进阶–补环境自吐 1

课前

@ 所有人 这个月开始进行 B 级攻坚能力培训，给大家分享一套补环境万能公式。明天晚上开始第一次分享，分享的内容主要是面对一些高强度混淆的 js 脚本，如何通过`补环境`自吐的方式降低逆向难度，实现加密反爬速通。分享前请大家先回忆以及思考以下几个问题：

1. 什么是`补环境`？怎么`补`？
2. 如何实现加密环境`自吐`？原理是什么？
3. 什么是`Proxy`、`Reflect`？他们的作用是什么？请先根据官方文档 https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Proxy 复习一下`Proxy`的简单用法。
4. 回忆一下`debugger`反调试：http://112.74.172.252:9080/exam/c/index.html

属性描述符

是描述属性例如是否可以修改等特性的属性

//获取所有属性

Object.getOwnPropertyDescriptors(Navigator.prototype)

//获取这个属性的属性哈描述符

Object.getOwnPropertyDescriptor(Navigator.prototype,'appCodeName')

//属性描述符的例子

set 为 undefined 无法修改
[native code] 原生方法，如果模拟的时候原生方法不返回[native code]可能就被识别为假了

```
//定义属性描述符
Object.defineProperty(x,'y',{
  get:() => 123,
})
```

proxy 和 reflect 的使用范例

对于对象的所有操作都可以拦截

JavaScript

```
const x = {}
const z = new Proxy(x, {
    construct(target, args) {
        //当是对象被new的时候，才有用
        console.log("new ${target.name} 传入的参数是:${JSON.stringify([...args])}")
        retrun
        new target(...args) //参数解构
    },
    apply() {
        //如果是函数，才有用，如果是对象则无效果
    },
    get(target, property, receiver) {
        //console.log('属性读取操作的捕捉器')
        const res = Reflect.get(target, property, receiver)
        console.log('对 ${target} 取了 ${property} 属性，结果是 ${JSON.stringify(res)}')
        return res
    },
    set(target, property, value, receiver) {
        //console.log("属性设置操作的捕捉器")
        const res = Reflect.set(target, property, value, receiver)
        console.log("对 ${target.name} 设置了 ${property} 属性,设置的属性值是 ${value}")
        return res
    },
    has(target, prop) {
        //console.log("in操作符的的捕捉器")
        const res = Reflect.has(target, prop)
        return res
    },
    getOwnPropertyDescriptor(obj, prop) {
        //console.log("取属性符的操作可以拦截下来")
        const res = Reflect.getOwnPropertyDescriptor(obj, prop)
        return res
    },
    defineProperty(target, prop, descriptor) {
        //console.log("定义属性描述的时候拦截")
        const res = Reflect.defineProperty(target, prop, descriptor)
        return res
```

```
    },
    deleteProperty(target, property) {
        //console.log("定义属性描述的时候拦截")
        const res = Reflect.deleteProperty(target, property)
        return res
    },
    ownKeys(target) {
        //console.log("获取了target的属性列表")
        const res = Reflect.ownKeys(target)
        return res
    }

});

const y = z.c;
z["c"]
```

智海给的成熟的代理对象

JavaScript

```
function common_proxy(obj, opts = {}) {
    let {
        identifier,
        prototype,
        stringify,
        native
    } = opts
    if (native) {
        func_set_natvie(obj)
    }
    if (!hook) return obj
    let obj_type = typeof obj
    if (obj_type != "object" && obj_type != "function") return

    if (!identifier) {
        if (prototype) {
            identifier = prototype.name + '.prototype.' + obj.name
        } else {
            identifier = obj instanceof Function ? `[function ${obj.name}]` : obj.toStrin
        }
    }
    if (!identifier.startsWith('[')) {
        identifier = `[` + identifier + ']'
    }
    if (stringify) {
        _stringify_prototypes.push(obj)
    }
    return new Proxy(obj, {
        construct() {
            let result = Reflect.construct.apply(this, arguments);
            console.log(`${identifier}.new => `, arguments, result);
```

```
            return constructor_excepts.find(e => result instanceof e) ? result : common_p
        },
        apply() {
            try {
                let result = Reflect.apply.apply(this, arguments);
                console.log(`${identifier}.apply => `, _stringify(arguments[1]), argument
                return result
            } catch (e) {
                console.log(`${identifier}.apply => `, _stringify(arguments[1]), argument
                throw e
            }
        },
        // 代理这个对象的属性设置  a.b , a["b"]
        get: function () {
            let result = Reflect.get.apply(this, arguments)
            if (typeof arguments[1] == "string" && !arguments[1].startsWith("_")) {
                if (hook_funcs.includes(arguments[1])) {
                    result = common_proxy(result, {
                        identifier: identifier + `.` + arguments[1]
                    })
                }
                console.log(
                    `${identifier}.get => `, arguments[1], _stringify(result)
                )
            }
            return result
        },
        // 代理这个对象的属性设置  a.b = 1, a["b"]  1
        set: function () {
            let result = Reflect.set.apply(this, arguments)
            if (typeof arguments[1] == "string" && !arguments[1].startsWith("_")) {
                console.log(`${identifier}.set => `, arguments[1], _stringify(arguments[2
            }
            return result
        },
        // in 操作符的捕捉器  "xx" in a
        has: function () {
            let result = Reflect.has.apply(this, arguments)
            console.log(`${identifier}.has => `, arguments[1], result);
            return result
        },
        // Object.getOwnPropertyNames(a) 方法和 Object.getOwnPropertySymbols(a) 方法的捕捉器
        ownKeys: function () {
            let result = Reflect.ownKeys.apply(this, arguments)
            console.log(`${identifier}.ownKeys => `, result.length);
            return result
        },
        // delete a.xxx
        deleteProperty: function () {
            let result = Reflect.deleteProperty.apply(this, arguments)
            console.log(`${identifier}.delete => `, arguments, result);
            return result
```

```
        },
    })
```

补环境万能开头

JavaScript

```
const hook = true, compress = true

delete process
delete global
delete require
delete module
delete Buffer
delete __dirname
delete __file__

const hook_funcs = ['toString', 'hasOwnProperty']
const constructor_excepts = [Date, RegExp]

const $toString = Function.toString;
const myFunction_toString_symbol = Symbol('('.concat('', ')_', (Math.random() + '').toStr
const myToString = function () {
    return typeof this == 'function' && this[myFunction_toString_symbol] || (result = $to
};

function set_native(func, key, value) {
    return Object.defineProperty(func, key, {
        "enumerable": false,
        "configurable": true,
        "writable": true,
        "value": value
    })
};
delete Function.prototype['toString'];
set_native(Function.prototype, "toString", myToString);
set_native(Function.prototype.toString, myFunction_toString_symbol, "function toString()
function func_set_natvie(func) {
    return set_native(func, myFunction_toString_symbol, `function ${myFunction_toString_s
}

let _origin_string_idx_of = String.prototype.indexOf
String.prototype.indexOf = function () {
    let result = _origin_string_idx_of.apply(this, arguments)
    console.log(`[string ${this}].indexOf`, arguments, result)
    return result
}
func_set_natvie(String.prototype.indexOf)

let _origin_regexp_test = RegExp.prototype.test
RegExp.prototype.test = function () {
    let result = _origin_regexp_test.apply(this, arguments)
```

```
    console.log(`[regexp ${this}].test`, arguments, result)
    return result
}
func_set_natvie(RegExp.prototype.test)

let _stringify_prototypes = []

function _stringify(e) {
    let ret = _stringify_prototypes.find(k => e instanceof k)
    return ret ? `[object ${ret.name}]` : e
}

function common_proxy(obj, opts = {}) {
    let {
        identifier,
        prototype,
        stringify,
        native
    } = opts
    if (native) {
        func_set_natvie(obj)
    }
    if (!hook) return obj
    let obj_type = typeof obj
    if (obj_type != "object" && obj_type != "function") return

    if (!identifier) {
        if (prototype) {
            identifier = prototype.name + '.prototype.' + obj.name
        } else {
            identifier = obj instanceof Function ? `[function ${obj.name}]` : obj.toStrin
        }
    }
    if (!identifier.startsWith('[')) {
        identifier = `[` + identifier + ']'
    }
    if (stringify) {
        _stringify_prototypes.push(obj)
    }
    return new Proxy(obj, {
        construct() {
            let result = Reflect.construct.apply(this, arguments);
            console.log(`${identifier}.new => `, arguments, result);
            return constructor_excepts.find(e => result instanceof e) ? result : common_p
        },
        apply() {
            try {
                let result = Reflect.apply.apply(this, arguments);
                console.log(`${identifier}.apply => `, _stringify(arguments[1]), argument
                return result
            } catch (e) {
                console.log(`${identifier}.apply => `, _stringify(arguments[1]), argument
```

```
                throw e
            }
        },
        // 代理这个对象的属性设置  a.b , a["b"]
        get: function () {
            let result = Reflect.get.apply(this, arguments)
            if (typeof arguments[1] == "string" && !arguments[1].startsWith("_")) {
                if (hook_funcs.includes(arguments[1])) {
                    result = common_proxy(result, {
                        identifier: identifier + `.` + arguments[1]
                    })
                }
                console.log(
                    `${identifier}.get => `, arguments[1], _stringify(result)
                )
            }
            return result
        },
        // 代理这个对象的属性设置  a.b = 1, a["b"]  1
        set: function () {
            let result = Reflect.set.apply(this, arguments)
            if (typeof arguments[1] == "string" && !arguments[1].startsWith("_")) {
                console.log(`${identifier}.set => `, arguments[1], _stringify(arguments[2
            }
            return result
        },
        // in 操作符的捕捉器  "xx" in a
        has: function () {
            let result = Reflect.has.apply(this, arguments)
            console.log(`${identifier}.has => `, arguments[1], result);
            return result
        },
        // Object.getOwnPropertyNames(a) 方法和 Object.getOwnPropertySymbols(a) 方法的捕捉器
        ownKeys: function () {
            let result = Reflect.ownKeys.apply(this, arguments)
            console.log(`${identifier}.ownKeys => `, result.length);
            return result
        },
        // delete a.xxx
        deleteProperty: function () {
            let result = Reflect.deleteProperty.apply(this, arguments)
            console.log(`${identifier}.delete => `, arguments, result);
            return result
        },
    })
}

class Window {
    constructor() {}
    get[Symbol.toStringTag]() {
        return "Window"
    }
```

```
}

window = common_proxy(new Window)
window.Window = common_proxy(Window, {
    native: true,
    stringify: true
})
```

实战公式

首先复制 JS+ 万能开头

运行，查看缺什么

把缺的东西在浏览器上运行并查找原型链

原型链的查找： document.__proto__.__proto__.__proto__.....

根据原型链的情况，依据原型链一层在 class 上创建

JavaScript

```
class X extend {
    constructor() {}
    get[Symbol.toStringTag]() {
        return "X"
    }
}
X = common_proxy(new X)
window.X = X
```