gain Unauthorized Access

4.14.1 Review any vulnerabilities you have identified within the various authentication functions, and identify any that you can leverage to achieve your objectives in attacking the application. This typically involves attempting to authenticate as a different user — if possible, a user with administrative privileges.

4.14.2 Before mounting any kind of automated attack, note any account lockout defenses you have identified. For example, when performing username enumeration against a login function, submit a common password with each request rather than a completely arbitrary value so as not to waste a failed login attempt on every username discovered. Similarly, perform any password-guessing attacks on a breadth-first, not depth-first, basis. Start your word list with the most common weak passwords, and proceed through this list, trying each item against every enumerated username.

4.14.3 Take account of the password quality rules and the completeness of password validation when constructing word lists to use in any `passwordguessing` attack to avoid impossible or superfluous test cases.

4.14.4 Use the techniques described in Chapter 14: Automating Customized Attacks to automate as much work as possible and maximize the speed and effectiveness of your attacks.

`5 Test the Session Management Mechanism`

5.1 Understand the Mechanism

5.1.1 Analyze the mechanism used to manage sessions and state. Establish whether the application uses session tokens or some other method of handling the series of requests received from each user. Note that some authentication technologies (such as HTTP authentication) may not require a full session mechanism to reidentify users post-authentication. Also, some applications use a `sessionless` state mechanism in which all state information is transmitted via the client, usually in an encrypted or obfuscated form.

5.1.2 If the application uses session tokens, confirm precisely which pieces of data are actually used to reidentify users. Items that may be used to transmit tokens include HTTP cookies, query string parameters, and hidden form fields. Several different pieces of data may be used collectively to reidentify the user, and different items may be used by different back-end components. Often, items that look like session tokens may not actually be employed as such by the application, such as the default cookie generated by the web server.

5.1.3 To verify which items are actually being employed as session tokens, find a page or function that is certainly session-dependent (such as a user-specific `My Details page`). Then make several requests for it, systematically removing each item you suspect is being used as a session token. If removing an item stops the session-dependent page from being returned, this may confirm that the item is a session token. `Burp Repeater` is a useful tool for performing these tests.

5.1.4 Having established which items of data are actually being used to reidentify users, for each token confirm whether it is being validated in its entirety, or whether some subcomponents of the token are ignored. Change the token's value `1 byte at a time`, and check whether the modified value is still accepted. If you find that certain portions of the token are not actually used to maintain session state, you can exclude these from further analysis.

5.2 Test Tokens for Meaning

5.2.1 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, log in with a series of similar usernames that have small variations, such as `A, AA, AAA, AAAA, AAAB, AAAC, AABA`, and so on. If other user-specific data is submitted at the login or is stored in user profiles (such as an e-mail address), perform a similar exercise to modify that data systematically and capture the resulting tokens.

5.2.2 Analyze the tokens you receive for any correlations that appear to be related to the username and other user-controllable data.

5.2.3 Analyze the tokens for any detectable encoding or obfuscation. Look for a correlation between the length of the username and the length of the token, which strongly indicates that some kind of obfuscation or encoding is in use. Where the username contains a sequence of the same character, look for a corresponding character sequence in the token, which may indicate the use of `XOR obfuscation`. Look for sequences in the token that contain only hexadecimal characters, which may indicate hexadecimal encoding of an `ASCII string` or other information. Look for sequences ending in an equals sign and/or containing only the other valid Base64 characters: a to z, A to Z, 0 to 9, +, and /.

5.2.4 If you can identify any meaningful data within your sample of session tokens, consider whether this is sufficient to mount an attack that attempts to guess the tokens recently issued to other application users. Find a page of the application that is session-dependent, and use the techniques described in Chapter 14: Automating Customized Attacks to automate the task of generating and testing possible tokens.

5.3 Test Tokens for Predictability

5.3.1 Generate and capture a large number of session tokens in quick succession, using a request that causes the server to return a new token (for example, a successful login request)

5.3.2 Attempt to identify any patterns within your sample of tokens. In all cases you should use `Burp Sequencer`, as described in Chapter 7: Attacking Session Management , to perform detailed statistical tests of the randomness properties of the application's tokens. Depending on the results, it may also be useful to perform the following manual analysis:

- Apply your understanding of which tokens and subsequences the application actually uses to reidentify users. Ignore any data that is not used in this way, even if it varies between samples.
- If it is unclear what type of data is contained in the token, or in any individual component of it, try applying various `decodings` (for example, Base64) to see if any more meaningful data emerges. It may be necessary to apply several `decodings` in sequence.
- Try to identify any patterns in the sequences of values contained in each decoded token or component. Calculate the differences between successive values. Even if these appear to be chaotic, there may be a fixed set of observed differences, which narrows down the scope of any brute-force attack considerably.
- Obtain a similar sample of tokens after waiting for a few minutes, and repeat the same analysis. Try to detect whether any of the tokens' content is `time-dependent.`

5.3.3 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.

5.3.4 If you can identify any exploitable sequences or time dependencies, consider whether this is sufficient to mount an attack that attempts to guess the tokens recently issued to other application users. Use the techniques described in Chapter 14: Automating Customized Attacks to automate the task of generating and testing possible tokens. Except in the simplest kind of sequences, it is likely that your attack will need to involve a customized script of some kind.

5.3.5 If the session ID appears to be custom-written, use the `bit flipper` payload source in Burp Intruder to sequentially modify each bit in the session token in turn. Grep for a string in the response that indicates whether modifying the token has not resulted in an invalid session, and
whether the session belongs to a different user.

5.4 Check for Insecure Transmission of Tokens

5.4.1 Walk through the application as normal, starting with unauthenticated content at the start URL, proceeding through the login process, and then going through all the application's functionality. Make a note of every occasion on which a new session token is issued, and which
portions of your communications use HTTP and which use HTTPS. You can use the logging function of your intercepting proxy to record this information.

5.4.2 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.

5.4.3 Determine whether, in the normal use of the application, session tokens are ever transmitted over an `HTTP connection`. If so, they are vulnerable to interception.

5.4.4 In cases where the application uses HTTP for unauthenticated areas and switches to HTTPS for the login and/or authenticated areas of the application, verify whether a new token is issued for the HTTPS portion of the communications, or whether a token issued during the
HTTP stage remains active when the application switches to HTTPS. If a token issued during the HTTP stage remains active, the token is vulnerable to interception.

5.4.5 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.

5.5 Check for Disclosure of Tokens in Logs

5.5.1 If your application mapping exercises identified any logging, monitoring, or diagnostic functionality, review these functions closely to determine whether any session tokens are disclosed within them. Confirm who is normally authorized to access these functions. If they are intended for administrators only, determine whether any other vulnerabilities exist that could enable a lower-privileged user to access them.

5.5.2 Identify any instances where session tokens are transmitted within the URL. It may be that tokens are generally transmitted in a more secure manner, but that developers have used the URL in specific cases to work around a particular problem. If so, these may be transmitted in
the Referrer header when users follow any off-site links. Check for any functionality that enables you to inject arbitrary off-site links into pages viewed by other users.

5.5.3 If you find any way to gather valid session tokens issued to other users, look for a way to test each token to determine whether it belongs to an administrative user (for example, by attempting to access a `privileged function` using the token).

5.6 Check Mapping of Tokens to Sessions

5.6.1 Log in to the application twice using the same user account, either from different browser processes or from different computers. Determine whether both sessions remain active concurrently. If they do, the application supports concurrent sessions, enabling an attacker who has compromised another user's credentials to use these without risk of detection.
5.6.2 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. If the latter occurs, the application is not really employing proper session tokens, but is
using unique persistent strings to reidentify each user. In this situation, there is no way to protect against concurrent logins or properly enforce session timeout.
5.6.3 If tokens appear to contain any structure and meaning, attempt to separate out components that may identify the user from those that appear to be inscrutable. Try to modify any user-related components of the token so that they refer to other known users of the application. Verify whether
the application accepts the resulting token and whether it enables you to masquerade as that user. See Chapter 7: Attacking Session Management for examples of this kind of subtle vulnerability.

5.7 Test Session Termination

5.7.1 When testing for session timeout and logout flaws, focus solely on the server's handling of sessions and tokens, rather than any events that occur on the client. In terms of session termination, nothing much depends on what happens to the token within the client browser.

5.7.2 Check whether session expiration is implemented on the server:

- Log in to the application to obtain a valid session token.
- Wait for a period without using this token, and then submit a request for a protected page (such as My Details) using the token.
- If the page is displayed normally, the token is still active.
- Use trial and error to determine how long any session expiration timeout is, or whether a token can still be used days after the previous request that used it. `Burp Intruder` can be configured to increment the time interval between successive requests to automate this task.

5.7.3 Check whether a logout function exists. If it does, test whether it effectively invalidates the user's session on the server. After logging out, attempt to reuse the old token, and determine whether it is still valid by requesting a protected page using the token. If the session is still active,
users remain vulnerable to some session hijacking attacks even after they have "logged out." You can use `Burp Repeater` to keep sending a specific request from the proxy history to see whether the application responds differently after you log out.

5.8 Check for Session Fixation

5.8.1 If the application issues session tokens to unauthenticated users, obtain a token and perform a login. If the application does not issue a fresh token following a successful login, it is vulnerable to `session fixation`.
5.8.2 Even if the application does not issue session tokens to unauthenticated users, obtain a token by logging in, and then return to the login page. If the application is willing to return this page even though you are already authenticated, submit another login as a different user using the **same token**. If the application does not issue a fresh token after the second login, it is vulnerable to session fixation.

5.8.3 Identify the format of session tokens that the application uses. Modify your token to an invented value that is validly formed, and attempt to log in. If the application allows you to create an authenticated session using an invented token, it is vulnerable to session fixation.
5.8.4 If the application does not support login, but processes sensitive user information (such as personal and payment details) and allows this to be displayed after submission (such as on a Verify My Order page), carry out the preceding three tests in relation to the pages displaying
sensitive data. If a token set during anonymous usage of the application can later be used to retrieve sensitive user information, the application is vulnerable to session fixation.

5.9 Check for CSRF

5.9.1 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.
5.9.2 Review the application's key functionality, and identify the specific requests that are used to perform sensitive actions. If an attacker can fully determine in advance parameters for any of these requests (that is, they do not contain any session tokens, unpredictable data, or other secrets), the application is almost certainly vulnerable.
5.9.3 Generate the CSRF POC from this link ⇒ https://github.com/merttasci/csrf-poc-generator
5.9.4 If the application uses additional tokens within requests in an attempt to prevent CSRF attacks, test the robustness of these in the same manner as for session tokens. Also test whether the application is vulnerable to UI redress attacks, in order to defeat the anti-CSRF defenses (see Chapter 13: Attacking Users: Other Techniques for more details).

5.10 Check Cookie Scope

5.10.1 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.
5.10.2 If the application explicitly liberalizes its cookies' scope to a parent domain or parent directory, it may be leaving itself vulnerable to attacks via other web applications that are hosted within the parent domain or directory.
5.10.3 If the application sets its cookies' domain scope to its own domain name (or does

not specify a domain attribute), it may still be exposed to attacks via any applications hosted on subdomains. This is a consequence of how cookie scoping works. It cannot be avoided other than by not hosting any other applications on a subdomain of a `securitysensitive application`.

5.10.4 Determine any reliance on segregation by path, such as `/site/main` and `/site/demo`, which can be subverted in the event of a cross-site scripting attack.
5.10.5 Identify all the possible domain names and paths that will receive the cookies that the application issues. Establish whether any other web applications are accessible via these domain names or paths that you may be able to leverage to capture the cookies issued to users of the
target application.

`6 Test Access Controls`

6.1 Understand the Access Control Requirements

6.1.1 Based on the core functionality implemented within the application, understand the broad requirements for access control in terms of vertical segregation (different levels of users have access to different types of functionality) and horizontal segregation (users at the same privilege
level have access to different subsets of data). Often, both types of segregation are present. For example, ordinary users may be able to access their own data, while administrators can access everyone's data.
6.1.2 Review your application mapping results to identify the areas of functionality and types of data resources that represent the most fruitful targets for `privilege escalation` attacks.
6.1.3 To perform the most effective testing for access control vulnerabilities, you should ideally obtain a number of different accounts with different vertical and horizontal privileges. If self-registration is possible, you can probably obtain the latter directly from the application. To obtain the former, you will probably need the cooperation of the application owner (or need to exploit some vulnerability to gain access to a high-privileged account). The availability of different kinds of accounts will affect the types of testing you can perform, as described next.

6.2 Test with Multiple Accounts

6.2.1 If the application enforces vertical privilege segregation, first use a powerful account to locate all the functionality it can access. Then use a less-privileged account and attempt to access each item of this functionality.
6.2.1.1 Using `Burp`, browse all the application's content within one user context.
6.2.1.2 Review the contents of Burp's `site map` to ensure you have identified all the functionality you want to test. Then, log out of the application and log back in using a different user context.
Use the context menu to select the `compare site maps` feature to determine which high-privileged requests may be accessible to the lower-privileged user. See Chapter 8: Attacking Access Controls for more details on this technique.
6.2.2 If the application enforces horizontal privilege segregation, perform the equivalent test using two different accounts at the same privilege level, attempting to use one account to access data belonging to the other account. This typically involves replacing an identifier (such as
a document ID) within a request to specify a resource belonging to the other user.
6.2.3 Perform manual checking of key access control logic.

 6.2.3.1 For each user privilege, review resources available to a user.
 Attempt to access those resources from an unauthorized user
 account by replaying the request using the unauthorized user's session token.

6.2.4 When you perform any kind of access control test, be sure to test every step of multistage functions individually to confirm whether access controls have been properly implemented at each stage, or whether the application assumes that users who access a later stage must have passed security checks implemented at the earlier stages. For example, if an administrative
page containing a form is properly protected, check whether the actual form submission is also subjected to proper access controls.

6.3 Test with Limited Access

6.3.1 If you do not have prior access to accounts at different privilege levels, or to multiple accounts with access to different data, testing for broken access controls is not quite as straightforward. 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.

6.3.2 In your application mapping exercises that use a low-privileged account, you may have identified the URLs for privileged functions such as administrative interfaces. If

these are not adequately protected, you will probably already know about this.
6.3.3 Decompile all compiled clients that are present, and extract any references to server-side functionality.
6.3.4 Most data that is subject to horizontal access controls is accessed using an identifier, such as an account number or order reference. To test whether access controls are effective using only a single account, you must try to guess or discover the identifiers associated with other users'
data. If possible, generate a series of identifiers in quick succession (for example, by creating several new orders). Attempt to identify any patterns that may enable you to predict the identifiers issued to other users. If there is no way to generate new identifiers, you are probably restricted to analyzing those you already have and guessing on that basis.
6.3.5 If you find a way to predict the identifiers issued to other users, use the techniques described in Chapter 14: Automating Customized Attacks to mount an automated attack to harvest interesting data belonging to other users. Use the Extract Grep function in `Burp Intruder` to capture the relevant information from within the application's responses.

6.4 Test for Insecure Access Control Methods

6.4.1 Some applications implement access controls based on request parameters in an inherently unsafe way. Look for parameters such as `edit=false` or `access=read` in any key requests, and modify these in line with their apparent role to try to interfere with the application's access control logic.
6.4.2 Some applications base access control decisions on the HTTP `Referer` header. For example, an application may properly control access to `/admin.jsp` and accept any request showing this as its `Referer`. To test for this behavior, attempt to perform some privileged actions to which you are authorized, and submit a missing or modified `Referer` header. If this change causes the application to block your request, it may be using the `Referer` header in an unsafe way. Try performing the same action as an unauthorized user, but supply the original `Referer` header and see whether the action succeeds.
6.4.3 If `HEAD` is an allowed method on the site, test for insecure `containermanaged` access control to URLs. Make a request using the HEAD method to determine whether the application permits it.

7 Test for Input-Based Vulnerabilities

7.1 Fuzz All Request Parameters

7.1.1 Review the results of your application mapping exercises and identify every distinct client request that submits parameters that the server-side application processes. Relevant parameters include items within the `URL query string`, `parameters` in the request body, and HTTP cookies. Also include any other items of user input that have been observed to have an
effect on the application's behavior, such as the `Referer` or `User-Agent` headers.

7.1.2 To fuzz the parameters, you can use your own scripts or a ready-made fuzzing tool. For example, to use `Burp Intruder`, load each request in turn into the tool. An easy way to do this is to intercept a request in Burp Proxy and select the Send to Intruder action, or right-click an item in the
Burp Proxy history and select this option. Using this option configures Burp Intruder with the contents of the request, along with the correct target host and port. It also automatically marks the values of all request parameters as payload positions, ready for fuzzing.

7.1.3 Using the payloads tab, configure a suitable set of attack payloads to probe for vulnerabilities within the application. You can enter payloads manually, load them from a file, or select one of the preset payload lists. Fuzzing every request parameter within the application typically entails
issuing a large number of requests and reviewing the results for anomalies. If your set of attack strings is too large, this can be counterproductive and generate a prohibitively large amount of output for you to review. Hence, a sensible approach is to target a range of common vulnerabilities that can often be easily detected in anomalous responses to specific crafted inputs and that often manifest themselves anywhere within the application rather than within specific types of functionality. Here is a suitable set of payloads that you can use to test for some common
categories of vulnerabilities:

SQL Injection:

```
‘
‘--
‘; waitfor delay ‘0:30:0’--
1; waitfor delay ‘0:30:0’--
```

XSS and Header Injection:

```
xsstest
“><script>alert(‘xss’)</script>
```

OS Command Injection:

```
|| ping -i 30 127.0.0.1 ; x || ping -n 30 127.0.0.1 &
| ping –i 30 127.0.0.1 |
| ping –n 30 127.0.0.1 |
& ping –i 30 127.0.0.1 &
& ping –n 30 127.0.0.1 &
; ping 127.0.0.1 ;
%0a ping –i 30 127.0.0.1 %0a
` ping 127.0.0.1 `
```

Path Traversal:

```
../../../../../../../../../../../../etc/passwd
../../../../../../../../../../../../boot.ini
..\..\..\..\..\..\..\..\..\..\..\..\etc\passwd
..\..\..\..\..\..\..\..\..\..\..\..\boot.ini
```

Script Injection:

```
;echo 111111
echo 111111
response.write 111111
:response.write 111111
```

File Inclusion:

```
http://<your server name>/
http://<nonexistent IP address>/
```

7.1.4 All the preceding payloads are shown in their literal form. The characters `?, ;, &, +, =`, and space need to be URL-encoded because they have special meaning within HTTP requests. By default, Burp Intruder performs the necessary encoding of these characters, so ensure that this option has not been disabled. (To restore all options to their defaults following earlier customization, select Burp , Restore Defaults.)

7.1.5 In the Grep function of Burp Intruder, configure a suitable set of strings to flag some common error messages within responses. For example:

```
error
exception
illegal
invalid
fail
stack
access
directory
file
not found
varchar
ODBC
SQL
SELECT
111111
=======================
Note that the string 111111 is included to test for successful script injection attacks.
The payloads in step 7.1.3 involve writing this value into  the server’s response.
```

7.1.6 Also select the Payload Grep option to flag responses that contain the payload itself, indicating a potential XSS or header injection vulnerability.

7.1.7 Set up a web server or `netcat` listener on the host you specified in the first file inclusion payload. This helps you monitor for connection attempts received from the server resulting from a successful remote file inclusion attack.

7.1.8 Launch the attack. When it has completed, review the results for anomalous responses indicating the presence of vulnerabilities. Check for divergences in the HTTP status code, the response length, the response time, the appearance of your configured expressions, and the appearance of the payload itself. You can click each column heading in the results table to sort the results by the values in that column (and Shift-click to reverse-sort the results). This enables you to quickly identify any anomalies that stand out from the other results.

7.1.9 For each potential vulnerability indicated by the results of your fuzz testing, refer to the following sections of this methodology. They describe the detailed steps you should take in relation to each category of problem to verify the existence of a vulnerability and successfully exploit it.

7.1.10 After you have configured Burp Intruder to perform a fuzz test of a single request, you can quickly repeat the same test on other requests within the application. Simply select each target request within Burp Proxy and choose the Send to Intruder option. Then immediately launch the attack within Intruder using the existing attack configuration. In this way, you can launch a large number of tests simultaneously in separate windows and manually review the results as each test completes its work.

7.1.11 If your mapping exercises identified any out-of-band input channels whereby user-controllable input can be introduced into the application's processing, you should perform a similar fuzzing exercise on these input channels. Submit various crafted data designed to trigger common vulnerabilities when processed within the web application. Depending on the nature of the input channel, you may need to create a custom script or other harness for this purpose.

7.1.12 In addition to your own fuzzing of application requests, if you have access to an automated web application vulnerability scanner, you should run it against the target application to provide a basis for comparison with your own findings.

7.2 Test for SQL Injection

7.2.1 If the SQL attack strings listed in step 7.1.3 result in any anomalous responses, probe the application's handling of the relevant parameter manually to determine whether a SQL injection vulnerability is present.
7.2.2 If any database error messages were returned, investigate their meaning. Use the section "SQL Syntax and Error Reference" in 🏪 Chapter 9: Attacking Data Stores to help interpret error messages on some common database platforms.
7.2.3 If submitting a single quotation mark in the parameter causes an error or other anomalous behavior, submit two single quotation marks. If this input causes the error or anomalous behavior to disappear, the application is probably vulnerable to SQL injection.
7.2.4 Try using common SQL string concatenator functions to construct a string that is equivalent to some benign input. If this causes the same response as the original benign input, the application is probably vulnerable. For example, if the original input is the expression FOO, you can perform

this test using the following items (in the third example, note the space between the two quotes):

```
‘||’FOO
‘+’FOO
‘ ‘FOO
=======
As always, be sure to URL-encode characters such as + and space that
have special meaning within HTTP requests.
```

7.2.5 If the original input is numeric, try using a mathematical expression that is equivalent to the original value. For example, if the original value was 2, try submitting 1+1 or 3–1. If the application responds in the same way, it may be vulnerable, particularly if the value of the numeric expression
has a systematic effect on the application’s behavior.
7.2.6 If the preceding test is successful, you can gain further assurance that a SQL injection vulnerability is involved by using SQL-specific mathematical expressions to construct a particular value. If the application’s logic can be systematically manipulated in this way, it is almost certainly vulnerable to SQL injection. For example, both of the following items are equivalent to the number 2: `67-ASCII(‘A’)` `51-ASCII(1)`
7.2.7 If either of the fuzz test cases using the `waitfor` command resulted in an abnormal time delay before the application responded, this is a strong indicator that the database type is MS-SQL and the application is vulnerable to SQL injection. Repeat the test manually, specifying different values in the `waitfor` parameter, and determine whether the time taken to respond varies systematically with this value. Note that your attack payload may be inserted into more than one SQL query, so the time delay observed may be a fixed multiple of the value specified.
7.2.8 If the application is vulnerable to SQL injection, consider what kinds of attacks are feasible and likely to help you achieve your objectives. Refer to 🏪 Chapter 9: Attacking Data Stores for the detailed steps needed to carry out any of the following attacks:

- Modify the conditions within a `WHERE` clause to change the application’s logic (for example, by injecting or 1=1-- to bypass a login).
- Use the `UNION` operator to inject an arbitrary `SELECT` query and combine the results with those of the application’s original query.
- Fingerprint the database type using database-specific SQL syntax.

- If the database type is MS-SQL and the application returns ODBC error messages in its responses, leverage these to enumerate the database structure and retrieve arbitrary data.
- If you cannot find a way to directly retrieve the results of an arbitrary injected query, use the following advanced techniques to extract data:
  - Retrieve string data in numeric form, one byte at a time.
  - Use an out-of-band channel.
- If you can cause different application responses based on a single arbitrary condition, use Absinthe to extract arbitrary data one bit at a time.
- If you can trigger time delays based on a single arbitrary condition, exploit these to retrieve data one bit at a time.
- If the application is blocking certain characters or expressions that you require to perform a particular attack, try the various bypass techniques described in 🏪 Chapter 9: Attacking Data Stores to circumvent the input filter.
- If possible, escalate the attack against the database and the underlying server by leveraging any vulnerabilities or powerful functions within the database.

7.3 Test for XSS and Other Response Injection

7.3.1 Identify Reflected Request Parameters

7.3.1.1 Sort the results of your fuzz testing by clicking the Payload Grep column, and identify any matches corresponding to the XSS payloads listed in step 7.1.3 These are cases where the XSS test strings were returned unmodified within the application's responses.

7.3.1.2 For each of these cases, review the application's response to find the location of the supplied input. If this appears within the response body, test for XSS vulnerabilities. If the input appears within any HTTP header, test for header injection vulnerabilities. If it is used in the Location header of a 302 response, or if it is used to specify a redirect in some other way, test for redirection vulnerabilities. Note that the same input might be copied into multiple locations within the response, and that more than one type of reflected vulnerability might be present.

7.3.2 Test for Reflected XSS

7.3.2.1 For each place within the response body where the value of the request parameter appears, review the surrounding HTML to identify possible ways of crafting your input to cause execution of arbitrary JavaScript. For example, you can inject `<script>` tags, inject into an existing script, or place a crafted value into a tag attribute.
7.3.2.2 Use the different methods of beating signature-based filters described in Chapter 12: Attacking Users: Cross-Site Scripting as a reference for the different ways in which crafted input can be used to cause execution of JavaScript.
7.3.2.3 Try submitting various possible exploits to the application, and monitor its responses to determine whether any filtering or sanitization of input is being performed. If your attack string is returned unmodified, use a browser to verify conclusively that you have succeeded in executing arbitrary JavaScript (for example, by generating an alert dialog).

7.3.2.4 If you find that the application is blocking input containing certain characters or expressions you need to use, or is HTML-encoding certain characters, try the various filter bypasses described in Chapter 12: Attacking Users: Cross-Site Scripting.

7.3.2.5 If you find an XSS vulnerability in a POST request, this can still be exploited via a malicious website that contains a form with the required parameters and a script to automatically submit the form. Nevertheless, a wider range of attack delivery mechanisms is available if the exploit
can be delivered via a GET request. Try submitting the same parameters in a GET request, and see if the attack still succeeds. You can use the Change Request Method action in Burp Proxy to convert the request for you.

7.3.3 Test for HTTP Header Injection

7.3.3.1 For each place within the response headers where the value of the request parameter appears, verify whether the application accepts data containing URL-encoded carriage-return (%0d) and line-feed (%0a) characters and whether these are returned `unsanitized` in its response. (Note that you are looking for the actual newline characters themselves to appear in the server's response, not their URL-encoded equivalents.)
7.3.3.2 If a new line appears in the server's response headers when you supply crafted input, the application is vulnerable to HTTP header injection. This can be leveraged to perform various attacks, as described in Chapter 13: Attacking Users: Other

Techniques.
7.3.3.3 If you find that only one of the two newline characters gets returned in the server's responses, it may still be possible to craft a working exploit, depending on the context and the target user's browser.
7.3.3.4 If you find that the application blocks input containing newline characters, or sanitizes those characters in its response, try the following items of input to test the filter's effectiveness:

```
foo%00%0d%0abar
foo%250d%250abar
foo%%0d0d%%0a0abar
```

7.3.4 Test for Open Redirection

7.3.4.1 If the reflected input is used to specify the target of a redirect of some kind, test whether it is possible to supply crafted input that results in an arbitrary redirect to an external website. If so, this behavior can be exploited to lend credibility to a `phishing-style` attack.

7.3.4.2 If the application ordinarily transmits an absolute URL as the parameter's value, modify the domain name within the URL, and test whether the application redirects you to the different domain.
7.3.4.3 If the parameter normally contains a relative URL, modify this into an absolute URL for a different domain, and test whether the application redirects you to this domain.
7.3.4.4 If the application carries out some validation on the parameter before performing the redirect, in an effort to prevent external redirection, this is often vulnerable to bypasses. Try the various attacks described in Chapter 13: Attacking Users: Other Techniques to test the robustness of the filters.

7.3.5 Test for Stored Attacks

7.3.5.1 If the application stores items of user-supplied input and later displays these on-screen, after you have fuzzed the entire application you may observe some of your attack strings being returned in responses to requests that did not themselves contain those strings. Note any instances where this occurs, and identify the original entry point for the data that is being stored.
7.3.5.2 In some cases, user-supplied data is stored successfully only if you complete a

multistage process, which does not occur in `basic fuzz testing`. If your application mapping exercises identified any functionality of this kind, manually walk through the relevant process and test the stored data for XSS vulnerabilities.
7.3.5.3 If you have sufficient access to test it, review closely any administrative functionality in which data originating from low-privileged users is ultimately rendered on-screen in the session of more privileged users. Any stored XSS vulnerabilities in functionality of this kind typically lead directly to privilege escalation.
7.3.5.4 Test every instance where user-supplied data is stored and displayed to users. Probe these for XSS and the other response injection attacks described previously.
7.3.5.5 If you find a vulnerability in which input supplied by one user is displayed to other users, determine the most effective attack payload with which you can achieve your objectives, such as `session hijacking` or `request forgery`. If the stored data is displayed only to the same user from whom it originated, try to find ways of chaining any other vulnerabilities you have discovered (such as broken access controls) to inject an attack into other users' sessions.

7.3.5.6 If the application allows upload and download of files, always probe this functionality for stored XSS attacks. If the application allows HTML, JAR, or text fi es, and does not validate or sanitize their contents, it is almost certainly vulnerable. If it allows JPEG files and does not validate that they contain valid images, it is probably vulnerable to attacks against Internet Explorer users. Test the application's handling of each file type it supports, and confirm how browsers handle responses containing HTML instead of the normal content type.
7.3.5.7 In every location where data submitted by one user is displayed to other users but where the application's filters prevent you from performing a stored XSS attack, review whether the application's behavior leaves it vulnerable to on-site request forgery.

7.4 Test for OS Command Injection

7.4.1 If any of the command injection attack strings listed in step 7.1.3 resulted in an abnormal time delay before the application responded, this is a strong indicator that the application is vulnerable to OS command injection. Repeat the test, manually specifying different values in the `-i` or `-n` parameter, and determine whether the time taken to respond varies systematically with this value.
7.4.2 Using whichever of the injection strings was found to be successful, try injecting a more interesting command (such as `ls` or `dir`), and determine whether you can

retrieve the results of the command to your browser.
7.4.3 If you are unable to retrieve results directly, other options are open to you:

- You can attempt to open an out-of-band channel back to your computer. Try using TFTP to copy tools up to the server, using telnet or `netcat` to create a reverse shell back to your computer, and using the mail command to send command output via SMTP.
- You can redirect the results of your commands to a file within the web root, which you can then retrieve directly using your browser. For example: `dir > c:\inetpub\wwwroot\foo.txt`

7.4.4 If you find a way to inject commands and retrieve the results, you should determine your privilege level (by using `whoami` or a similar command, or attempting to write a harmless file to a protected directory). You may then seek to escalate privileges, gain backdoor access to sensitive
application data, or attack other hosts that can be reached from the compromised server.

7.4.5 If you believe that your input is being passed to an OS command of some kind, but the attack strings listed are unsuccessful, see if you can use the `< or >` character to direct the contents of a file to the command's input or to direct the command's output to a fi le. This may enable you
to read or write arbitrary file contents. If you know or can guess the actual command being executed, try injecting command-line parameters associated with that command to modify its behavior in useful ways (for example, by specifying an output file within the web root).
7.4.6 If you find that the application is escaping certain key characters you need to perform a command injection attack, try placing the escape character before each such character. If the application does not escape the escape character itself, this usually leads to a bypass of this defensive measure. If you find that whitespace characters are blocked or sanitized, you may be able to use `$IFS` in place of spaces on UNIX-based platforms.

7.5 Test for Path Traversal

7.5.1 For each fuzz test you have performed, review the results generated by the path traversal attack strings listed in step 7.1.3. You can click the top of the payload column

in Burp Intruder to sort the results table by payload and group the results for these strings. For any cases where
an unusual error message or a response with an abnormal length was received, review the response manually to determine whether it contains the contents of the specified fi le or other evidence that an anomalous file operation occurred.
7.5.2 In your mapping of the application's attack surface, you should have noted any functionality that specifically supports the reading and writing of files on the basis of user-supplied input. In addition to the general fuzzing of all parameters, you should manually test this functionality
very carefully to identify any path traversal vulnerabilities that exist.

7.5.3 Where a parameter appears to contain a filename, a portion of a filename, or a directory, modify the parameter's existing value to insert an arbitrary subdirectory and a single traversal sequence. For example, if the application submits this parameter: `file=foo/file1.txt`
try submitting this value: `file=foo/bar/../file1.txt` If the application's behavior is identical in the two cases, it may be vulnerable, and you should proceed to the next step. If the behavior is different, the application may be blocking, stripping, or sanitizing traversal sequences, resulting in an invalid file path. Try using the encoding and other attacks described in Chapter 10: Attacking Back-End Components in an attempt to bypass the filters.

7.5.4 If the preceding test of using traversal sequences within the base directory is successful, try using additional sequences to step above the base directory and access known files on the server's operating system. If these attempts fail, the application may be imposing various filters or checks before file access is granted. You should probe further to understand the controls that are implemented and whether any bypasses exist.
7.5.5 The application may be checking the file extension being requested and allowing access to only certain kinds of fi les. 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`

7.5.6 The application may be checking that the user-supplied file path starts with a particular directory or stem. Try appending traversal sequences after a known accepted stem in an attempt to bypass the filter. For example:

`/images/../../../../../../../etc/passwd`

7.5.7 If these attacks are unsuccessful, try combining multiple bypasses, working initially entirely within the base directory in an attempt to understand the filters in place and the ways in which the application handles unexpected input.
7.5.8 If you succeed in gaining read access to arbitrary fi les on the server, attempt to retrieve any of the following files, which may enable you to escalate your attack:

- Password files for the operating system and application
- Server and application configuration files, to discover other vulnerabilities or fine-tune a different attack
- Include files that may contain database credentials
- Data sources used by the application, such as MySQL database files or XML files
- The source code to server-executable pages, to perform a code review in search of bugs
  Application log files that may contain information such as usernames and session tokens.

7.5.9 If you succeed in gaining write access to arbitrary fi les on the server, examine whether any of the following attacks are feasible in order to escalate your attack:

- Creating scripts in users' startup folders
- Modifying files such as `in.ftpd` to execute arbitrary commands when a user next connects
- Writing scripts to a web directory with execute permissions and calling them from your browser

7.6 Test for Script Injection

7.6.1 For each fuzz test you have performed, review the results for the string `111111` on its own (that is, not preceded by the rest of the test string). You can quickly identify these in Burp Intruder by Shift-clicking the heading for the 111111 Grep string to group all the results containing this string. Look for any that do not have a check in the Payload Grep column. Any cases identified are likely to be vulnerable to injection of scripting commands.
7.6.2 Review all the test cases that used script injection strings, and identify any containing scripting error messages that may indicate that your input is being executed

but caused an error. These may need to be fi ne-tuned to perform successful script injection.
7.6.3 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’)`

7.7 Test for File Inclusion

7.7.1 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.
7.7.2 Review the results of the file inclusion test cases, and identify any that caused an anomalous delay in the application’s response. In these cases, it may be that the application itself is vulnerable but that the resulting HTTP requests are timing out due to network-level filters.

7.7.3 If you find a remote fi le inclusion vulnerability, deploy a web server containing a malicious script specific to the language you are targeting, and use commands such as those used to test for script injection to verify that your script is being executed.

`8 Test for Function-Specific Input VULNs`

8.1 Test for SMTP Injection

8.1.1 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 in step 7.1 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>
<youremail>%0aBcc:<youremail>
<youremail>%0d%0aBcc:<youremail>
```

```
%0aDATA%0afoo%0a%2e%0aMAIL+FROM:+<youremail>%0aRCPT+TO:+<youremail>
%0aDATA%0aFrom:+<youremail>%0aTo:+<youremail>%0aSubject:+test%0afoo %0a%2e%0a
%0d%0aDATA%0d%0afoo%0d%0a%2e%0d%0aMAIL+FROM:+<youremail>%0d%0aRCPT+TO:+
<youremail>%0d%0aDATA%0d%0aFrom:+<youremail>%0d%0aTo:+<youremail>
%0d%0aSubject:+test%0d%0afoo%0d%0a%2e%0d%0a
```

8.1.2 Review the results to identify any error messages the application returns. If these appear to relate to any problem in the e-mail function, investigate whether you need to fine-tune your input to exploit a vulnerability.
8.1.3 Monitor the e-mail address you specified to see if any e-mail messages are received.
8.1.4 Review closely the HTML form that generates the relevant request. It may contain clues regarding the server-side software being used. It may also contain a hidden or disabled field that is used to specify the To address of the e-mail, which you can modify directly.

8.2 Test for Native Software Vulnerabilities

8.2.1 Test for Buffer Overflows

8.2.1.1 For each item of data being targeted, submit a range of long strings with lengths somewhat longer than common buffer sizes. Target one item of data at a time to maximize the coverage of code paths in the application. You can use the character blocks payload source in Burp Intruder to
automatically generate payloads of various sizes. The following buffer sizes are suitable to test:
1100 4200 33000

8.2.1.2 Monitor the application's responses to identify any anomalies. An uncontrolled overflow is almost certain to cause an exception in the application, although diagnosing the nature of the problem remotely may be difficult. Look for any of the following anomalies:

- An HTTP 500 status code or error message, where other malformed (but not overlong) input does not have the same effect
- An informative message indicating that a failure occurred in some external, native code component
- A partial or malformed response being received from the server

- The TCP connection to the server closing abruptly without returning a response
- The entire web application no longer responding
- Unexpected data being returned by the application, possibly indicating that a string in memory has lost its null terminator

8.2.2 Test for Integer Vulnerabilities

8.2.2.1 When dealing with native code components, identify any integer-based data, particularly length indicators, which may be used to trigger integer vulnerabilities.
8.2.2.2 Within each targeted item, send suitable payloads designed to trigger any vulnerabilities. For each item of data being targeted, send a series of different values in turn, representing boundary cases for the signed and unsigned versions of different sizes of integer. For example:

- 0x7f and 0x80 (127 and 128)
- 0xff and 0x100 (255 and 256)
- 0x7ffff and 0x8000 (32767 and 32768)
- 0xffff and 0x10000 (65535 and 65536)
- 0x7fffffff and 0x80000000 (2147483647 and 2147483648)
- 0xffffffff and 0x0 (4294967295 and 0)

8.2.2.3 When the data being modified is represented in hexadecimal form, send both little-endian and big-endian versions of each test case, such as `ff7f` and `7fff`. If hexadecimal numbers are submitted in `ASCII form`, use the same case as the application itself uses for alphabetic characters to ensure that these are decoded correctly.
8.2.2.4 Monitor the application's responses for anomalous events, as described in step 8.2.1.2.

8.2.3 Test for Format String Vulnerabilities

8.2.3.1 Targeting each parameter in turn, submit strings containing long sequences of different format specifiers. For example:

```
%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n
%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s
%1!n!%2!n!%3!n!%4!n!%5!n!%6!n!%7!n!%8!n!%9!n!%10!n! etc...
%1!s!%2!s!%3!s!%4!s!%5!s!%6!s!%7!s!%8!s!%9!s!%10!s! etc...
```

Remember to URL-encode the % character as `%25`.
8.2.3.2 Monitor the application's responses for anomalous events, as described in step 8.2.1.2.

8.3 Test for SOAP Injection

8.3.1 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.
8.3.2 If an error was received, submit instead a valid opening and closing tag pair, such as `<foo></foo>`. If this causes the error to disappear, the application may be vulnerable.
8.3.3 If the item you submit is copied back into the application's responses, submit the following two values in turn. If you find that either item is returned as the other, or as simply test, you can be confident that your input is being inserted into an XML-based message. `test <foo/>` `test<foo></foo>`
8.3.4 If the HTTP request contains several parameters that may be being placed into a SOAP message, try inserting the opening comment character `<!-- into one parameter and the closing comment character !-->` into another parameter. Then switch these (because you have no way
of knowing in which order the parameters appear). This can have the effect of commenting out a portion of the server's SOAP message, which may change the application's logic or result in a different error condition that may divulge information.

8.4 Test for LDAP Injection

8.4.1 In any functionality where user-supplied data is used to retrieve information from a directory service, target each parameter in turn to test for potential injection into an LDAP query.
8.4.2 Submit the `*` character. If a large number of results are returned, this is a good indicator that you are dealing with an LDAP query.
8.4.3 Try entering a number of closing parentheses: `))))))))))`
This input invalidates the query syntax, so if an error or other anomalous behavior results, the application may be vulnerable (although many other application functions and injection situations may behave in the same way).
8.4.4 Try entering various expressions designed to interfere with different types of queries, and see if these allow you to influence the results being returned. The `cn`

attribute is supported by all LDAP implementations and is useful if you do not know any details about the directory you are querying:
`)(cn=*`
`))(|(cn=`
`*))%00`
8.4.5 Try adding extra attributes to the end of your input, using commas to separate each item. Test each attribute in turn. An error indicates that the attribute is not valid in the present context. The following attributes are commonly used in directories queried by LDAP:

```
cn
c
mail
givenname
o
ou
dc
l
uid
objectclass
postaladdress
dn
sn
```

8.5 Test for XPath Injection

8.5.1 Try submitting the following values, and determine whether they result
 n different application behavior without causing an error:
`' or count(parent::` `[position()=1])=0 or 'a'='b`
`' or count(parent::` `[position()=1])>0 or 'a'='b`
8.5.2 If the parameter is numeric, also try the following test strings:

`1 or count(parent::` `[position()=1])=0` `1 or count(parent::` `[position()=1])>0`
8.5.3 If any of the preceding strings causes differential behavior within the application without causing an error, it is likely that you can extract arbitrary data by crafting test conditions to extract 1 byte of information at a time. Use a series of conditions with the following form to determine
the name of the current node's parent: `substring(name(parent::*[position()=1]),1,1)='a'`

8.5.4 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'`

8.6 Test for Back-End Request Injection

8.6.1 Locate any instance where an internal server name or IP address is specified in a parameter. Submit an arbitrary server and port, and monitor the application for a timeout. Also submit localhost, and finally your own IP address, monitoring for incoming connections on the port specified.
8.6.2 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 were unmodified, there is a chance of HTTP parameter injection vulnerabilities. Attempt to attack the back-end request by injecting known parameter name/value pairs that may alter the back-end logic, as described in Chapter 10: Attacking Back-End Components .

8.7 Test for XXE Injection

8.7.1 If users are submitting XML to the server, an external entity injection attack may be possible. If a fi eld 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>
```

If no known field can be found, specify an external entity of “http://192.168.1.1:25” and monitor the page response time. If the page takes significantly longer to return or times out, it may be vulnerable.

`9 Test for Logic Flaws`

9.1 Identify the Key Attack Surface

9.1.1 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.
9.1.2 Review the results of your application mapping exercises, and identify any instances of the following features:

- Multistage processes
- Critical security functions, such as login
- Transitions across trust boundaries (for example, moving from being anonymous to being self-registered to being logged in)
- Context-based functionality presented to a user
- Checks and adjustments made to transaction prices or quantities

9.2 Test Multistage Processes

9.2.1 When a multistage process involves a defined sequence of requests, attempt to submit these requests out of the expected sequence. Try skipping certain stages, accessing a single stage more than once, and accessing earlier stages after later ones.
9.2.2 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. Be sure to understand fully the mechanisms that the application is employing to deliver access to distinct stages.

9.2.3 In addition to interfering with the sequence of steps, try taking parameters that are submitted at one stage of the process and submitting them at a different stage. If the relevant items of data are updated within the application's state, you should investigate whether you can leverage this behavior to interfere with the application's logic.
9.2.4 If a multistage process involves different users performing operations on the same set of data, try taking each parameter submitted by one user and submitting it as another. If they are accepted and processed as that user, explore the implications of this behavior, as described previously.
9.2.5 From the context of the functionality that is implemented, try to understand what assumptions the developers may have made and where the key attack surface lies. Try to identify ways of violating those assumptions to cause undesirable behavior within the application.
9.2.6 When multistage functions are accessed out of sequence, it is common to encounter a variety of anomalous conditions within the application, such as variables

with null or uninitialized values, partially defined or inconsistent state, and other unpredictable behavior. Look for interesting error messages and debug output, which you can use to better understand the application's internal workings and thereby fine-tune the current or a different attack.

9.3 Test Handling of Incomplete Input

9.3.1 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.
9.3.2 For each parameter in turn, remove both the name and value of the parameter from the request. Monitor the application's responses for any divergence in its behavior and any error messages that shed light on the logic being performed.
9.3.3 If the request you are manipulating is part of a multistage process, follow the process through to completion, because the application may store data submitted in earlier stages within the session and then process this at a later stage.

9.4 Test Trust Boundaries

9.4.1 Probe how the application handles transitions between different types of trust of the user. Look for functionality where a user with a given trust status can accumulate an amount of state relating to his identity. For example, an anonymous user could provide personal information
during self-registration, or proceed through part of an account recovery process designed to establish his identity.
9.4.2 Try to find ways to make improper transitions across trust boundaries by accumulating relevant state in one area and then switching to a different area in a way that would not normally occur. For example, having completed part of an account recovery process, attempt to
switch to an authenticated user-specific page. Test whether the application assigns you an inappropriate level of trust when you transition in this way.
9.4.3 Try to determine whether you can harness any higher-privileged function directly or indirectly to access or infer information.

9.5 Test Transaction Logic

9.5.1 In cases where 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.
9.5.2 Examine whether you can use a series of successive transactions to bring about a state that you can exploit for a useful purpose. For example, you may be able to perform several low-value transfers between accounts to accrue a large balance that the application's logic was intended to
prevent.
9.5.3 If the application adjusts prices or other sensitive values based on criteria that are determined by user-controllable data or actions, first understand the algorithms used by the application, and the point within its logic where adjustments are made. Identify whether these adjustments are
made on a one-time basis, or whether they are revised in response to further actions performed by the user.
9.5.4 Try to find ways to manipulate the application's behavior to cause it to get into a state where the adjustments it has applied do not correspond to the original criteria intended by its designers.

`10 Test for Shared Hosting Vulnerabilities`

10.1 Test Segregation in Shared Infrastructures

10.1.1 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. Consider the following questions:

- Does the remote access facility use a secure protocol and suitably hardened infrastructure?
- Can customers access files, data, and other resources that they do not legitimately need to access?
- Can customers gain an interactive shell within the hosting environment and execute arbitrary commands?

10.1.2 If a proprietary application is used to allow customers to configure and customize a shared environment, consider targeting this application as a way to compromise the environment itself and individual applications running within it.

10.1.3 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.

10.2 Test Segregation Between ASP-Hosted Applications

10.2.1 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. Attempt to leverage these to compromise the shared portion of the application and thereby attack other individual
applications.

10.2.2 If a common database is used within any kind of shared environment, perform a comprehensive audit of the database configuration, patch level, table structure, and permissions using a database scanning tool such as `NGSSquirrel`. Any defects within the database security model may provide a way to escalate an attack from within one application to another.

`11 Test for Application Server Vulnerabilities`

11.1 Test for Default Credentials

11.1.1 Review the results of your application mapping exercises to identify the web server and other technologies in use that may contain accessible administrative interfaces.
11.1.2 Perform a `port scan` of the web server to identify any administrative interfaces running on a different port than the main target application.
11.1.3 For any identified interfaces, consult the manufacturer's documentation and common default password listings to obtain default credentials.
11.1.4 If the default credentials do not work, use the steps listed in section 4 to attempt to guess valid credentials.
11.1.5 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.

11.2 Test for Default Content

11.2.1 Review the results of your `Nikto` scan (step 1.4.1) to identify any default content that may be present on the server but that is not an integral part of the application.
11.2.2 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. If feasible, carry out a local installation of these, and review them for any default functionality
that you may be able to leverage in your attack.
11.2.3 Examine the default content for any functionality or vulnerabilities that you may be able to leverage to attack the server or the application.

11.3 Test for Dangerous HTTP Methods

11.3.1 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.
11.3.2 Try each reported method manually to confirm whether it can in fact be used.
11.3.3 If you find that some `WebDAV` methods are enabled, use a `WebDAVenabled` client for further investigation, such as Microsoft FrontPage or the Open as Web Folder option in Internet Explorer.

11.4 Test for Proxy Functionality

11.4.1 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.
11.4.2 Using both `GET` and `CONNECT` requests, attempt to connect to different IP addresses and ports within the hosting infrastructure.
11.4.3 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.

11.5 Test for Virtual Hosting Misconfiguration

11.5.1 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)

11.5.2 Compare the responses to these requests. A common result is that directory listings are obtained when the server’s IP address is used in the Host header. You may also find that different default content is accessible.
11.5.3 If you observe different behavior, repeat the application mapping exercises described in section 1 using the hostname that generated different results. Be sure to perform a `Nikto` scan using the `-vhost` option to identify any default content that may have been overlooked during initial application mapping.

11.6 Test for Web Server Software Bugs

11.6.1 Run Nessus and any other similar scanners you have available to identify any known vulnerabilities in the web server software you are attacking.
11.6.2 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.
11.6.3 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.
11.6.4 If possible, consider performing a local installation of the software you are attacking, and carry out your own testing to find new vulnerabilities that have not been discovered or widely circulated.

11.7 Test for Web Application Firewalling

11.7.1 Submit an arbitrary parameter name to the application with a clear attack payload in the value, ideally somewhere the application includes the name and/or value in the response. If the application blocks the attack, this is likely to be due to an external defense.
11.7.2 If a variable can be submitted that is returned in a server response, submit a range of fuzz strings and encoded variants to identify the behavior of the application defenses to user input.
11.7.3 Confirm this behavior by performing the same attacks on variables within the application.

11.7.4 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.
11.7.5 If a particular request is blocked, try submitting the same parameter in a different location or context. For instance, submit the same parameter in the URL in a GET request, within the body of a POST request, and within the URL in a POST request.
11.7.6 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.
11.7.7 Review all the other methods of introducing user input provided in Chapter 4: Mapping the Application , picking any that are not protected.
11.7.8 Determine locations where user input is (or can be) submitted in a nonstandard format such as serialization or encoding. If none is available, build the attack string by concatenation and/or by spanning it across multiple variables. (Note that if the target is ASP.NET, you may be able
to use HPP to concatenate the attack using multiple specifications of the same variable.)

`12 Miscellaneous Checks`

12.1 Check for DOM-Based Attacks

12.1.1 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 (both static and dynamically generated).

12.1.2 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
document.referrer
window.location
```

12.1.3 Trace the relevant data through the code to identify what actions are performed

with it. If the data (or a manipulated form of it) is passed to one of the following APIs, the application may be vulnerable to XSS:
`document.write()`
`document.writeln()`
`document.body.innerHtml`
`eval()`
`window.execScript()`
`window.setInterval()`
`window.setTimeout()`
12.1.4 If the data is passed to one of the following APIs, the application may be vulnerable to a redirection attack:
`document.location`
`document.URL`
`document.open()`
`window.location.href`
`window.navigate()`
`window.open()`

12.2 Check for Local Privacy Vulnerabilities

12.2.1 Review the 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.
12.2.2 If a persistent cookie is set that contains any sensitive data, a local attacker may be able to capture this data. Even if the data is encrypted, an attacker who captures it will be able to resubmit the cookie to the application and gain access to any data or functionality that this allows.
12.2.3 If any application pages containing sensitive data are accessed over `HTTP`, look for any cache directives within the server's responses. If any of the following directives do not exist (either within the HTTP headers or within HTML metatags), the page concerned may be cached by one
or more browsers:
`Expires: 0`
`Cache-control: no-cache`
`Pragma: no-cache`
12.2.4 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.
12.2.5 For all forms that are used to capture sensitive data from the user (such as credit

card details), review the form's HTML source. If the attribute `autocomplete=off` is not set, within either the form tag or the tag for the individual input field, data entered is stored within browsers that support autocomplete, provided that the user has not disabled this feature.

12.2.6 Check for technology-specific local storage.

12.2.6.1 Check for Flash local objects using the `BetterPrivacy` plug-in for Firefox.

12.2.6.2 Check any Silverlight isolated storage in this directory: `C:\Users{username}\AppData\LocalLow\Microsoft\Silverlight\`

12.2.6.3 Check any use of HTML5 local storage.

12.3 Check for Weak SSL Ciphers

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

12.3.2 If any weak or obsolete ciphers and protocols are supported, a suitably positioned attacker may be able to perform an attack to downgrade or decipher the SSL communications of an application user, gaining access to his sensitive data.

12.3.3 Some web servers advertise certain weak ciphers and protocols as supported but refuse to actually complete a handshake using these if a client requests them. This can lead to false positives when you use the `THCSSLCheck` tool. You can use the 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.

12.4 Check Same-Origin Policy Configuration

12.4.1 Check for the `/crossdomain.xml` file. If the application allows unrestricted access (by specifying `<allow-access-from domain="*" />`), Flash object from any other site can perform two-way interaction, riding on the sessions of application users. This would allow all data to be retrieved, and any user actions to be performed, by any other domain.

12.4.2 Check for the /clientaccesspolicy.xml fi le. Similar to Flash, if the `<cross-domain-access>` configuration is too permissive, other sites can perform two-way interaction with the site under assessment.

12.4.3 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.

---

13 Follow Up Any Information Leakage

13.1 In all your probing of the target application, monitor its responses for error messages that may contain useful information about the error's cause, the technologies in use, and the application's internal structure and functionality.
13.2 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. For example:

`”unable to retrieve” filetype:php`

13.3 Review the search results, looking both for any discussion about the error message and for any other websites in which the same message has appeared. Other applications may produce the same message in a more verbose context, enabling you to better understand what kind of conditions
give rise to the error. Use the search engine cache to retrieve examples of error messages that no longer appear within the live application.
13.4 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. For example:
`unable\ to\ retrieve lang:php package:mail`
13.5 If you receive error messages with stack traces containing the names of library and third-party code components, search for these names on both types of search engine.

Made with ❤️ By **@sl4x0**