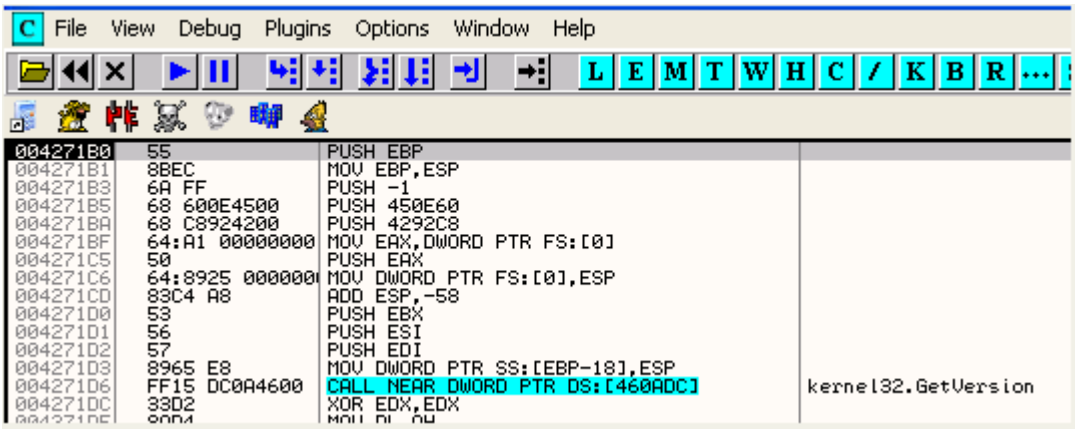
第五十二章-ASProtect v2.3.04.26 脱壳-Part2

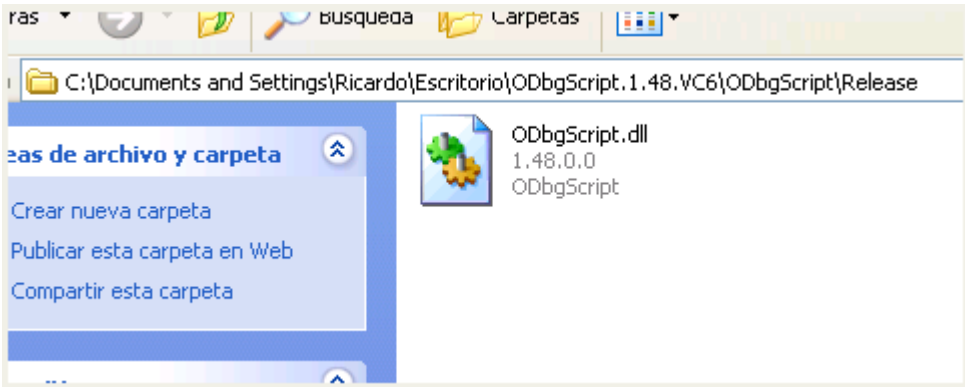
上次给大家搞的一个小比赛已经结束了。由于难度比较大,所以我仅仅只收到了提交来的少数几份答案。尽管如此,还是有童鞋脱颖而出了,Hiei 童鞋就成功编写出脚本将加了 ASProtect 壳的 UnPackMe 的 AntiDump 修复了。

这里需要提一下,很多时候我们的 OD 加载了 OllyAdvanced 插件,调试加了 ASProtect 壳的程序无法正常运行。这是因为 OllyAdvanced 这款插件中的 AntiRDTSC 这个功能加载的驱动导致的,AntiRDTSC 这个反反调试功能并不兼容所有机器。不知道大家对 RDTSC 了不了解,很多壳利用该指令检测某段程序执行的时间间隔以此来判断该程序是否正在被调试。但是现在开始流行多核了,已经不宜再用 RDTSC 指令来测试指令周期和时间了。我看了一下 ASProtect 的最新版本,已经没有采用 RDTSC 指令来进行反调试了,很可能是考虑到多核系统的兼容问题,所以我们这里就不必纠结这个问题了。

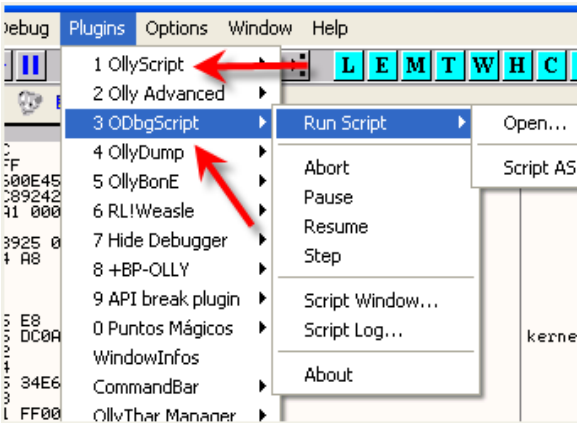
好了,下面我们用上一章中介绍的方法定位到 OEP,下面我们来执行下面的这个脚本看看效果。



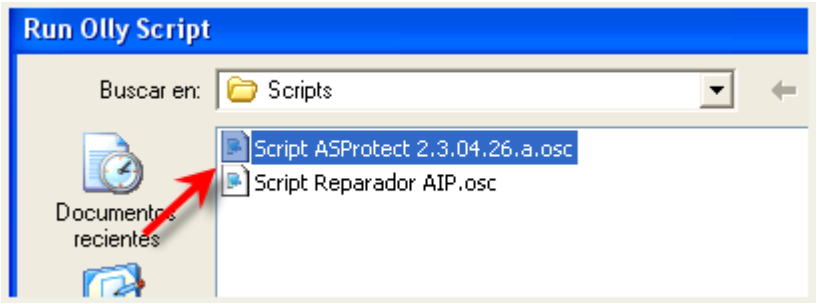
现在我们处于 OEP 处,这里大家需要注意一下,Hiei 童鞋的这个脚本不能在老版本的 OllyScript 下执行,大家需要使用新版本的 OllyScript 插件,这里我找到了 OllyScript 的新版,大家自行下载附件中的 ODbgScript 添加到 OD 的插件目录下就行。



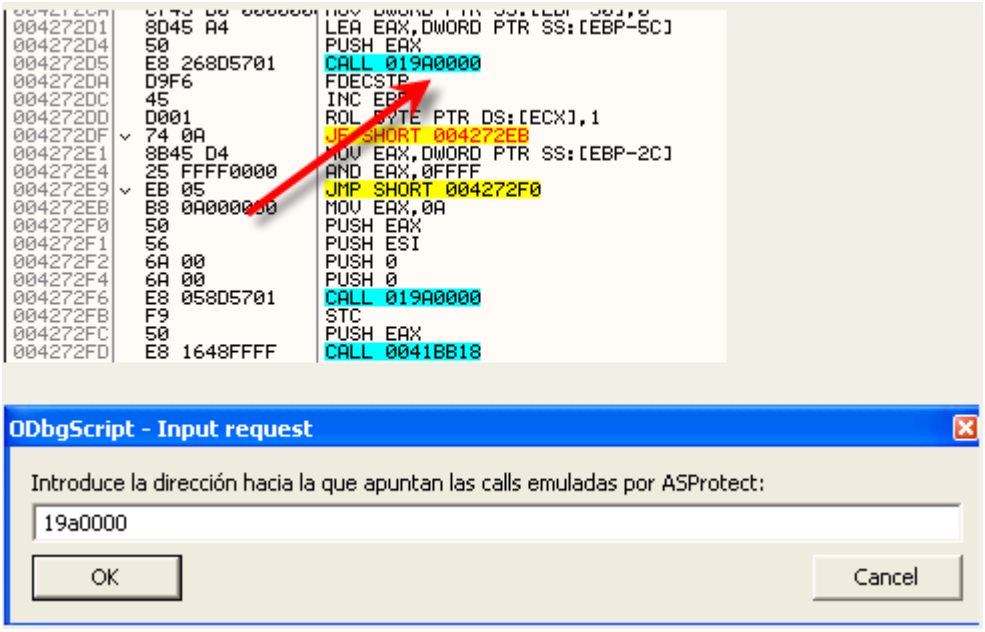
添加了新版的 OllyScript 插件以后,我们重启 OD,再次来到了 OEP 处,大家不要忘了关闭 break-on execute 这个选项(PS:这里并没有使用作者介绍的这个方法来定位 OEP,作者介绍的方法已经失灵了,所以不必关闭 break-on execute 这个选线带来的影响),不然会出问题的。



虽然这是 OllyScript 插件新的版本,但是由于与老版本的名字并不相同,所以 OD 插件菜单中新老两个插件都显示出来了。我们选择新版的 OllyScript,然后定位到 Hiei 童鞋的脚本。

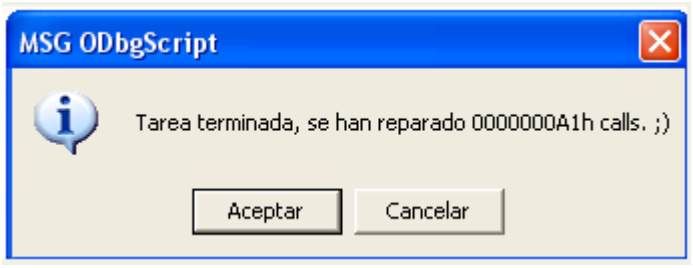


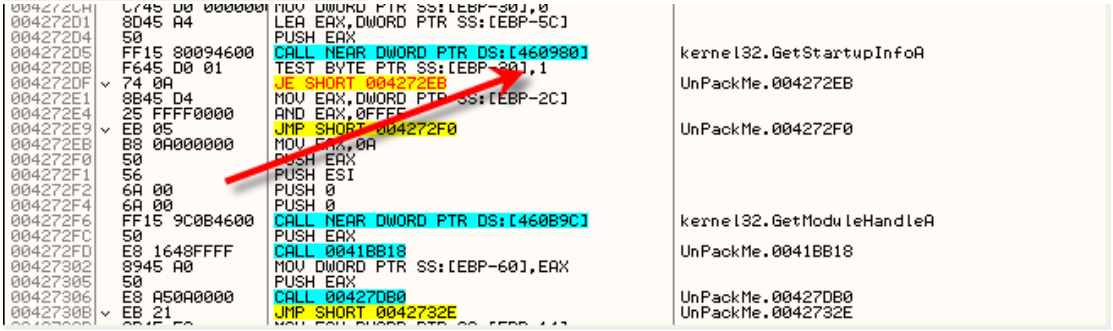
这里脚本启动以后,要求我们输入待修复 CALL 的地址,这里我的机器该地址是 19A0000。(PS:大家机器上这个待修复的地址可能不尽相同,请根据自己机器的地址来输入)



地址输入完以后,单击 OK,该脚本就开始执行了。

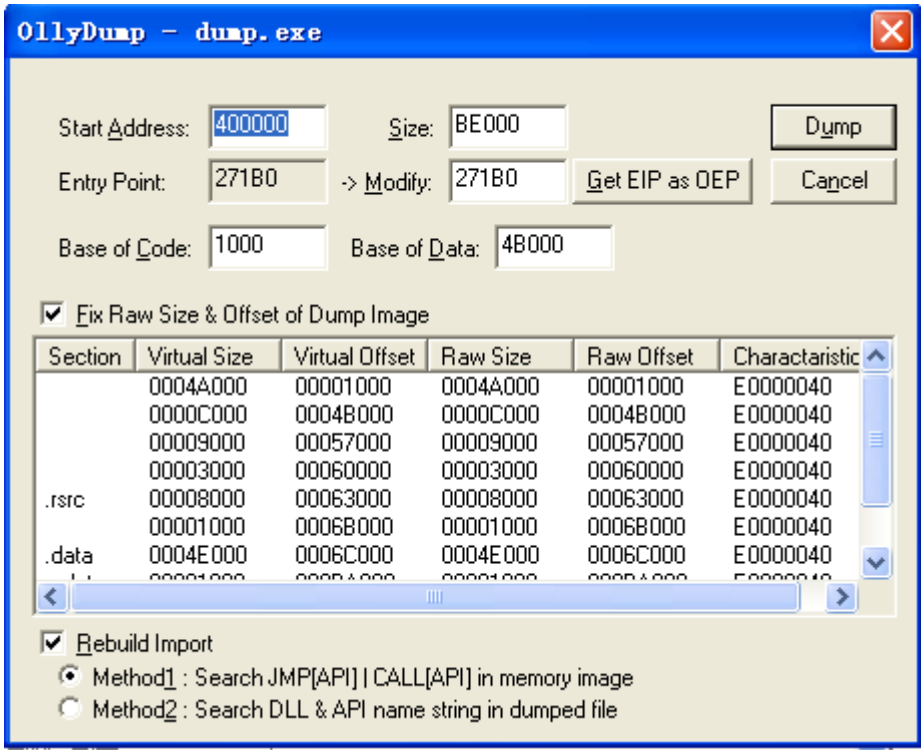
稍等片刻。



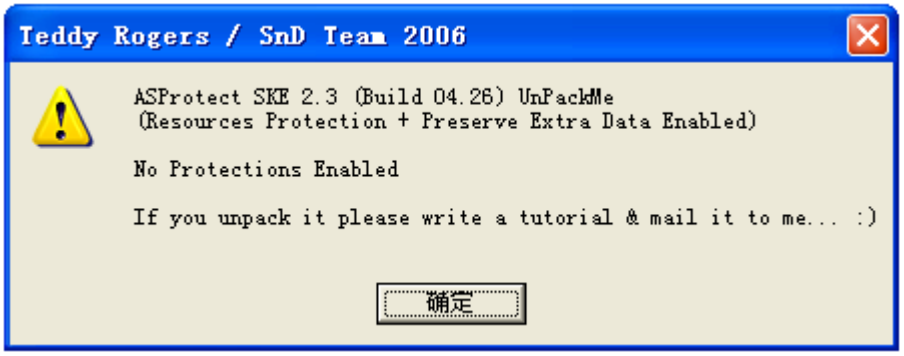


这里我们可以看到之前那些 CALL 19A0000 的指令已经被修复了,嘿嘿,HIEI 童鞋写的脚本很好用,赞一个。

接着使用 OllyDump 对其进行 dump(PS:这里我就不按作者的习惯来了,我直接用 OllyDump 来修复 IAT,所以选中了 Rebuild Import 这一项中 Method1)。



接下来直接运行 dump 并修复 IAT 的程序看看效果。



我们可以看到完美运行。

下面是 Hiei 童鞋的脚本,Hiei 童鞋加了详细的注释,接下来我来给大家一一做讲解。

```
/*
    -------------------------------------------------------------------
                                .:[CracksLatinoS]:.

        本脚本作者:Hiei

        目的:修复 ASProtect v2.3 SKE AntiDump

        目标程序:UnPackMe_ASProtect.2.3.04.26.a.exe.

        配置: ODbgScript v1.3x 或者更高版本,执行到 OEP 处,忽略所有异常

        日期: 2006 年 8 月 5 日

                              - = [备注] = -

        感谢:Ricardo Narvaja 和 Martian 的指导

        没有上面两位老师的指导,我就不能顺利编写这个脚本).

    -------------------------------------------------------------------
*/

var var_oep
var var_codebase
var var_codesize
var var_base_aspr
var var_base_aip
var var_ini_iat
var var_dir
var var_dir_iat
var var_sig
var var_dest
var var_api
var var_count
```

```
    cmp $VERSION,"1.30"                    // 检查 OllyScript 插件的版本
    jb err_version
    ask "请输入 ASProtect 构造 CALL 的地址:"
    cmp $RESULT,0
    je to_leave
    mov var_base_aip,$RESULT               // 保存用户输入的待修复的地址
    mov var_oep,eip                        // 将 OEP 保存到变量 var_oep 中
    gmi eip,codebase                       // 获取主程序代码段的基地址
    mov var_codebase,$RESULT               // 将代码段的基地址保存到变量 var_codebase 中
    gmi eip,codesize                       // 获取代码段的大小
    mov var_codesize,$RESULT               // 将代码段的大小保存到变量 var_codesize 中
    add var_codesize,var_codebase          // 计算代码段的结束位置
    mov var_ini_iat,460814                 // 将 IAT 的起始地址保存到变量 var_ini_iat 中
    mov var_base_aspr,[46C048]             // 保存我们可得知要调用哪个 API 函数指令地址所在区段的基地址(该区段由 ASProtect 创建)
    add var_base_aspr,3B02E                // 在基地址的基础上加上这个常量就可以定位到关键地址了
    bphws var_base_aspr,"x"                // 对该关键地址设置硬件执行断点
    jmp to_lookfor

to_lookfor:
    find var_codebase,#E8????????#         // 从代码段基地址处开始查找以机器码 E8 开头的 CALL
    cmp $RESULT,0                          // 如果没有找到,就输出总共找到的 CALL 个数,并退出脚本
    je no_calls
    mov var_dir,$RESULT                    // 如果找到的以机器码 E8 开头的 CALL,就把该 CALL 的地址保存到变量 var_dir 中
    mov var_sig,var_dir                    // 将找到的以机器码 E8 开头的 CALL 的地址保存到变量 var_sig 中
    mov var_dest,var_dir                   // 将找到的以机器码 E8 开头的 CALL 的地址保存到变量 var_dest 中
    add var_sig,5                          // 计算该 CALL 下一条指令的地址并保存到变量 var_sig 中
    inc var_dest                           // 将指针指向偏移量,并将指向偏移量的指针保存到变量 var_dest 中
    mov var_dest,[var_dest]                // 获取操作码 E8 后的偏移量,并保存到变量 var_dest 中
    add var_dest,var_sig                   // 计算 CALL 的目标地址
    cmp var_dest,var_base_aip              // 判断 CALL 的目标地址是否与用户输入的地址相同,如果相同则跳转到 to_execute 标签处进行下一步处理
    je to_execute
    inc var_dir                            // 如果 CALL 的目标地址与用户输入的地址不相同,则继续查找待修复 CALL 的地址
    mov var_codebase,var_dir               // 更新待查找的地址并保存到 var_codebase 中
    jmp to_lookfor

to_execute:                                // 是用户输入的待修复 CALL 的目标地址
    mov eip,var_dir                        // 将 EIP 修改为待修复 CALL 所在的地址
    run                                    // 运行起来

eob to_verify                              // 如果断下来了,就跳转到 to_verify 标签处

to_verify:
    cmp eip,var_base_aspr                  // 判断断下来的地方是不是关键地址(执行到该关键地址处时,此时的 EDX 中保存了实际要调用 API 函数的地址)
    jne unexpected                         // 如果断下来的地方不是关键地址处,说明发生异常了,直接跳转到 unexpected 标签处,进行异常处理
```

```
    mov var_api,edx                                  // 如果断下来的地方是关键地址处,则将 EDX 寄存器中实际要调用 API 函数的地址保存到变量 var_api 中
    jmp to_lookfor_api                               // 跳转到 to_lookfor_api 标签处,进行下一步的处理

to_lookfor_api:
    cmp var_ini_iat, 460F28                          // 判断是否超出了 IAT 的范围
    je error                                         // 如果超出了 IAT 的范围,说明并没有在 IAT 中定位到 API 函数对应的 IAT 项
    cmp [var_ini_iat],var_api                        // 判断是否是与该 API 函数地址对应的 IAT 项
    je to_repair                                     // 如果是与该 API 函数对应的 IAT 项,则跳转到 to_repair 标签处
    add var_ini_iat,4                                // 如果不是与该 API 函数对应的 IAT 项,将指向 IAT 的指针往后移 4 个字节,便于下一次查找
    jmp to_lookfor_api                               // 跳转到 to_lookfor_api 标签处继续查找 IAT 项

to_repair:
    mov var_dir_iat,var_ini_iat                      // 将与该 API 函数对应的 IAT 项的指针保存到变量 var_dir_iat 中
    ref var_dir                                      // 搜索该待修复的 CALL 的参考引用处
    cmp $RESULT,0                                    // 判断待修复的 CALL 的指令有没有参考引用
    jne repair_jump                                  // 如果待修复 CALL 的指令有参考引用,则跳转到 repair_jump 标签处
    eval "Call dword[{var_dir_iat}]"
    asm var_dir,$RESULT                              // 将待修复的 CALL 汇编成 CALL DWORD[对应 IAT 项的指针]的形式
    inc var_count                                    // 更新已修复项的计数
    inc var_dir
    mov var_codebase,var_dir
    mov var_ini_iat,460814                           // 将 var_ini_iat 变量重新设置为 IAT 的起始位置,以便下次查找
    jmp to_lookfor

repair_jump:
    eval "Jmp dword[{var_dir_iat}]"
    asm var_dir,$RESULT                              // 将待修复的 CALL 汇编成 JMP DWORD[对应 IAT 项的指针]的形式
    inc var_count
    inc var_dir
    mov var_codebase,var_dir
    mov var_ini_iat,460814                           // 将 var_ini_iat 变量重新设置为 IAT 的起始位置,以便下次查找
    jmp to_lookfor

unexpected:
    msg "发生异常了,是否继续"
    cmp $RESULT,0
    je to_leave
    run

error:
    eval "错误,请手动修改 CALL 的地址: {var_dir}h"
    msg $RESULT
    run

no_calls:
```

```
    bphwc var_base_aspr
    eval "任务完成！总共修复{var_count}h 个 CALL?;)"
    msg $RESULT
    jmp to_leave

err_version:
    msg "错误,OllyScript 版本过低!"
    ret

to_leave:
    bphwc var_base_aspr
    mov eip,var_oep
    ret
```

附几张脚本的截图:

```
0000 /*
0001     ------------------------------------------------------------------
0002                           .:[CracksLatinoS]:.
0003
0004         本脚本作者:Hiei
0005
0006         目的:修复ASProtect v2.3 SKE AntiDump
0007
0008         目标程序:UnPackMe_ASProtect.2.3.04.26.a.exe.
0009
0010         配置: ODbgScript v1.3x或者更高版本,执行到OEP处,忽略所有异常
0011
0012         日期: 2006年8月5日
0013
0014                           - = [备注] = -
0015
0016         感谢:Ricardo Narvaja和Martian的指导
0017         没有上面两位老师的指导,我就不能顺利编写这个脚本).
0018
0019     ------------------------------------------------------------------
0020 */
0021
0022 var var_oep
0023 var var_codebase
0024 var var_codesize
0025 var var_base_aspr
0026 var var_base_aip
0027 var var_ini_iat
0028 var var_dir
0029 var var_dir_iat
0030 var var_sig
0031 var var_dest
0032 var var_api
0033 var var_count
0034
0035    cmp $VERSION,"1.30"                    // 检查OllyScript插件的版本
0036    jb err_version
0037    ask "请输入ASProtect构造CALL的地址:"
0038    cmp $RESULT,0
0039    je to_leave
0040    mov var_base_aip,$RESULT               // 保存用户输入的待修复的地址
0041    mov var_oep,eip                        // 将OEP保存到变量var_oep中
0042    gmi eip,codebase                       // 获取主程序代码段的基地址
0043    mov var_codebase,$RESULT               // 将代码段的基地址保存到变量var_codebase中
0044    gmi eip,codesize                       // 获取代码段的大小
0045    mov var_codesize,$RESULT               // 将代码段的大小保存到变量var_codesize中
```

```
   add var_codesize,var_codebase              // 计算代码段的结束位置
   mov var_ini_iat,460814                     // 将IAT的起始地址保存到变量var_ini_iat中
   mov var_base_aspr,[46C048]                 // 保存我们可得知要调用哪个API函数指令地址所在区段的基地址(该区段由ASProtect创建)
   add var_base_aspr,3B02E                    // 在基地址的基础上加上这个常量就可以定位到关键地址了
   bphws var_base_aspr,"x"                    // 对该关键地址设置硬件执行断点
   jmp to_lookfor

to_lookfor:
   find var_codebase,#E8????????#             // 从代码段基地址处开始查找以机器码E8开头的CALL
   cmp $RESULT,0                              // 如果没有找到,就输出总共找到的CALL个数,并退出脚本
   je no_calls
   mov var_dir,$RESULT                        // 如果找到的以机器码E8开头的CALL,就把该CALL的地址保存到变量var_dir中
   mov var_sig,var_dir                        // 将找到的以机器码E8开头的CALL的地址保存到变量var_sig中
   mov var_dest,var_dir                       // 将找到的以机器码E8开头的CALL的地址保存到变量var_dest中
   add var_sig,5                              // 计算该CALL下一条指令的地址并保存到变量var_sig中
   inc var_dest                               // 将指针指向偏移量,并将指向偏移量的指针保存到变量var_dest中
   mov var_dest,[var_dest]                    // 获取操作码E8后的偏移量,并保存到变量var_dest中
   add var_dest,var_sig                       // 计算CALL的目标地址
   cmp var_dest,var_base_aip                  // 判断CALL的目标地址是否与用户输入的地址相同,如果相同则跳转到to_execute标签处进行下一步处理
   je to_execute
   inc var_dir                                // 如果CALL的目标地址与用户输入的地址不相同,则继续查找待修复CALL的地址
   mov var_codebase,var_dir                   // 更新待查找的地址并保存到var_codebase中
   jmp to_lookfor

to_execute:                                   // 是用户输入的待修复CALL的目标地址
   mov eip,var_dir                            // 将EIP修改为待修复CALL所在的地址
   run                                        // 运行起来

eob to_verify                                 // 如果断下来了,就跳转到to_verify标签处

to_verify:
   cmp eip,var_base_aspr                      // 判断断下来的地方是不是关键地址(执行到该关键地址处时,此时的EDX中保存了实际要调用API函数的地址)
   jne unexpected                             // 如果断下来的地方不是关键地址处,说明发生异常了,直接跳转到unexpected标签处,进行异常处理
   mov var_api,edx                            // 如果断下来的地方是关键地址处,则将EDX寄存器中实际要调用API函数的地址保存到变量var_api中
   jmp to_lookfor_api                         // 跳转到to_lookfor_api标签处,进行下一步的处理

to_lookfor_api:
   cmp var_ini_iat, 460F28                    // 判断是否超出了IAT的范围
   je error                                   // 如果超出了IAT的范围,说明并没有在IAT中定位到API函数对应的IAT项
   cmp [var_ini_iat],var_api                  // 判断是否是与该API函数地址对应的IAT项
   je to_repair                               // 如果是与该API函数对应的IAT项,则跳转到to_repair标签处
   add var_ini_iat,4                          // 如果不是与该API函数对应的IAT项,将指向IAT的指针往后移4个字节,便于下一次查找
   jmp to_lookfor_api                         // 跳转到to_lookfor_api标签处继续查找IAT项
```

```
to_repair:
   mov var_dir_iat,var_ini_iat                // 将与该API函数对应的IAT项的指针保存到变量var_dir_iat中
   ref var_dir                                // 搜索该待修复CALL的参考引用处
   cmp $RESULT,0                              // 判断待修复CALL的指令有没有参考引用处
   jne repair_jump                            // 如果待修复CALL的指令有参考引用,则跳转到repair_jump标签处
   eval "Call dword[{var_dir_iat}]"
   asm var_dir,$RESULT                        // 将待修复的CALL汇编成CALL DWORD[对应IAT项的指针]的形式
   inc var_count                              // 更新已修复项的计数
   inc var_dir
   mov var_codebase,var_dir
   mov var_ini_iat,460814                     // 将var_ini_iat变量重新设置为IAT的起始位置,以便下次查找
   jmp to_lookfor

repair_jump:
   eval "Jmp dword[{var_dir_iat}]"
   asm var_dir,$RESULT                        // 将待修复的CALL汇编成JMP DWORD[对应IAT项的指针]的形式
   inc var_count
   inc var_dir
   mov var_codebase,var_dir
   mov var_ini_iat,460814                     // 将var_ini_iat变量重新设置为IAT的起始位置,以便下次查找
   jmp to_lookfor

unexpected:
   msg "发生异常了,是否继续"
   cmp $RESULT,0
   je to_leave
   run

error:
   eval "错误,请手动修改CALL的地址: {var_dir}h"
   msg $RESULT
   run

no_calls:
   bphwc var_base_aspr
   eval "任务完成! 总共修复{var_count}h 个CALL?;)"
   msg $RESULT
   jmp to_leave

err_version:
   msg "错误,OllyScript版本过低!"
   ret

to_leave:
   bphwc var_base_aspr
   mov eip,var_oep
   ret
```

我感觉上面脚本中的注释已经够详细了,不需要我再一条语句一条语句的解释了。这里我需要解释的就是其如何实现在任意机器上定位 ASProtect 的函数的,这里它并没有采用我说的内存访问断点的方式。

这里是保存 ASProtect 创建区段的起始地址。

```
mov var_base_aspr,[46C048]          // 保存我们可得知要调用哪个 API 函数指令地址所在区段的基地址(该区段由 ASProtect 创建)

add var_base_aspr,3B02E             // 在基地址的基础上加上这个常量就可以定位到关键地址了

bphws var_base_aspr,"x"             // 对该关键地址设置硬件执行断点

jmp to_lookfor
```

在 46c048 这个地址指向的内存单元中我们可以定位到关键地址(由该关键地址指令所在处我们可以得知要实际要调用哪个 API

函数)所在区段的基地址,然后加上一个常量就可以得到关键地址,这样就可以实现在各个机器上通用。因为不同的机器上可以该基地址可能会不相同,但是关键地址的偏移量是相同的,这里的关键地址的偏移量就是 3B02E。所在接着对该关键地址设置硬件执行断点。

好了,下面从起始地址为 401000 的代码段开始搜索目标地址为用户输入 CALL 的地址。

```
to_execute:                          // 是用户输入的待修复 CALL 的目标地址
    mov eip,var_dir                  // 将 EIP 修改为待修复 CALL 所在的地址
    run                              // 运行起来
```

这里将 EIP 设置为待修复 CALL 的地址,然后运行起来,紧接着下面的 eob to_verify 命令是遇到中断则跳转到 to_verify 标签处。

```
eob to_verify                        // 如果断下来了,就跳转到 to_verify 标签处
```

这里是验证是否断在了之前设置的硬件执行断点的指令处,如果是,则跳转到 to_verify 标签进行下一步处理。

```
to_verify:
    cmp eip,var_base_aspr            // 判断断下来的地方是不是关键地址(到执行到该关键地址处时,此时的 EDX 中保存了实际要调用 API 函数的地址)
    jne unexpected                   // 如果断下来的地方不是关键地址处,说明发生异常了,直接跳转到 unexpected 标签处,进行异常处理
    mov var_api,edx                  // 如果断下来的地方是关键地址处,则将 EDX 寄存器中实际要调用 API 函数的地址保存到变量 var_api 中
    jmp to_lookfor_api               // 跳转到 to_lookfor_api 标签处,进行下一步的处理
```

这里是判断是不是断在了关键地址处(该关键地址处,EDX 寄存器中保存了实际要调用 API 函数的地址),如果不是断在关键地址处,就表示出错了,就跳转到 unexpected 标签处(异常处理标签),如果断在了关键地址处,则保存 EDX 中的 API 函数地址,接着跳转到 to_lookfor_api 标签处进行下一步处理。

```
to_lookfor_api:
    cmp var_ini_iat, 460F28          // 判断是否超出了 IAT 的范围
    je error                         // 如果超出了 IAT 的范围,说明并没有在 IAT 中定位到 API 函数对应的 IAT 项
    cmp [var_ini_iat],var_api        // 判断是否是与该 API 函数地址对应的 IAT 项
    je to_repair                     // 如果是与该 API 函数对应的 IAT 项,则跳转到 to_repair 标签处
    add var_ini_iat,4                // 如果不是与该 API 函数对应的 IAT 项,将指向 IAT 的指针往后移 4 个字节,便于下一次查找
    jmp to_lookfor_api               // 跳转到 to_lookfor_api 标签处继续查找 IAT 项
```

这里判断待查找的 IAT 指针是否超出了 IAT 的范围,如果指针已经超出 IAT 的范围,说明没找到与之匹配的 IAT 项,则跳转到 error 标签处,请用户重新输入待修复的地址。

如果在 IAT 中查找到了 API 函数对应的 IAT 项,则跳转到 to_repair 标签处进行修复处理。

修复处理如下:

首先判断待修复 CALL 指令是否存在参考引用,如果不存在存在引用处则将其汇编为 CALL [对应 IAT 项的地址]的形式,如果存在参考引用,则将其汇编为 JMP [对应 IAT 项的地址]的形式。

然后按照上面的步骤继续修复其他的项,直到所有项都修复为止。

这里非常感谢 HIHE 童鞋提供的这个脚本,条理非常清晰,注释也很详细。很少有人编写的脚本注释的如此之详细。为 HIEI 童鞋点赞。为了感谢 HIEI 童鞋,HIEI 童鞋将会免费获得一张去 XXX 旅行的机票。嘿嘿,不过你得亲自过来取,哈哈哈。

好了,接下来还有一个小比赛,比起这个来说要简单的多。分为两个部分(我猜这次参赛的人可能会多一些,哈哈)。

首先大家需要定位到 TPPpack 这款壳的 OEP,然后修复它的 Stolen bytes,之前 Martian 先生已经写过一个手脱 TPPpack 的教程了,哈哈。

大家可以随意使用 HideDebugger,HideOd 等反反调试插件,以及 ODdbgScript 等脚本插件。

第二部分,大家需要编写一个脚本来修复 TPPpack 的 IAT。大家可能分别为两部分各编写一个脚本。

简而言之:

Part1:编写脚本定位 OEP 并修复 Stolen bytes

Part2:编写脚本修复 IAT

上面说过了大家可以使用上面提到的这 3 个插件。(但是请大家不要再问能不能使用 CmdBar 命令栏插件这样的问题(PS:提这样

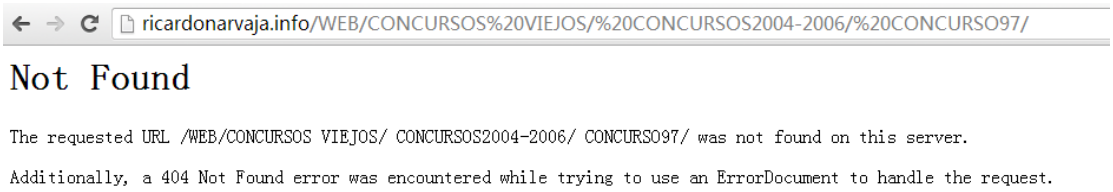
的问题确实有点弱智,哈哈哈),嘿嘿）

比赛截止日期:8 月 30 号,在截止日期之前完成任务的童鞋可以将你们的答案邮件给我,两个答案一起发,分开发都可以。

Martian 写的手脱 TPPpack 的教程在我的博客上,网址如下:

http://ricardonarvaja.info/WEB/CONCURSOS%20VIEJOS/%20CONCURSOS2004-2006/%20CONCURSO97/

(PS:貌似该页面已经被移除了)



本章到此结束,感谢大家的观看。

希望在第 53 章中看到你们的名字,嘿嘿。