**渗透测试高级技巧（三）：被前端加密后的漏洞测试**

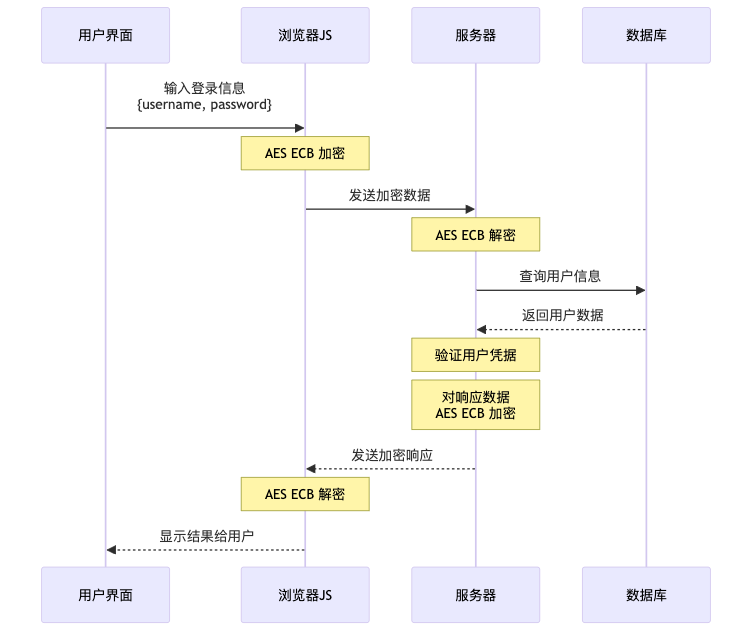
**背景**

在之前的高级前端加密和动态加密流程中，我们基本可以实现针对一些 JS 加密后的表单进行发包测试，但是仅仅是发包还不够，我们发现要进行大量测试的话，现有的基础设施不经过一些改动或者自定义，无法加速测试过程。

我们在本篇文章中，就会尝试针对一个有漏洞但是被前端加密的接口进行测试，在测试过程中，将会使用热加载技术，在恰当的时候进行恰当的自动加密解密，从而让测试变得容易。

**前端加密后的 SQL 注入**

我们考虑以下登陆场景，在这个场景中，用户界面和服务器之间通信使用浏览器 JS 加密，前后都用 AES ECB 加密。



这个时序图展示了整个加密登录流程：

1. **用户界面到 JS 层**：

* 用户在界面输入用户名和密码
* 数据以 JSON 格式传递给 JS 层处理

1. **JS 层加密处理**：

* JS 接收到原始 JSON 数据
* 使用 AES ECB 模式进行加密

1. **客户端到服务器传输**：

* 发送加密后的数据到服务器

1. **服务器处理**：

* 服务器接收加密数据并进行解密
* 查询数据库获取用户信息
* 验证用户凭据
* 准备响应数据并进行加密

1. **服务器到客户端响应**：

* 发送加密的响应数据回客户端

1. **客户端处理响应**：

* JS 层解密服务器响应
* 将解密后的结果显示在用户界面

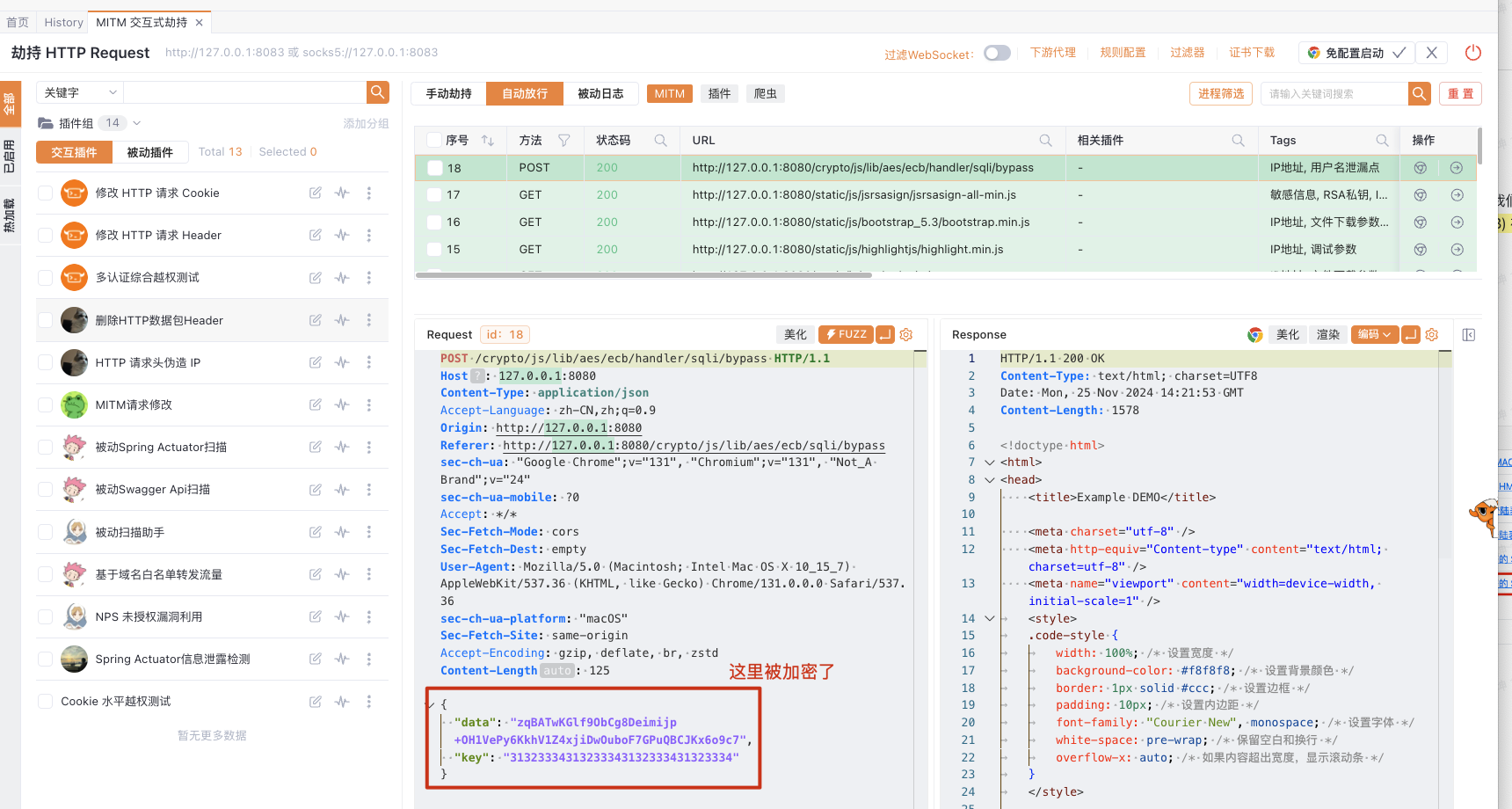
**准备工作**

1. **启动靶场**

在开始之前，大家需要先启动 Yakit 的 Vulinbox 靶场，在这个靶场中，我们将会开始我们今天要操作的加密和解密处理。点击 2024 - 11 - 25 之后的靶场，在最新的靶场中，将会看到 CryptoJS.AES(ECB) 被前端加密的 SQL 注入（Bypass认证）这个靶场，点击进入后会看到一个登录框和一些基本提示。



2. **Yakit 抓包：发现关键加密请求**



3. **分析 JS 加密算法**

因为在前两篇文章中，我们已经讲解了 JS 加密算法如何分析，在这里就简略一点，把篇幅留给重要的章节：



通过 “右键” 查看源代码发现，AES-ECB 的加密方式，密钥为 1234123412341234 这个其实非常简单，我们可以很快得到它对应的加密解密代码。

|  |
| --- |
| JavaScript raw = codec.DecodeBase64(`zqBATwKGlf9ObCg8Deimijp+OH1VePy6KkhV1Z4xjiDwOuboF7GPuQBCJKx6o9c7`)~ result = codec.AESECBDecrypt(`1234123412341234`, raw,"")~ dump(result) |

我们把上个数据包的秘文和密码取出来，使用 Yak Runner 写出证明我们解密成功了：



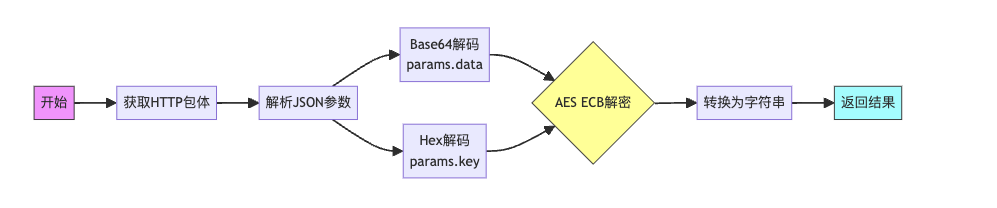
得到这么一段代码之后，再包装出来，包装成一个 Decrypt 函数，要求这个函数可以对整个数据包进行解密，我已经写好了这段代码，大家可以参考一下，主要涉及到数据包取 body 对应字段之后，再解密，假定我们的数据包长这个样子：

|  |
| --- |
| HTTP POST /crypto/js/lib/aes/ecb/handler/sqli/bypass HTTP/1.1 Host: 127.0.0.1:8080 Content-Type: application/json  {  "data": "zqBATwKGlf9ObCg8Deimijp+OH1VePy6KkhV1Z4xjiDwOuboF7GPuQBCJKx6o9c7",  "key": "31323334313233343132333431323334" } |

针对上面这个数据包，我们编写一个函数如下：

|  |
| --- |
| JavaScript decryptData = (packet) => {  body = poc.GetHTTPPacketBody(packet)  params = json.loads(body)  raw = codec.DecodeBase64(params.data)~  key = codec.DecodeHex(params.key)~  result = codec.AESECBDecrypt(key, raw, nil)~  return string(result) } |

在这个函数中，我们经过如下步骤得到了解密的结果：



具体执行起来是什么样？用户可以根据如下代码在自己的 Yak Runner 运行一下这个函数感受一下：

|  |
| --- |
| JavaScript decryptData = (packet) => {  body = poc.GetHTTPPacketBody(packet)  params = json.loads(body)  raw = codec.DecodeBase64(params.data)~  key = codec.DecodeHex(params.key)~  result = codec.AESECBDecrypt(key, raw, nil)~  return string(result) }  packet = <<<TEXT POST /crypto/js/lib/aes/ecb/handler/sqli/bypass HTTP/1.1 Host: 127.0.0.1:8080 Content-Type: application/json  {  "data": "zqBATwKGlf9ObCg8Deimijp+OH1VePy6KkhV1Z4xjiDwOuboF7GPuQBCJKx6o9c7",  "key": "31323334313233343132333431323334" } TEXT result = decryptData(packet) println(result) |



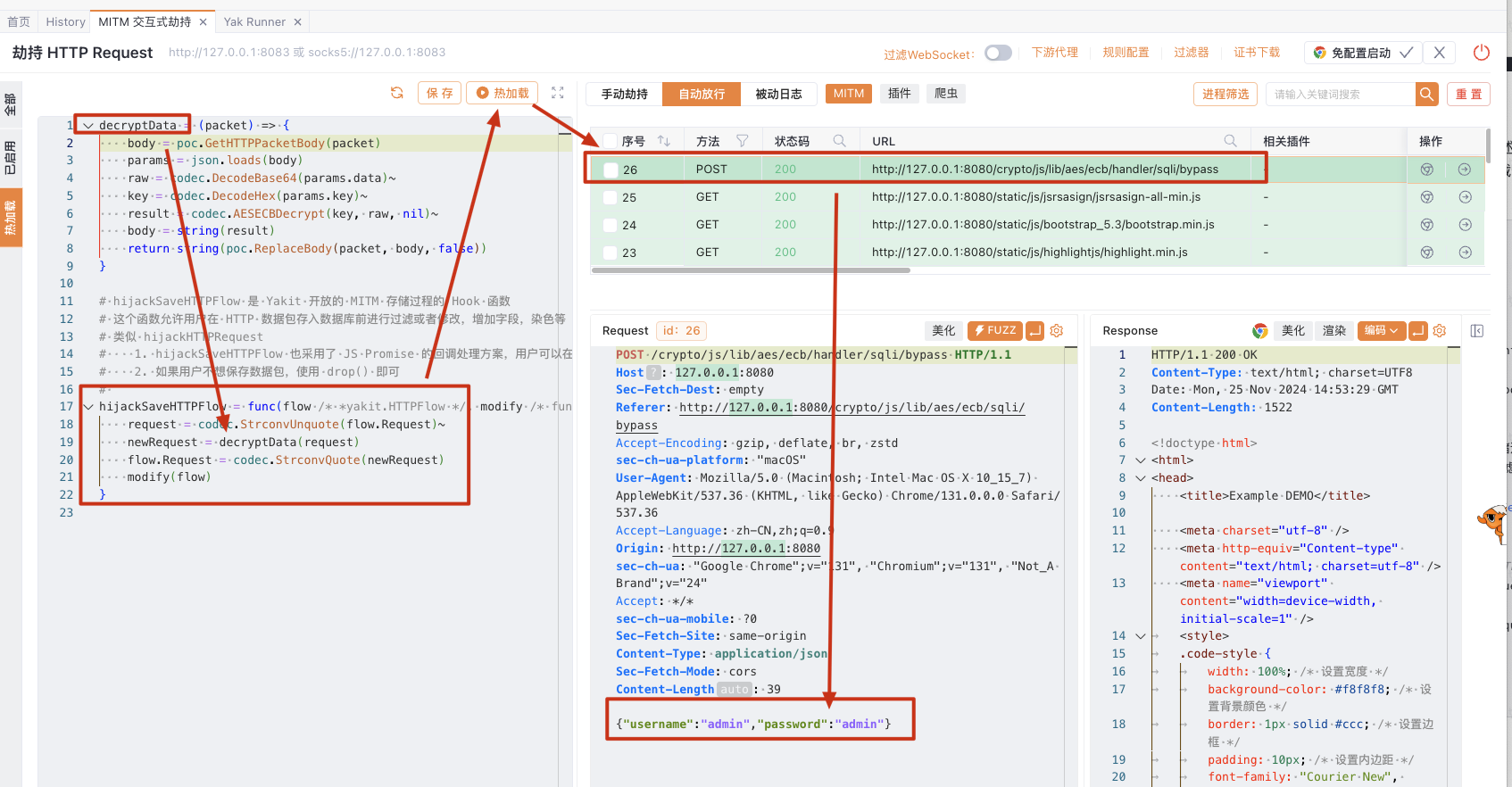
**如何让 MITM 看到请求包明文？**

经过上述简单的分析和实践，我想大家已经知道基本解密上述的代码了，那么可以正式开始我们的第一个任务：让 MITM 看到明文数据包：我们直接把上述的函数复制到热加载代码中，我们再修改一下代码，让 encryptData 直接返回整个数据包，这样就可以直接保存到数据库了。

|  |
| --- |
| JavaScript decryptData = (packet) => {  body = poc.GetHTTPPacketBody(packet)  params = json.loads(body)  raw = codec.DecodeBase64(params.data)~  key = codec.DecodeHex(params.key)~  result = codec.AESECBDecrypt(key, raw, nil)~  body = string(result)  return string(poc.ReplaceBody(packet, body, false)) }  # hijackSaveHTTPFlow 是 Yakit 开放的 MITM 存储过程的 Hook 函数 # 这个函数允许用户在 HTTP 数据包存入数据库前进行过滤或者修改，增加字段，染色等 # 类似 hijackHTTPRequest # 1. hijackSaveHTTPFlow 也采用了 JS Promise 的回调处理方案，用户可以在这个方法体内进行修改，修改完通过 modify(flow) 来进行保存 # 2. 如果用户不想保存数据包，使用 drop() 即可 #  hijackSaveHTTPFlow = func(flow /\* \*yakit.HTTPFlow \*/, modify /\* func(modified \*yakit.HTTPFlow) \*/, drop/\* func() \*/) {  request = codec.StrconvUnquote(flow.Request)~  newRequest = decryptData(request)  flow.Request = codec.StrconvQuote(newRequest)  modify(flow) } |

在这里我们需要使用到 hijackSaveHTTPFlow 这个函数，这个函数可以在数据包进入数据库之前进行一次修改，我们可以在这里解密数据包，保证数据包传入的是正确的。

跟随如下步骤，点击热加载，我们就发现，请求包已经变成了明文：



**如何在 Web Fuzzer 自动加密数据包？**

现在我们在 MITM 中可以看到明文的请求包了，那么如何发送这个数据包，同时让他在发送的时候，自动进行加密？

类似的，我们首先需要准备一下加密数据包的函数：

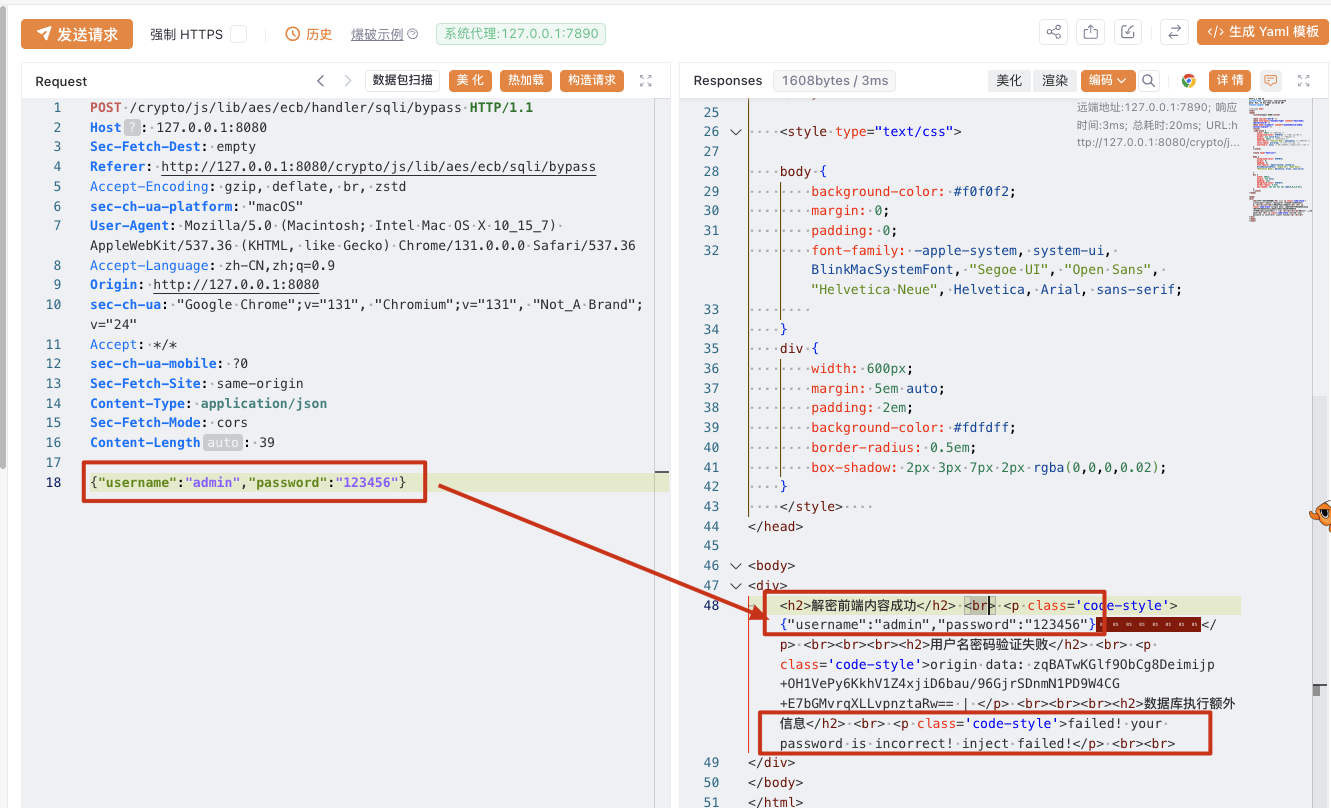
|  |
| --- |
| JavaScript encryptData = (packet, key) => {  body = poc.GetHTTPPacketBody(packet)  result = string(codec.AESECBEncrypt(key, body, nil)~)  data = {  "data": codec.EncodeBase64(result),  "key": codec.EncodeToHex(key),  }  body = json.dumps(data)  return string(poc.ReplaceBody(packet, body /\*type: []byte\*/, false)) } |

我们在这个数据包发送之前，最后进行一步处理即可。

接下来，点开 Web Fuzzer ，把刚才的解密后的数据包放在这里，并且在热加载中处理好相应的代码：



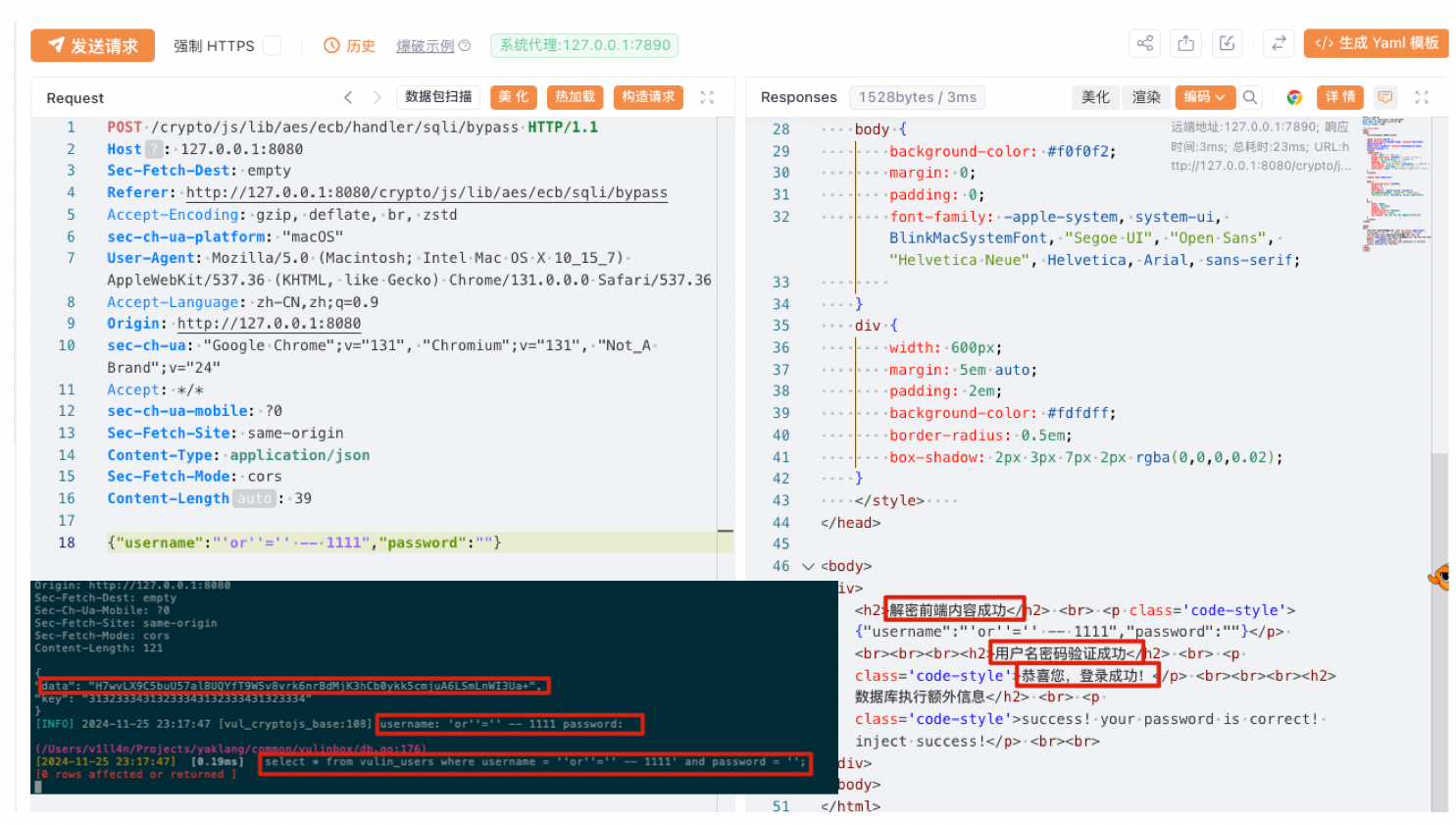
经过上面的处理，我们发送这个数据包将会看到如下结果：



虽然我们解密成功了，但是认证密码却失败了，不过不重要，我们在这个时候已经可以让测试的成本变低了，接下来只需要调整或者爆破就行了。

**SQL注入绕过认证**

虽然这一步是最简单的，但是我们可以把这一步当成是一个胜利的象征：



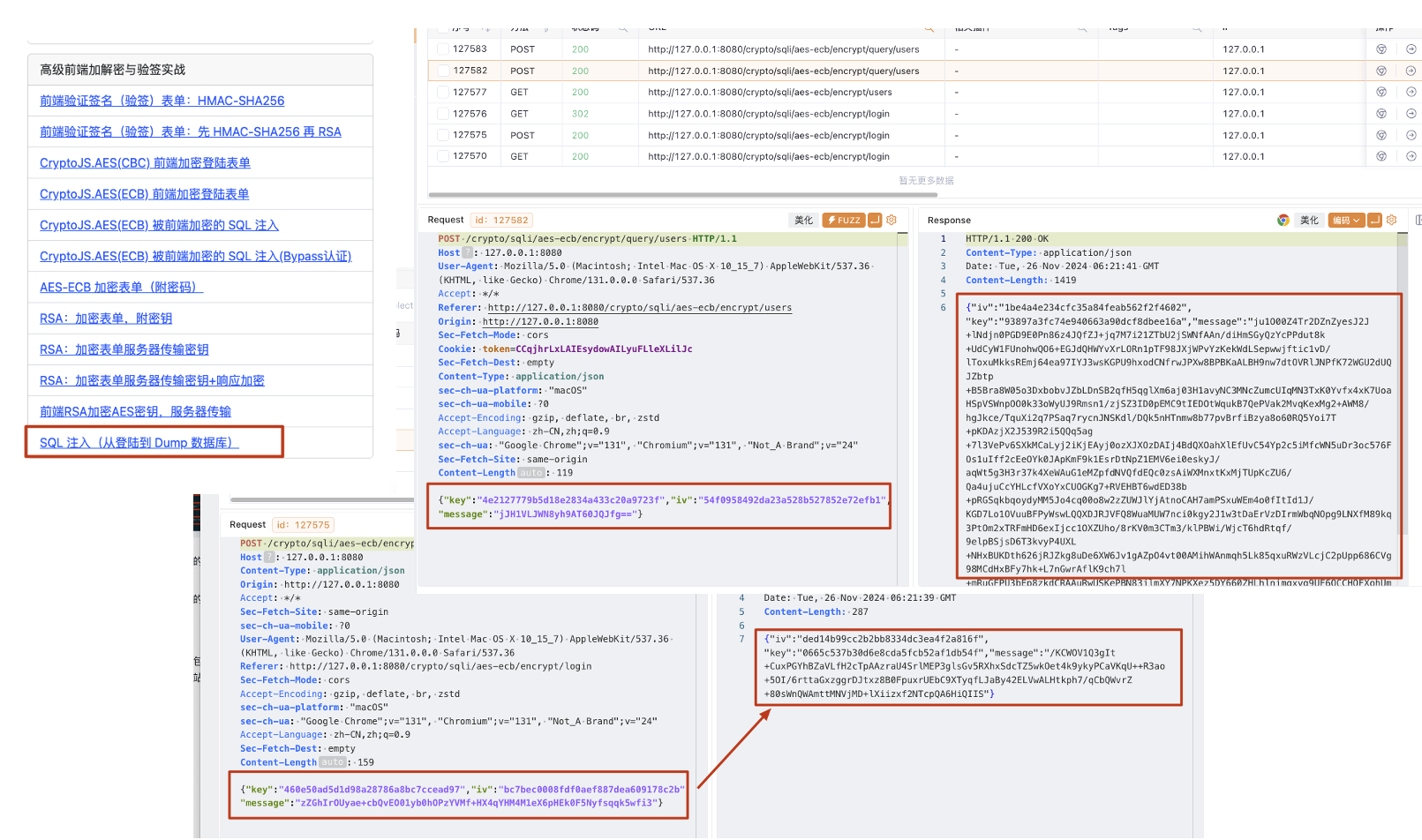
直接发送上述数据包，服务器接收到的核心数据是已经加密后的内容，返回的内容包含 “解密成功” - “密码验证成功”，“登陆成功”。

我们通过热加载主动去修改了数据包的内容，进行了加密，直接绕过了上述加密和解密内容，成功测试了这个漏洞。

**响应也被加密怎么办？**

细心的朋友发现，我们上面这个数据包只是加密了请求。当然为了方便做教程，我们只写了提交请求的部分加密流程。但是实际上，我们往往会遇到全站加密的问题：除了静态资源之外，几乎所有的数据传输都经过了加密。

例如我们下面这个例子：



我们发现，数据包中请求中包含 key, iv 和 message 三个字段，响应包中也包含着三个字段，这给我们的测试造成了巨大的障碍，甚至重放数据包都有点费劲。那么我们应该怎么处理这种问题呢？

**Web Fuzzer 全自动加密解密**

根据我们前面讲到的一些基本手法，大概看一下解密过程：



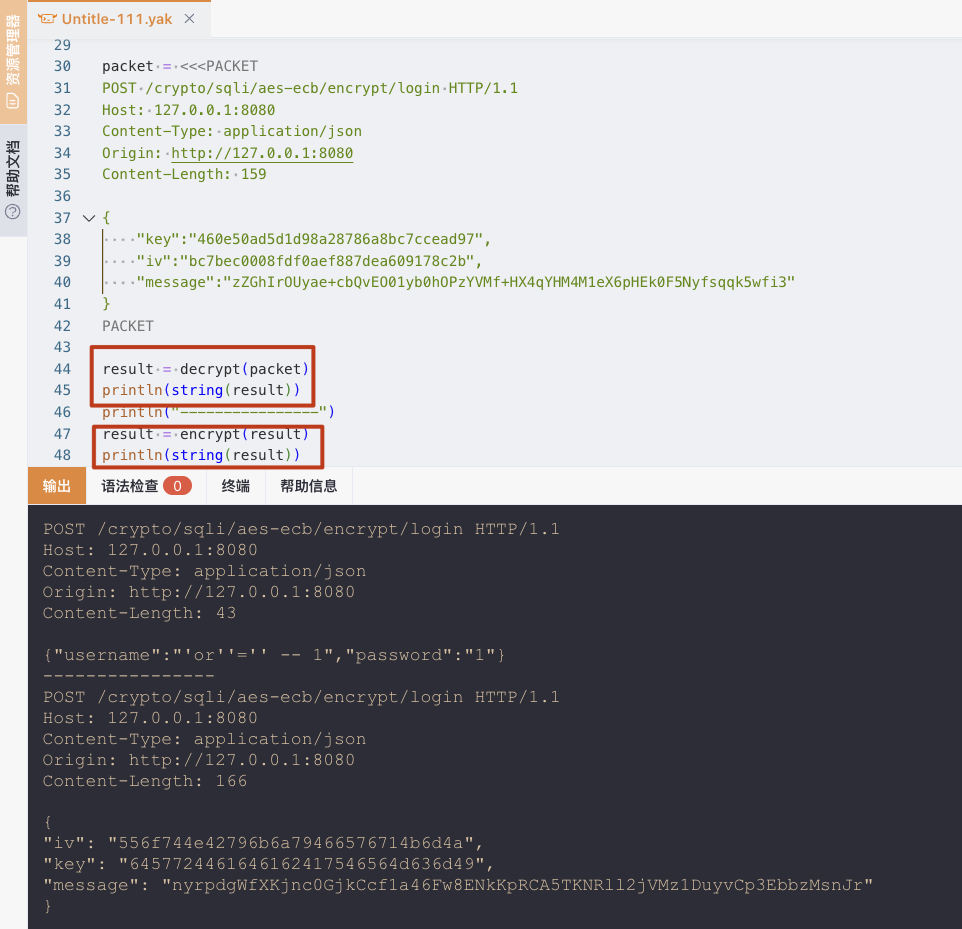
随机 key 和随机初始偏移量，AES CBC 加密，Pkcs7Padding，并且我们发现数据包内已经带上了 iv 和 key 的 hex 编码后的内容，类似如下的格式：

|  |
| --- |
| JavaScript **POST** /crypto/sqli/aes-ecb/encrypt/login **HTTP/1.1** **Host**: 127.0.0.1:8080 **Content-Type**: **application/json** **Origin**: http://127.0.0.1:8080 **Content-Length**: 159  {  **"key"**:**"460e50ad5d1d98a28786a8bc7ccead97"**,  **"iv"**:**"bc7bec0008fdf0aef887dea609178c2b"**,  **"message"**:**"zZGhIrOUyae+cbQvEO01yb0hOPzYVMf+HX4qYHM4M1eX6pHEk0F5Nyfsqqk5wfi3"** } |

其对应的 Yaklang 的核心加密解密代码应该如下：

|  |
| --- |
| JavaScript decrypt = packet => {  body = poc.GetHTTPPacketBody(packet)  obj = json.loads(body)  if "iv" in obj && "key" in obj && "message" in obj {  iv = codec.DecodeHex(obj.iv)~  key = codec.DecodeHex(obj.key)~  msg = codec.DecodeBase64(obj.message)~  newBody = string(codec.AESCBCDecrypt(key, msg, iv)~)  return poc.ReplaceBody(packet, newBody, false)  }  return packet }  encrypt = packet => {  body = poc.GetHTTPPacketBody(packet)  iv = randstr(16)  key = randstr(16)  msg = string(body)  enc := codec.AESCBCEncryptWithPKCS7Padding(key, msg, iv /\*type: []byte\*/)~  newBodyObj = {  "iv": codec.EncodeToHex(iv),  "key": codec.EncodeToHex(key),  "message": codec.EncodeBase64(enc),  }  newBody = json.dumps(newBodyObj)  packet = poc.ReplaceHTTPPacketBody(packet /\*type: []byte\*/, newBody)  return packet } |

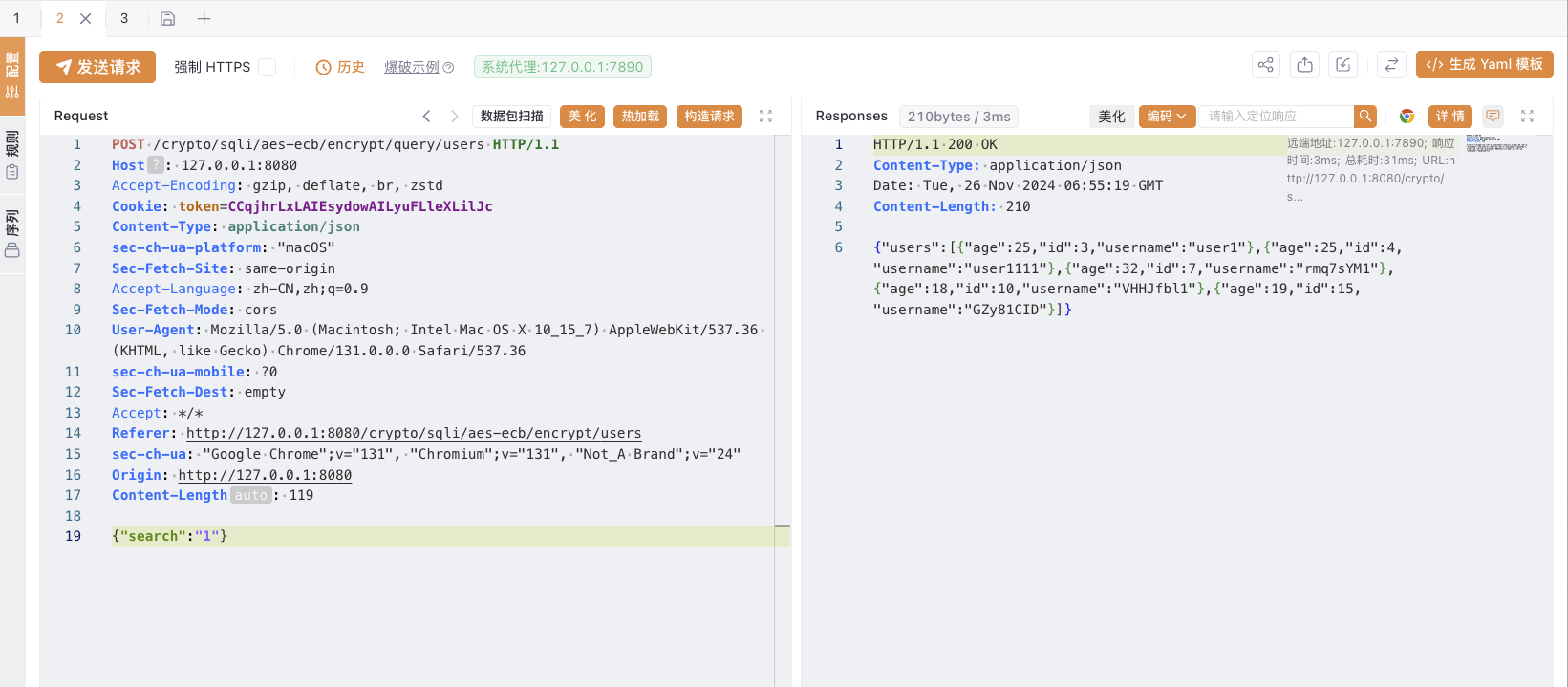
当我们写出这两个函数之后，可以快速验证一下函数写的对不对，可以接下来执行下面的代码快速验证：



直接调用我们发现解密和加密都看起来比较正常，那么就可以直接在热加载中使用这一对儿函数了：



我们直接在 beforeRequest 和 afterRequest 直接使用我们的加密解密函数，这样就可以直接得到如下效果：



我们直接使用明文请求 {"search": "1"} 就可以发送成功，并且请求包自动被替换成了明文。

我们通过使用 beforeRequest 和 afterRequest 两个魔术方法，直接可以让测试人员看到明文，隐藏掉加密解密的逻辑和过程。

**MITM 自动保存明文数据包**

这个我们测试成功 Web Fuzzer 之后，想在不影响数据包交互的情况下，自动把解密后数据存储到数据库？

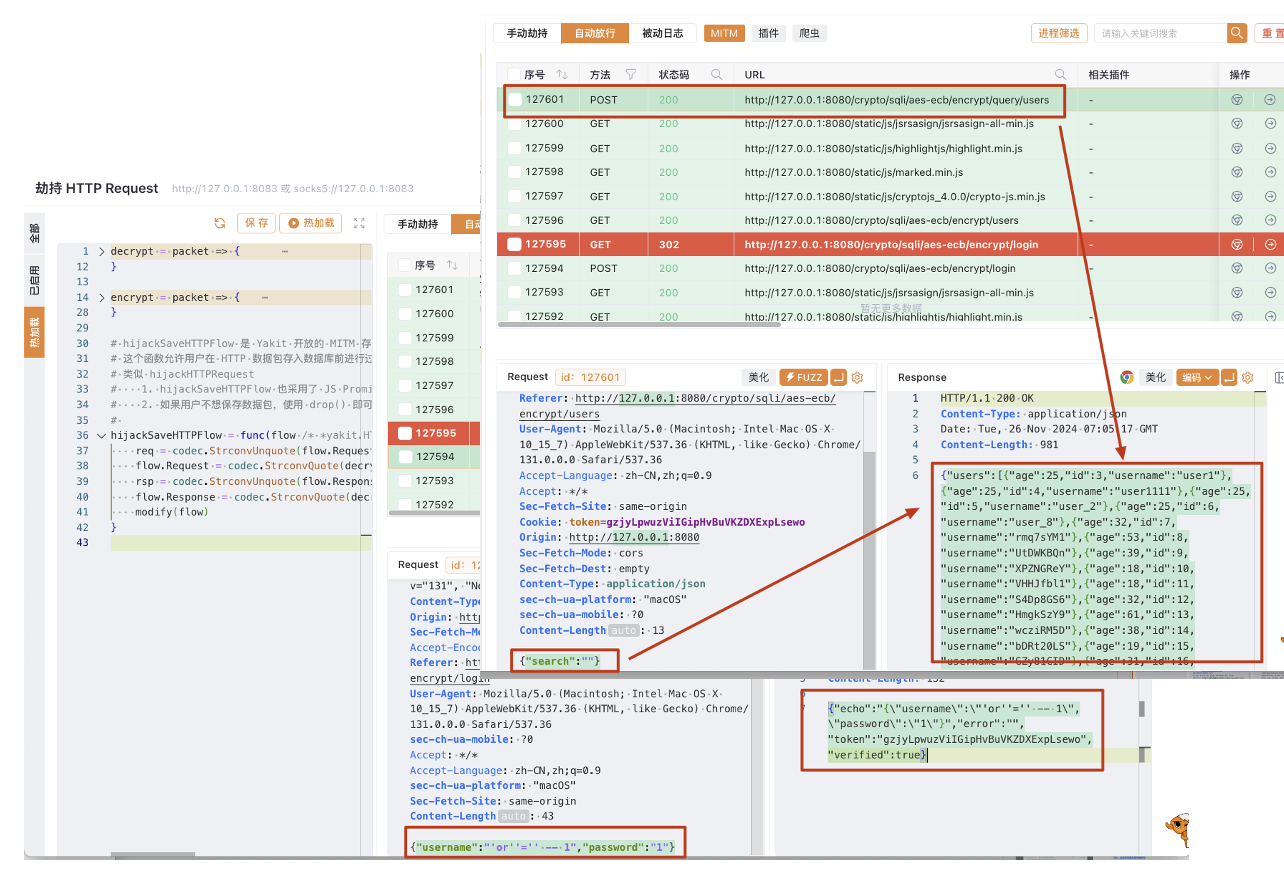
那自然也应该去对热加载进行一些修改，和 Web Fuzzer 热加载十分类似，同样的，我们也在最一开始的案例中，写过类似的代码：



我们复制上加密解密函数之后，直接使用下面的代码：

|  |
| --- |
| JavaScript hijackSaveHTTPFlow = func(flow /\* \*yakit.HTTPFlow \*/, modify /\* func(modified \*yakit.HTTPFlow) \*/, drop/\* func() \*/) {  req = codec.StrconvUnquote(flow.Request)~  flow.Request = codec.StrconvQuote(decrypt(req))  rsp = codec.StrconvUnquote(flow.Response)~  flow.Response = codec.StrconvQuote(decrypt(rsp))  modify(flow) } |

随后点击热加载按钮，然后过流量：



我们发现和一开始的流量有着显著区别，iv, key 和 message 都没了，直接变成了大家喜闻乐见的明文。

这样我们就可以直接把 MITM 的数据包发送到 Web Fuzzer，直接修改明文数据，通过 Web Fuzzer 热加载去加密数据包发送，并且保证展示也是被解密的。

**总结**

本文介绍了两个更贴近实际的靶场：

1. 被加密了请求的 SQL 注入（用以学习基本工具使用）
2. 请求和响应都被加密的场景（增加熟练度）

当然，这个场景并不是 MITM 的全部用法，实际上如果你有其他工具可以测试漏洞，但是无法适配加密套件，Yakit MITM 还可以直接充当加密套件来辅助用户的其他工具测试，等有机会的话，我们再来介绍后续的场景。