Vulnerability Title:
Buffer overflow observed in *parse_statement() and tokenizer code in js.c*

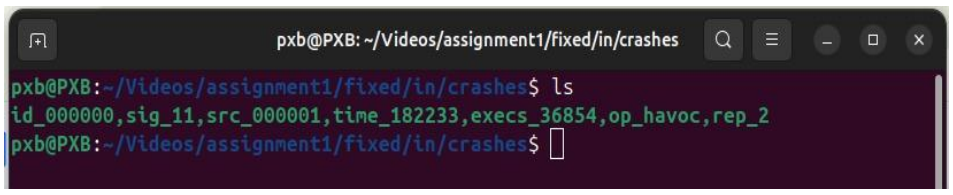
Security Level:
Critical

Vulnerable Component:
Function: *parse_statement() and next_token()* tokenizer.

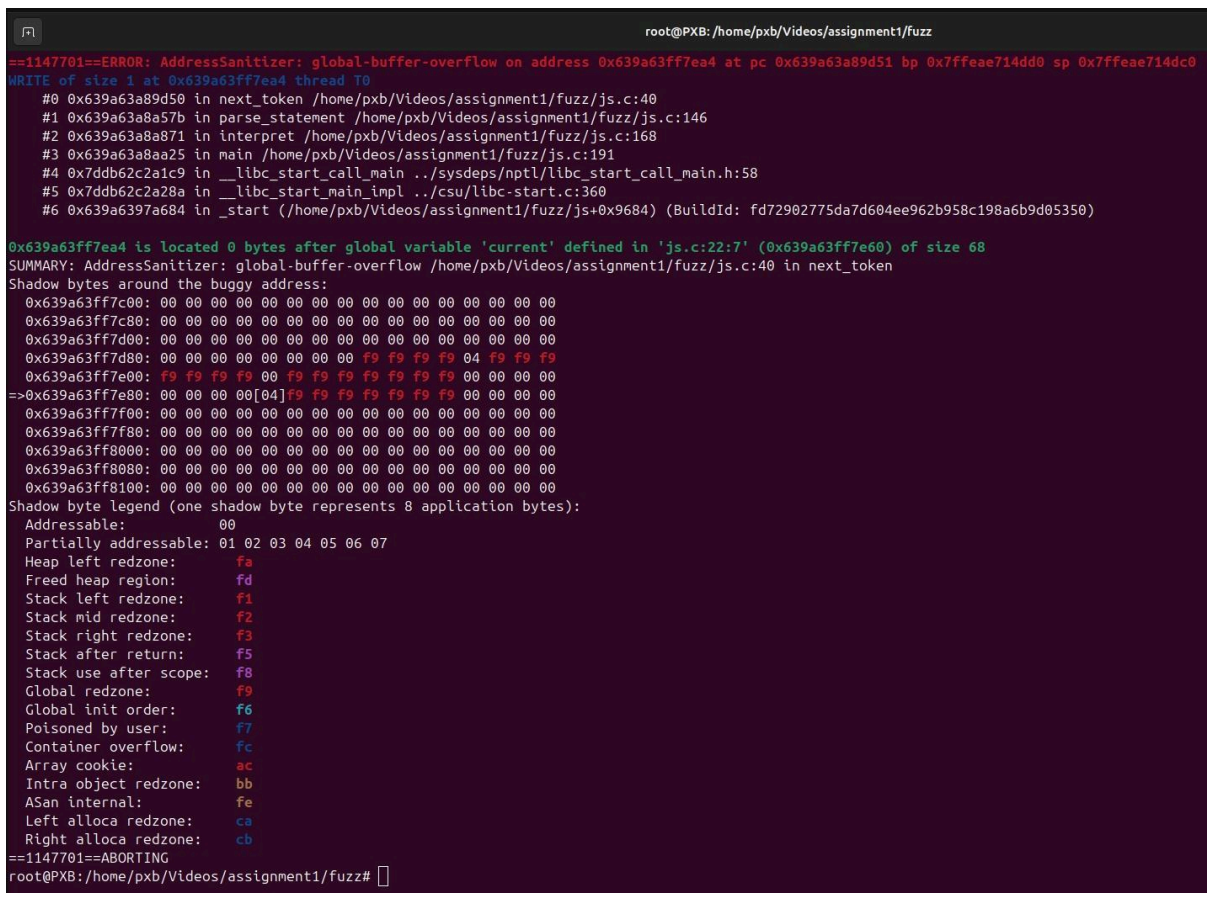
***File** : js.c*

Vulnerability Details :
Fuzzer discovered a crash caused by a buffer overflow in the file js.c. The program improperly copies unbounded strings into the fixed-size buffer (current.text and name[64]) using unsafe copying logic that lacks bound check, such as the use of strcpy(). This allows an attacker-supplied input to override adjacent memory and may potentially lead to control-flow corruption, remote code execution etc.

Evidence:
In the given screenshot below, the crash file was discovered



In the given screenshot below, ASAN(Address Sanitizer) output is observed when running the crash against the original vulnerable binary.



The ASAN report in the screenshot confirms a global buffer overflow triggered by the fuzzed input. The overflow occurred inside the next_token() function where characters are copied into current.text without size validation. ASAN provides the specific memory address, stack trace, and shadow bytes that identify the out-of-bounds write. This evidence not only verifies the vulnerability but also aligns with the behavior observed in the fuzzing crash file.

Root Cause Analysis:

- The tokenizer (next_token()) copies numeric/identifiers tokens into the global current.text[64] buffer without checking that the token would fit in the 64 bytes buffer. If a very long sequence of digits or letters is produced from mutated input, it can exceed the size of the current text, leading to memory corruption in adjacent global memory. This issue has been observed as the overwriting of variable and return addresses.
- In the parse_statement() function, after parsing let <id> = <expr>, a new variable is added to the vars[] array with vars[var_count++] = (Var){ .value = val }. The name is copied using strcpy(vars[var_count - 1].name, name). However, `strcpy()` does not check the size of the destination buffer, which can lead to buffer overflow vulnerability.

Rationale behind the patch:

- Token copying (numbers & identifiers) — add bounds checking while copying into current.text so we never write more than sizeof(current.text)-1 bytes and always terminated by a null.

```c
if (*src == '\0') { current.type = TOKEN_EOF; return; }

if (isdigit(*src)) {
    int i = 0;
    while (isdigit(*src) || *src == '.') current.text[i++] = *src++;
    current.text[i] = '\0';
    current.type = TOKEN_NUMBER;
    return;
}
f (isdigit(*src)) {
int i = 0;
while (isdigit(*src) || *src == '.') {
    if (i < (int)sizeof(current.text) - 1) current.text[i++] = *src;
    src++;
}
current.text[i] = '\0';
current.type = TOKEN_NUMBER;
return;
```

- Insertion into vars[] and copying the name — check capacity of vars[] before appending; use bounded copy (strncpy) when storing name; always terminated by a null.

```c
next_token();
double val = parse_expr();
if (current.type == TOKEN_SEMI) next_token();
vars[var_count++] = (Var){ .value = val };
strcpy(vars[var_count - 1].name, name);

/* Prevent overflow of vars[]: check capacity before inserting. */
if (var_count < (int)(sizeof(vars) / sizeof(vars[0]))) {
    vars[var_count].value = val;
    /* ensure name fits into vars[].name (both are 64 bytes) */
    strncpy(vars[var_count].name, name, sizeof(vars[var_count].name) - 1);
    vars[var_count].name[sizeof(vars[var_count].name) - 1] = '\0';
    var_count++;
} else {
    fprintf(stderr, "Too many variables (overflow attempt)\n");
    exit(1);
}
```

Patch Implementation:

1. **Token bounds checks**: By checking i < (int)sizeof(current.text) - 1 before copying each byte, the code prevents writes beyond the buffer's last usable byte. The final current.text[i] = '\0'; ensures the stored token is always a valid C string.
2. **Safe insertion to vars[]**: The var_count capacity check prevents an out-of-bounds write into the vars array when it is full. Replacing strcpy() with strncpy() (plus explicit null-termination) prevents name from overflowing the vars[].name buffer. Together, these eliminate two different overflow vectors: overflowing current.text into adjacent global and overflowing a variable's name into adjacent memory.

```
pxb@PXB:~/Videos/assignment1/fixed$ sudo ./run.sh
[sudo] password for pxb:
[*] Docker detected — building and running container...
[+] Building 1.3s (11/11) FINISHED                                                          docker:default
 => [internal] load build definition from Dockerfile                                                  0.0s
 => => transferring dockerfile: 553B                                                                  0.0s
 => [internal] load metadata for docker.io/library/ubuntu:22.04                                       1.0s
 => [auth] library/ubuntu:pull token for registry-1.docker.io                                         0.0s
 => [internal] load .dockerignore                                                                     0.0s
 => => transferring context: 2B                                                                       0.0s
 => [1/5] FROM docker.io/library/ubuntu:22.04@sha256:104ae83764a5119017b8e8d6218fa0832b09df65aae7d5a6de29a85d813da2fb  0.0s
 => => resolve docker.io/library/ubuntu:22.04@sha256:104ae83764a5119017b8e8d6218fa0832b09df65aae7d5a6de29a85d813da2fb  0.0s
 => [internal] load build context                                                                     0.0s
 => => transferring context: 1.44MB                                                                   0.0s
 => CACHED [2/5] WORKDIR /app                                                                         0.0s
 => CACHED [3/5] RUN apt-get update && apt-get install -y --no-install-recommends     build-essential clang patch  nano vim git ca-certificates curl     && rm -rf /var/lib/apt/l  0.0s
 => [4/5] COPY . /app                                                                                 0.0s
 => [5/5] RUN chmod +x /app/run.sh                                                                    0.2s
 => exporting to image                                                                                0.0s
 => => exporting layers                                                                               0.0s
 => => writing image sha256:a7542d1d593543dc99ace1cf93dfb33fb19e2508623466cbfb7c1e41f2650186          0.0s
 => => naming to docker.io/library/js_test_image                                                      0.0s
[*] Applying patch...
patching file js.c
Patch applied successfully ✅
[*] Using crash file: in/crashes/id_000000,sig_11,src_000001,time_182233,execs_36854,op_havoc,rep_2

[*] Compiling ASan (diagnostic) binary...
[+] ASan binary built: js_asan

[*] Running ASan binary on crash...
=== ASan output (first 20 lines) ===
Result: 1e+63
==================================

=== RESULT: Crash no longer occurs — issue fixed ✅ ===

[*] Building fast instrumented binary for fuzzing...
[+] Fast binary built: js_fast
pxb@PXB:~/Videos/assignment1/fixed$
```

3. **Result:** The previous ASAN detected global buffer overflow arises from writing past current into *vars* — the token bounds checks stop current overflow; the vars capacity/name copy checks stop overflow on *vars* insertion.

References :

https://nvd.nist.gov/vuln-metrics/cvss/v3-calculator
https://www.cve.org/CVERecord/SearchResults?query=buffer+overflow
https://docs.docker.com/build/concepts/overview/
https://clang.llvm.org/docs/AddressSanitizer.html
https://aflplus.plus//