Web Security Basics

In the early days of the Internet, the vast majority of websites were static. This means the content served to a user was largely fixed and identical for everyone, regardless of who was viewing it or when. The primary flow of information was in a single direction, exclusively from the server to the browser. User interaction was minimal, mostly limited to clicking links. Authentication and login mechanisms were either absent or rudimentary. Due to this lack of interactivity and personalized content, a compromise of an early website generally did not expose critical user or secret business information.

Today, the Web is dominated by web applications: highly interactive and complex systems that extend far beyond simple content delivery. These applications are the backbone of e-commerce, social media, and business operations. A bidirectional flow of information is now the norm, requiring constant interaction and communication between the user (client) and the server. This heavy communication extensively on Application Programming Interfaces (APIs) to facilitate the exchange of data and services. Modern web applications are composed of multiple, interconnected components that work together (e.g., Databases, Scripting Code, Mail Servers, Cloud Storage, etc.). Web pages are dynamically generated in real-time. Their content is highly personalized and dependent on factors like the user's authentication status and the specific systems making the request. Crucially, web application typically store, process, and transmit critical information, including personal data, financial records, and proprietary business secrets. This makes robust security and authentication non-negotiable, as a compromise can have significant, high-impact consequences.

1.1 The Core Security Challenge: Arbitrary Input

The main security problem facing modern web application sis the handling of arbitrary input. Attackers can deliberately manipulate inputs to a web application, such as URL parameters, POST request bodies, cookies, or HTTP headers, to alter the application's intended behavior. This manipulation can be used for various malicious purposes, including bypassing security controls, fraud, impersonation, data theft and so on.

It is critical to enforce strict security and validation over every piece of data a user provides. The basic defense mechanism must revolve around the fundamental acknowledgment that a web application can receive arbitrary, potentially hostile inputs. This necessitates proactive measures, which include:

- *Handling User Authentication Data*: Ensuring that mechanisms prevent one user from successfully logging in or acting as another user.
- *Handling Possible Malformed Inputs*: Validating and sanitizing all inputs to prevent data that is structurally incorrect or contains malicious code from being processed.
- *Handling Potential Attacker Actions*: Implementing controls to prevent denial-of-service (DoS) style actions, such as slowing down or overwhelming the system through multiple subsequent, rapid operations.

- *Administrator Actions to Log Application Activity*: Maintaining comprehensive logs of application activity to monitor for suspicious behavior and provide an audit trail for forensic analysis.

Handling User Authentication Data. Securing a web application begins with reliably determining and controlling who a user is and what they are allowed to do. This involves three interconnected concepts: Authentication, Session Management, and Access Control.

(1) Authentication is the process of correctly verifying a user's claimed identity. While the lowest degree of access is anonymous, this is largely avoided or restricted in modern applications that handle sensitive data. Credential-based authentication is the most straightforward and common method, and relies typically on a username and password. However, relying solely on standard username/password combination is often considered weak due to their susceptibility to various attacks (e.g., brute-forcing, dictionary attacks, or credential stuffing). To counter these weaknesses, multifactor authentication (MFA) is widely employed. MFA requires the user to provide two or more distinct verification factors.

(2) Once a user is authenticated, Session Management provides a powerful way for the application to handle multiple, subsequent requests from the same user without requiring them to log in again for every action. Applications typically generate a unique, cryptographically secure session token that identifies the user's active session. This token is then transmitted (usually via a cookie or HTTP header) with every authenticated communication between the browser and the server. Because this token is the *de facto* identity of the user for the duration of the session, the security of these tokens is crucial. If an attacker steals a valid session token (a process known as session hijacking), they can impersonate the legitimate user.

(3) Finally, Access Control determines which contents, resources, or functions an authenticated user is permitted to utilize or view. By default, access control ensures that unauthorized users cannot access specific contents meant only for logged-in or administrative users. Access control is also used to limit the actions of authenticated users based on their role. For example, in a banking application, money transfers might be limited to users who have completed extra verification steps, or an administrative panel may only be accessible by users with the "Administrator" role.

Handling Malformed Inputs. A fundamental step in securing a web application is the rigorous evaluation of all data submitted by the user. Arbitrary input can be deliberately malformed or crafted to trick the application into parsing unexpected data or executing hostile commands. To mitigate this, applications employ several strategies. Many forms put limitations on what the user can insert (e.g., character limits, allowed formats), while others may employ hidden form values: data sent with the form that the user is not intended to see or deliberately manipulate. However, these checks are easily bypassed by savvy attackers who can manipulate the raw HTTP request, meaning server-side validation is always mandatory. The primary defensive techniques for handling malformed inputs are:

- *Reject Known Bad*: This technique involves blacklisting specific input patterns, keywords, or characters known to be associated with attacks. The application scans the input for malicious signatures, such as SQL keywords (`SELECT` , `UNION` , `DROP`) or script tags (`<script>`) associated with SQL Injection or Cross-Site Scripting (XSS) attacks. Blacklisting is easily bypassed by attackers who user obfuscation. For example, blocking `SELECT` can be circumvented by using `SeLeCt` or encoding techniques. Attackers may use a NULL-byte attack (injecting `%00`) in the input string. In some older or poorly written applications, the server-side logic might stop processing the string at the first null byte, causing the security filter to not analyze the rest of the string. However, the subsequent application logic (like a database query) might still execute the full, malicious payload.

- *Accept Known Good and Sanitization*: This approach is significantly more robust than blacklisting because it defines and accepts only a safe subset of possible inputs. Accept Known Good employs a whitelist of explicitly accepted characters, formats, or values. If an input contains anything outside this approved list, it is rejected. This is often the most effective way to filter input, especially for fields with known constraints (like a phone number or a specific set of drop-down options). Sanitization provides automatic fixes to input that is potentially malicious but cannot be automatically be rejected. Instead of rejecting the whole input, sanitization modifies the dangerous parts: for instance, replacing `<` , `>` , and `&` with their harmless HTML entities (`&lt;` , `&gt;` , `&amp;`) before the data is stored or displayed.

- *Safe Data Handling and Semantic Checks*: These techniques focus on processing data securely, regardless of the input source. Safe Data Handling means utilizing formally correct coding procedures to prevent the application's underlying code from misinterpreting user input as code. The best example is using parameterized queries (or prepared statements) for all database interactions. This technique ensures user input is always treated purely as data and can never be executed as part of an SQL command, virtually eliminating most forms of SQL Injection. While an input may be syntactically correct and not contain malicious characters, it can still be used in malicious ways based on the context of the transaction. A semantic check ensures that the data makes logical sense in the context of the application's business logic. For example, a user attempting to purchase an item might pass a price parameter that is technically a valid number, but a semantic check would reject it if the price doesn't match the actual, server-side price of the product.

It is tempting to believe that simple data sanitization at the point of entry is sufficient to secure a web application. This approach assumes that one only needs to worry about the raw input data and not the internal web application logic, leading to the idea that all checks can be performed as a single, monolithic, a priori phase. However, relying on a single input validation step is insufficient. Web applications perform multiple data processing steps across various internal components. Data that is safe at the entry point might become hostile after an intermediate processing step. The variety and complexity of potential attacks are too high to be fixed in one phase. Furthermore, the sanitization techniques effective against one attack (e.g., removing angle brackets for XSS) might not be compatible or effective against another (e.g., preventing a logic flaw). The correct approach is Boundary Validation, which means making checks in multiple stages throughout the application's lifecycle. The necessary checks depend heavily on which components are being run by the web app. For instance, input passed to a database requires parameterized queries, while input being rendered back to a user's browser requires HTML entity encoding. Checks are performed by considering the input that is passed at each state: at the external boundary and at every internal boundary.

Handling Attackers. When a web application is directly targeted, defensive strategies shift from passive prevention (like input validation) to active defense, monitoring, and reaction. The core objective is to reduce the attacker's motivation and opportunity by implementing robust error handling, detailed logging, and a swift incident response plan.

- *Handling Errors*: When an unexpected event occurs (such as trying to visit a non-existent page or a server-side exception), the application must avoid leaking sensitive information. The primary goal of secure error handling is to mask internal server details. Generic error messages (e.g., "An unexpected error occurred") should be shown to the user, while detailed information (e.g., stack traces, database query errors, server paths) must be logged internally but never displayed publicly. Leaking such details can provide attackers with clues about the application's architecture and potential vulnerabilities.
- *Maintaining Audit Logs*: Audit logs are indispensable tools for monitoring system activity, conducting forensic analysis, and detecting suspicious behavior. Logs should comprehensively monitor critical user activities, including: successful and failed login attempts, payment transactions and other business-critical actions, and blocked access attempts and security violations. Every log record must contain essential information to identify and trace the event, such as timestamp, source IP address, and the session ID or user account.
- *Alerting Administrators and Reacting to Attacks*: Automatic alerts are essential for enabling prompt administrative reactions to ongoing attacks. Alerting systems are typically triggered by anomalies, such as:
  - Usage Anomalies: Sudden spikes in failed login attempts or unusually high request volumes.
  - Business Anomalies: Transactions outside typical parameters (e.g., an unreasonably large money transfer).
  - Attack Signatures: Requests containing known malicious strings or attempts to manipulate hidden form data.

  Upon detection, automated or manual reactions can be deployed: intentionally slowing down requests from a suspicious source to hinder automated attacks, temporarily or permanently banning specific source IP addresses or user accounts. Automatically logging out users after a certain time or number of failed attempts.
- *Securing Administrative Functionalities*: The administrative components of a web application represent a high-value target and a severe vulnerability if compromised. Administrative Functionalities are often assumed to be more secure than public-facing parts, but vulnerabilities here can allow an attacker to compromise the entire application (e.g., gain full database access or execute remote code). Administrators possess powerful privileges, such as the ability to influence other users' sessions, view sensitive information, or grant powerful access rights. If an attacker gains access to the administrator panel, they acquire these same, potentially devastating, capabilities.

Web Application Protocols

2.1 HTTP

HTTP, or Hypertext Transfer Protocol, is the foundational communication protocol of the World Wide Web. The fundamental interaction involves a client sending a message (a request) to a server, which then processes it and sends a message back (a response) to the client. HTTP typically utilizes TCP (Transmission Control Protocol) connections for reliable data transfer. Crucially, HTTP is a stateless protocol, meaning that each request-response exchange is independent and self-contained; the server does not inherently remember past interactions with the client. While various requests might use different TCP connection variants, the protocol's core request/response mechanism remains autonomous.

Request. An HTTP Request is structured with several key parameters:

- *Method (or Verb)*: Specifies the action to be performed on the target resource (e.g., `GET` , `POST` , `PUT` , `DELETE`).
- *Resource*: Specifies the part of the URL identifying the target resource.
- *HTTP Protocol Version*: Indicates the version of the HTTP protocol being used (e.g., `HTTP/1.1` , `HTTP/2` , `HTTP/3`).
- *Host*: Specifies the hostname (domain name) of the server for the requested resource.
- *User-Agent*: Provides information about the client application (usually a browser) that generated the request, including its type and operating system.
- *Referer*: Indicates the URL of the page that linked to the requested resource.

```http
HTTP
POST /api/login HTTP/1.1
Host:  www.example.com
User-Agent:  Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
Content-Type:  application/json
Content-Length:  52
Referer:  https://www.example.com/login


{"username":  "alice123", "password":  "superSecret123"}
```

Figure 1 / Example HTTP POST request for user authentication

Other important request parameters (often included as headers) include:

- *Connection*: Controls whether the underlying network connection should remain open or be closed after the current transaction finishes (e.g., `keep-alive` , `close`).
- *Cache-Control*: Used to specify caching directories for both requests and responses, influencing how intermediaries and the client should store and reuse responses.

- *Upgrade-Insecure-Requests*: Sends a signal to the server (typically with a value of `1`) expressing the client's preference for an encrypted and authenticated response, often prompting a redirect to a secure HTTPS server.
- *Sec-Fetch-**: A set of experimental headers (like `Sec-Fetch-Mode` , `Sec-Fetch-Dest` , `Sec-Fetch-Site`) designed to provide contextual information about the request (e.g., its origin and purpose) to allow the server or intermediaries to determine a priori if the request should be served, primarily for security against cross-site leaks.
- *Accept*: Provides a list of media types (e.g., `text/html` , `application/json`) that the client is capable of processing. This can be refined by related headers such as `Accept-Encoding` and `Accept-Language` .

In a standard URL structure, parameters are often passed using a query string, which begins with a question mark (`?`). This character signifies the start of the key-value pairs used to pass data to the server. The query string is used to pass parameters or data to the resource identified by the URL. The ampersand character (`&`) is used as a delimiter to combine multiple parameters within the query string. REST (Representational State Transfer) architecture provides an alternative and often cleaner way to represent resource addresses, favoring path variables over query strings for identifying a resource or a collection of resources. In RESTful URLs, resources are typically identified by hierarchical paths that align with their logical structure, promoting better clarity and meaning. Instead of `.../users?id=123` , a RESTful approach might use `.../users/123` to uniquely identify user 123.

URL Encoding defines the set of Allowed ASCII codes as ranging from `0x20` to `0x7e` . Some problematic characters (for example: `0x20` for space, `0x0a` for line feed) are encoded using the percent sign followed by the character's hexadecimal value, like `%20` or `%0a` . This process is necessary to ensure that characters that have special meaning in a URL, or those that are non-printable, are transmitted safely. In some cases, especially for characters belonging to other languages, UNICODE is used. This typically involves a two-byte representation in older URL encoding standards, as seen in the example `%u2215` → `/` . UTF-8 is the modern and most common standard, being a multibyte way to represent characters. An example of a UTF-8 encoding is `%e2%89%a0` . This encoding is widely used for data transmitted across the web, including parameters like those found in HTTP cookies. Beyond URL and character set encodings, other formats are employed for data integrity and representation. HTML encoding uses entities to represent special characters that might conflict with the document's markup (e.g., `&quot` ; → "). Base64 encoding is useful to represent binary data (like cryptographic keys or images) as ASCII strings, so they can be safely included in text-based protocols like HTTP headers. Finally, Hex encoding is also useful to represent raw binary data in a readable, two-character hexadecimal format.

Response. The following parameters are typically found in the response header:

- *HTTP Protocol Version*: Indicates the version of the HTTP protocol the server used for the response.
- *Status Code*: A three-digit number that indicates the result of the request attempt. The first digit defines the class of the response: `1xx` (Informational), `2xx` (Success, e.g., 200 OK), `3xx` (Redirection, e.g., 301 Moved Permanently), `4xx` (Client Error, e.g., 404 Not Found), and `5xx` (Server Error, e.g., 500 Internal Server Error).

- *Reason Phrase*: A short, readable sentence that further explains the status code (e.g., for status code 200, the reason phrase is "OK").
- *Data*: Specifies the data and time when the response was generated by the server.
- *Accept-Ranges*: If this field is present and its value is different from `none` , it signifies that the server can accept and process partial requests for the resource.
- *Last-Modified*: The data and time of the last modification of the requested resource on the server. Clients can use this to optimize caching by sending conditional requests.
- *Access-Control-**: A group of fields (e.g., `Access-Control-Allow-Origin`) related to Cross-Origin Resource Sharing (CORS). These fields define which external domains are allowed to access the resource and how, acting as a crucial part of access control for web applications.

```http
HTTP/1.1 200 OK
Date:  Mon, 21 Oct 2024 14:30:00 GMT
Server:  Apache/2.4.41 (Ubuntu)
Last-Modified:  Fri, 18 Oct 2024 09:15:22 GMT
Accept-Ranges:  bytes
Content-Length:  1256
Content-Type:  text/html; charset=UTF-8
Access-Control-Allow-Origin:  *
Connection:  keep-alive


<!DOCTYPE html>
<html lang="en">
<head>
<title>Alice's Profile</title> ...
```

Figure 2 / Example HTTP response for successful login.

Cookies. Cookies are tokens that a server sends to the user's web browser. Their primary purpose is to help maintain state in the otherwise stateless HTTP protocol, allowing the server to remember information about the user across multiple requests. A cookie is initially created and sent by the server via the response header `Set-Cookie` (e.g., `Set-Cookie:  tracking=tI8rk7joMx44S2Uu85nSWc`); multiple cookies can be issued by sending multiple `Set-Cookie` headers in a single response. In subsequent requests to the same server, the client automatically retransmits the saved cookie data using the `Cookie` header (e.g., `Cookie:  tracking=tI8rk7joMx44S2Uu85nSWc`). Cookies are typically stored as key-value pairs, though they can also be a single string without spaces.
The behavior and score of a cookie are controlled by parameters set within the initial `Set-Cookie` header:

- *Expires*: Defines a specific date and time after which the cookie will be deleted by the browser. If set, this causes the browser to save the cookie to persistent storage on the user's hard drive, allowing it to be reused in subsequent browser sessions until the expiration data is reached.
- *Domain*: Specifies the domain for which the cookie is valid. The value must be the same domain that set the cookie or a parent domain. The browser will only send the cookie to

requests made to this specified domain or its subdomains.

- *Path*: Specifies the URL path on the server for which the cookie is valid. The cookie will only be submitted for requests whose path starts with this value.
- *Secure*: A flag indicating that the cookie should only be submitted by the browser over secure channels, meaning the cookie will only be sent with HTTPS requests, preventing transmission over unencrypted HTTP.

2.2 Server/Client-Side Technologies

Server-Side. Parameters are sent to the server in multiple ways: by using the query string (starting with `?` in the URL), by employing the REST interface style (where they are embedded in the URL path), by embedding them in HTTP cookies (sent via the `Cookie` header), or by embedding them in the request body when using `POST` requests. The server processes various parts of the HTTP request, and these parameters can have a huge impact on the response. For example, a certain value of the `User-Agent` header can influence the specific page or content that is visualized by the user (e.g., serving content optimized for a mobile browser). Multiple components are used on the server's side, including: scripting languages (such as PHP and Perl), web application platforms/frameworks, web servers (to handle requests), databases and filesystems (for asset storage).

Client-Side. The user interface is essential for enabling proper communication with the server, allowing results to be presented to the user and data to be sent back to the server.

- *Hyperlinks*: These are a compact way for a user to navigate and external URL (e.g., `<a href="https://www.example.com/products">View Products</a>`). HTML Forms are extensively used to collect data from the user and send it to the server, often using `GET` or `POST` requests.
- *CSS*: CSS (Cascading Style Sheets) is used to describe the presentation of a document written in markup languages (e.g., HTML). CSS instructs the browser on how to render the contents of a resource. CSS syntax uses selectors to define a class of markup elements (e.g., all paragraphs, or elements with a specific ID) to which a given set of visual and layout attributes should be applied.
- *JavaScript*: This is a scripting language that enables the client (browser) to perform actual data processing. This can improve the application's performance by offloading part of the workload from the server to the client. It also enhances usability by allowing parts of the user interface to be dynamically updated without full page reloads. JavaScript is used to:
  (1) Validate the user's data before it gets submitted to the server, catching errors locally.
  (2) Control the browser's behavior by updating the Document Object Model (DOM). The DOM is an abstract, tree-like representation of an HTML document that can be manipulated through APIs. It allows scripts to access and manipulate individual HTML elements.
- *Ajax*: Ajax (Asynchronous JavaScript and XML) is a set of techniques that employs scripting to handle certain user actions asynchronously. By doing this, the application can exchange data with the server and update parts of a page without requiring a full page reload, significantly improving responsiveness. In Ajax applications, the client communicates their action to the server using the `XMLHttpRequest` API (or the more modern `fetch` API). The server replies with compact data, often formatted as JSON. JSON (JavaScript Object Notation) is a lightweight data interchange format used to serialize data. This compact

JSON data is then received and further processed by the client-side scripting language do dynamically update the DOM.

Sessions. The concept of a Session is crucial for maintaining a sense of continuity for a user across the stateless HTTP protocol. Once a user is authenticated, they can perform multiple actions, and the web application must be sure that each subsequent request is issued by the same user. For this reason, the server maintains a data structure that holds the user's current state and related information. This server-side data structure is called the session. Since HTTP is stateless, a unique session identifier must be constantly sent by the client and received by the server to look up and update the user's activity. There are many ways to manage and implement sessions, with the most commonly used mechanism being HTTP Cookies. A unique session ID is typically stored in a cookie on the client side; this cookie is then retransmitted with every request. The reliance on these external parameters can be dangerous if an attacker can access or hijack them, potentially leading to session hijacking and unauthorized access to the user's account.

2.3 Burp

Burp Suite is a professional, modular, Java-based software suite designed for comprehensive security testing and analysis of web applications. It operates fundamentally as an HTTP/HTTPS proxy, positioning itself as a man-in-the-middle to intercept, inspect, and modify all traffic between the client's browser and the target server. Its modular design allows for numerous extensions (plugins) and includes tools for executing various types of attacks, such as:

- Web application enumeration.
- Bruteforcing request (Intruder).
- Manual manipulation and resending of requests (Repeater).
- Automated scanning for known vulnerabilities (Scanner).

The `Target` tool serves as the core hub for a penetration testing project, providing a consolidated, high-level overview of the target application's content and functionality. Its primary components and functions are:

- *Site map*: A hierarchical tree view that records all discovered application content (hosts, folders, files, parameters). This map is populated through traffic passing through the Proxy and through Live Passive Crawling. It helps the tester visualize the application's complete attack surface.
- *Scope*: This is used to explicitly define which hosts, protocols, and URLs are in-scope for the current testing project. Setting the scope is critical because it allows the tester to filter out irrelevant traffic in the Proxy history and Site map, and configure Burp's other tools to only process traffic directed at the intended target, preventing accidental attacks on out-of-scope systems.

Burp's security auditing capabilities are split into Active and Passive checks. Passive analysis is a fundamental, non-intrusive method of vulnerability identification. The passive scanner works by only analyzing the requests and responses that naturally pass through Burp's proxy. It does not send any new, modified, or specially crafted requests to the server, ensuring the target application is not affected or alerted.

The `Proxy` tool is the fundamental and most frequently used component of Burp Suite, acting as an intercepting web proxy. The proxy is configured to sit between your browser and the web application's server. All traffic must pass through Burp, making it a powerful MITM tool. The

primary feature is its ability to intercept all incoming and outgoing HTTP/HTTPS messages. When interception is turned on, the tool holds the request or response, preventing it from continuing its journey until the tester manually forwards it. While a message is intercepted, the tester can inspect and modify any part of the raw data, including the URL, headers, and the message body. The HTTP History sub-tab logs every request and response that passes through the proxy. This creates a complete, searchable record of the application's entire traffic flow, which is essential for discovery and later analysis.
The `Repeater` tool is designed for manual, iterative testing of individual HTTP and WebSocket requests. It is the core tool for manually exploring the behavior of a specific application endpoint. It allows a tester to take a single request, modify it, and resend it repeatedly to the server without needing to interact with the browser again. Repeater is ideal for testing input-based vulnerabilities through trial and error, such as:

- *Injection Flaws* (e.g., SQL Injection, XSS) by repeatedly adjusting parameters with various payloads.
- *Access Control Flaws* (e.g., IDORs) by changing parameter values like user or product ids to access unauthorized resources.
- *State Manipulation* by sending requests in a specific sequence to test multistep processes.

Each request sent to Repeater is given its own tab, allowing the tester to work on multiple distinct test cases simultaneously. Within each tab, a full history of the modifications and corresponding server responses is maintained, making it easy to track testing progress and compare different responses.

Spidering and Parameter Analysis. Spidering is an essential reconnaissance technique used to automatically discover and map the entire content and hierarchy of a web application, including its folders, files, and links. Automatic spidering tools work by mimicking a user's navigation to systematically build a complete site map. The tool starts with an initial URL, analyzes the HTML content of the page, automatically detects all embedded links, parses forms, and follows these links to recursively discover deeper parts of the site. The `robots.txt` file, typically located in the web app's main folder `root` , contains a list of directories and files that search engines are requested not to retrieve. It is important to note that this is only a convention. A malicious spider will often check this file precisely because it may inadvertently expose hidden or sensitive areas of the application.
Automatic tools are effective but face several limitations, especially with modern applications:

- *Dynamic Content*: They often cannot automatically parse content generated dynamically on the client-side (e.g., using complex JavaScript scripts), leading to skipped resources.
- *Non-Standard Content*: Links embedded within legacy objects like Flash or Java applets are frequently skipped.
- *Multi-State Functionality*: Complex workflows, such as multistep forms or multipage checkout processes, are often not parsed or traversed correctly as the tool may fail to maintain the necessary session state or required sequence of steps.
- *Authentication and Session*: Spiders generally cannot automatically handle authentication mechanisms. The tester must manually provide session tokens or configure Burp to manage the session state.

To overcome the limitations of automatic tools and to gain context, manual spidering is performed concurrently with the automatic process. The tester should manually attempt to access the paths listed in the `robots.txt` file, as the list itself is an information disclosure vulnerability. Manually reviewing JavaScript source code can reveal URLs, functions, and endpoints that are

never linked in the visible HTML but are called dynamically. The tester must manually log into the application and then browse through all privileged or user-specific pages. This action forces the traffic to pass through the Burp Proxy, populating the Site map with authenticated-session resources.
In single-URL applications, the functionality is determined not by a new URL path, but by modifying a parameter within the request to a single resource. Spidering the base URL repeatedly will be unproductive. The key is to analyze and test the request parameters. The internal application logic is often controlled by a specific parameter, such as a hidden form field or a query string value.

```http
POST /bank.jsp HTTP/1.1
Host:  wahh-bank.com
Content-Length:  106
servlet=TransferFunds&method=confirmTransfer&fromAccount=10372918&to
        Account=3910852&amount=291.23&Submit=Ok
```

In the request above, the functionality being accessed is completely controlled by the `servlet` and `method` parameters. To enumerate other possible functionalities, a tester would use the Repeater or Intruder tools to modify these parameter values and observe the server's response. This is essential for discovering hidden or undocumented application functions.