我们在爬取网站的时候，经常会遇到各种各样类似加密的情形，比如：

- 某个网站的 URL 带有一些看不懂的长串加密参数，想要抓取就必须要懂得这些参数是怎么构造的，否则我们连完整的 URL 都构造不出来，更不用说爬取了。
- 分析某个网站的 Ajax 接口的时候，可以看到接口的一些参数也是加密的，或者 Request Headers 里面也可能带有一些加密参数，如果不知道这些参数的具体构造逻辑就无法直接用程序来模拟这些 Ajax 请求。
- 翻看网站的 JavaScript 源代码，可以发现很多压缩了或者看不太懂的字符，比如 JavaScript 文件名被编码，JavaScript 的文件内容被压缩成几行，JavaScript 变量也被修改成单个字符或者一些十六进制的字符，导致我们不好轻易根据 JavaScript 找出某些接口的加密逻辑。

这些情况，基本上都是网站为了保护其本身的一些数据不被轻易抓取而采取的一些措施，我们可以把它归为两大类：

- 接口加密技术；
- JavaScript 压缩、混淆和加密技术。

本课时我们就来了解下这两类技术的实现原理。

数据保护

当今大数据时代，数据已经变得越来越重要，网页和 App 现在是主流的数据载体，如果其数据的接口没有设置任何保护措施，在爬虫工程师解决了一些基本的反爬如封 IP、验证码的问题之后，那么数据还是可以被轻松抓取到。

那么，有没有可能在接口或 JavaScript 层面也加上一层防护呢？答案是可以的。

接口加密技术

网站运营商首先想到防护措施可能是对某些数据接口进行加密，比如说对某些 URL 的一些参数加上校验码或者把一些 ID 信息进行编码，使其变得难以阅读或构造；或者对某些接口请求加上一些 token、sign 等签名，这样这些请求发送到服务器时，服务器会通过客户端发来的一些请求信息以及双方约定好的秘钥等来对当前的请求进行校验，如果校验通过，才返回对应数据结果。

比如说客户端和服务端约定一种接口校验逻辑，客户端在每次请求服务端接口的时候都会附带一个 sign 参数，这个 sign 参数可能是由当前时间信息、请求的 URL、请求的数据、设备的 ID、双方约定好的秘钥经过一些加密算法构造而成的，客户端会实现这个加密算法构造 sign，然后每次请求服务器的时候附带上这个参数。服务端会根据约定好的算法和请求的数据对 sign 进行校验，如果校验通过，才返回对应的数据，否则拒绝响应。

JavaScript 压缩、混淆和加密技术

接口加密技术看起来的确是一个不错的解决方案，但单纯依靠它并不能很好地解决问题。为什么呢？

对于网页来说，其逻辑是依赖于 JavaScript 来实现的，JavaScript 有如下特点：

- JavaScript 代码运行于客户端，也就是它必须要在用户浏览器端加载并运行。
- JavaScript 代码是公开透明的，也就是说浏览器可以直接获取到正在运行的 JavaScript 的源码。

由于这两个原因，导致 JavaScript 代码是不安全的，任何人都可以读、分析、复制、盗用，甚至篡改。

所以说，对于上述情形，客户端 JavaScript 对于某些加密的实现是很容易被找到或模拟的，了解了加密逻辑后，模拟参数的构造和请求也就是轻而易举了，所以如果 JavaScript 没有做任何层面的保护的话，接口加密技术基本上对数据起不到什么防护作用。

如果你不想让自己的数据被轻易获取，不想他人了解 JavaScript 逻辑的实现，或者想降低被不怀好意的人甚至是黑客攻击。那么你就需要用到 JavaScript 压缩、混淆和加密技术了。

这里压缩、混淆、加密技术简述如下。

- 代码压缩：即去除 JavaScript 代码中的不必要的空格、换行等内容，使源码都压缩为几行内容，降低代码可读性，当然同时也能提高网站的加载速度。
- 代码混淆：使用变量替换、字符串阵列化、控制流平坦化、多态变异、僵尸函数、调试保护等手段，使代码变得难以阅读和分析，达到最终保护的目的。但这不影响代码原有功能。是理想、实用的 JavaScript 保护方案。
- 代码加密：可以通过某种手段将 JavaScript 代码进行加密，转成人无法阅读或者解析的代码，如将代码完全抽象化加密，如 eval 加密。另外还有更强大的加密技术，可以直接将 JavaScript 代码用 C/C++ 实现，JavaScript 调用其编译后形成的文件来执行相应的功能，如 Emscripten 还有 WebAssembly。

下面我们对上面的技术分别予以介绍。

接口加密技术

数据一般都是通过服务器提供的接口来获取的，网站或 App 可以请求某个数据接口获取到对应的数据，然后再把获取的数据展示出来。

但有些数据是比较宝贵或私密的，这些数据肯定是需要一定层面上的保护。所以不同接口的实现也就对应着不同的安全防护级别，我们这里来总结下。

完全开放的接口

有些接口是没有设置任何防护的，谁都可以调用和访问，而且没有任何时空限制和频率限制。任何人只要知道了接口的调用方式就能无限制地调用。

这种接口的安全性是非常非常低的，如果接口的调用方式一旦泄露或被抓包获取到，任何人都可以无限制地对数据进行操作或访问。此时如果接口里面包含一些重要的数据或隐私数据，就能轻易被篡改或窃取了。

接口参数加密

为了提升接口的安全性，客户端会和服务端约定一种接口校验方式，一般来说会使用到各种加密和编码算法，如 Base64、Hex 编码，MD5、AES、DES、RSA 等加密。

比如客户端和服务器双方约定一个 sign 用作接口的签名校验，其生成逻辑是客户端将 URL Path 进行 MD5 加密然后拼接上 URL 的某个参数再进行 Base64 编码，最后得到一个字符串 sign，这个 sign 会通过 Request URL 的某个参数或 Request Headers 发送给服务器。服务器接收到请求后，对 URL Path 同样进行 MD5 加密，然后拼接上 URL 的某个参数，也进行 Base64 编码得到了一个 sign，然后比对生成的 sign 和客户端发来的 sign 是否是一致的，如果是一致的，那就返回正确的结果，否则拒绝响应。这就是一个比较简单的接口参数加密的实现。如果有人想要调用这个接口的话，必须要定义好 sign 的生成逻辑，否则是无法正常调用接口的。

以上就是一个基本的接口参数加密逻辑的实现。

当然上面的这个实现思路比较简单，这里还可以增加一些时间戳信息增加时效性判断，或增加一些非对称加密进一步提高加密的复杂程度。但不管怎样，只要客户端和服务器约定好了加密和校验逻辑，任何形式加密算法都是可以的。

这里要实现接口参数加密就需要用到一些加密算法，客户端和服务器肯定也都有对应的 SDK 实现这些加密算法，如 JavaScript 的 crypto-js，Python 的 hashlib、Crypto 等等。

但还是如上文所说，如果是网页的话，客户端实现加密逻辑如果是用 JavaScript 来实现，其源代码对用户是完全可见的，如果没有对 JavaScript 做任何保护的话，是很容易弄清楚客户端加密的流程的。

因此，我们需要对 JavaScript 利用压缩、混淆、加密的方式来对客户端的逻辑进行一定程度上的保护。

JavaScript 压缩、混淆、加密

下面我们再来介绍下 JavaScript 的压缩、混淆和加密技术。

JavaScript 压缩

这个非常简单，JavaScript 压缩即去除 JavaScript 代码中的不必要的空格、换行等内容或者把一些可能公用的代码进行处理实现共享，最后输出的结果都被压缩为几行内容，代码可读性变得很差，同时也能提高网站加载速度。

如果仅仅是去除空格换行这样的压缩方式，其实几乎是没有任何防护作用的，因为这种压缩方式仅仅是降低了代码的直接可读性。如果我们有一些格式化工具可以轻松将 JavaScript 代码变得易读，比如利用 IDE、在线工具或 Chrome 浏览器都能还原格式化的代码。

目前主流的前端开发技术大多都会利用 Webpack 进行打包，Webpack 会对源代码进行编译和压缩，输出几个打包好的 JavaScript 文件，其中我们可以看到输出的 JavaScript 文件名带有一些不规则字符串，同时文件内容可能只有几行内容，变量名都是一些简单字母表示。这其中就包含 JavaScript 压缩技术，比如一些公共的库输出成 bundle 文件，一些调用逻辑压缩和转义成几行代码，这些都属于 JavaScript 压缩。另外其中也包含了一些很基础的 JavaScript 混淆技术，比如把变量名、方法名替换成一些简单字符，降低代码可读性。

但整体来说，JavaScript 压缩技术只能在很小的程度上起到防护作用，要想真正提高防护效果还得依靠 JavaScript 混淆和加密技术。

JavaScript 混淆

JavaScript 混淆完全是在 JavaScript 上面进行的处理，它的目的就是使得 JavaScript 变得难以阅读和分析，大大降低代码可读性，是一种很实用的 JavaScript 保护方案。

JavaScript 混淆技术主要有以下几种：

- 变量混淆

将带有含意的变量名、方法名、常量名随机变为无意义的类乱码字符串，降低代码可读性，如转成单个字符或十六进制字符串。

- 字符串混淆

将字符串阵列化集中放置、并可进行 MD5 或 Base64 加密存储，使代码中不出现明文字符串，这样可以避免使用全局搜索字符串的方式定位到入口点。

- 属性加密

针对 JavaScript 对象的属性进行加密转化，隐藏代码之间的调用关系。

- 控制流平坦化

打乱函数原有代码执行流程及函数调用关系，使代码逻变得混乱无序。

- 僵尸代码

随机在代码中插入无用的僵尸代码、僵尸函数，进一步使代码混乱。

- 调试保护

基于调试器特性，对当前运行环境进行检验，加入一些强制调试 debugger 语句，使其在调试模式下难以顺利执行 JavaScript 代码。

- 多态变异

使 JavaScript 代码每次被调用时，将代码自身即立刻自动发生变异，变化为与之前完全不同的代码，即功能完全不变，只是代码形式变异，以此杜绝代码被动态分析调试。

- 锁定域名

使 JavaScript 代码只能在指定域名下执行。

- 反格式化

如果对 JavaScript 代码进行格式化，则无法执行，导致浏览器假死。

- 特殊编码

将 JavaScript 完全编码为人不可读的代码，如表情符号、特殊表示内容等等。

总之，以上方案都是 JavaScript 混淆的实现方式，可以在不同程度上保护 JavaScript 代码。

在前端开发中，现在 JavaScript 混淆主流的实现是 javascript-obfuscator 这个库，利用它我们可以非常方便地实现页面的混淆，它与 Webpack 结合起来，最终可以输出压缩和混淆后的 JavaScript 代码，使得可读性大大降低，难以逆向。

下面我们会介绍下 javascript-obfuscator 对代码混淆的实现，了解了实现，那么自然我们就对混淆的机理有了更加深刻的认识。

javascript-obfuscator 的官网地址为：https://obfuscator.io/，其官方介绍内容如下：

> A free and efficient obfuscator for JavaScript (including ES2017). Make your code harder to copy and prevent people from stealing your work.

它是支持 ES8 的免费、高效的 JavaScript 混淆库，它可以使得你的 JavaScript 代码经过混淆后难以被复制、盗用，混淆后的代码具有和原来的代码一模一样的功能。

怎么使用呢？首先，我们需要安装好 Node.js，可以使用 npm 命令。

然后新建一个文件夹，比如 js-obfuscate，随后进入该文件夹，初始化工作空间：

```
npm init
```

这里会提示我们输入一些信息，创建一个 package.json 文件，这就完成了项目初始化了。

接下来我们来安装 javascript-obfuscator 这个库：

```
npm install --save-dev javascript-obfuscator
```

接下来我们就可以编写代码来实现混淆了，如新建一个 main.js 文件，内容如下：

```
const code = `
let x = '1' + 1
console.log('x', x)
`

const options = {
   compact: false,
   controlFlowFlattening: true

}

const obfuscator = require('javascript-obfuscator')
function obfuscate(code, options) {
   return obfuscator.obfuscate(code, options).getObfuscatedCode()
}
console.log(obfuscate(code, options))
```

在这里我们定义了两个变量，一个是 code，即需要被混淆的代码，另一个是混淆选项，是一个 Object。接下来我们引入了 javascript-obfuscator 库，然后定义了一个方法，传入 code 和 options，来获取混淆后的代码，最后控制台输出混淆后的代码。

代码逻辑比较简单，我们来执行一下代码：

```
node main.js
```

输出结果如下：

```
var _0x53bf = ['log'];
(function (_0x1d84fe, _0x3aeda0) {
   var _0x10a5a = function (_0x2f0a52) {
       while (--_0x2f0a52) {
           _0x1d84fe['push'](_0x1d84fe['shift']());
       }
```

```
  };
   _0x10a5a(++_0x3aeda0);
}(_0x53bf, 0x172));
var _0x480a = function (_0x4341e5, _0x5923b4) {
   _0x4341e5 = _0x4341e5 - 0x0;
   var _0xb3622e = _0x53bf[_0x4341e5];
   return _0xb3622e;
};
let x = '1' + 0x1;
console[_0x480a('0x0')]('x', x);
```

看到了吧，这么简单的两行代码，被我们混淆成了这个样子，其实这里我们就是设定了一个“控制流扁平化”的选项。

整体看来，代码的可读性大大降低，也大大加大了 JavaScript 调试的难度。

好，接下来我们来跟着 javascript-obfuscator 走一遍，就能具体知道 JavaScript 混淆到底有多少方法了。

代码压缩

这里 javascript-obfuscator 也提供了代码压缩的功能，使用其参数 compact 即可完成 JavaScript 代码的压缩，输出为一行内容。默认是 true，如果定义为 false，则混淆后的代码会分行显示。

示例如下：

```
const code = `
let x = '1' + 1
console.log('x', x)
`
const options = {
   compact: false
}
```

这里我们先把代码压缩 compact 选项设置为 false，运行结果如下：

```
let x = '1' + 0x1;
console['log']('x', x);
```

如果不设置 compact 或把 compact 设置为 true，结果如下：

```
var _0x151c=['log'];(function(_0x1ce384,_0x20a7c7){var _0x25fc92=function(_0x188aec){while(--_0x188aec){_0x1ce384['push'](_0x1ce384['shift']());}};_0x25fc92(++_0x20a7c7);}(_0x151c,0x1b
```

可以看到单行显示的时候，对变量名进行了进一步的混淆和控制流扁平化操作。

变量名混淆

变量名混淆可以通过配置 identifierNamesGenerator 参数实现，我们通过这个参数可以控制变量名混淆的方式，如 hexadecimal 则会替换为 16 进制形式的字符串，在这里我们可以设定如下值：

- hexadecimal：将变量名替换为 16 进制形式的字符串，如 0xabc123。
- mangled：将变量名替换为普通的简写字符，如 a、b、c 等。

该参数默认为 hexadecimal。

我们将该参数修改为 mangled 来试一下：

```
const code = `
let hello = '1' + 1
console.log('hello', hello)
`
const options = {
  compact: true,
  identifierNamesGenerator: 'mangled'
}
```

运行结果如下：

```
var a=['hello'];(function(c,d){var e=function(f){while(--f){c['push'](c['shift']());}};e(++d);}(a,0x9b));var b=function(c,d){c=c-0x0;var e=a[c];return e;};let hello='1'+0x1;console['lo
```

可以看到这里的变量命名都变成了 a、b 等形式。

如果我们将 identifierNamesGenerator 修改为 hexadecimal 或者不设置，运行结果如下：

```
var _0x4e98=['log','hello'];(function(_0x4464de,_0x39de6c){var _0xdffdda=function(_0x6a95d5){while(--_0x6a95d5){_0x4464de['push'](_0x4464de['shift']());}};_0xdffdda(++_0x39de6c);}(_0x4
```

可以看到选用了 mangled，其代码体积会更小，但 hexadecimal 其可读性会更低。

另外我们还可以通过设置 identifiersPrefix 参数来控制混淆后的变量前缀，示例如下：

```
const code = `
let hello = '1' + 1
console.log('hello', hello)
`
const options = {
  identifiersPrefix: 'germey'
}
```

运行结果：

```
var germey_0x3dea=['log','hello'];(function(_0x348ff3,_0x5330e8){var _0x1568b1=function(_0x4740d8){while(--_0x4740d8){_0x348ff3['push'](_0x348ff3['shift']());}};_0x1568b1(++_0x5330e8);
```

可以看到混淆后的变量前缀加上了我们自定义的字符串 germey。

另外 renameGlobals 这个参数还可以指定是否混淆全局变量和函数名称，默认为 false。示例如下：

```
const code = `
var $ = function(id) {
  return document.getElementById(id);
};
`
const options = {
  renameGlobals: true
}
```

运行结果如下：

```
var _0x4864b0=function(_0x5763be){return document['getElementById'](_0x5763be);};
```

可以看到这里我们声明了一个全局变量 $，在 renameGlobals 设置为 true 之后，$ 这个变量也被替换了。如果后文用到了这个 $ 对象，可能就会有找不到定义的错误，因此这个参数可能导致代码执行不通。

如果我们不设置 renameGlobals 或者设置为 false，结果如下：

```
var _0x239a=['getElementById'];(function(_0x3f45a3,_0x583dfa){var _0x2cade2=function(_0x28479a){while(--_0x28479a){_0x3f45a3['push'](_0x3f45a3['shift']());}};_0x2cade2(++_0x583dfa);}(_
```

可以看到，最后还是有 $ 的声明，其全局名称没有被改变。

字符串混淆

字符串混淆，即将一个字符串声明放到一个数组里面，使之无法被直接搜索到。我们可以通过控制 stringArray 参数来控制，默认为 true。

我们还可以通过 rotateStringArray 参数来控制数组化后结果的元素顺序，默认为 true。
还可以通过 stringArrayEncoding 参数来控制数组的编码形式，默认不开启编码，如果设置为 true 或 base64，则会使用 Base64 编码，如果设置为 rc4，则使用 RC4 编码。
还可以通过 stringArrayThreshold 来控制启用编码的概率，范围 0 到 1，默认 0.8。

示例如下：

```
const code = `
var a = 'hello world'
`
const options = {
  stringArray: true,
  rotateStringArray: true,
  stringArrayEncoding: true, // 'base64' or 'rc4' or false
  stringArrayThreshold: 1,
}
```

运行结果如下：

```
var _0x4215=['aGVsbG8gd29ybGQ='];(function(_0x42bf17,_0x4c348f){var _0x328832=function(_0x355be1){while(--_0x355be1){_0x42bf17['push'](_0x42bf17['shift']());}};_0x328832(++_0x4c348f);}
```

可以看到这里就把字符串进行了 Base64 编码，我们再也无法通过查找的方式找到字符串的位置了。

如果将 stringArray 设置为 false 的话，输出就是这样：

```
var a='hello\x20world';
```

字符串就仍然是明文显示的，没有被编码。

另外我们还可以使用 unicodeEscapeSequence 这个参数对字符串进行 Unicode 转码，使之更加难以辨认，示例如下：

```
const code = `
var a = 'hello world'
`
const options = {
  compact: false,
  unicodeEscapeSequence: true
}
```

运行结果如下：

```
var _0x5c0d = ['\x68\x65\x6c\x6c\x6f\x20\x77\x6f\x72\x6c\x64'];
(function (_0x54cc9c, _0x57a3b2) {
  var _0xf833cf = function (_0x3cd8c6) {
    while (--_0x3cd8c6) {
      _0x54cc9c['push'](_0x54cc9c['shift']());
    }
};
_0xf833cf(++_0x57a3b2);
}(_0x5c0d, 0x17d));
var _0x28e8 = function (_0x3fd645, _0x2cf5e7) {
  _0x3fd645 = _0x3fd645 - 0x0;
  var _0x298a20 = _0x5c0d[_0x3fd645];
  return _0x298a20;
};
var a = _0x28e8('0x0');
```

可以看到，这里字符串被数字化和 Unicode 化，非常难以辨认。

在很多 JavaScript 逆向的过程中，一些关键的字符串可能会作为切入点来查找加密入口。用了这种混淆之后，如果有人想通过全局搜索的方式搜索 hello 这样的字符串找加密入口，也没法搜到了。

代码自我保护

我们可以通过设置 selfDefending 参数来开启代码自我保护功能。开启之后，混淆后的 JavaScript 会强制以一行形式显示，如果我们将混淆后的代码进行格式化（美化）或者重命名，该段代码将无法执行。

例如：

```
const code = `
console.log('hello world')
`
const options = {
  selfDefending: true
}
```

运行结果如下：

```
var _0x26da=['log','hello\x20world'];(function(_0x190327,_0x57c2c0){var _0x577762=function(_0xc9dabb){while(--_0xc9dabb){_0x190327['push'](_0x190327['shift']());}};var _0x35976e=functi
```

如果我们将上述代码放到控制台，它的执行结果和之前是一模一样的，没有任何问题。
如果我们将其进行格式化，会变成如下内容：

```
var _0x26da = ['log', 'hello\x20world'];
(function (_0x190327, _0x57c2c0) {
    var _0x577762 = function (_0xc9dabb) {
        while (--_0xc9dabb) {
            _0x190327['push'](_0x190327['shift']());
        }
    };
    var _0x35976e = function () {
        var _0x16b3fe = {
            'data': {
                'key': 'cookie',
                'value': 'timeout'
            },
            'setCookie': function (_0x2d52d5, _0x16feda, _0x57cadf, _0x56056f) {
                _0x56056f = _0x56056f || {};
                var _0x5b6dc3 = _0x16feda + '=' + _0x57cadf;
                var _0x333ced = 0x0;
                for (var _0x333ced = 0x0, _0x19ae36 = _0x2d52d5['length']; _0x333ced < _0x19ae36; _0x333ced++) {
                    var _0x409587 = _0x2d52d5[_0x333ced];
                    _0x5b6dc3 += ';\x20' + _0x409587;
                    var _0x4aa006 = _0x2d52d5[_0x409587];
                    _0x2d52d5['push'](_0x4aa006);
                    _0x19ae36 = _0x2d52d5['length'];
                    if (_0x4aa006 !== !![]) {
                        _0x5b6dc3 += '=' + _0x4aa006;
                    }
                }
                _0x56056f['cookie'] = _0x5b6dc3;
            }, 'removeCookie': function () {
                return 'dev';
            }, 'getCookie': function (_0x30c497, _0x51923d) {
                _0x30c497 = _0x30c497 || function (_0x4b7e18) {
                    return _0x4b7e18;
                };
                var _0x557e06 = _0x30c497(new RegExp('(?:^|;\x20)' + _0x51923d['replace'](/([.$?*|{}()[]\/+^])/g, '$1') + '=([^;]*)'));
```

```
                var _0x817646 = function (_0xf3fae7, _0x5d8208) {
                    _0xf3fae7(++_0x5d8208);
                };
                _0x817646(_0x577762, _0x57c2c0);
                return _0x557e06 ? decodeURIComponent(_0x557e06[0x1]) : undefined;
            }
        };
        var _0x4673cd = function () {
            var _0x4c6c5c = new RegExp('\x5cw+\x20*\x5c(\x5c)\x20*{\x5cw+\x20*[\x27|\x22].+[\x27|\x22];?\x20*}');
            return _0x4c6c5c['test'](_0x16b3fe['removeCookie']['toString']());
        };
        _0x16b3fe['updateCookie'] = _0x4673cd;
        var _0x5baa80 = '';
        var _0x1faf19 = _0x16b3fe['updateCookie']();
        if (!_0x1faf19) {
            _0x16b3fe['setCookie'](['*'], 'counter', 0x1);
        } else if (_0x1faf19) {
            _0x5baa80 = _0x16b3fe['getCookie'](null, 'counter');
        } else {
            _0x16b3fe['removeCookie']();
        }
    };
    _0x35976e();
}(_0x26da, 0x140));
var _0x4391 = function (_0x1b42d8, _0x57edc8) {
    _0x1b42d8 = _0x1b42d8 - 0x0;
    var _0x2fbeca = _0x26da[_0x1b42d8];
    return _0x2fbeca;
};
var _0x197926 = function () {
    var _0x10598f = !![];
    return function (_0xffa3b3, _0x7a40f9) {
        var _0x48e571 = _0x10598f ? function () {
            if (_0x7a40f9) {
                var _0x2194b5 = _0x7a40f9['apply'](_0xffa3b3, arguments);
                _0x7a40f9 = null;
                return _0x2194b5;
            }
        } : function () {};
        _0x10598f = ![];
        return _0x48e571;
    };
}();
var _0x2c6fd7 = _0x197926(this, function () {
    var _0x4828bb = function () {
            return '\x64\x65\x76';
        },
        _0x35c3bc = function () {
            return '\x77\x69\x6e\x64\x6f\x77';
        };
    var _0x456070 = function () {
        var _0x4576a4 = new RegExp('\x5c\x77\x2b\x20\x2a\x5c\x28\x5c\x29\x20\x2a\x7b\x5c\x77\x2b\x20\x2a\x5b\x27\x7c\x22\x5d\x2e\x2b\x5b\x27\x7c\x22\x5d\x3b\x3f\x20\x2a\x7d');
        return !_0x4576a4['\x74\x65\x73\x74'](_0x4828bb['\x74\x6f\x53\x74\x72\x69\x6e\x67']());
    };
    var _0x3fde69 = function () {
        var _0xabb6f4 = new RegExp('\x28\x5c\x5c\x5b\x78\x7c\x75\x5d\x28\x5c\x77\x29\x7b\x32\x2c\x34\x7d\x29\x2b');
        return _0xabb6f4['\x74\x65\x73\x74'](_0x35c3bc['\x74\x6f\x53\x74\x72\x69\x6e\x67']());
    };
    var _0x2d9a50 = function (_0x58fdb4) {
        var _0x2a6361 = ~~0x1 >> 0x1 + 0xff % 0x0;
        if (_0x58fdb4['\x69\x6e\x64\x65\x78\x4f\x66']('\x69' === _0x2a6361)) {
            _0xc388c5(_0x58fdb4);
        }
    };
    var _0xc388c5 = function (_0x2073d6) {
        var _0x6bb49f = ~~0x4 >> 0x1 + 0xff % 0x0;
        if (_0x2073d6['\x69\x6e\x64\x65\x78\x4f\x66']((!![] + '')[0x3]) !== _0x6bb49f) {
            _0x2d9a50(_0x2073d6);
        }
    };
    if (!_0x456070()) {
        if (!_0x3fde69()) {
            _0x2d9a50('\x69\x6e\x64\u0435\x78\x4f\x66');
        } else {
            _0x2d9a50('\x69\x6e\x64\x65\x78\x4f\x66');
        }
    } else {
        _0x2d9a50('\x69\x6e\x64\u0435\x78\x4f\x66');
    }
});
_0x2c6fd7();
console[_0x4391('0x0')](_0x4391('0x1'));
```

如果把这段代码放到浏览器里面，浏览器会直接卡死无法运行。这样如果有人对代码进行了格式化，就无法正常对代码进行运行和调试，从而起到了保护作用。

控制流平坦化

控制流平坦化其实就是将代码的执行逻辑混淆，使其变得复杂难读。其基本思想是将一些逻辑处理块都统一加上一个前驱逻辑块，每个逻辑块都由前驱逻辑块进行条件判断和分发，构成一个个闭环逻辑，导致整个执行逻辑十分复杂难读。

我们通过 controlFlowFlattening 变量可以控制是否开启控制流平坦化，示例如下：

```
const code = `
(function(){
    function foo () {
        return function () {
            var sum = 1 + 2;
            console.log(1);
            console.log(2);
            console.log(3);
            console.log(4);
            console.log(5);
            console.log(6);
        }
    }

    foo()();
})();
`
const options = {
  compact: false,
  controlFlowFlattening: true
}
```

输出结果如下：

```
var _0xbaf1 = [
    'dZwUe',
    'log',
    'fXqMu',
```

```
    '0|1|3|4|6|5|2',
    'chYMl',
    'IZEsA',
    'split'
];
(function (_0x22d342, _0x4f6332) {
    var _0x43ff59 = function (_0x5ad417) {
        while (--_0x5ad417) {
            _0x22d342['push'](_0x22d342['shift']());
        }
    };
    _0x43ff59(++_0x4f6332);
}(_0xbaf1, 0x192));
var _0x1a69 = function (_0x8d64b1, _0x5e07b3) {
    _0x8d64b1 = _0x8d64b1 - 0x0;
    var _0x300bab = _0xbaf1[_0x8d64b1];
    return _0x300bab;
};
(function () {
    var _0x19d8ce = {
        'chYMl': _0x1a69('0x0'),
        'IZEsA': function (_0x22e521, _0x298a22) {
            return _0x22e521 + _0x298a22;
        },
        'fXqMu': function (_0x13124b) {
            return _0x13124b();
        }
    };
    function _0x4e2ee0() {
        var _0x118a6a = {
            'LZAQV': _0x19d8ce[_0x1a69('0x1')],
            'dZwUe': function (_0x362ef3, _0x352709) {
                return _0x19d8ce[_0x1a69('0x2')](_0x362ef3, _0x352709);
            }
        };
        return function () {
            var _0x4c336d = _0x118a6a['LZAQV'][_0x1a69('0x3')]('|'), _0x2b6466 = 0x0;
            while (!![]) {
                switch (_0x4c336d[_0x2b6466++]) {
                case '0':
                    var _0xbfa3fd = _0x118a6a[_0x1a69('0x4')](0x1, 0x2);
                    continue;
                case '1':
                    console['log'](0x1);
                    continue;
                case '2':
                    console[_0x1a69('0x5')](0x6);
                    continue;
                case '3':
                    console[_0x1a69('0x5')](0x2);
                    continue;
                case '4':
                    console[_0x1a69('0x5')](0x3);
                    continue;
                case '5':
                    console[_0x1a69('0x5')](0x5);
                    continue;
                case '6':
                    console[_0x1a69('0x5')](0x4);
                    continue;
                }
                break;
            }
        };
    }
    _0x19d8ce[_0x1a69('0x6')](_0x4e2ee0)();
}());
```

可以看到，一些连续的执行逻辑被打破，代码被修改为一个 switch 语句，我们很难再一眼看出多条 console.log 语句的执行顺序了。

如果我们将 controlFlowFlattening 设置为 false 或者不设置，运行结果如下：

```
var _0x552c = ['log'];
(function (_0x4c4fa0, _0x59faa0) {
    var _0xa01786 = function (_0x409a37) {
        while (--_0x409a37) {
            _0x4c4fa0['push'](_0x4c4fa0['shift']());
        }
    };
    _0xa01786(++_0x59faa0);
}(_0x552c, 0x9b));
var _0x4e63 = function (_0x75ea1a, _0x50e176) {
    _0x75ea1a = _0x75ea1a - 0x0;
    var _0x59dc94 = _0x552c[_0x75ea1a];
    return _0x59dc94;
};
(function () {
    function _0x507f38() {
        return function () {
            var _0x17fb7e = 0x1 + 0x2;
            console[_0x4e63('0x0')](0x1);
            console['log'](0x2);
            console['log'](0x3);
            console[_0x4e63('0x0')](0x4);
            console[_0x4e63('0x0')](0x5);
            console[_0x4e63('0x0')](0x6);
        };
    }
    _0x507f38()();
}());
```

可以看到，这里仍然保留了原始的 console.log 执行逻辑。

因此，使用控制流扁平化可以使得执行逻辑更加复杂难读，目前非常多的前端混淆都会加上这个选项。

但启用控制流扁平化之后，代码的执行时间会变长，最长达 1.5 倍之多。

另外我们还能使用 controlFlowFlatteningThreshold 这个参数来控制比例，取值范围是 0 到 1，默认 0.75，如果设置为 0，那相当于 controlFlowFlattening 设置为 false，即不开启控制流扁平化 。

僵尸代码注入

僵尸代码即不会被执行的代码或对上下文没有任何影响的代码，注入之后可以对现有的 JavaScript 代码的阅读形成干扰。我们可以使用 deadCodeInjection 参数开启这个选项，默认为 false。
示例如下：

```
const code = `
(function(){
    if (true) {
        var foo = function () {
            console.log('abc');
```

```
            console.log('cde');
            console.log('efg');
            console.log('hij');
        };

        var bar = function () {
            console.log('klm');
            console.log('nop');
            console.log('qrs');
        };

        var baz = function () {
            console.log('tuv');
            console.log('wxy');
            console.log('z');
        };

        foo();
        bar();
        baz();
    }
})();
`
const options = {
  compact: false,
  deadCodeInjection: true
}
```

运行结果如下：

```
var _0x5024 = [
    'zaU',
    'log',
    'tuv',
    'wxy',
    'abc',
    'cde',
    'efg',
    'hij',
    'QhG',
    'TeI',
    'klm',
    'nop',
    'qrs',
    'bZd',
    'HMx'
];
var _0x4502 = function (_0x1254b1, _0x583689) {
    _0x1254b1 = _0x1254b1 - 0x0;
    var _0x529b49 = _0x5024[_0x1254b1];
    return _0x529b49;
};
(function () {
    if (!![]) {
        var _0x16c18d = function () {
            if (_0x4502('0x0') !== _0x4502('0x0')) {
                console[_0x4502('0x1')](_0x4502('0x2'));
                console[_0x4502('0x1')](_0x4502('0x3'));
                console[_0x4502('0x1')]('z');
            } else {
                console[_0x4502('0x1')](_0x4502('0x4'));
                console[_0x4502('0x1')](_0x4502('0x5'));
                console[_0x4502('0x1')](_0x4502('0x6'));
                console[_0x4502('0x1')](_0x4502('0x7'));
            }
        };
        var _0x1f7292 = function () {
            if (_0x4502('0x8') === _0x4502('0x9')) {
                console[_0x4502('0x1')](_0x4502('0xa'));
                console[_0x4502('0x1')](_0x4502('0xb'));
                console[_0x4502('0x1')](_0x4502('0xc'));
            } else {
                console[_0x4502('0x1')](_0x4502('0xa'));
                console[_0x4502('0x1')](_0x4502('0xb'));
                console[_0x4502('0x1')](_0x4502('0xc'));
            }
        };
        var _0x33b212 = function () {
            if (_0x4502('0xd') !== _0x4502('0xe')) {
                console[_0x4502('0x1')](_0x4502('0x2'));
                console[_0x4502('0x1')](_0x4502('0x3'));
                console[_0x4502('0x1')]('z');
            } else {
                console[_0x4502('0x1')](_0x4502('0x4'));
                console[_0x4502('0x1')](_0x4502('0x5'));
                console[_0x4502('0x1')](_0x4502('0x6'));
                console[_0x4502('0x1')](_0x4502('0x7'));
            }
        };
        _0x16c18d();
        _0x1f7292();
        _0x33b212();
    }
}());
```

可见这里增加了一些不会执行到的逻辑区块内容。

如果将 deadCodeInjection 设置为 false 或者不设置，运行结果如下：

```
var _0x402a = [
    'qrs',
    'wxy',
    'log',
    'abc',
    'cde',
    'efg',
    'hij',
    'nop'
];
(function (_0x57239e, _0x4747e8) {
    var _0x3998cd = function (_0x34a502) {
        while (--_0x34a502) {
            _0x57239e['push'](_0x57239e['shift']());
        }
    };
    _0x3998cd(++_0x4747e8);
}(_0x402a, 0x162));
var _0x5356 = function (_0x2f2c10, _0x2878a6) {
    _0x2f2c10 = _0x2f2c10 - 0x0;
    var _0x4cfe02 = _0x402a[_0x2f2c10];
    return _0x4cfe02;
```

```
};
(function () {
    if (!![]) {
        var _0x60edc1 = function () {
            console[_0x5356('0x0')](_0x5356('0x1'));
            console[_0x5356('0x0')](_0x5356('0x2'));
            console[_0x5356('0x0')](_0x5356('0x3'));
            console['log'](_0x5356('0x4'));
        };
        var _0x56405f = function () {
            console[_0x5356('0x0')]('klm');
            console['log'](_0x5356('0x5'));
            console['log'](_0x5356('0x6'));
        };
        var _0x332d12 = function () {
            console[_0x5356('0x0')]('tuv');
            console[_0x5356('0x0')](_0x5356('0x7'));
            console['log']('z');
        };
        _0x60edc1();
        _0x56405f();
        _0x332d12();
    }
}());
```

另外我们还可以通过设置 deadCodeInjectionThreshold 参数来控制僵尸代码注入的比例，取值 0 到 1，默认是 0.4。

僵尸代码可以起到一定的干扰作用，所以在有必要的时候也可以注入。

对象键名替换

如果是一个对象，可以使用 transformObjectKeys 来对对象的键值进行替换，示例如下：

```
const code = `
(function(){
    var object = {
        foo: 'test1',
        bar: {
            baz: 'test2'
        }
    };
})();
`
const options = {
  compact: false,
  transformObjectKeys: true
}
```

输出结果如下：

```
var _0x7a5d = [
    'bar',
    'test2',
    'test1'
];
(function (_0x59fec5, _0x2e4fac) {
    var _0x231e7a = function (_0x46f33e) {
        while (--_0x46f33e) {
            _0x59fec5['push'](_0x59fec5['shift']());
        }
    };
    _0x231e7a(++_0x2e4fac);
}(_0x7a5d, 0x167));
var _0x3bc4 = function (_0x309ad3, _0x22d5ac) {
    _0x309ad3 = _0x309ad3 - 0x0;
    var _0x3a034e = _0x7a5d[_0x309ad3];
    return _0x3a034e;
};
(function () {
    var _0x9f1fd1 = {};
    _0x9f1fd1['foo'] = _0x3bc4('0x0');
    _0x9f1fd1[_0x3bc4('0x1')] = {};
    _0x9f1fd1[_0x3bc4('0x1')]['baz'] = _0x3bc4('0x2');
}());
```

可以看到，Object 的变量名被替换为了特殊的变量，这也可以起到一定的防护作用。

禁用控制台输出

可以使用 disableConsoleOutput 来禁用掉 console.log 输出功能，加大调试难度，示例如下：

```
const code = `
console.log('hello world')
`
const options = {
    disableConsoleOutput: true
}
```

运行结果如下：

```
var _0x3a39=['debug','info','error','exception','trace','hello\x20world','apply','{}.constructor(\x22return\x20this\x22)(\x20)','console','log','warn'];(function(_0x2a157a,_0x5d9d3b){v
```

此时，我们如果执行这段代码，发现是没有任何输出的，这里实际上就是将 console 的一些功能禁用了，加大了调试难度。

调试保护

我们可以使用 debugProtection 来禁用调试模式，进入无限 Debug 模式。另外我们还可以使用 debugProtectionInterval 来启用无限 Debug 的间隔，使得代码在调试过程中会不断进入断点模式，无法顺畅执行。
示例如下：

```
const code = `
for (let i = 0; i < 5; i ++) {
    console.log('i', i)
}
`
const options = {
    debugProtection: true
}
```

运行结果如下：

```
var _0x41d0=['action','debu','stateObject','function\x20*\x5c(\x20*\x5c)','\x5c+\x5c+\x20*(?:_0x(?:[a-f0-9]){4,6}|(?:\x5cb|\x5cd)[a-z0-9]{1,4}(?:\x5cb|\x5cd))','init','test','chain','i
```

如果我们将代码粘贴到控制台，其会不断跳到 debugger 代码的位置，无法顺畅执行。

域名锁定

我们可以通过控制 domainLock 来控制 JavaScript 代码只能在特定域名下运行，这样就可以降低被模拟的风险。

示例如下：

```
const code = `
console.log('hello world')
`
const options = {
    domainLock: ['cuiqingcai.com']
}
```

运行结果如下：

```
var _0x3203=['apply','return\x20(function()\x20','{}.constructor(\x22return\x20this\x22)(\x20)','item','attribute','value','replace','length','charCodeAt','log','hello\x20world'];(func
```

这段代码就只能在指定域名 cuiqingcai.com 下运行，不能在其他网站运行，不信你可以试试。

特殊编码

另外还有一些特殊的工具包，如使用 aaencode、jjencode、jsfuck 等工具对代码进行混淆和编码。

示例如下：

```
var a = 1
```

jsfuck 的结果：

```
[][(![]+[])[!+[]+!![]+!![]]+([]+{})[+!![]]+(!![]+[])[+!![]]+(!![]+[])[+[]]][([]+{})[!+[]+!![]+!![]+!![]]+([]+{})[+!![]]+([][[]]+[])[+!![]]+(![]+[])[!+[]+!![]+!![]]+(!![]+[])[+[]]+
```

aaencode 的结果：

```
ﾟωﾟﾉ= / `m´）ﾉ ~┻━┻ / ['_']; o=(ﾟｰﾟ) =_=3; c=(ﾟΘﾟ) =(ﾟｰﾟ)-(ﾟｰﾟ); (ﾟДﾟ) =(ﾟΘﾟ)= (o^_^o)/ (o^_^o);(ﾟДﾟ)={ﾟΘﾟ: '_' ,ﾟωﾟﾉ : ((ﾟωﾟﾉ==3) +'_') [ﾟΘﾟ] ,ﾟｰﾟﾉ :(ﾟωﾟﾉ+ '_')[o^_^o -(ﾟΘﾟ)] ,ﾟДﾟﾉ:((ﾟｰﾟ==3) +'_')[ﾟｰﾟ] };
```

jjencode 的结果：

```
$=~[];$={___:++$,$$$$:(![]+"")[$],__$:++$,$_$_:(![]+"")[$],_$_:++$,$_$$:({}+"")[$],$$_$:($[$]+"")[$],_$$:++$,$$$_:(!""+"")[$],$__:++$,$_$:++$,$$__:({}+"")[$],$$_:++$,$$$:++$,$___:++$,$
```

这些混淆方式比较另类，但只需要输入到控制台即可执行，其没有真正达到强力混淆的效果。

以上便是对 JavaScript 混淆方式的介绍和总结。总的来说，经过混淆的 JavaScript 代码其可读性大大降低，同时防护效果也大大增强。

JavaScript 加密

不同于 JavaScript 混淆技术，JavaScript 加密技术可以说是对 JavaScript 混淆技术防护的进一步升级，其基本思路是将一些核心逻辑使用诸如 C/C++ 语言来编写，并通过 JavaScript 调用执行，从而起到二进制级别的防护作用。

其加密的方式现在有 Emscripten 和 WebAssembly 等，其中后者越来越成为主流。
下面我们分别来介绍下。

Emscripten

现在，许多 3D 游戏都是用 C/C++ 语言写的，如果能将 C / C++ 语言编译成 JavaScript 代码，它们不就能在浏览器里运行了吗？众所周知，JavaScript 的基本语法与 C 语言高度相似。于是，有人开始研究怎么才能实现这个目标，为此专门做了一个编译器项目 Emscripten。这个编译器可以将 C / C++ 代码编译成 JavaScript 代码，但不是普通的 JavaScript，而是一种叫作 asm.js 的 JavaScript 变体。

因此说，某些 JavaScript 的核心功能可以使用 C/C++ 语言实现，然后通过 Emscripten 编译成 asm.js，再由 JavaScript 调用执行，这可以算是一种前端加密技术。

WebAssembly

如果你对 JavaScript 比较了解，可能知道还有一种叫作 WebAssembly 的技术，也能将 C/C++ 转成 JavaScript 引擎可以运行的代码。那么它与 asm.js 有何区别呢？

其实两者的功能基本一致，就是转出来的代码不一样：asm.js 是文本，WebAssembly 是二进制字节码，因此运行速度更快、体积更小。从长远来看，WebAssembly 的前景更光明。

WebAssembly 是经过编译器编译之后的字节码，可以从 C/C++ 编译而来，得到的字节码具有和 JavaScript 相同的功能，但它体积更小，而且在语法上完全脱离 JavaScript，同时具有沙盒化的执行环境。

利用 WebAssembly 技术，我们可以将一些核心的功能利用 C/C++ 语言实现，形成浏览器字节码的形式。然后在 JavaScript 中通过类似如下的方式调用：

```
WebAssembly.compile(new Uint8Array(`
  00 61 73 6d  01 00 00 00  01 0c 02 60  02 7f 7f 01
  7f 60 01 7f  01 7f 03 03  02 00 01 07  10 02 03 61
  64 64 00 00  06 73 71 75  61 72 65 00  01 0a 13 02
  08 00 20 00  20 01 6a 0f  0b 08 00 20  00 20 00 6c
  0f 0b`.trim().split(/[\s\r\n]+/g).map(str => parseInt(str, 16))
)).then(module => {
  const instance = new WebAssembly.Instance(module)
  const { add, square } = instance.exports
  console.log('2 + 4 =', add(2, 4))
  console.log('3^2 =', square(3))
  console.log('(2 + 5)^2 =', square(add(2 + 5)))
})
```

这种加密方式更加安全，因为作为二进制编码它能起到的防护效果无疑是更好的。如果想要逆向或破解那得需要逆向 WebAssembly，难度也是很大的。

总结

以上，我们就介绍了接口加密技术和 JavaScript 的压缩、混淆和加密技术，知己知彼方能百战不殆，了解了原理，我们才能更好地去实现 JavaScript 的逆向。
本节代码：https://github.com/Python3WebSpider/JavaScriptObfuscate

参考文献

- https://www.ruanyifeng.com/blog/2017/09/asmjs_emscripten.html
- https://juejin.im/post/5cfcb9d25188257e853fa71c#heading-23
- https://www.jianshu.com/p/326594cbd4fa
- https://github.com/javascript-obfuscator/javascript-obfuscator
- https://obfuscator.io/
- https://www.sojson.com/jjencode.html
- http://dean.edwards.name/packer/