Constructing and understanding a CTF Challenge IT Security Beginners Practical 2024

Jakob Weber
4265377
jakob.weber@stud.uni-heidelberg.de [1]

***Abstract*—Understanding vulnerabilities is a crucial aspect of cybersecurity. This paper, along with the corresponding Capture-The-Flag (CTF) challenge, aims to explore various vulnerabilities, examining how they work, how they emerge, how they are exploited, and how they are patched. Additionally, it provides guidance for programmers on how to avoid these vulnerabilities in the first place.**

I. Introduction

Solving a CTF-Challenge is a commonly used way to learn more about vulnerabilities and exploits. Similarly setting up a CTF challenge yourself can help to understand how vulnerabilities are introduced into software and what steps could have been taken to avoid them.

Exactly this was the goal in setting up the particular CTF challenge [2] for the beginners practical for university that is referenced in this paper.

The constructed CTF aims to cover a wide range of vulnerabilities while trying to stick to a realistic setup for a website, which consists of a frontend for the website, a backend that handles database interaction and a monitoring server that checks the frontend for availability.

The vulnerabilities covered in the CTF include Stored Cross Site Scripting, Injection (SQL Injection, PHP shellcode injection), Broken Access Control (CVE-2019-14287, CVE-2019-5736), Identification and Authentication Failures (CVE-2018-15473), Cryptographic Failures (weak RSA Modulus) and Vulnerable and Outdated Components (CVE-2019-14287, CVE-2019-5736, CVE-2018-15473) and therefore cover a big part of the latest OWASP Top 10:2021 list [3] for common Security Risks in web applications.

II. Setting up the Challenge

A. Website

The frontend consists of a Flask application and raw socket communication with the backend to improve performance, which was previously hindered by using a Flask API for the backend. The backend interacts with a `.db` file through Python.

The frontend employs the `| safe` keyword on the homepage, allowing for XSS vulnerabilities. It also uses the PHP command-line interface to process PHP code, enabling the attacker to upload and execute a reverse shell. This setup simulates a typical Apache web server that displays the contents of the database on the website and allows administrators to write new pages that may include PHP.

In the backend, SQL inputs are directly inserted into a premade string, mimicking a raw `pg_query()` request in PHP. Unlike `pg_query_params()`, this approach is vulnerable to SQL injection when the input is not properly sanitized. Python was used to replace the standard procedures with raw PHP and JavaScript to minimize development costs while maintaining high flexibility, ensuring the desired vulnerabilities could be incorporated.

B. Backend Server

The backend server was set up with `OpenSSH 7.7p2` and `Sudo 1.8.21p2` to showcase two additional CVEs. To enhance the challenge, information such as the login name or root access was not freely provided, making the challenge more engaging. An unprivileged user was configured with a common Unix username and password, making it easily brute-forceable.

C. Monitoring

To demonstrate why `OpenSSH` announced a switch to a minimum RSA modulus of 1024 bits in the release notes for version 7.2, a custom SSH-type shell was programmed in Python. This shell uses full RSA encryption with a very weak modulus of only 48 bits, significantly smaller than the minimum ever used by SSH, making it easier to brute-force.

To make the XSS vulnerability on the frontend exploitable, the Python script for monitoring uses `Playwright`, which has an option to render JavaScript. While a more realistic approach for a monitoring server might involve using `requests` to fetch the HTML and check if the page responds, executing the JavaScript from the XSS is necessary to give the attacker an opportunity to exploit the vulnerability. It's worth noting that this XSS vulnerability could still be dangerous in a real scenario, as an attacker could wait for an admin to visit the front page while logged in, thereby gaining access to the admin panel, or wait for any logged-in user to trigger the XSS and log in as that user.

Finally, a modified version of the `get_flag` script was used to demonstrate a buffer overflow that allows the attacker to obtain a root shell. The vulnerable `fgets` function is called after the `seteuid(0)` call to read secrets from the root directory. Initially, the plan was to hide the buffer address in normal output and enable ASLR, requiring the attacker to find the changing address by debugging the program. However, this approach would only work if the monitoring user could

execute the debugger with root privileges, as the `get_flag` binary is owned by root and has the `setuid` bit set.

Since granting the monitoring user root privileges for the debugger could potentially allow them to bypass the binary altogether and compile and execute their own binary with root privileges (making it easier to obtain a root shell), ASLR was disabled, and the buffer address was provided directly. An interesting detail is that after the buffer overflow is exploited, the `uid` is indeed set to `0`, while the `gid` and `groups` remain inherited from the monitoring user. Therefore, running `su` is necessary to obtain a "full" root shell.

It should also be noted that although the binary is compiled with `-z execstack` to allow for the injecion of shellcode into the buffer. Even without this option the `fgets` function is dangerous since the return address can be overwritten by the one of a function in the original binary if it is known, possibly executing this function at a time that was not originally intended.

D. Monitoring Host

Setting up the monitoring host was by far the most challenging aspect of creating the CTF. The rules for the CTF dictate that all components meant to be exploited must run within a Docker container, which required showcasing a container breakout vulnerability by running a Docker container inside another Docker container.

Initially, I attempted to set this up manually by installing Docker in a standard Ubuntu container. However, since Docker is not a virtualization framework but rather encapsulates the filesystem, memory, etc., this task proved more difficult than expected. Additionally, I wanted to install an older, vulnerable version of Docker inside the container while using an up-to-date version to run the overall challenge. This approach avoids the risk of requiring participants to install an unsafe version of Docker on their systems, which could easily be forgotten and reused in other projects, posing a serious security risk.

To address these challenges, I decided to use a Docker-in-Docker image that contained the desired older Docker version. Once this issue was resolved, I could finally begin building a scenario where the CVE could be exploited. Exploiting this vulnerability requires running `docker exec` at least twice. To facilitate this, I set up a script inside the monitoring container that checks whether the monitoring service is running and restarts it if it is not.

This vulnerability is especially dangerous if no such automated `docker exec` call is made. To ensure the exploit can be triggered by default, the `/bin/bash` binary is replaced, so the exploit activates when someone attempts to get a shell inside the container.

III. The CVEs

A. CVE-2018-15473 [4] - OpenSSH $\leq$ 7.7 - User Enumeration

This CVE was first discovered by the Qualys Security Advisory team in August 2018 [5] while they were reviewing the then - latest OpenSSH commits [6]. They identified a discrepancy in the authentication process: when a malformed packet was parsed, the authentication would fail with `SSH2_MSG_USERAUTH_FAILURE` if the user was invalid, but it would call `fatal()` — resulting in the connection being closed — if the user was valid but the parsing failed due to the malformation.

This difference could be exploited to scan for existing users by sending a malformed authentication request for each user being tested. The vulnerability was addressed by delaying the failure exit when an invalid user attempted to authenticate until after the packet was fully parsed. This change ensured that all malformed packets would fail at the same point, providing uniform feedback regardless of whether the user existed on the host machine.

B. CVE-2019-14287 [7] - `sudo` < 1.8.28 - Privilege Escalation

This CVE, discovered by Joe Vennix in October 2019 [8], allows a user to execute commands with root privileges if the user has `sudo` privileges to execute commands as anyone except `root` (for example, if the `/etc/sudoers` file contains an entry like `<user> <hostname>=(!root) <command>`).

Although this configuration was intended to prevent the user from executing commands as `root`, calling `sudo -u#-1 <command>` would in fact execute the command as `root`. According to a write-up from the official sudo website [9], the issue resulted from `sudo` first setting the `euid/esuid` to 0 to have the rights to switch to the `euid/esuid` specified for the command. Later, `sudo` used the `setresuid(2)` [10] and `setreuid(2)` [11] system calls, both of which accept `0xffffffff` as a special value that results in no change to the `euid/esuid`. This led to the `euid/esuid` remaining set to `0`, running the command with `root` privileges, regardless of the `-u#-1` or `-u#4294967295` argument.

The vulnerability was fixed by preventing `#-1` from being used as an argument for the `-u` option. [12]

C. CVE-2019-5736 [13]
`runc` $\leq$ 1.0-rc6 (`docker` < 18.09.2) - Container Breakout

This CVE was first discovered by Adam Iwaniuk and Borys Poplawski in February 2019 [14] and is by far the most complex vulnerability in this CTF challenge. To understand how it works and how it is patched, one must understand the proc filesystem (`procfs`). `procfs` is a virtual filesystem that contains information about running processes and is mounted under `/proc`. Each process has information that may be needed for execution or system interaction under `/proc/<pid>/`, but it can also access its own information via the `/proc/self/` process-specific symlink.

The crucial components for this exploit are `/proc/self/exe` and `/proc/[pid]/exe`, which point to the binary executed by the referenced process, and `/proc/self/fd/`, which holds symlinks to the file descriptors used by a process. To exploit this vulnerability (using `docker exec`; `docker run` is also vulnerable

via a similar process but is not covered here), the attacker needs to know a binary called via `docker exec` (typically `/bin/bash`) and have root privileges within the container.

The exploit begins by overwriting the binary so that `/proc/self/exe` is executed in the container (by `execve` or using the shebang `#!/proc/self/exe`). When the binary is called via `docker exec`, `/proc/self/exe` becomes a symbolic link to the `runc` binary used to execute the `docker exec` call. The attacker then obtains a read file descriptor on the `runc` binary and enters a waiting loop. This loop tries to acquire a write file descriptor on the `runc` binary via `/proc/<attacking process>/fd/<runc binary read fd>`. This succeeds precisely when the `runc` process terminates, as the binary is no longer in use, but the symlink remains valid because the `/proc/self/` directory of the overwritten binary has not yet been deleted.

Once the attacker acquires the write file descriptor, the `runc` binary on the host can be overwritten with an arbitrary binary, such as a reverse shell, granting access to the host when the binary is executed again. It is important to note that this exploit is also applicable to other containerization frameworks that use `runc`, as listed in the original report linked above. The vulnerability was finally patched by cloning the `runc` binary before executing `docker exec` or the equivalent command in other frameworks, ensuring that only the copied binary could be overwritten while the original binary remained untouched for future commands. [15]

IV. Walkthrough

A. Frontend

1) **Inspect the homepage:** Begin by examining the homepage for potential vulnerabilities or useful information.

2) **Find the monitoring site:** The homepage may reveal the presence of a monitoring site, indicating that a service is accessing the site with privileges to edit or update it.

3) **Locate the comment section:** Identify a comment section that appears vulnerable to Cross-Site Scripting (XSS) attacks.

4) **Extract the session cookie:** Inspect the session cookie associated with your user session.

5) **Craft an XSS payload:** Create a comment that includes a script designed to steal the session cookie.

6) **Wait for execution:** Allow the monitoring service to execute the XSS payload. Once executed, the session cookie of the monitoring service should appear as a comment on the page.

7) **Copy the session cookie:** Copy the session cookie and use it to access the admin panel.

8) **Inspect the admin panel:** Explore the admin panel and find the option to add a new page.

9) **Upload a PHP reverse shell:** Use the add page option to upload a PHP reverse shell, which will be used to compromise the first server.

10) **Execute the reverse shell:** Trigger the reverse shell to gain access to the server.

11) **Extract the first flag:** Once you have access, execute `/root/get_flag` to obtain the first flag.

B. Backend

1) Unprivileged:

1) **Inspect the `/etc/hosts` file:** Read the `/etc/hosts` file to discover potential targets.

2) **Port scan targets:** Perform a port scan on `backend.ctf-challenge.edu` and `monitoring.ctf-challenge.edu` to find open ports.

3) **Exploit SSH vulnerability:** Use `nmap` to scan `backend.ctf-challenge.edu` for open port 22. Identify `OpenSSH 7.7p2`, which is vulnerable to username enumeration (CVE-2018-15473).

4) **Find valid usernames:** Utilize the provided `unix_users.txt` file to enumerate valid usernames.

5) **Analyze `main.py`:** Review `main.py` and note that input sanitization, especially SQL sanitization, is handled in the frontend.

6) **Craft SQL injection exploit:** Use the `db_client.DatabaseClient` to write a SQL injection exploit with a UNION query, targeting the `get-username-from-session` endpoint.

7) **Extract admin hash:** Extract the hash for the admin account using SQL injection.

8) **Crack the admin hash:** Use online tools or the provided `unix-password.txt` file to crack the hash.

9) **SSH into the server:** Connect to the server via SSH using the cracked password for the `dbadmin` account.

10) **Extract the second flag:** Execute `/home/dbadmin/get_flag` to obtain the second flag.

2) Privileged:

1) **Check sudo version:** Run `sudo --version` to find that it is version `1.8.21p2`, which is vulnerable to a buffer overflow (CVE-2019-14287).

2) **List sudo privileges:** Execute `sudo -l` and discover that the `dbadmin` user can run all commands as any user except root.

3) **Exploit sudo vulnerability:** Run `sudo -u#-1 su` to obtain a root shell.

4) **Extract the third flag:** Execute `/root/get_flag` to obtain the third flag.

C. Monitoring

1) Unprivileged:

1) **Inspect logs:** Read the logs in the root directory to find that `coolsh_server.py` also runs on `monitoring.ctf-challenge.edu`, providing the admin with easier, permanent access.

2) **Identify RSA weakness:** Notice that the RSA encryption modulus is unusually small.

3) **Factorize the modulus:** Write an exploit to factorize the modulus and calculate the private exponent of the admin.

4) **Adapt `coolsh_client.py`:** Modify `coolsh_client.py` to use public key authentication with the cracked admin key pair.

5) **Extract the fourth flag:** Execute `/home/monitoring/get_flags` to obtain the fourth flag.

2) Privileged:

1) **Analyze `get_flags.c`:** Inspect the `/home/monitoring/get_flags.c` file and find that it uses the `gets()` function, which is vulnerable to a buffer overflow and is executed with an effective user ID (`euid`) of `0` to read the secrets from the `/root` directory.

2) **Write buffer overflow exploit:** Use the information from `get_flags` to craft a payload that launches a reverse shell with root privileges.

3) **Obtain a full root shell:** Run `su` to gain a full root shell (setting the `gid` and `groups` to `0`).

4) **Retrieve the fifth flag:** Execute `/home/monitoring/get_flags` again with elevated privileges to obtain the fifth flag.

D. Monitoring Host

1) **Inspect Docker logs:** Review the logs to discover that the Docker version is `18.09.1`, which is vulnerable to a container breakout (CVE-2019-5736).

2) **Examine logs for `check_monitoring.sh`:** Identify that `check_monitoring.sh` is regularly executed via `docker exec`.

3) **Analyze network configuration:** Observe that the network stack is shared between the host and the container.

4) **Modify `check_monitoring.sh`:** Edit `check_monitoring.sh` to invoke the `runc` binary.

5) **Write a container breakout exploit:** Develop an exploit to obtain a read file descriptor for the `runc` binary, wait for the `runc` process to end, then obtain a write file descriptor and overwrite it with a reverse shell pointing to localhost.

6) **Retrieve the sixth flag:** Execute `/root/get_flags` again with elevated privileges to obtain the sixth flag.

A more detailed walkthrough is available in the `walkthrough` branch of the repository [16].

V. Conclusion

To understand how programmers can avoid the vulnerabilities discussed in this paper (as well as many others), we must take a step back to examine how these vulnerabilities were introduced into the web application in the first place. While some specific steps to create the vulnerabilities may seem unrealistic in real-world scenarios, it's important to remember that they were likely chosen to illustrate the vulnerabilities themselves. For example, using the `| safe` keyword to display user input (such as comments) on our website might seem like an obvious security risk, but the same effect could be achieved in a PHP application where comments are fetched from a database and directly inserted into a prewritten HTML template. If HTML tags are allowed in this context to enable users to format their input, improper sanitization could lead to the exact same Stored XSS vulnerability, though through less obvious means.

The next vulnerability discussed was SQL Injection, which can easily occur, as mentioned earlier, by using `pg_query` instead of `pg_query_params`. A backend developer who is not focused on cybersecurity might find the former function easier to use. Additionally, if data used in the SQL query is sanitized on the frontend instead of the database server before querying, this could result from overtrusting a controlled server without considering that it might be compromised and become untrustworthy in the future.

The subsequent vulnerabilities were introduced by outdated and vulnerable components, and in the case of the `sudo` vulnerability, by setting permissions through a blacklist instead of whitelisting the permissions the user actually needs explicitly.

The factorization of a weak RSA modulus can be attributed to underestimating the complexity of software, overlooking the security measures others have implemented, and not understanding the rationale behind those measures, leading to their omission.

The buffer overflow vulnerability, like the SQL Injection, resulted from using an insecure function (`fgets`) for reading user input.

The final container breakout was again a consequence of using an outdated and vulnerable component.

In conclusion, the challenge highlights some key practices to help avoid these vulnerabilities:

1) Use appropriate libraries and packages whenever possible. Since there is always a possibility that what we are programming could be relevant to security, we should use libraries that are typically employed in such contexts, as they are likely to address the associated security concerns and help avoid vulnerabilities.

2) Read the documentation. While researching the reasons why certain functions are vulnerable (`pg_query`, `setreuid(2)`, `setresuid(2)`) and others are not, I observed that in all these cases, the potential vulnerabilities were mentioned in the documentation, although sometimes not as clearly as they should be.

3) Always treat user input as potentially malicious. At every point where a user can input data, they could attempt to inject payloads, execute exploits, and so on. This must be considered for every component of the software individually. For example, while distrusting user-submitted comments might be more obvious than distrusting queries sent by our own frontend server, it is essential to sanitize input and not trust it blindly at every stage.

4) Keep software updated to the latest stable build, especially if the changelog mentions vulnerabilities that have been fixed. Although keeping software up to date may require additional effort, it is crucial to ensure that our applications are as secure as possible.

While following these guidelines will result in more secure applications, it is important to remember that no application of even moderate complexity can be definitively proven to be secure. In both 2022 and 2023, over 25,000 CVEs were reported each year, continuing a clear upward trend since CVEs began being reported publicly in 1999. We must conclude that most CVEs exist in software for a significant amount of time before they are discovered and patched, and they may have been exploited by malicious actors who have no intention of informing the public for the same duration.

REFERENCES

[1] Jakob Weber, "Email," mailto:jakob.weber@stud.uni-heidelberg.de.

[2] ——, "CTF-Challenge," https://github.com/KunoVonHagen/CTF-Challenge.

[3] OWASP, "OWASP Top 10: 2021 List for Common Security Risks in Web Applications," https://owasp.org/Top10/.

[4] CVE Mitre, "CVE-2018-15473 - OpenSSH $\leq$ 7.7 - User Enumeration," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-15473.

[5] Qualys Security Advisory Team, "First Discovered by the Qualys Security Advisory Team in August 2018," https://www.openwall.com/lists/oss-security/2018/08/15/5.

[6] OpenBSD, "OpenSSH Commits," https://github.com/openbsd/src/commit/779974d35b4859c07bc3cb8a12c74b43b0a7d1e0?diff=split&w=0#diff-2d35b193519c52b0b39fea601d22b4a6cfa65bcdf2282eeaf1ab88ecccee2e99.

[7] CVE Mitre, "CVE-2019-14287 - sudo $<$ 1.8.28 - Privilege Escalation," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-14287.

[8] Joe Vennix, "Discovered by Joe Vennix in October 2019," https://seclists.org/bugtraq/2019/Oct/21.

[9] Sudo Project, "Write-up on CVE-2019-14287," https://www.sudo.ws/security/advisories/minus_1_uid/.

[10] Linux Man Pages, "setresuid(2) - Set User ID," https://man7.org/linux/man-pages/man2/setresuid.2.html.

[11] ——, "setreuid(2) - Set Real and Effective User IDs," https://man7.org/linux/man-pages/man2/setreuid.2.html.

[12] millert, "Commit to Fix CVE-2019-14287," https://github.com/sudo-project/sudo/commit/f752ae5cee163253730ff7cdf293e34a91aa5520.

[13] CVE Mitre, "CVE-2019-5736 - runc $\leq$ 1.0-rc6 (docker ¡ 18.09.2) - Container Breakout," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-5736.

[14] SecurityFocus, "runc $\leq$ 1.0-rc6 Container Breakout Discovery," https://web.archive.org/web/20210121140808/https://www.securityfocus.com/bid/106976/info.

[15] cyphar, "Commit to Fix CVE-2019-5736," https://github.com/opencontainers/runc/commit/0a8e4117e7f715d5fbeef398405813ce8e88558b#diff-61c35e83761595b20bb8e8ecfc1d2a2fa55662c227a5e14a29d26380cdcf46ce.

[16] Jakob Weber, "Walkthrough," https://github.com/KunoVonHagen/CTF-Challenge/tree/walkthrough.