reverse eng

Created by	Ⓜ Manikandan N
Created time	@December 25, 2025 1:18 PM
Last edited by	Ⓜ Manikandan N
Last updated time	@December 25, 2025 1:18 PM
Category	blue team room what i learned and completed

🔍 Reverse Engineering Room

Category: Blue Team Room – What I Learned and Completed

✅ Reverse Engineering Room – What I Did 🧩

✔️ File identification 📄

I first identified that the given file was a **C source file** related to the **PHP zlib extension**.

This helped me understand that the code was part of a **core PHP component**, not a normal application file.

✔️ Focused code scanning (not full reading) 👀

Instead of reading the entire code line by line, I searched for:

- User-controlled inputs (HTTP headers)
- Dangerous functions related to execution

This helped me avoid confusion from legitimate compression logic.

✔️ User input tracing 🔗

I observed that the code was accessing:

- `$_SERVER` variables

- Specifically the **User-Agent HTTP header** (internally referenced as `HTTP_USER_AGENTT`)

This indicated that **external user input** was being processed.

✔️ **Detection of dangerous function** 🚨

I identified the use of:

- `zend_eval_string()`

This function executes PHP code dynamically, which is extremely dangerous when used with user input.

✔️ **Vulnerability identification** 🛑

I concluded that:

- User-controlled HTTP header data was executed as PHP code
- This results in **Code Execution vulnerability**

✔️ **Commit-level impact understanding** 🧾

I learned that the vulnerability was introduced by adding **only 11 lines of code**, showing how a very small change can introduce a critical backdoor.

⚠️ What I Struggled With (REAL STRUGGLES) 🤯

1️⃣ **Code looked overwhelming at first**

The C code was very large and confusing. I initially tried to understand everything, which made the task difficult.

2️⃣ **Understanding what to ignore**

I struggled to differentiate between:

- Legitimate compression logic
- Malicious backdoor code

Later, I learned that most of the file was noise from a security perspective.

3️⃣ **Why a trusted library would contain a backdoor**

It was confusing to accept that a core PHP extension could contain malicious code. This taught me about **supply-chain risks**.

4️⃣ **Counting lines of code correctly**

I initially miscounted the number of lines added. I learned that:

- Each physical line added matters
- Wrapped function arguments are also counted as separate lines

---

🧠 Small → Big Things I Need to Remember 📈

🔷 Small (Basics)

- Not all code needs to be understood
- `eval` like functions are always red flags
- HTTP headers are user-controlled

🔷 Medium (Code Review)

- Trace **input → processing → execution**
- Look for context mismatch (code that doesn't belong)
- Small commits can introduce huge risks

🔷 Big (SOC Thinking) 🧠

- Core libraries are high-value attack targets
- Supply-chain attacks can be subtle
- Manual code review is critical
- Trust ≠ secure

🧪 Key Indicators Identified

- User-controlled **User-Agent header**
- Execution via `zend_eval_string()`
- Hidden logic unrelated to compression
- Minimal code change with maximum impact

🧠 Final SOC Conclusion (MY WORDS) 🛡️

I identified a hidden code execution backdoor in the PHP zlib extension where user-controlled HTTP header data was executed using `zend_eval_string()`. The vulnerability was introduced with only 11 lines of code and could be exploited

using a crafted User-Agent header. This lab helped me understand how small changes in trusted components can lead to critical security issues and improved my ability to analyze code from a blue team perspective.