

变量检测

构造检测程序

处理变量检测

call的检测原理

查找检测变量

变量检测处理

堆栈检测

堆栈检测原理 基于本地

堆栈检测原理 基于服务器

处理一层堆栈检测

处理多层堆栈检测

变量检测

构造检测程序

接下来我们来了解一下游戏call检测的内容。由于口袋西游这个程序没有call检测，这里我们自己写了一个dll用于模拟call的检测，原理和实际的检测call是一样的。

首先打开吃药call检测，正常在游戏里吃药是不会有任何反应的



但是当我们点击吃药call的时候会弹出一个警告框。这样就构造了一个call的检测(吃药call参数写死了，记得把药品放到第一个格子)

处理变量检测

call的检测原理

对call的检测，大体来说分两种，第一种是堆栈检测，第二种是变量检测。这里用的是第二种变量检测。我们需要先搞懂一件事，才会更容易明白检测是如何设计的。

当我们正常在游戏中吃药是没有检测的，那是因为我们完完整整的把游戏的代码全部调用了；而我们直接点吃药call的时候执行的代码是不完全的。

那么游戏的开发者就可以在吃药call的外层设置一个变量，并且给变量赋值，然后在吃药call内层的某一个位置对这个变量的值进行检测。如果值不对，说明代码没有被完全执行。

在检测完成之后，再修改这个变量的值到原始状态，然后进行下一次检测。整个流程的伪代码如下：

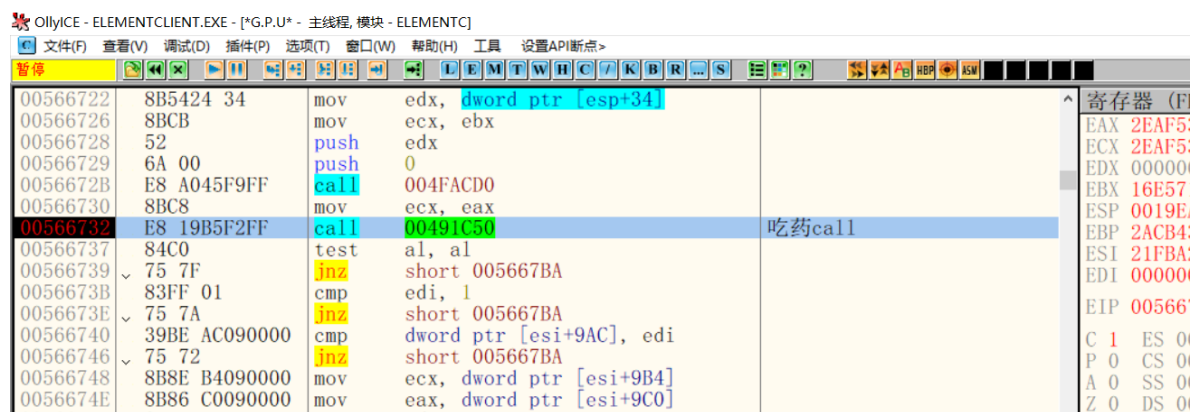
```
//用于检测的全局变量
int status=0;

function()
{
    //调用其他代码.....
    status=1;
    吃药call();
    //调用其他代码.....
}

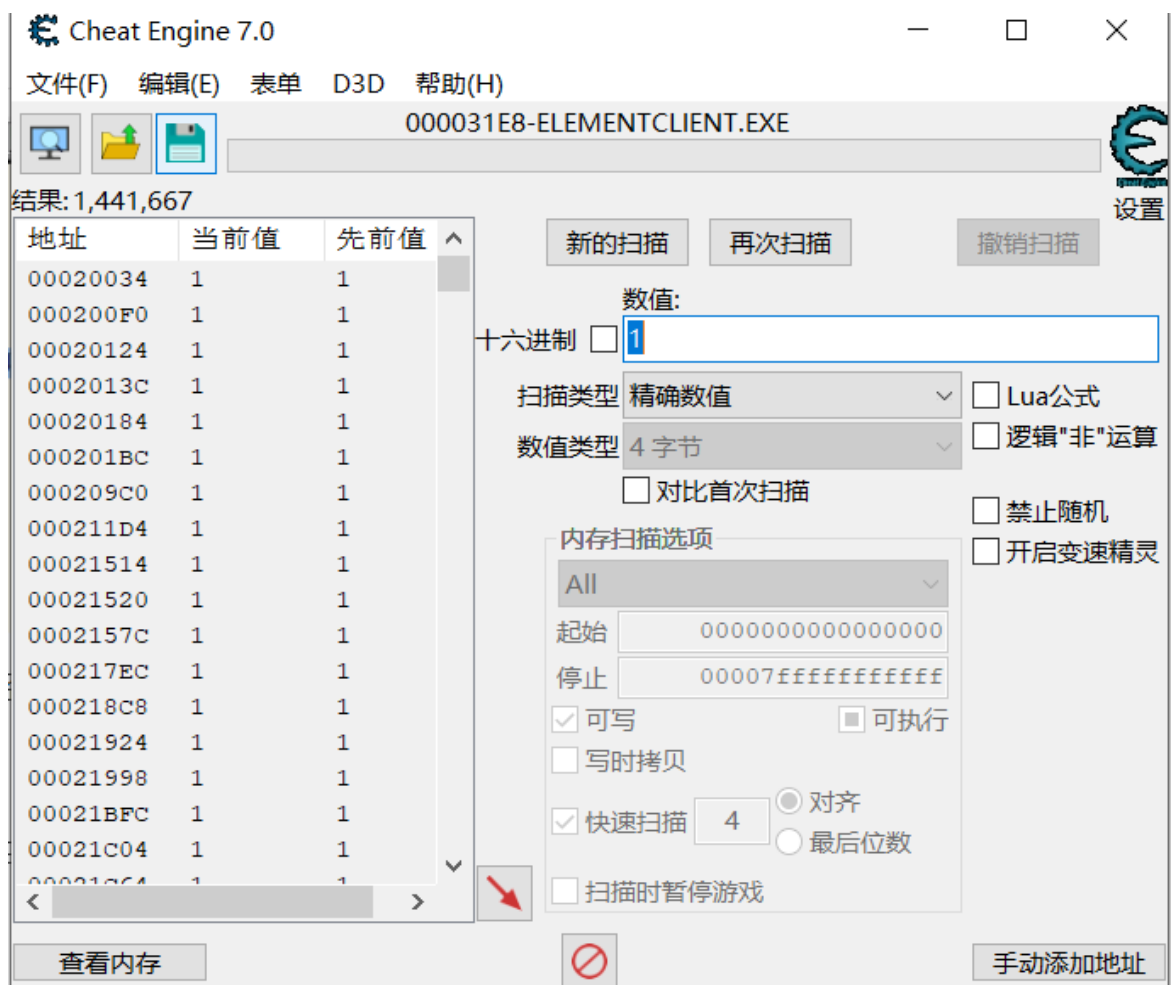
吃药call()
{
    //调用其他代码.....
    if(status==1)
    {
        //正常执行代码
    }
    else
    {
        //检测到外挂 进行处理
    }
    //再对status进行赋初始值 以便进行下一次检测
    status=0;
    //调用其他代码.....
}
```

查找检测变量

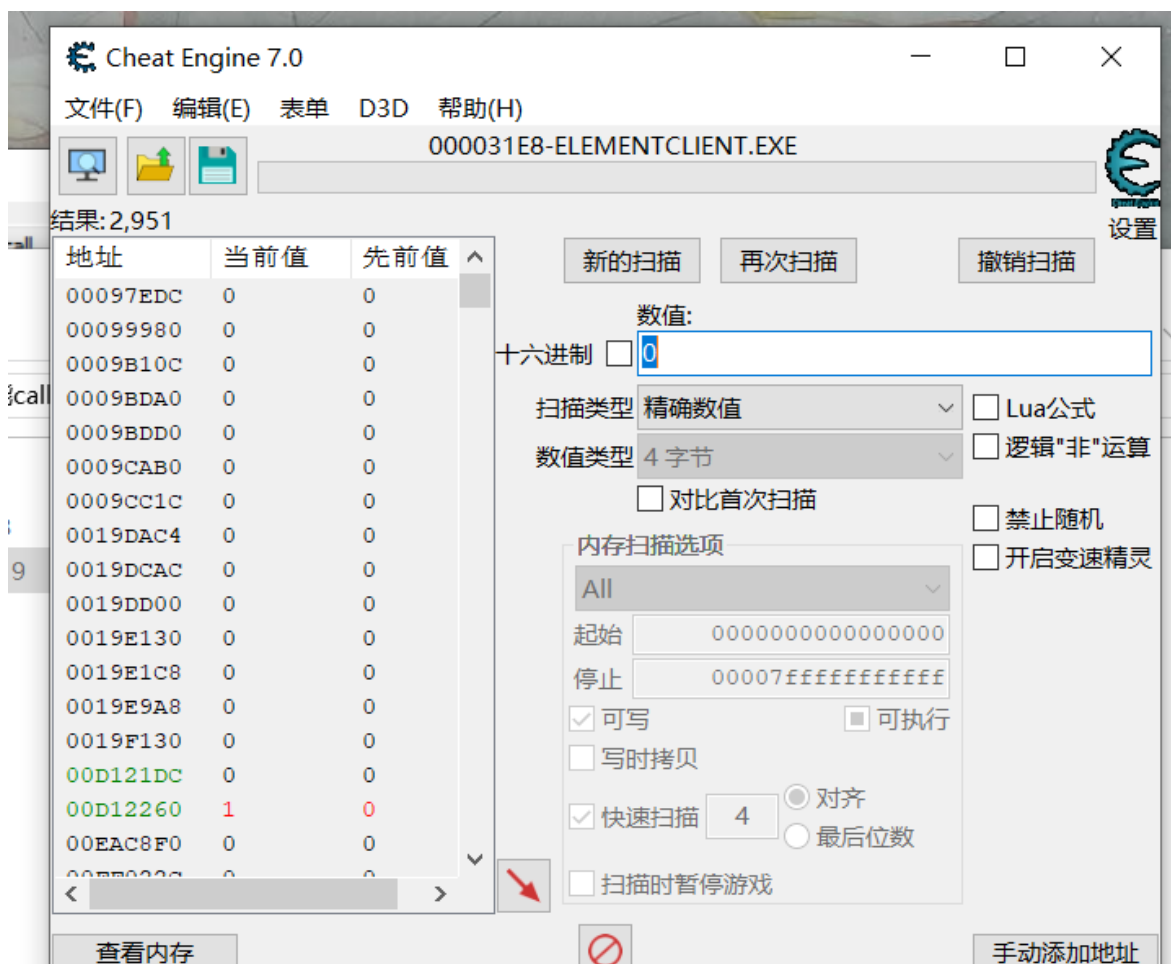
这个变量的值其实就相当于一个标志位，0和1的可能性比较大。所以我们可以通过搜索0和1的方式来找到这个变量，如果不行再尝试未知的初始值。



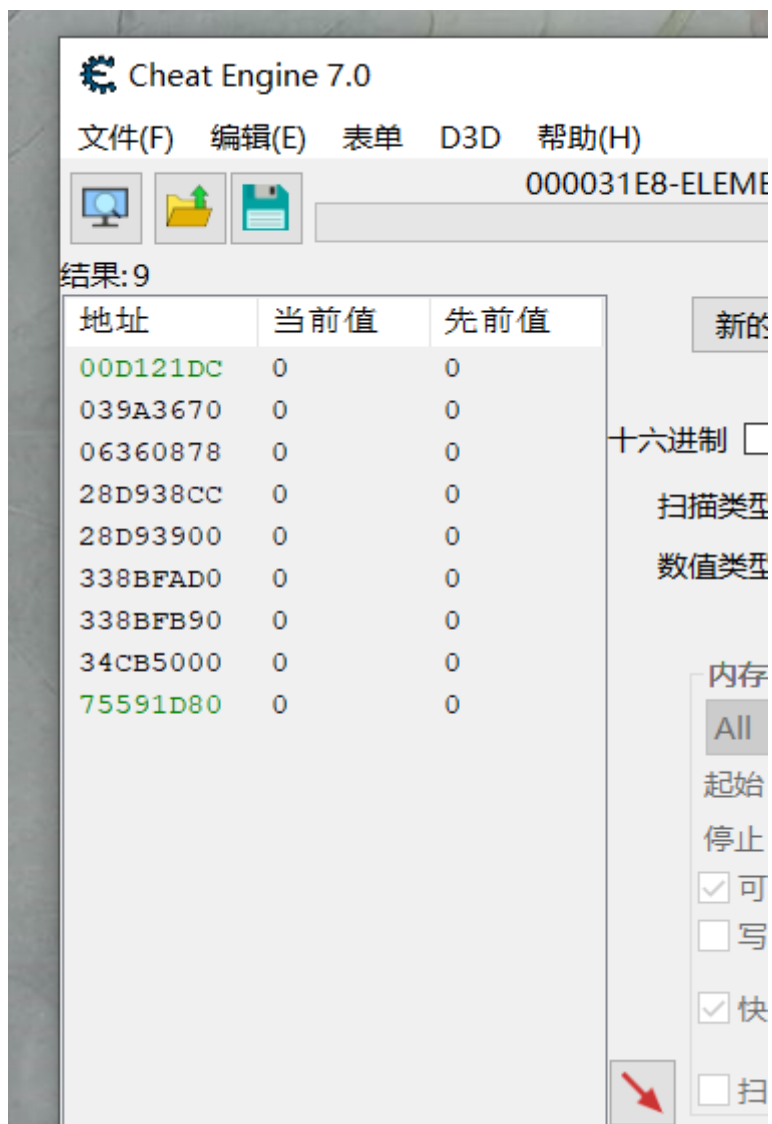
首先在吃药call下断，正常吃药让程序断下，这个时候外层的检测变量已经被赋值了



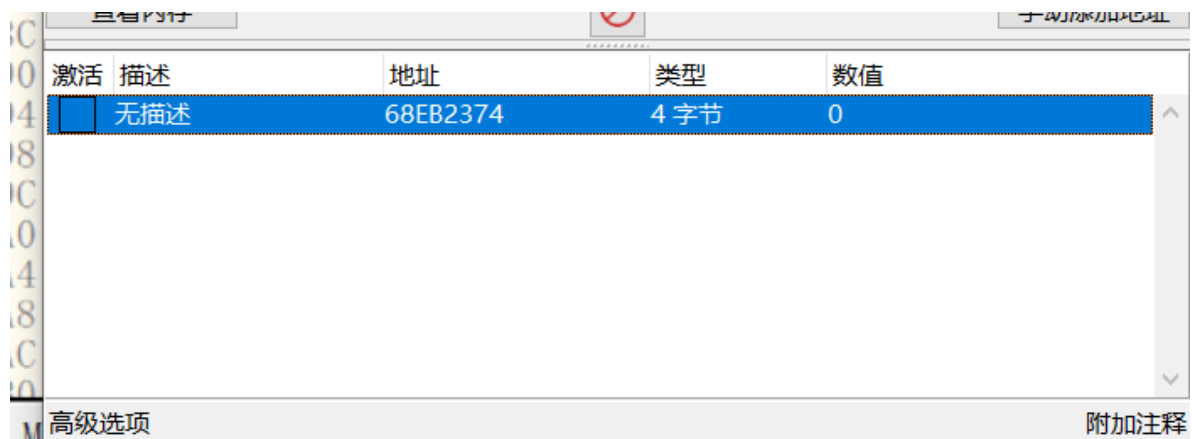
我们在CE里扫描1。然后F9运行程序，这个时候吃药call执行完毕，检测标志位也被恢复到原始状态



这个时候我们搜索0。

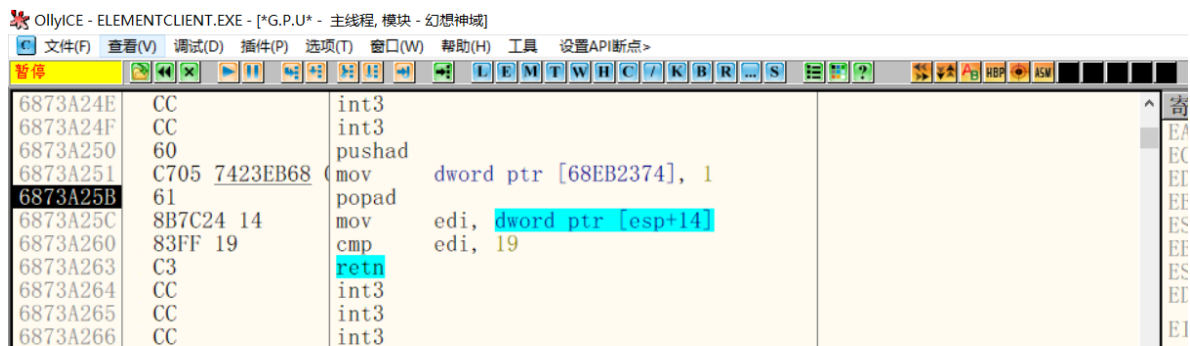


一直重复这个过程，最后剩下十个左右的地址就可以停止搜索了。



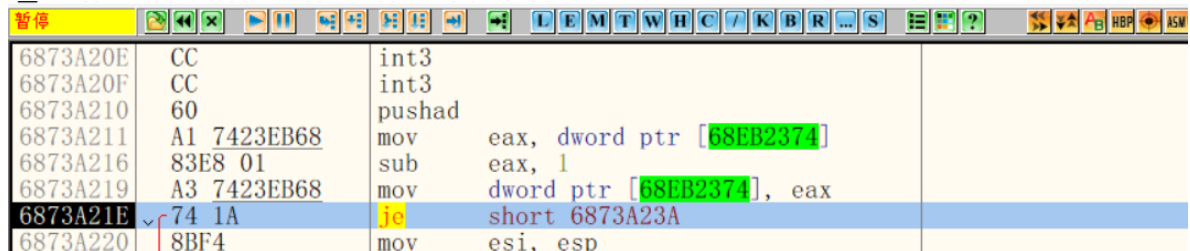
利用二分法进行测试，将这些变量逐个修改为1，看看哪一次调用call不会被检测到。这样可以定位到唯一的一个检测标志位。

然后我们在这个标志位下一个硬件写入断点，吃药让程序断下



```
OllyICE - ELEMENTCLIENT.EXE - [G.P.U* - 主线程, 模块 - 幻想神域]
文件(F) 查看(V) 调试(D) 插件(P) 选项(T) 窗口(W) 帮助(H) 工具 设置API断点>
暂停
6873A24E CC int3
6873A24F CC int3
6873A250 60 pushad
6873A251 C705 7423EB68 (mov dword ptr [68EB2374], 1
6873A25B 61 popad
6873A25C 8B7C24 14 mov edi, dword ptr [esp+14]
6873A260 83FF 19 cmp edi, 19
6873A263 C3 retn
6873A264 CC int3
6873A265 CC int3
6873A266 CC int3
```

第一次断下的位置, [68EB2374]这个地址会被赋值为1, F9运行程序

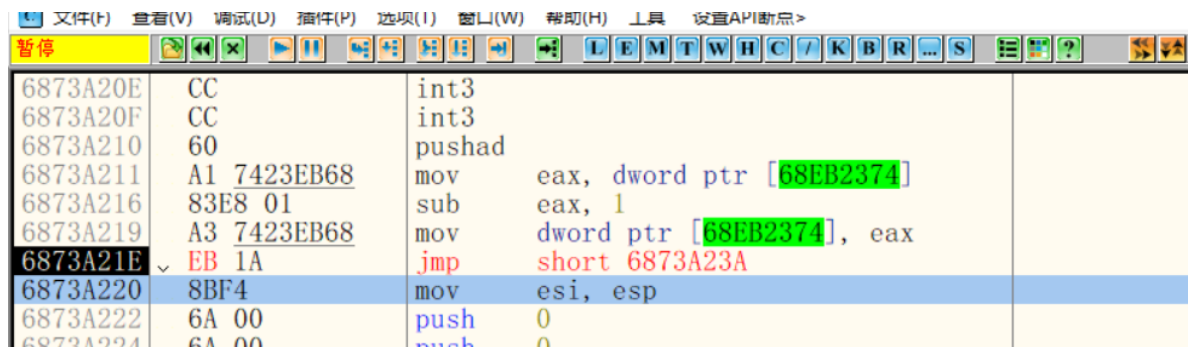


```
6873A20E CC int3
6873A20F CC int3
6873A210 60 pushad
6873A211 A1 7423EB68 mov eax, dword ptr [68EB2374]
6873A216 83E8 01 sub eax, 1
6873A219 A3 7423EB68 mov dword ptr [68EB2374], eax
6873A21E 74 1A je short 6873A23A
6873A220 8BF4 mov esi, esp
```

第二次断下的位置, 这个地方会在原来的的值的基础上减一。接着判断地址的值是否为0, 不为零则弹出警告框

变量检测处理

这里有两种处理方法, 第一种直接修改游戏代码, 不执行检测call。



```
6873A20E CC int3
6873A20F CC int3
6873A210 60 pushad
6873A211 A1 7423EB68 mov eax, dword ptr [68EB2374]
6873A216 83E8 01 sub eax, 1
6873A219 A3 7423EB68 mov dword ptr [68EB2374], eax
6873A21E EB 1A jmp short 6873A23A
6873A220 8BF4 mov esi, esp
6873A222 6A 00 push 0
6873A224 6A 00 push 0
```

第二种就是在调用call的时候, 手动将找到的检测变量赋值, 这种方法也是可以的

堆栈检测

变量检测有一个很明显的缺点, 就是需要一个实时去修改一个全局变量, 这种方法很容易用CE搜索到。那么能不能通过其他方式去传递检测变量而又不被CE扫描到呢? 答案是可以。

这个就是堆栈检测, 通过堆栈来传递检测变量。

堆栈检测原理 基于本地

堆栈检测分两种情况, 第一种就是检测堆栈中的返回地址。如果游戏中一个正常的call被执行的话, 那么堆栈中的所有的返回地址一定是本模块的; 相反, 如果是通过自己写的dll注入到游戏内部来调用的话, 那么堆栈中返回地址就全部是自己的dll模块的地址。

基于这个特点, 程序只要在功能call内部去读特定的堆栈返回地址, 检测当前地址是否属于本模块, 就可以检测出当前call是否被游戏调用。

堆栈检测原理 基于服务器

另外一种情况比较少见，程序会通过一个参数将当前堆栈的一部分数据发送到服务器，服务器接收到这些数据以后，再对参数内的堆栈地址进行检测。

这种方式不太实用，首先发送这部分堆栈数据很容易被调用者发现，其次会增大服务器压力，所以这种检测方式相对比较少见。

处理一层堆栈检测

这种检测的处理方法也有两种，第一种单步跟找到这个堆栈检测call，处理掉。但是这种检测call可能处在代码的任何一个位置，找起来不是那么容易。

第二种就是修改我们自己的程序代码。首先在进入call之后将返回地址修改为游戏模块内正确返回地址，这样就可以过掉检测call。然后在离开call之前将返回地址修改为我们自己的返回地址，这样可以保证程序正常执行。其实就是写两个HOOK修改堆栈地址。

伪代码如下：

```
HOOK函数头()
{
    //保存原程序的返回地址
    mov g_retaddr,[esp];
    //修改返回地址为游戏内的返回地址
    mov [esp],0x12345678
}
HOOK函数尾()
{
    //修改返回地址为原程序返回地址
    mov [esp],g_retaddr;
}
```

处理多层堆栈检测

如果堆栈检测只检测一层返回地址的话，那么就可以用上面的方式过掉检测。但是游戏往往会检测堆栈内的多个返回地址。

多层堆栈检测处理也很简单。处理一层堆栈检测要用HOOK的方式是因为在调用游戏内部的功能call的时候，我们没有办法去修改这个call内部的代码。

但如果是多层堆栈检测，就可以在调用这个功能call之前，就布局好当前的堆栈返回地址。伪代码如下：

```
function()
{
    sub esp,0x28;
    //修改第一层返回地址
    mov [esp],0x12345678;
    //修改第二层返回地址
    mov [esp+0x14],0x66666666;
    //调用功能call
    call xxxxxxx;
    //还原堆栈
    add esp,0x28;
}
```

利用上面这种方式，不管游戏内部检测多少层堆栈返回地址，我们都可以通过直接伪造堆栈的方式达到过掉检测的目的。

相关工具:

<https://github.com/TonyChen56/GameReverseNote>