游戏抓包工具制作（五）—— 伪造和发包

原创 阿宽 游戏小测试 2024-04-08 19:07 广东

最近这段时间有点忙，距离上一次发文章差不多快过去一个月了，实在抽不出身

今天整理一下如何伪造客户端的协议，主要讲三种方式，分别是**复制粘贴法，参数生成法**，以及**代理接口法**。不过由于直接打印的bytes字节流的文本中可能存在部分转义，且直接阅读不是那么美观，我们先将bytes字节流换成WPE那种数字和大写字母的十六进制显示形式，便于阅读和分析。

准备工作：编写一个bytes字节流转换方法

```python
def bytes_to_str(data):
    """
    将bytes字节流转化成WPE显示样式的字符串
    :param data:
    :return:
    """
    # 首先将字节流转化成十六进制ASCII表示形式
    base = binascii.b2a_hex(data)
    # 然后将其转化成字符串，因为转化后是b'xxxxxxxxxx'的格式，所以要去掉前2后1个字符
    base_str = str(base)[2:-1]
    # 将字符串设置成WPE的格式，并转成大写
    return re.sub(r"(?<=\w)(?=(?:\w\w)+$)", " ", base_str.upper())
```

因为后面我们还需要将这个字符串粘贴修改并当作字节流发送，所以我们还需要写个将WPE格式的字符串转换成bytes字节流的方法

```python
def str_to_bytes(string):
    """
    将WPE格式的十六进制字符串还原为bytes字节流
    :param string:
    :return:
    """
    # 首先去除空格
    base_msg = string.replace(' ', '')
    # 然后还原成字节流
    return binascii.a2b_hex(base_msg)
```

接下来我们写个测试方法来看一下是不是我想要的效果：

```python
if __name__ == '__main__':
    data = b'*\xf8\x00e\xa8\xf3\x9d\x11\xe6'
    print('原始的字节流:', data)
    msg = bytes_to_str(data)
    print('转换成WPE格式进行显示:', msg)
    new_data = str_to_bytes(msg)
    print('还原成字节流进行显示：', new_data)
```

执行后打印结果如下：

```
原始的字节流: b'*\xf8\x00e\xa8\xf3\x9d\x11\xe6'
转换成WPE格式进行显示: 2A F8 00 65 A8 F3 9D 11 E6
还原成字节流进行显示:  b'*\xf8\x00e\xa8\xf3\x9d\x11\xe6'

进程已结束,退出代码0
```

PS：为了让大家更清楚它的过程，所以我把它拆分了多步以便注释，有需要的同学可以将代码进行简化，如第一个方法可以省略前两步：

```python
    return re.sub(r"(?<=\w)(?=(?:\w\w)+$)", " ", str(binascii.b2a_hex(data))[2
```

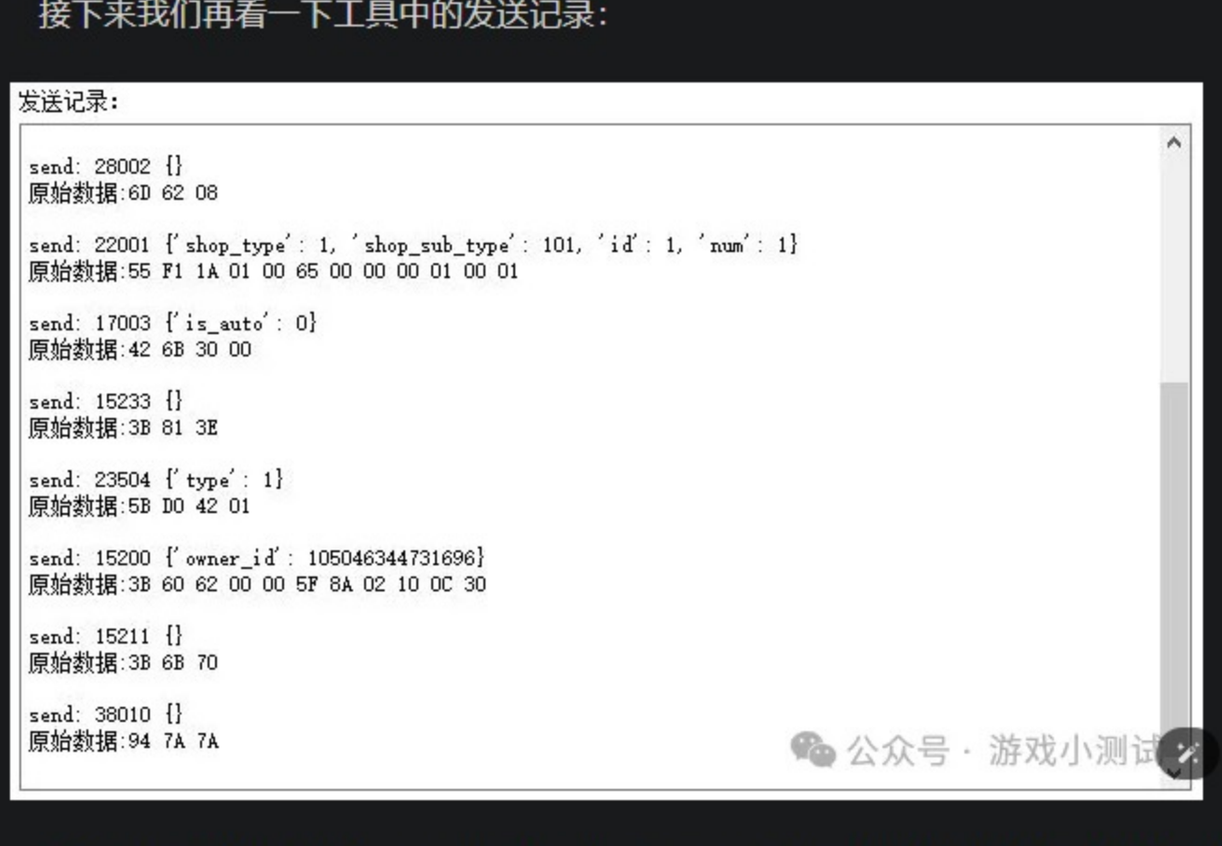
准备工作做完了，接下来就进入正题

一、复制客户端的协议进行修改，然后通过工具进行发送

我们先从复制粘贴法开始，这个比较简单，既然是复制，那么首先需要有一个复制源，在前几篇文章中，我们已经可以将字节流解析成明文进行显示了，现在我们对那个方法 ***(client_to_server()方法中)*** 进行一点小小的修改，将字节流转换成WPE格式的字符串一同显示出来。

```python
然后再调用对象明文化的方法，获取到对象的明文字典
lf.ui_thread.send_signal.emit(f'Send C{proto_id}: {dispose_obj(proto_obj)}\n')

找到上面这段代码，将其替换为下面这个样子：

然后再调用对象明文化的方法，获取到对象的明文字典
lf.ui_thread.send_signal.emit(f'Send C{proto_id}: {dispose_obj(proto_obj)}\n'
                              f'原始数据:{bytes_to_str(true_proto_data)}\n')
```

接下来我们再看一下工具中的发送记录：



OK，现在字符串已经有了，也可以复制了，下一步就是如何**将复制的十六进制字符串发出去**，如果大家还记得的话，在我们之前写的方法中，我们有两个转发线程，一个是从客户端处接收字节流，明文化之后直接通过与服务器连接的socket进行转发，另一个则是从服务器处接收字节流，明文化之后通过与客户端连接的socket进行转发，接下来我们要**在agent类中再写一个方法，用来发送我们复制或伪造的网络封包**，这个方法不用写在线程中。

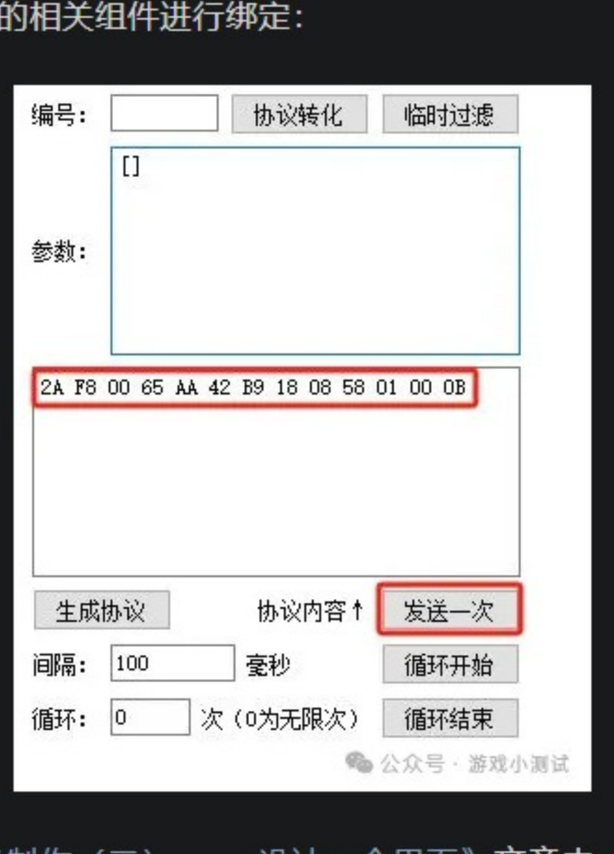
从简单的开始，先来写一下tcp连接方式下的实现方法：

```python
def insert_tcp_bytes(self, data):
    """
    通过server_socket发送一个tcp字节流（包含了协议号和协议内容）
    说明一下，按照我们之前定义的协议格式里面，一个完整的tcp封包分别包含：
    长度：除去协议号之外的，实际的协议内容的长度
    协议号：用4位int32表示协议号
    协议内容：实际的协议参数等内容
    有些可能还包含用来验证的验证码什么的，在这里我们先不考虑
    所以一个tcp封包的长度，实际指的是协议内容的长度，而我们传进来的协议内容是包含协议号
    所以在下面的方法里面，我们在对长度进行转化和拼接时，需要减去4
    这个要根据项目的实际情况来做对应的调整
    :param data: 包含了协议号和协议内容的字节流
    :return:
    """
    self.server_socket.send(struct.pack('!H', len(data)-4) + data)
```

websocket由于加入了mask掩码加密，所以与tcp的生成方式有较多不同，我会尽量把注释写的详细一些，直接上代码：

```python
def insert_ws_bytes(self, data):
    """
    通过server_socket发送一个websocket字节流（包含了协议号和协议内容）
    跟tcp的封包格式不同的是，ws的网络封包是包含了mask掩码加密的
    并且ws的封包中，长度是包含了协议号的
    如果要让我们的封包能正常通过服务器的验证，我们需要实现一套跟它一样的加密方法
    :param data: 包含了协议号和协议内容的字节流
    :return:
    """
    ws_data = b'\x82'       # 代表二进制序列帧
    length = len(data)      # ws协议的长度是需要包含协议号的
    if length < 126:
        ws_data += chr(128 | length).encode('latin-1')
    elif length < 65535:
        ws_data += chr(128 | 0x7e).encode('latin-1')
        ws_data += struct.pack("!H", length)
    else:
        ws_data += chr(128 | 0x7f).encode('latin-1')
        ws_data += struct.pack("!Q", length)
    # 生成4位掩码
    mask_key = os.urandom(4)
    # 将mask_key和data字节流分别转化成字节数组
    mask_value = array.array("B", mask_key)
    data_value = array.array("B", data)
    # 获取系统默认的字节序
    native_byteorder = sys.byteorder
    # 获取协议数据的长度
    data_len = len(data_value)
    # 将协议数据从字节转换为整数
    data_value = int.from_bytes(data_value, native_byteorder)
    # 扩展掩码值的长度，使其与数据值的长度相匹配
    mask_value = int.from_bytes(mask_value * (data_len // 4) + mask_value[: c
    # 对数据和掩码值进行异或操作，得到经过加密处理后的数据值，然后和之前的内容进行拼接
    ws_send_data = ws_data + mask_key + (data_value ^ mask_value).to_bytes(da
    # 通过server_socket将字节流发送给服务器
    self.server_socket.send(ws_send_data)
```

接下来我们跟工具的相关组件进行绑定：



在《游戏抓包工具制作（二）—— 设计一个界面》文章中，我在最后画了一个界面，由于后面功能很多，所以我之后会参照那个界面来分享，忘记的小伙伴可以回去翻一下那个界面。

为了减少各种封包内容错误导致的意外，我们在`WpeTools`那个类中先写一个检测方法，判断我们上图中的WPE文本转化的字节流，是否可以正常被协议对象解析：

```python
def check_data(self, data):
    """
    判断数据是否可以正常被解析
    :param data:
    :return:
    """
    # 根据字节流解析出协议号
    proto = struct.unpack(f"!H", data[0:4])[0]
    try:
        # 生成协议对象，调用decode方法，查看字节流是否可以正常被协议对象解析
        # 如果可以，则返回true，如果报错了，则返回false
        obj = eval(f'C{proto}')()
        obj.decode(data[4:])
        return True
    except Exception as e:
        print(f'协议检测失败：协议解析出现{e}错误，请检查是否可以通过 {proto} 协议
        return False
```

然后我们写一个点击"发送一次"按钮之后的处理逻辑：

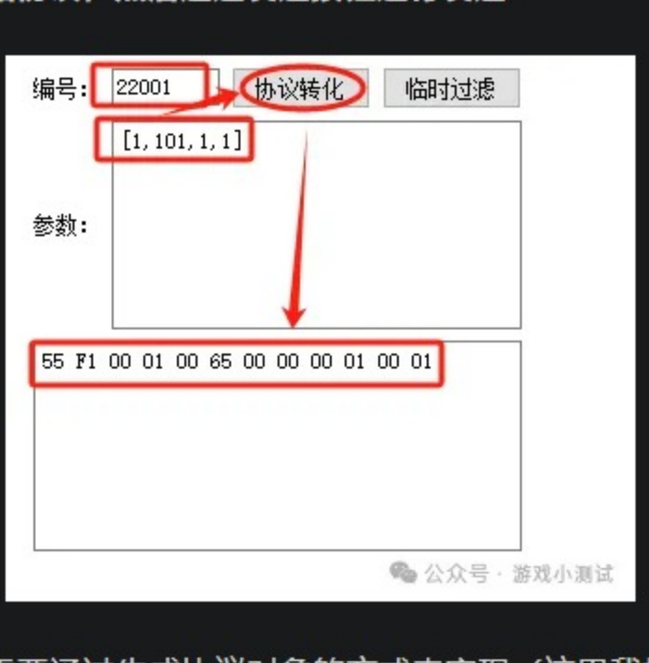
```python
def on_once_send_btn_click(self):
    """
    点击发送一次按钮之后的处理方法，以ws方法为例，tcp只是换个方法名而已
    读取文本框中的内容，转换为bytes，组合成一个完整协议，并发送给服务器
    :return:
    """

    try:
        try:
            # 首先从发送框获得字符串，并将其转化成字节流（这个文本框的组件名是textEd
            data = str_to_bytes(self.ui.textEdit.toPlainText())
            # 检查一下封包内容是否符合正确格式，如果是，则发送，否则，提醒格式不正确
            if self.agent.alive:
                if self.check_data(data):
                    self.agent.insert_ws_bytes(data)
                else:
                    print("协议检查失败，填入的协议无法正常解析")
            else:
                print("代理服务未启动，请先启动代理服务！")
        except SyntaxError:
            print("协议解析失败，请填入正确的WPE格式字节流数据")
        except AttributeError:
            print("协议发送失败，请先确保客户端已正常通过代理连接到服务器！")
        except Exception as e:
            print(f"协议检查出错，错误内容为：{e}")
    except Exception as e:
        print(f'协议发送失败，错误信息为：{e}')
```

最后将这个按钮调用方法跟界面上的按钮组件进行绑定（我一般写在界面初始化的时候进行绑定）

```python
# 绑定发送一次按钮事件
self.ui.once_send_btn.clicked.connect(self.on_once_send_btn_click)
```

到这里整个流程就完成了，大家如果能通过工具连上各自的游戏的话，可以从发送记录那里复制一份协议，粘贴到这个文本框中，然后点击"发送一次"的按钮，看看是否可以发送成功（需要注意的是，文本框中这里属于原始数据，如果你的项目的收发协议有其他类似验证或者加密方法的话，记得也要进行响应的处理哦）

二、通过参数生成原始协议，然后通过发送按钮进行发送



这里我们依然是需要通过生成协议对象的方式来实现（这里我加了一些try，防止输入格式错误，如能保证不输入错误的格式的话，可以去掉这些try），代码如下：

```python
def on_creat_proto_btn_click(self):
    """
    点击生成协议按钮之后的处理方法
    1.首先判断协议编号是否正确
    2.判断参数格式是否正确
    3.生成一个实例，然后调用对应的协议编号的encode方法，看是否正确返回
    4.将生成的字节流转换成WPE字符串显示格式
    :return:
    """
    # 判断协议号和协议参数是否有内容
    if self.ui.proto_id_line.text() and self.ui.proto_msg_line.toPlainText():
        try:
            # 解析出协议号，这里加个try，预防输入错误
            proto_id = int(self.ui.proto_id_line.text())
            try:
                # 解析参数列表，同样加个try，预防输入错误
                args = eval(self.ui.proto_msg_line.toPlainText())
                try:
                    # 看参数是否赋予协议对象，并调用encode方法
                    obj = eval(f'C{proto_id}')(*args)
                    proto, length, buf = obj.encode()
                    # 将协议号和协议内容的字节流，转化成WPE十六进制字符串，填入指定位
                    self.ui.textEdit.setText(bytes_to_str(struct.pack('!I', p
                except Exception:
                    print("协议生成失败：如确认填写正确，则可能是参数不符合协议格式
            except Exception:
                print("协议生成失败：协议参数不正确，请输入正确的参数，并以列表[]形
        except Exception:
            print("协议生成失败：协议编号格式不正确，请输入正确的数字")
    else:
        self.ui.textEdit.setText("协议号和协议内容不能为空！！！")
```

同样的，将按钮和方法进行绑定

```python
# 绑定生成协议按钮事件
self.ui.creat_proto_btn.clicked.connect(self.on_creat_proto_btn_click)
```

之后的操作就跟方法一一样了，只要生成协议内容，再点击发送按钮就可以将这个协议发送给服务器了。

三、给agent写一个生成协议的接口

```python
def send_proto(self, proto_id, *args):
    """
    根据协议id和参数进行协议发送
    :param proto_id:
    :param args:
    :return:
    """
    # 首先生成协议对象，传入参数实例化
    obj = eval(f'C{proto_id}')(*args)
    # 调用协议的encode方法，获得协议序列化之后的字节流
    proto, length, base_data = obj.encode()
    # 因为这个字节流是协议参数的，我们将它和协议号拼接一下
    data = struct.pack('!I', proto) + base_data
    # 根据连接类型，调用不同的协议发送方法(我这里两个都写上了，但实际只需要1个)
    # if 是tcp连接类型:
    self.insert_tcp_bytes(data)
    # elif 是websocket连接类型
    self.insert_ws_bytes(data)
```

这种方法跟界面关系不大，并非是通过工具上的组件调用的（虽然也可以），更多时候，我是在编写一些自动化脚本的时候来进行调用，比如我写个测试抽卡概率测试的伪代码，大概是这样：

```python
def probability_test(self, times):
    """
    测试一定次数的抽卡概率
    假设抽卡协议号是12345
    参数是抽卡类型，1是单抽，2是十连抽，考虑到效率，我们用10连
    考虑到背包可能会满，所以每抽卡1000次，清除一次背包
    :param times: 抽卡次数
    :return:
    """
    # 设置一个空结果集（在self.record_results()方法中对其进行更新）
    self.result = {}
    # 首先清除背包(自定义脚本)
    self.clean_bag()
    # 然后添加抽卡道具(自定义脚本)
    self.add_test_item()
    # 循环抽卡，先根据次数判断需要执行多少个大循环（需要清背包的）
    for i in range(int(times/1000)):
        for j in range(1000):
            self.send_proto(12345, 2)
        # 循环完成之后记录结果(自定义脚本)并清理背包
        self.record_results()
        self.clean_bag()
    # 循环剩余的次数
    for t in range(times % 1000):
        self.send_proto(12345, 2)
    # 记录结果
    self.record_results()
    return self.result
```

至此三种方法就介绍的差不多了，还有一种方法是调用新界面自动生成协议模板的，大概是这样：

篇幅有限，这里就先不讲了，有兴趣的同学可以自己研究一下。

求关注，求点赞，求转发，有问题的小伙伴可以私聊我，谢谢

#抓包工具 15

抓包工具 · 目录

〈上一篇
游戏抓包工具制作（四）—— 其他连接方式的实现（WebSocket）

下一篇〉
游戏抓包工具制作（六）—— socks代理非侵入式抓包