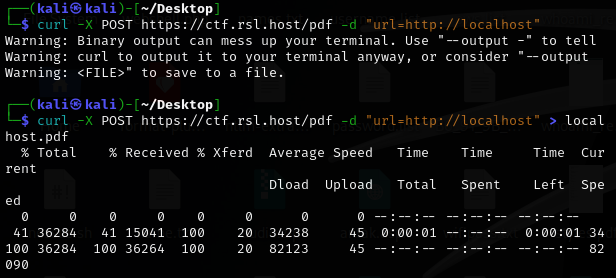
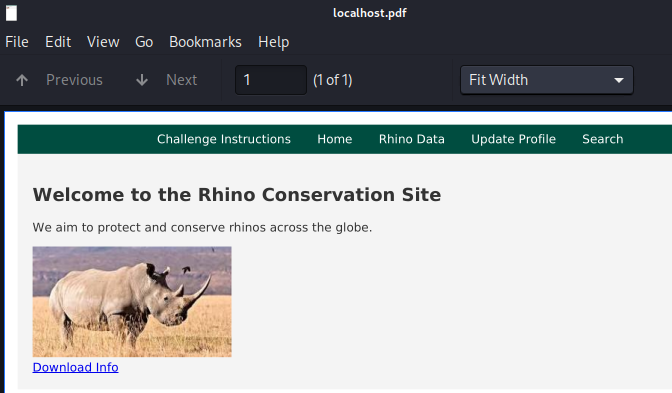
Web Challenge Report

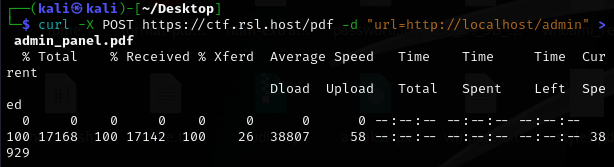
Joshua Babcock  
jobabcoc@syr.edu

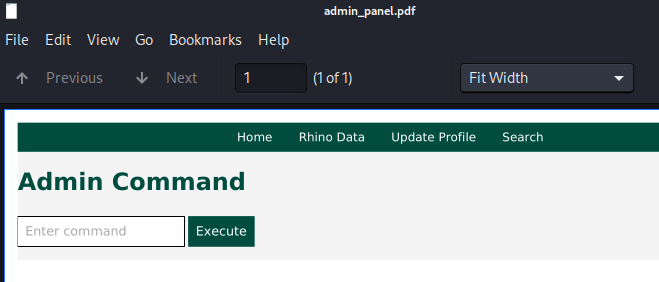
1. Name: SSRF -> RCE via PDF Generation
   * + Risk Rating: [**Critical** / High / Medium / Low]
     + Exploitation Likelihood: [**Critical** / High / Medium / Low]
     + Potential Impact: [**Critical** / High / Medium / Low]
   1. Description
      1. The PDF generation endpoint (*/pdf*) accepts a user-controlled *url* parameter that is processed by the *wkhtmltopdf* library without proper validation. This creates a Server-Side Request Forgery (SSRF) vulnerability that allows attackers to make arbitrary HTTP requests from the server. More critically, this SSRF can access an internal admin panel at *http://localhost/admin* that provides command execution capabilities. The combination of SSRF and the exposed admin interface results in Remote Code Execution with root privileges, representing a complete system compromise.
   2. Remediation
      1. Implement strict URL validation and whitelisting for the PDF generation endpoint
      2. Remove or properly secure the admin interface with authentication and authorization
      3. Use a separate, isolated service for PDF generation with restricted network access
      4. Implement network segmentation to prevent access to internal services
      5. Apply the principle of least privilege - run the application with a non-root user
   3. Testing Process
      1. Identified the hidden PDF form in the homepage source code with a *url* parameter
      2. Tested basic SSRF functionality: *curl -X POST https://ctf.rsl.host/pdf -d "url=http://localhost"*
      3. Discovered internal admin panel: *curl -X POST https://ctf.rsl.host/pdf -d "url=http://localhost/admin"*
      4. Achieved RCE through admin interface: *curl -X POST https://ctf.rsl.host/pdf -d "url=http://localhost/admin?cmd=whoami"*
      5. Confirmed root access and full system compromise: *curl -X POST https://ctf.rsl.host/pdf -d "url=http://localhost/admin?cmd=id"*

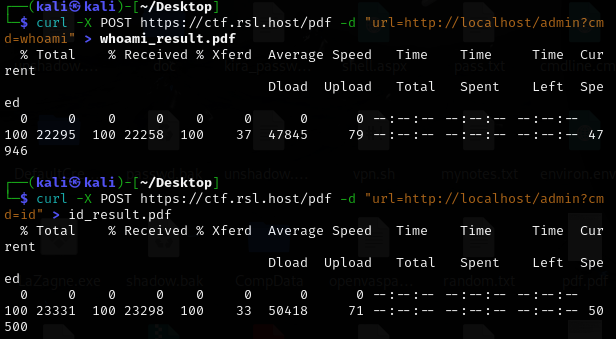


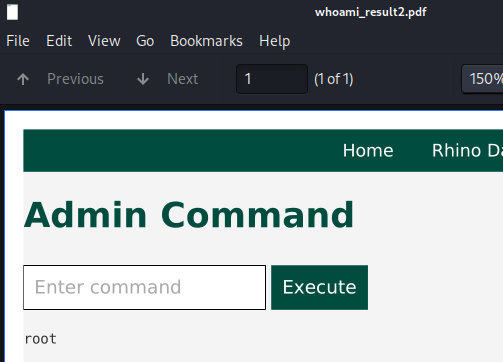


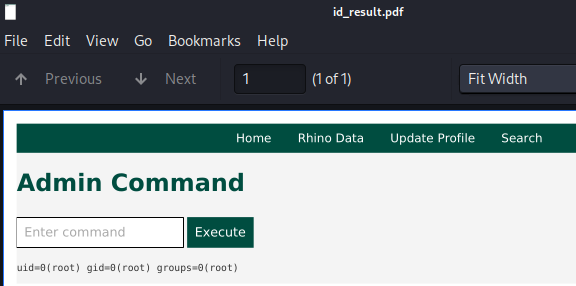




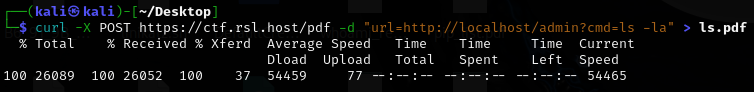


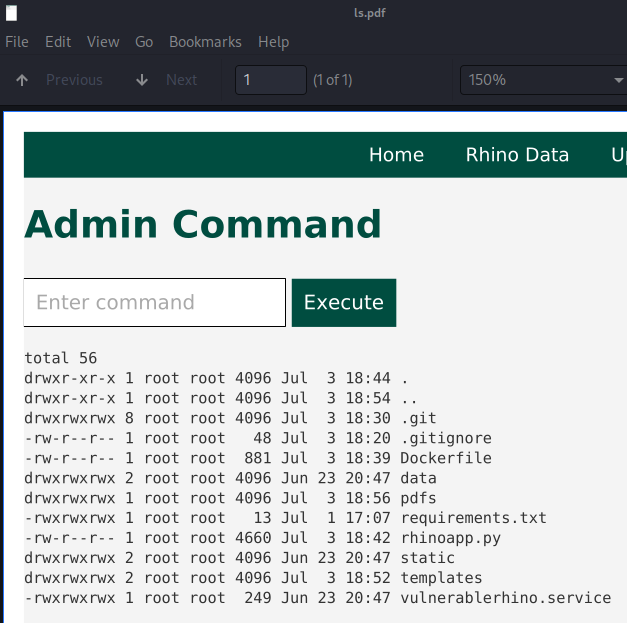


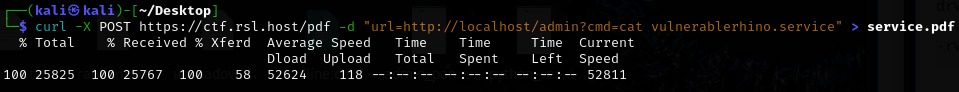


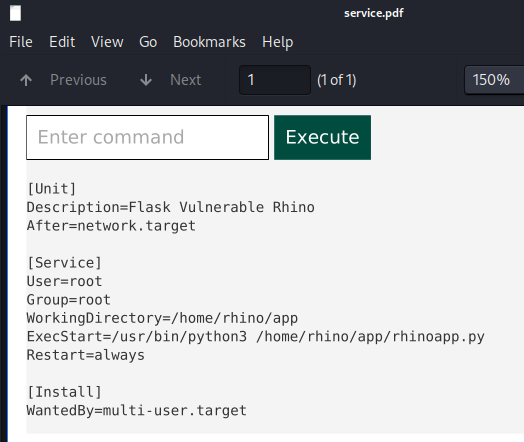


1. Name: Application Running as Root User
   * + Risk Rating: [Critical / **High** / Medium / Low]
     + Exploitation Likelihood: [Critical / **High** / Medium / Low]
     + Potential Impact: [**Critical** / High / Medium / Low]
   1. Description
      1. The Flask application is configured to run with root privileges (UID 0), violating the fundamental security principle of least privilege. This configuration significantly amplifies the impact of any code execution vulnerabilities, as demonstrated by the SSRF→RCE chain that immediately provides root access. Running web applications as root creates an unnecessarily large attack surface and can lead to complete system compromise from otherwise limited vulnerabilities.
   2. Remediation
      1. Create a dedicated non-privileged user account for the application
      2. Update the *systemd* service file to use *User=appuser* and *Group=appgroup*
      3. Ensure the application user has only the minimum permissions required
      4. Implement proper file system permissions and ownership
      5. Use containers or sandboxing technologies to further isolate the application
   3. Testing Process
      1. Leveraged the RCE vulnerability to execute *whoami* command
      2. Confirmed root user: *curl -X POST https://ctf.rsl.host/pdf -d "url=http://localhost/admin?cmd=whoami"*
      3. Verified with *id* command showing *uid=0(root) gid=0(root)*
      4. Examined service configuration file showing *User=root* and *Group=root*

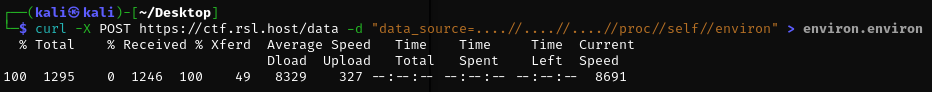






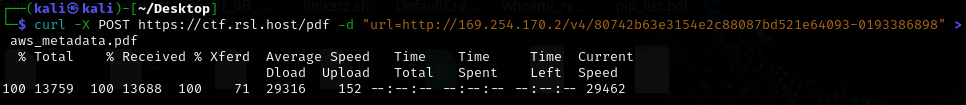


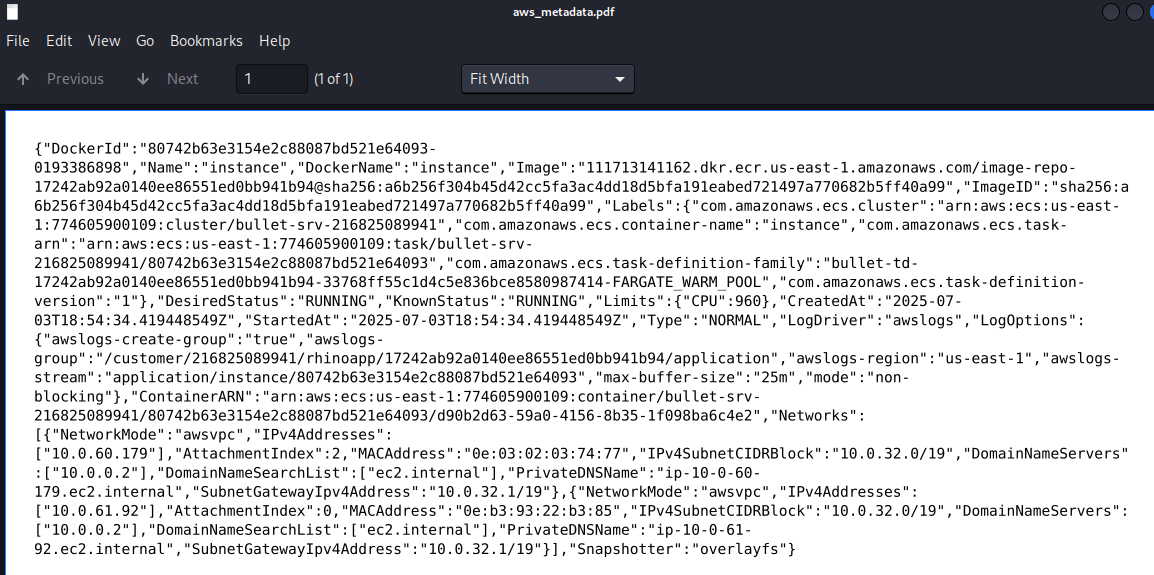
1. Name: AWS Metadata Service Access via SSRF
   * + Risk Rating: [Critical / **High** / Medium / Low]
     + Exploitation Likelihood: [Critical / **High** / Medium / Low]
     + Potential Impact: [Critical / **High** / Medium / Low]
   1. Description
      1. The SSRF vulnerability allows access to AWS ECS metadata endpoints, exposing sensitive cloud infrastructure information. The application running in AWS ECS Fargate exposes container metadata through *169.254.170.2*, revealing AWS account details, container ARNs, network configuration, and task definitions. This information can be used for reconnaissance, lateral movement, and potential privilege escalation within the AWS environment.
   2. Remediation
      1. Implement network-level restrictions to block access to metadata endpoints
      2. Use IMDSv2 (Instance Metadata Service Version 2) which requires session tokens
      3. Configure VPC endpoints and security groups to restrict metadata access
      4. Implement proper IAM roles with minimal permissions
      5. Monitor and log access to metadata services
   3. Testing Process
      1. Discovered ECS metadata endpoints in environment variables through RCE
      2. Accessed metadata service: *curl -X POST https://ctf.rsl.host/pdf -d "url=http://169.254.170.2/v4/* *80742b63e3154e2c88087bd521e64093-0193386898*"
      3. Extracted sensitive information including AWS account ID (774605900109)
      4. Retrieved container configuration, network details, and task definitions



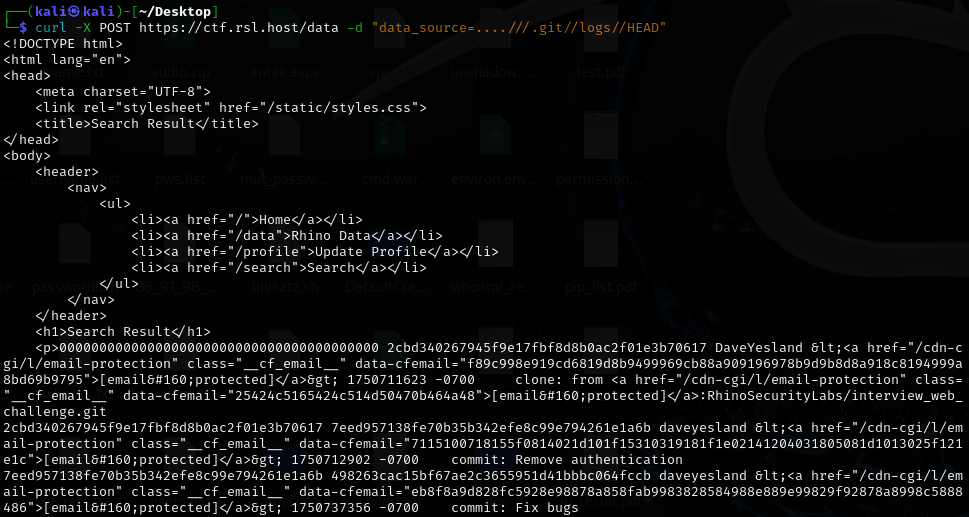
A screenshot of a computer

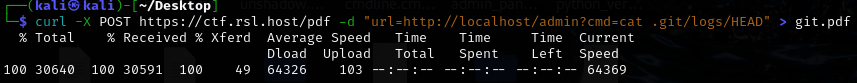
AI-generated content may be incorrect.

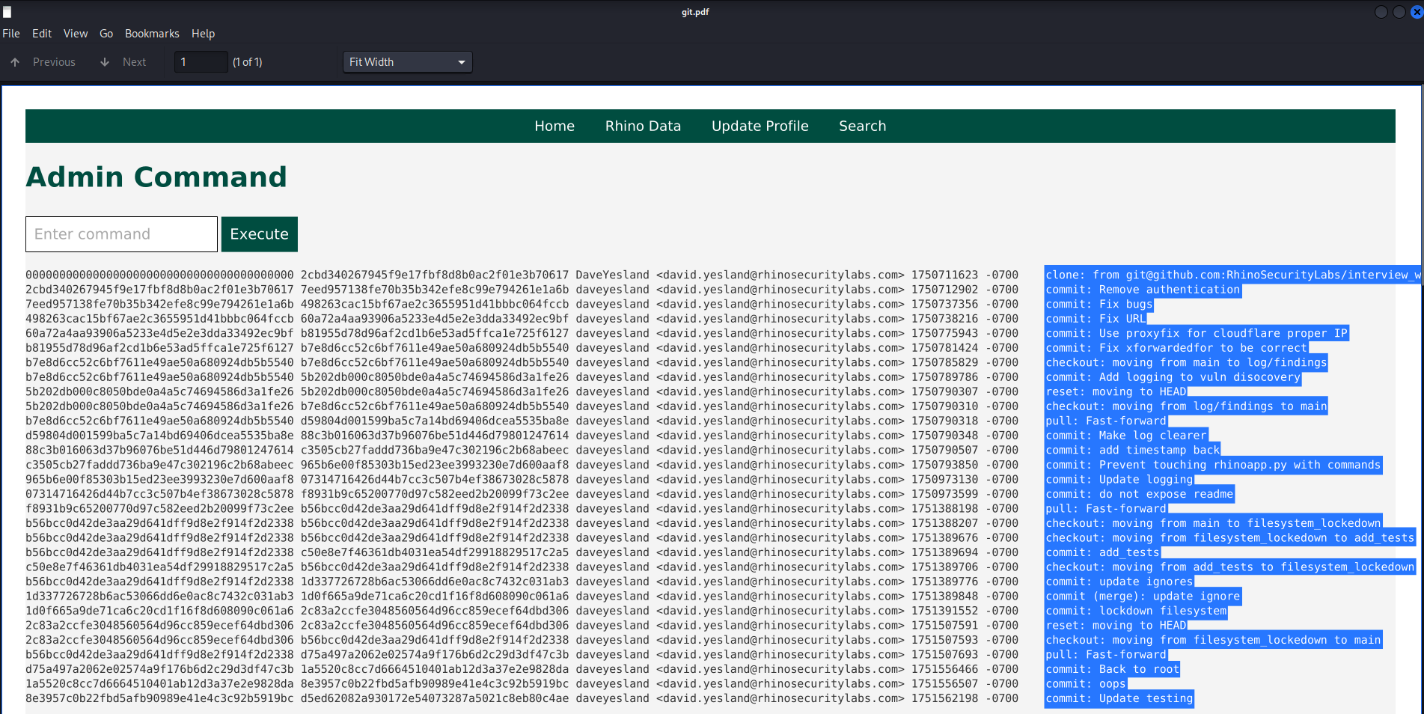




1. Name: Information Disclosure via Exposed Git Repository
   * + Risk Rating: [Critical / **High** / Medium / Low]
     + Exploitation Likelihood: [Critical / **High** / Medium / Low]
     + Potential Impact: [Critical / **High** / Medium / Low]
   1. Description
      1. The application's *.git* directory is accessible through the path traversal vulnerability, exposing the complete development history and sensitive information. The git logs reveal developer email addresses, repository locations, and commits. This information disclosure provides valuable reconnaissance data for attackers and may contain credentials, API keys, or other sensitive data in commit history.
   2. Remediation
      1. Remove *.git* directories from production deployments
      2. Use proper build pipelines that exclude development artifacts
      3. Implement *.gitignore* files to prevent committing sensitive data
      4. Use environment variables or secure vaults for configuration data
      5. Regular security audits of git history for exposed secrets
   3. Testing Process
      1. Discovered *.git* directory through admin panel file listing
      2. Accessed git logs: *curl -X POST https://ctf.rsl.host/data -d "data\_source=....//....//....///.git//logs//HEAD"*
      3. Extracted developer information: *david.yesland@rhinosecuritylabs.com*
      4. Identified repository: *git@github.com:RhinoSecurityLabs/interview\_w*







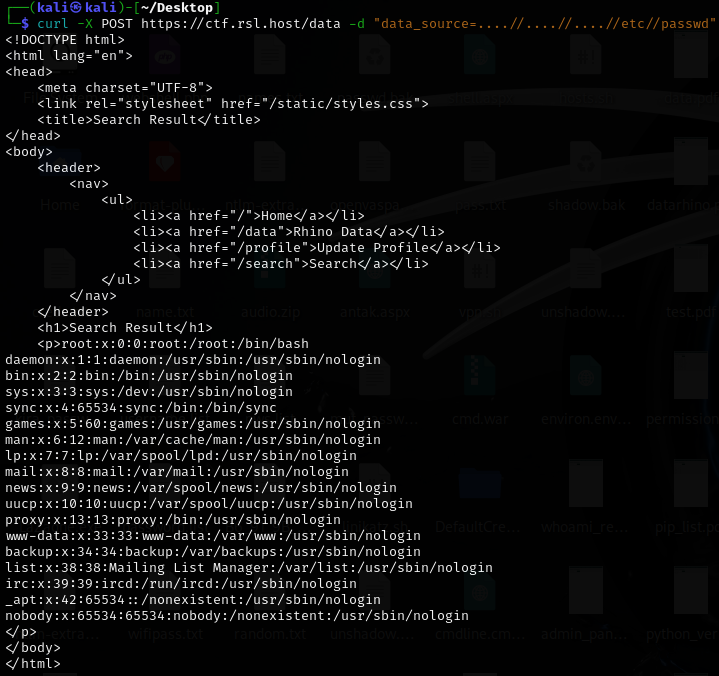
1. Name: Missing Security Headers
   * + Risk Rating: [Critical / High / **Medium** / Low]
     + Exploitation Likelihood: [Critical / High / **Medium** / Low]
     + Potential Impact: [Critical / High / **Medium** / Low]
   1. Description
      1. The application lacks essential HTTP security headers that protect against common client-side attacks. Missing headers include *X-Content-Type-Options*, *X-Frame-Options*, *Content-Security-Policy*, *X-XSS-Protection*, *Strict-Transport-Security*, and *Referrer-Policy*. The absence of these headers makes the application vulnerable to clickjacking, MIME sniffing attacks, cross-site scripting, and other client-side security issues.
   2. Remediation
      1. Implement *X-Content-Type-Options: nosniff* to prevent MIME sniffing
      2. Add *X-Frame-Options: DENY* or *SAMEORIGIN* to prevent clickjacking
      3. Configure *Content-Security-Policy* to restrict resource loading
      4. Enable *Strict-Transport-Security* for HTTPS enforcement
      5. Set appropriate *Referrer-Policy* and *X-XSS-Protection* headers
   3. Testing Process
      1. Analyzed HTTP response headers: *curl -I https://ctf.rsl.host/*
      2. Tested multiple endpoints to confirm consistent absence of security headers
      3. Verified lack of protection against common client-side attacks (*7* and *8* below)

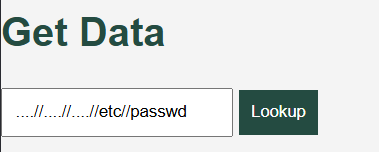


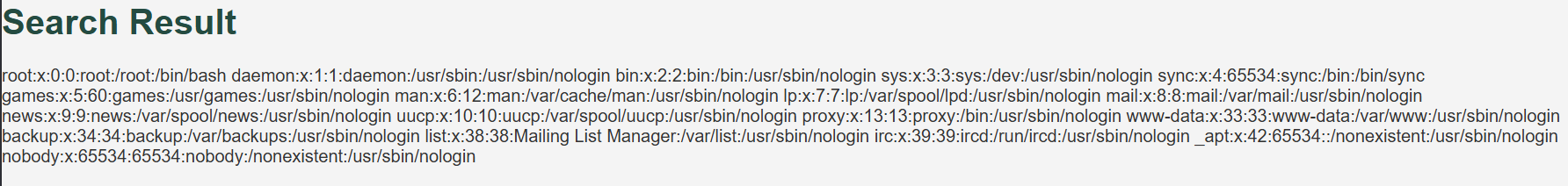
1. Name: Path Traversal - File Disclosure
   * + Risk Rating: [**Critical** / High / Medium / Low]
     + Exploitation Likelihood: [**Critical** / High / Medium / Low]
     + Potential Impact: [**Critical** / High / Medium / Low]
   1. Description
      1. The */data* endpoint contains a path traversal vulnerability that allows reading arbitrary files from the server filesystem. While basic *../* sequences are filtered, the protection can be bypassed using *....//* double-dot encoding. This vulnerability enables access to sensitive system files including */etc/passwd*, */etc/shadow*, and potentially application source code and configuration files.
   2. Remediation
      1. Implement proper input validation and sanitization
      2. Use allowlisting instead of denylisting for file access
      3. Employ secure file access APIs that prevent directory traversal
      4. Implement proper access controls and file system permissions
      5. Use chroot jails or containers to limit file system access
   3. Testing Process
      1. Tested normal functionality: *curl -X POST https://ctf.rsl.host/data -d "data\_source=rhino"*
      2. Attempted basic traversal (blocked): *curl -X POST https://ctf.rsl.host/data -d "data\_source=../../../etc/passwd"*
      3. Bypassed protection: *curl -X POST https://ctf.rsl.host/data -d "data\_source=....//....//....//etc//passwd"*
      4. Successfully accessed */etc/passwd*, and other system files

A computer screen shot of a computer screen

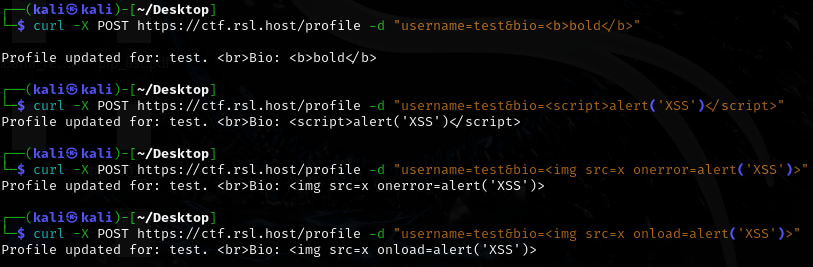
AI-generated content may be incorrect.

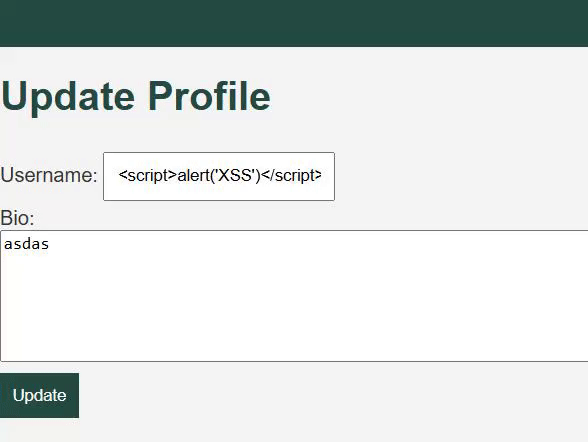


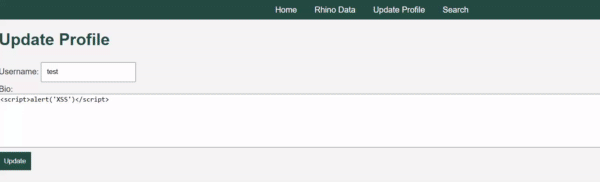


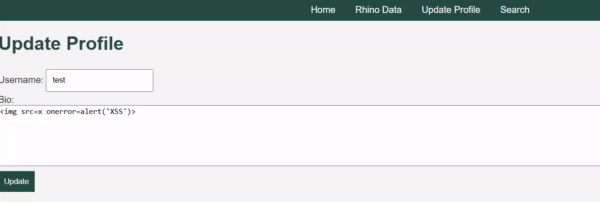


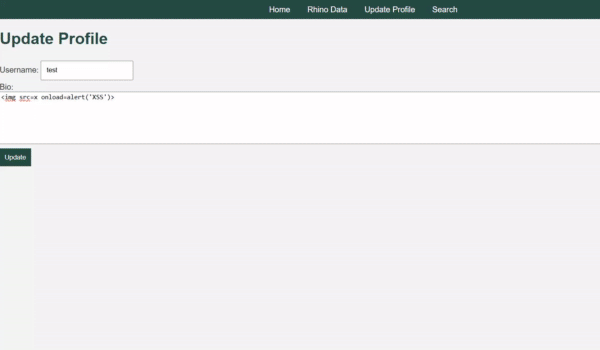
1. Name: Stored XSS in Bio Field
   * + Risk Rating: [Critical / **High** / Medium / Low]
     + Exploitation Likelihood: [Critical / **High** / Medium / Low]
     + Potential Impact: [Critical / **High** / Medium / Low]
   1. Description
      1. The profile bio field stores user input without proper sanitization, creating a stored cross-site scripting (XSS) vulnerability. While *<script>* tags are filtered, other HTML elements with event handlers like *<img src=x onerror=alert('XSS')>* are stored without modification. This allows attackers to inject malicious JavaScript that will execute when other users view the profile, potentially leading to session hijacking, credential theft, or further compromise.
   2. Remediation
      1. Implement comprehensive input sanitization using a whitelist approach
      2. Use Content Security Policy (CSP) to prevent inline script execution
      3. Apply proper output encoding when displaying user content
      4. Use established libraries like *DOMPurify* for HTML sanitization
      5. Implement input validation on both client and server sides
   3. Testing Process
      1. Tested basic HTML injection: *curl -X POST https://ctf.rsl.host/profile -d "username=test&bio=<b>test</b>"*
      2. Confirmed script tag filtering: *curl -X POST https://ctf.rsl.host/profile -d "username=test&bio=<script>alert('XSS')</script>"*
      3. Bypassed with img tag: *curl -X POST https://ctf.rsl.host/profile -d "username=test&bio=<img src=x onerror=alert('XSS')>"*
      4. Verified payload storage and potential execution in browser context











1. Name: Reflected XSS in Search Results
   * + Risk Rating: [Critical / **High** / Medium / Low]
     + Exploitation Likelihood: [Critical / **High** / Medium / Low]
     + Potential Impact: [Critical / **High** / Medium / Low]
   1. Description
      1. The search functionality reflects user input directly into HTML attributes without proper encoding. The search term is placed in both the *id* attribute of a div element and in the page content, allowing injection of malicious JavaScript through event handlers. Attackers can craft malicious URLs that execute JavaScript when victims click on them, leading to session theft, credential harvesting, or other client-side attacks.
   2. Remediation
      1. Implement proper output encoding for all user input
      2. Use context-aware encoding (HTML entity encoding for HTML context)
      3. Validate and sanitize input parameters
      4. Implement Content Security Policy to prevent inline script execution
      5. Use secure templating engines that auto-escape output
   3. Testing Process
      1. Tested normal search: *curl "https://ctf.rsl.host/search?search\_term=test"*
      2. Identified reflection in HTML: *<div id="test"><p>Results for: test</p></div>*
      3. Injected event handler: *curl "https://ctf.rsl.host/search?search\_term=test%22%20onmouseover%3D%22alert(1)%22"*
      4. Confirmed XSS execution in browser when hovering over results
      5. Tested multiple event handlers (onclick, onmouseover) with successful injection

A computer code on a black background

AI-generated content may be incorrect.

