**渗透测试高级技巧（二）：对抗前端动态密钥与非对称加密防护**

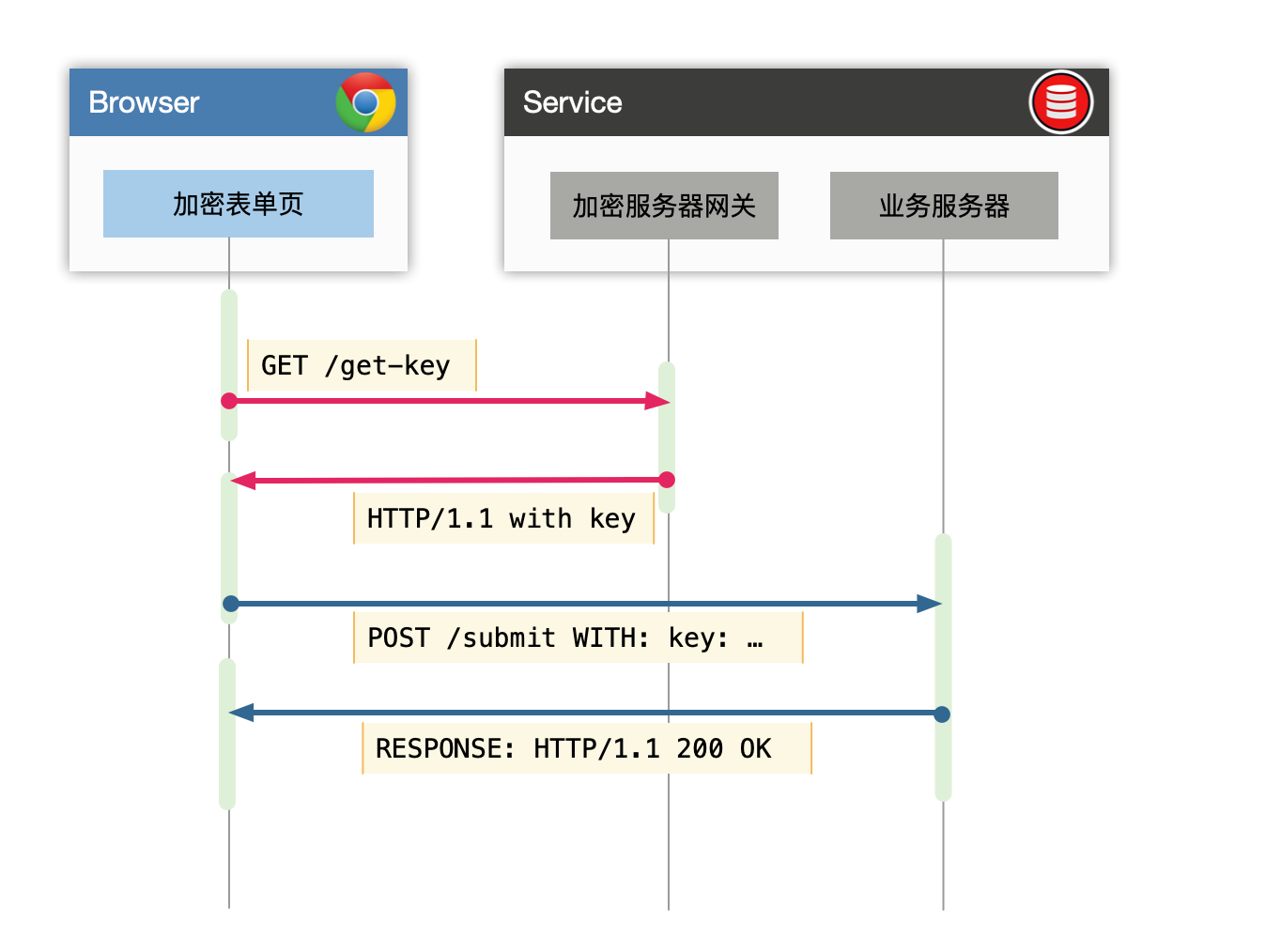
在前文的技术分享中，我们描述了验签和静态对称加密（静态密钥 AES）的常见场景，大家我相信遇到类似的加解密清醒，基本都可以通过热加载的基本使用获得破解前端加密解密的方法，达到一个比较好的测试状态。

在本文中，我们在保持同样的通用适配度的同时，将会来接触更加复杂的前端加密与解密场景：

1. 动态密钥：无法通过简单的阅读 JavaScript 直接获取加密密钥
2. 非对称加密技术：使用 RSA / SM2 技术来加密通信数据以避免明文传输

**案例：获取密钥和提交表单不在同一个请求中**

这个场景可能初步听起来略微抽象，但是实际上却非常常见，我们使用基础的时序图将会很容易理解这个案例：我们首先请求 /get-key 接口获取一个密钥，然后开始用获取到的 KEY 加密表单，在 POST 中提交加密后的数据（可能并不携带密钥），服务器端可以通过服务器的密钥来解密数据



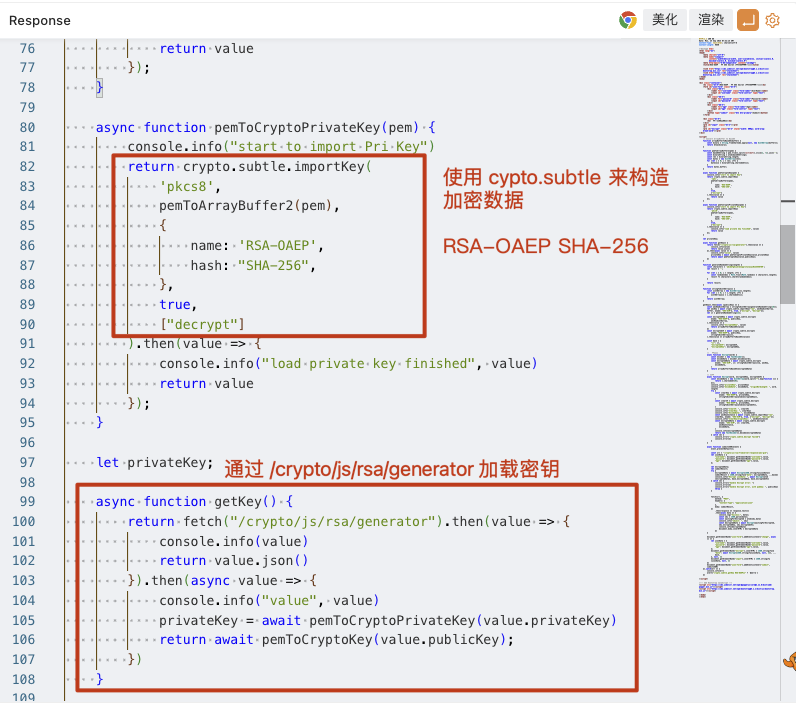
一般来说，这种加密方式并不是一个非常冷门的方式，反而很多网站会使用这种技术来防护自己的登录信息不会泄漏。为此，我们基于实际网站的前端加密混淆方式构建了一个非常经典的案例，来帮助大家掌握这些测试技术。我们在 Vulinbox 中可以获取到关于这个部分的靶场测试环境：



首先打开网站，抓包看一下通信流程发现网站请求完资源后会去加载一套证书：



因此我们需要去寻找 JavaScript 中的加密线索：



那么基于此，加密的基本脉络就很清楚了，我们需要先发一个请求去服务端获取私钥用来解密签名的数据（来自于服务器），获取公钥来签名需要发送到后端的数据。但是实际上这还不够，我们知道有一些基本资料：

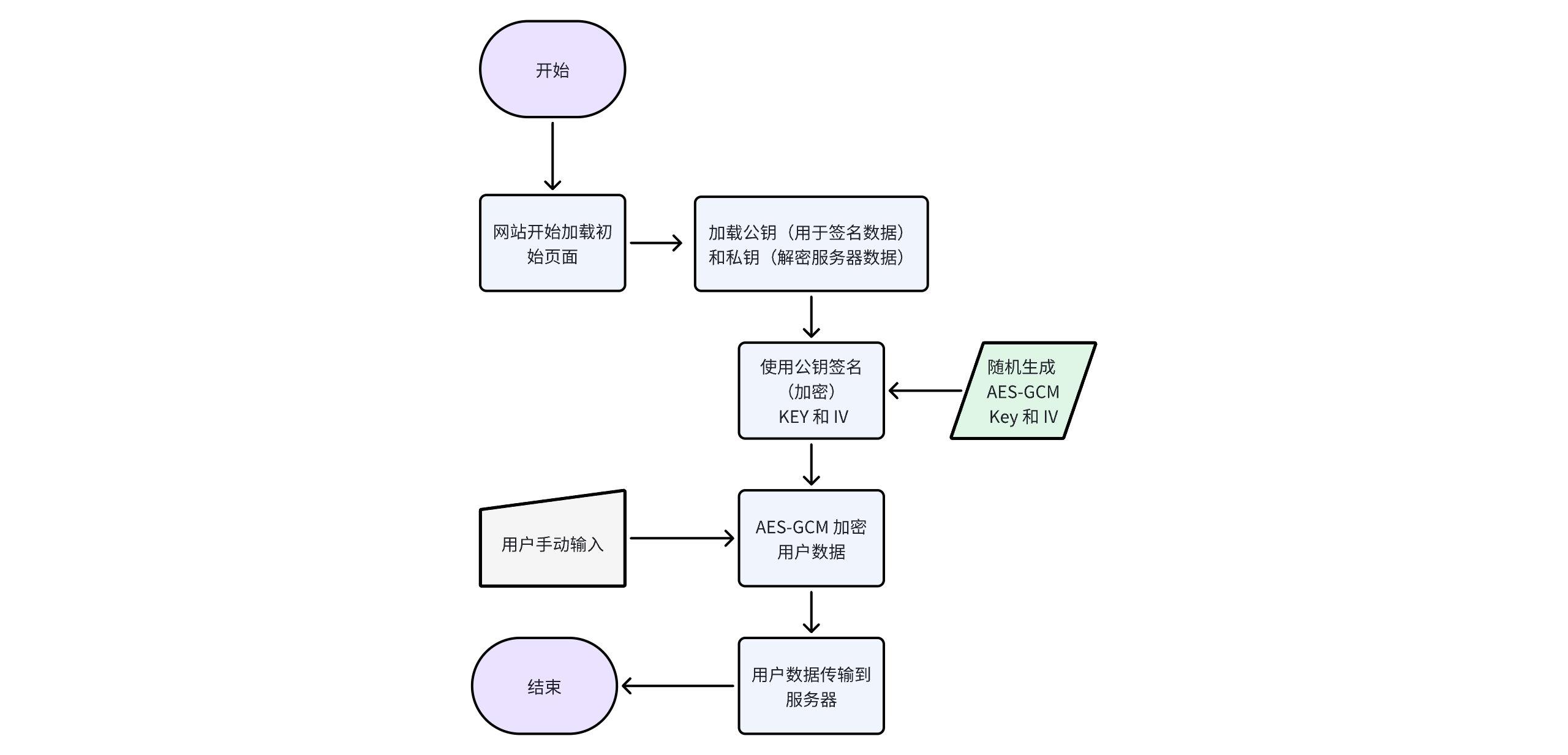
|  |
| --- |
| RSA-OAEP的最大加密长度取决于RSA的密钥长度和OAEP填充的长度。OAEP填充需要额外的空间，通常是两倍的哈希长度加上2（对于SHA-1和SHA-256，这将分别是42和66字节）。因此，对于一个n位的RSA密钥，最大的明文长度将是n/8减去OAEP填充的长度。 |

我们并不能直接使用 OAEP 去加密大段的数据，那这种情况下，我们知道 AES KEY 一般来说并不会太大，所以使用“混合加密（RSA-OAEP + AES）” 实际上是比较好的解决方案，否则我们就需要把大量数据进行切片传输，这实际上非常不符合常理。

那么我们继续从 JavaScript 中寻找线索：



**捋清思路，准备动手**



在这个超复杂情况中，我们发现实际执行的时候，需要两个请求，这个操作恰好符合我们 Web Fuzzer 的请求序列的定义。

1. 请求一：先执行 GET /crypto/js/rsa/generator 获取签名验证用的 RSA 密钥
2. 使用数据提取功能，提取请求一中的公钥的 PublicKey，用来加密 KEY 和 IV
3. 使用 AES 加密数据，并把加密后的 KEY 和 IV 一起发送到服务器

实际我们发现，这个结果处理起来也并不复杂，随机生成 KEY 和 IV 可以写死，这样用它加密真实数据即可，这个过程和我们在《渗透测试高级技巧（一）》中的操作非常类似，我们可以在热加载中很容易处理。

实际对我们来说，最大的工作量是把 KEY 和 IV 使用 RSA-OAEP 加密好，传输给服务器。这个步骤在 yaklang 中可以轻易使用 codec.RSAEncryptWithOAEP(pemPublic, data) 这个函数做到，因此我们的 Yaklang 热加载代码可以写成：

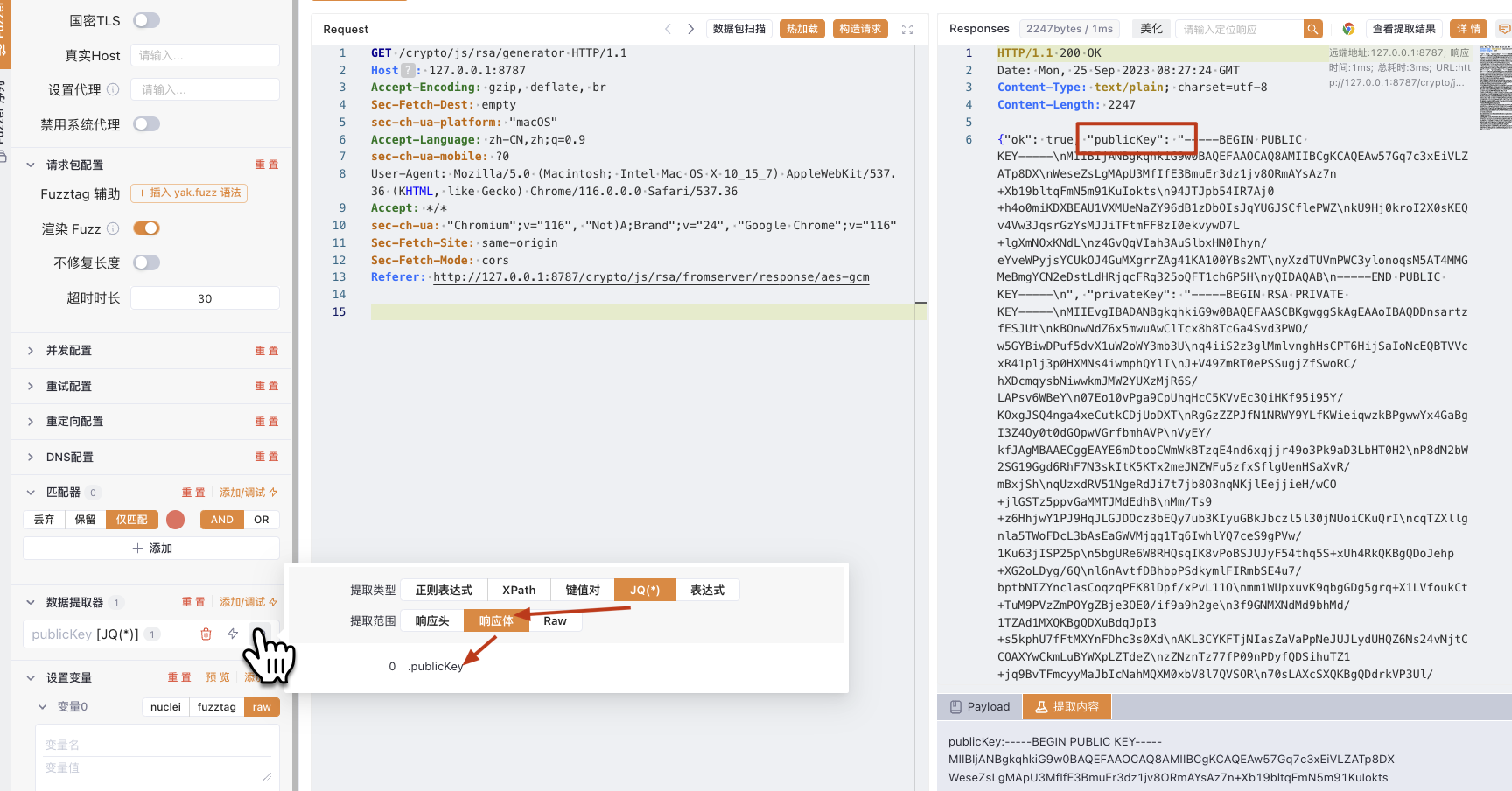
|  |
| --- |
| JavaScript aesKey = "aaaaaaaaaaaaaaaa" iv = "aaaaaaaaaaaa" aesGCM = data => {  return codec.EncodeBase64(codec.AESGCMEncryptWithNonceSize12(aesKey, data, iv)~) } rsaOAEP\_key = (pem) => {  return codec.EncodeBase64(codec.RSAEncryptWithOAEP(pem, aesKey)~) } rsaOAEP\_iv = (pem) => {  return codec.EncodeBase64(codec.RSAEncryptWithOAEP(pem, iv)~) } |

对应的，我们的使用标签函数就可以是 {{yak(rsaOAEP\_key|{{params(publicKey)}})和{{yak(rsaOAEP\_iv|{{params(publicKey)}})}}，如果使用加密数据，而且我们的 AES KEY 是硬编码的，可以直接写成 {{yak(aesGCM|...)}}`

到现在，基本核心功能有了非常好的实现，我们可以设置热加载代码并且完成我们的 Web Fuzzer 序列了。

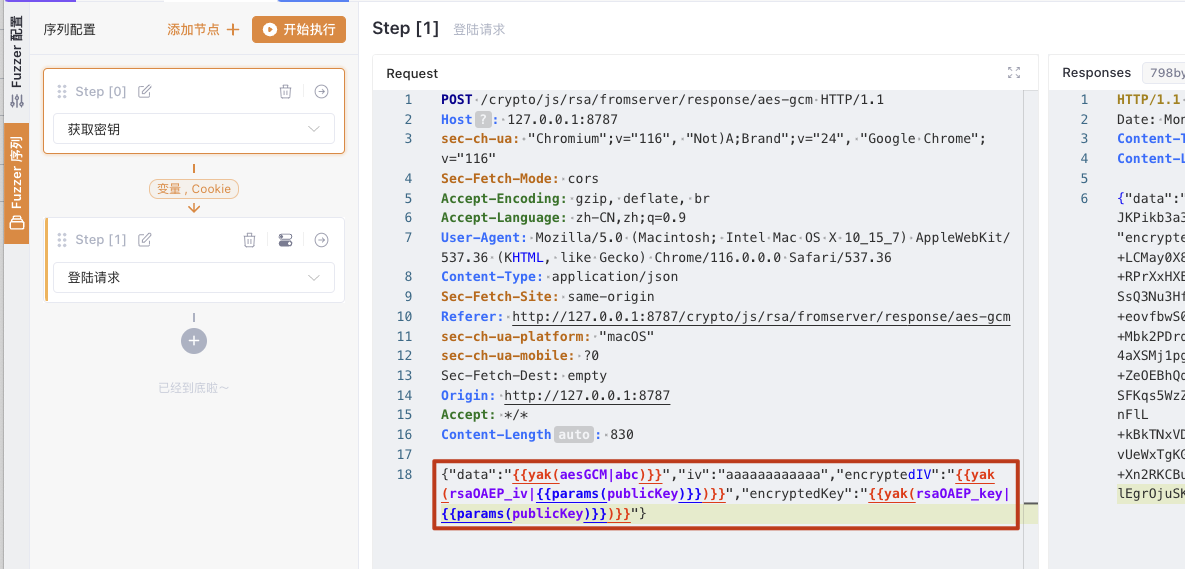
首先我们在序列一中设置好数据提取器，数据提取器来提取服务端的 publicKey

**请求一：获取签名（加密）用的公钥**



我们在第一个请求中，使用“数据提取器”添加一个配置项目，设置 JQ(\*) 提取数据类型，范围设置成“响应体”，把提取字段名设置为 publicKey，这样我们在后续的请求中，可以用 {{param(publicKey)}} 安全引用上一个请求中提取到的变量。

**请求二：继承请求一中的密钥，加密请求中参数**



我们在实际的发送的时候，发现 AES 密钥和 IV 可以硬编码，因此根据原数据包，我们把 JSON 中的关键字符串使用热加载 fuzztag 替换掉，形成如下效果

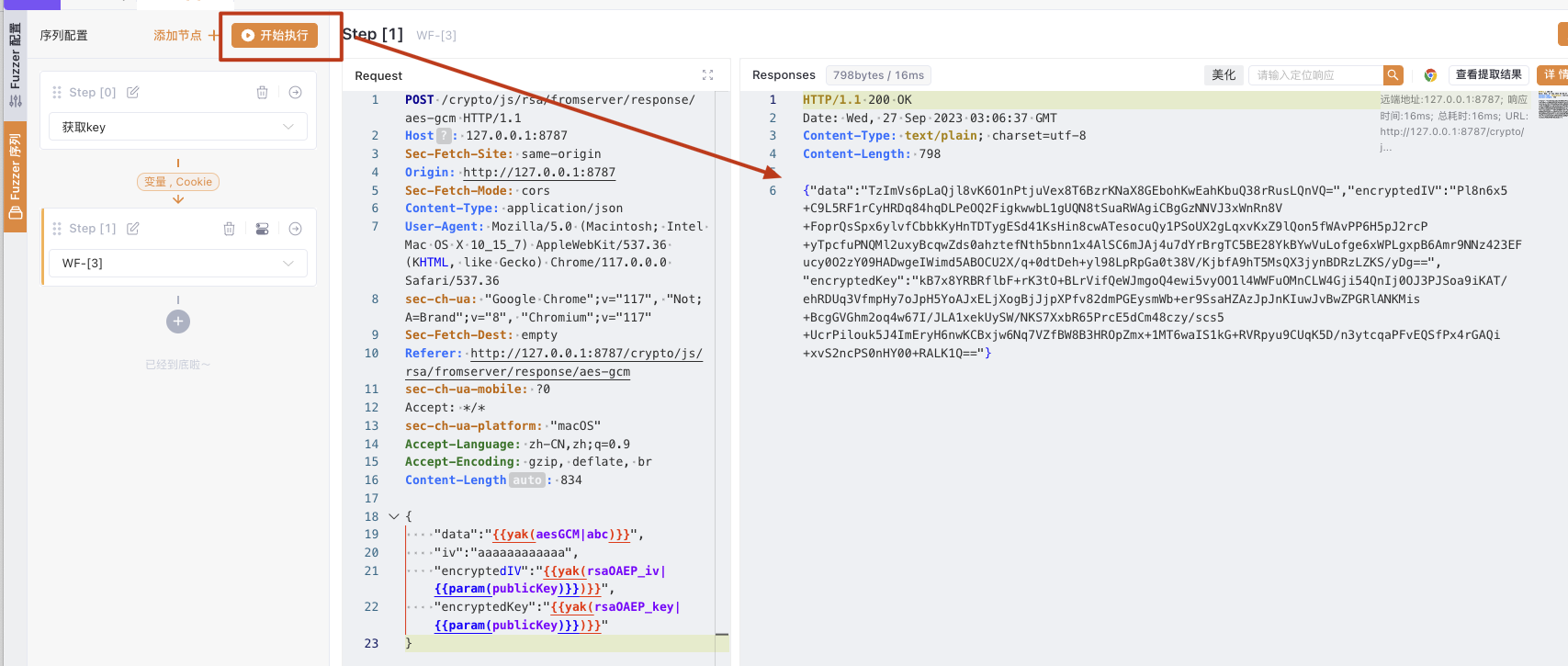
|  |
| --- |
| JavaScript {  "data":"{{yak(aesGCM|abc)}}",  "iv":"aaaaaaaaaaaa",  "encryptedIV":"{{yak(rsaOAEP\_iv|{{param(publicKey)}})}}",  "encryptedKey":"{{yak(rsaOAEP\_key|{{param(publicKey)}})}}" } |

上述 Payload 中对应的热加载代码为：

|  |
| --- |
| JavaScript aesKey = "aaaaaaaaaaaaaaaa" iv = "aaaaaaaaaaaa" aesGCM = data => {  return codec.EncodeBase64(codec.AESGCMEncryptWithNonceSize12(aesKey, data, iv)~) } rsaOAEP\_key = (pem) => {  return codec.EncodeBase64(codec.RSAEncryptWithOAEP(pem, aesKey)~) } rsaOAEP\_iv = (pem) => {  return codec.EncodeBase64(codec.RSAEncryptWithOAEP(pem, iv)~) } |

实际上就是把加密内容进行 Base64 写回原来数据包，并且 aes 和 iv 为了方便测试，已经写好了！

**Wait a Minute，事情还没结束！**



经过上述的操作，我们请求确实发送出去了，并且得到了正确的回应，但是我们似乎遇到了更大的问题：我们居然没有办法直接根据响应内容判断我们到底登陆成功了没有。

那么如何判断登陆成功了？我们发现服务端返回的信息中，包含了 encryptedIV 还有 encryptedKey 还有 data，大致直接可以推断出，返回的信息也被加密了。这也是一个非常棘手的问题

我们想要解决这种问题，那就需要找一个地方可以获取到相应的信息，并且可以加密解密。这里就需要进入我们本地技巧集的第二部分：

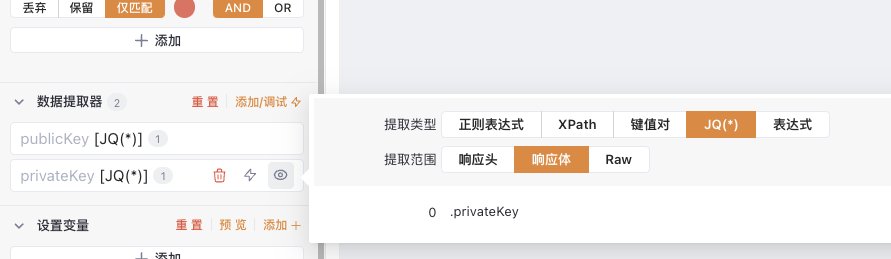
**难点：解决 HTTP 响应加密问题**

* 思路一：继续利用 Web Fuzzer 序列，添加第三个请求，处理第二个请求中响应的信息。
* 思路二：通过热加载 afterRequest 或 mirrorHTTPFlow 来编写代码解密响应中的信息

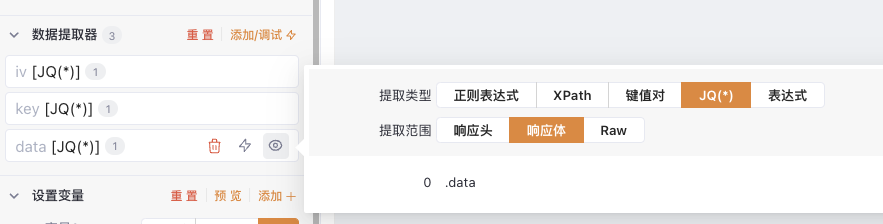
我们有了思路之后，可以尝试动手来做

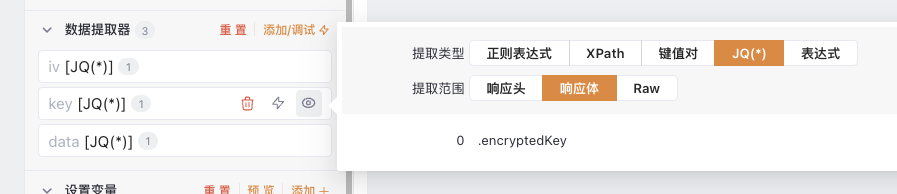
**使用 Fuzzer 序列来解密响应**

类似 publicKey 的使用，我们可以在“请求一”中添加 privateKey 的提取配置，使用 .privateKey JQ 来提取初始化中的响应信息



在第二个响应中，我们需要获取到响应的加密信息来解密响应中的数据，从而判断请求提交的真实结果，因此我们需要在请求二中设置提取 encryptedIV / encryptedKey / data 三个变量，通过数据提取器的配置：



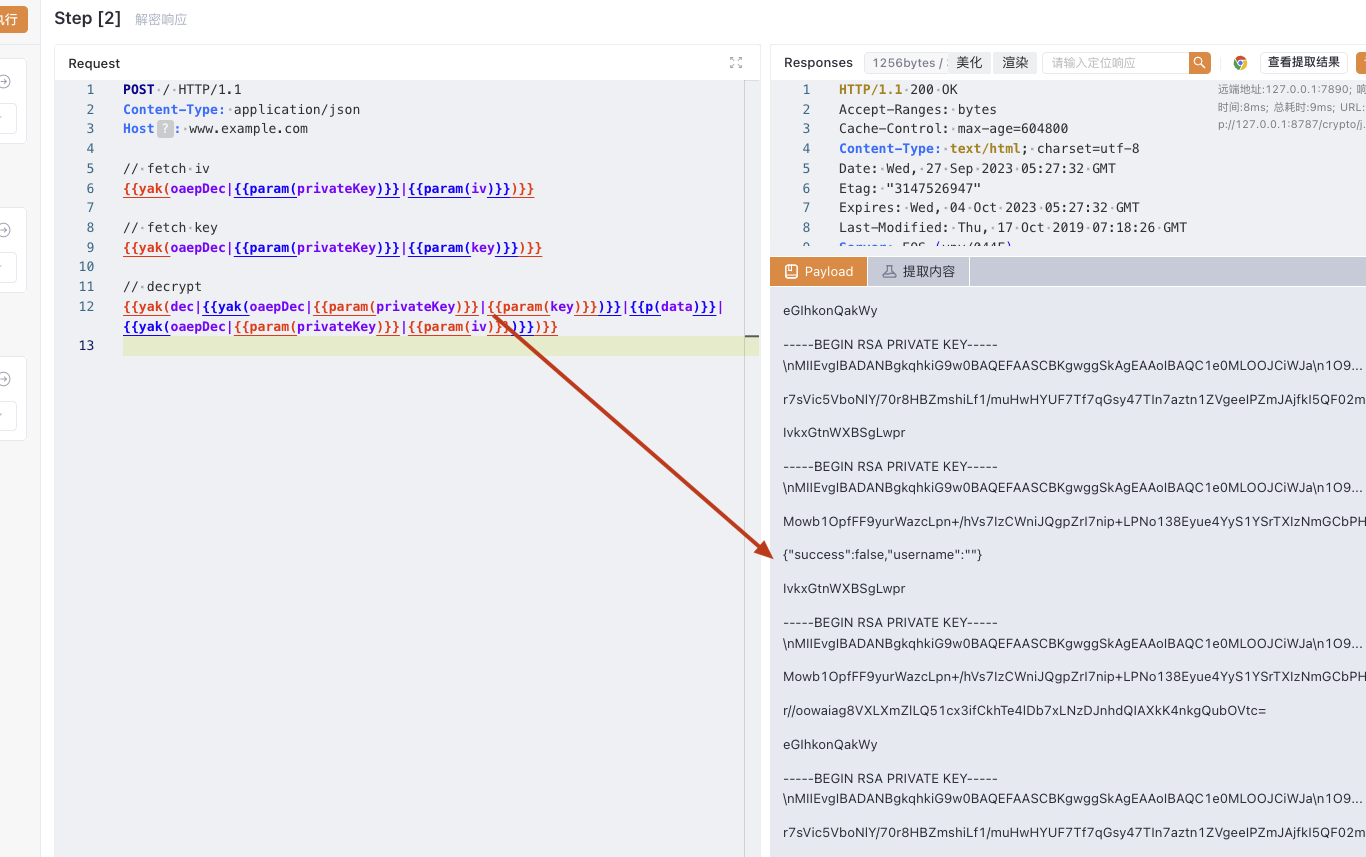


可以很容易配置成功，接下来，通过第三个请求中使用热加载标签，执行这两个函数即可马上解密 RSA-OAEP 的内容。

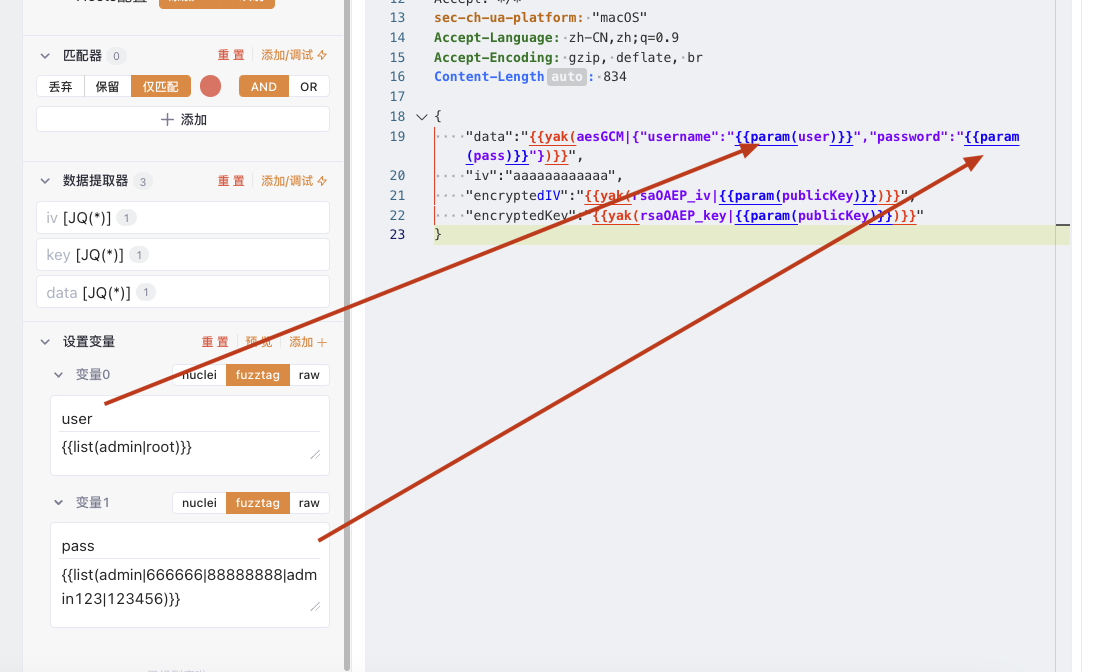
|  |
| --- |
| JavaScript oaepDec = (pem,data) => {  return string(codec.RSADecryptWithOAEP(pem, codec.DecodeBase64(data)~)~) } dec = (key,data,iv) => {  dump(key, data, iv)  return string(codec.AESGCMDecryptWithNonceSize12(key, codec.DecodeBase64(data)~, iv)~) } |

相对来说，oaepDec 直接使用 PEM 和 Data 可以解密 KEY，和 IV

然后再使用 key 和 iv 去解密 data 中的内容最终就可以实现对响应的解密

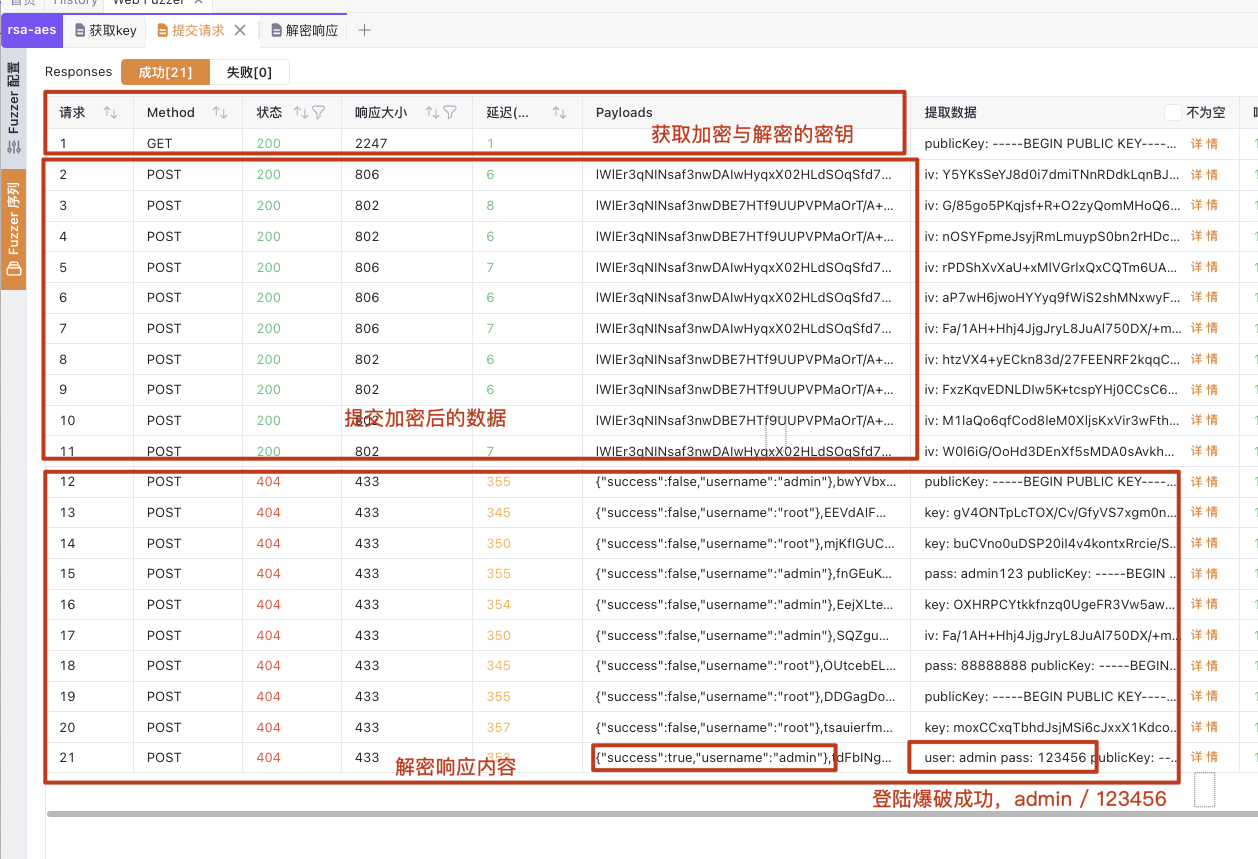


可以，非常棒，接下来我们就可以直接修改用户名和密码来进行爆破了对吧？利用这种特性，我们在第二个请求中，需要对用户名和密码进行爆破：



在第二个请求的 web fuzzer 配置中，增加设置变量 user 和 pass，然后这两个变量实际上数据被加密，并且能把 encryptedIV 和 encryptedKey 设置正确。

我们配置 user 为 admin|root 配置 pass 为 admin|666666|88888888|admin123|123456，实现了用户名和密码的爆破，总共应该有 2 \* 5 个结果，点击序列执行查看效果：



通过这种操作，我们实现了简单的“爆破功能”。真实情况是，这个场景已经非常复杂了，基本上属于“最严密防护”的那一类，在这种严密防护下，实际开发的过程也会非常艰辛，当然我们也很难实际真的遇到这种情况。但是如果这种情况我们可以解决，绝大多数弱于这种防护的策略我们都可以轻松突破。

**优化流程：更好地处理响应中加密的部分**

很多时候，响应的数据其实我们需要提炼一下信息或者认真处理一下加密的信息，不能总是使用序列来解决吧？那我们如何在一个 web fuzzer 流程中，处理好响应中的加密数据呢？其实并不是不可以解决的

Yaklang 的最新版本 1.2.7 及以上的版本，设置了一个热加载中的新 hook 函数帮助大家解决：

mirrorHTTPFlow 可以给大家提供一种 “编程提炼 Web Fuzzer 请求和响应数据” 的新路径：

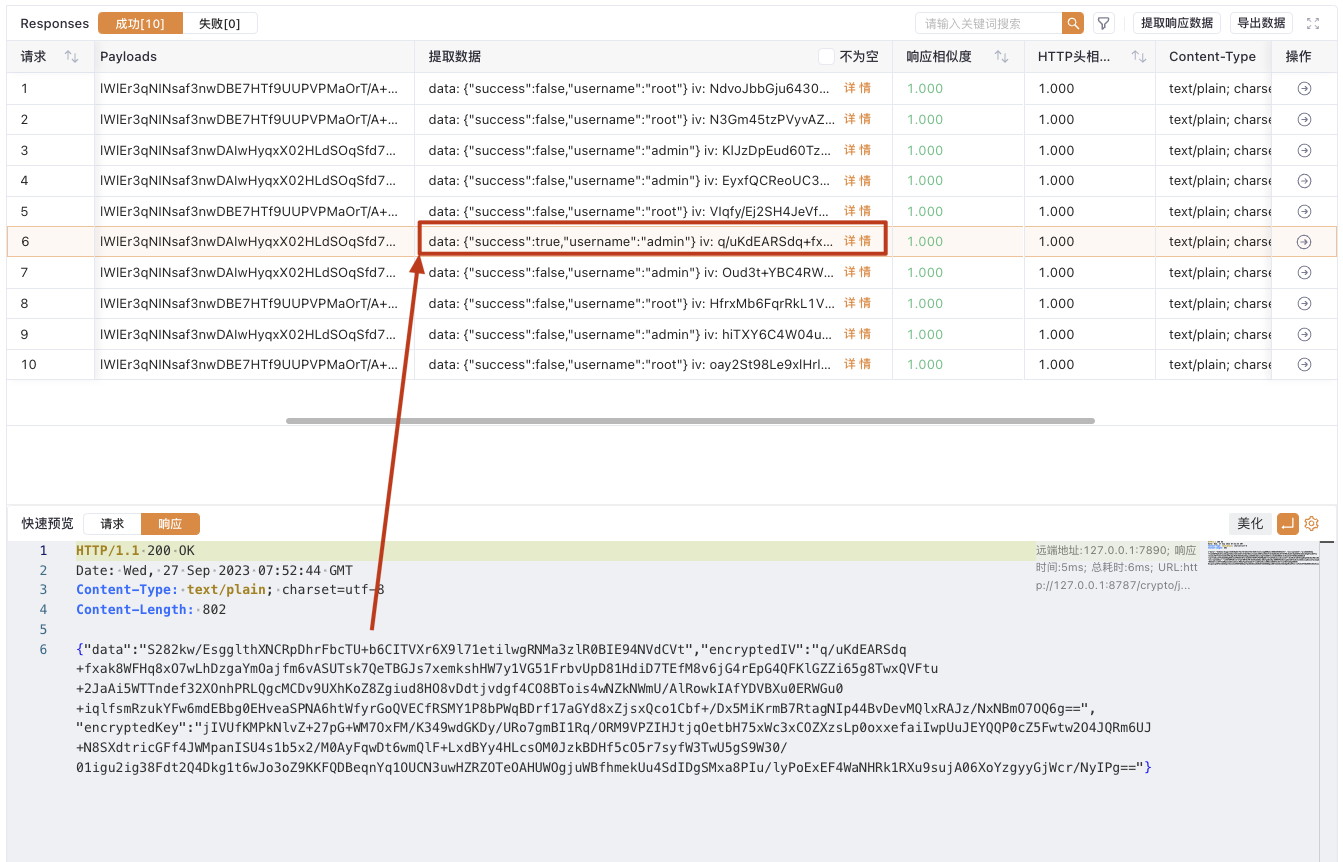
具体定义如下：

|  |
| --- |
| JavaScript mirrorHTTPFlow = (reqBytes, rspBytes, params) => {  return {"key": "value"} }  // 使用案例如下 // 如果响应中包含 “Success”，就在提取的变量中输出一个“登陆”：“成功”的变量 mirrorHTTPFlow = (req, rsp, params) => {  data = {}  if string(rsp).Contains("Success") {  data["登陆"] = "成功"  }   return data } |

我们可以在热加载中添加这个函数，来获取上下文，甚至执行对请求响应的解密。

所以我们可以整理一下，在上一个解决方案中，还可以做的更简单，直接使用热加载调制一个函数来实现加密解密。

|  |
| --- |
| JavaScript  mirrorHTTPFlow = (req, rsp, params) => {  // 获取私钥以解密响应数据  pem = params.privateKey    // 切割响应中的数据，作为 JSON 加载  \_, body = poc.Split(rsp)  body = json.loads(body)    // 获取响应中的加密部分，data 为密文  // IV 和 KEY 分别是 AES-GCM 的加密密钥和 IV  data = body.data  ivEnc = body.encryptedIV  keyEnc = body.encryptedKey    obj = {}  // 使用 RSA-OAEP 解密 IV 和 KEY  iv = codec.RSADecryptWithOAEP(pem, codec.DecodeBase64(ivEnc)~)~  key = codec.RSADecryptWithOAEP(pem, codec.DecodeBase64(keyEnc)~)~    // 使用 AES-GCM 解密  obj["data"] = codec.AESGCMDecryptWithNonceSize12(key, codec.DecodeBase64(data)~, iv)~  return obj } |



通过我们新增加的 mirrorHTTPFlow 函数，我们同时处理请求，响应和序列变量三个部分，实现了对流量的解密，并且很方便的识别出我们的爆破到底成功没有。

**总结**

在本篇“高级技巧”中，我们明显和《渗透测试高级技巧（一）》中的验签有了非常明显的区分，我们大量使用了 Web Fuzzer 序列来实现有上下文关联的部分。并且在这个案例中，因为响应的加密，导致区分“爆破”结果变得十分困难，我们也给出了一些处理技巧和实际方案。

当然本篇内容的靶场和使用案例也并不是空想，而是来源于真实案例的总结和抽象。在实际使用中，加密算法和接口名称与当前案例略有差别，但是笔者相信，在完整实现过这个流程之后，你一定已经对 Web Fuzzer 序列的使用更加得心应手了。