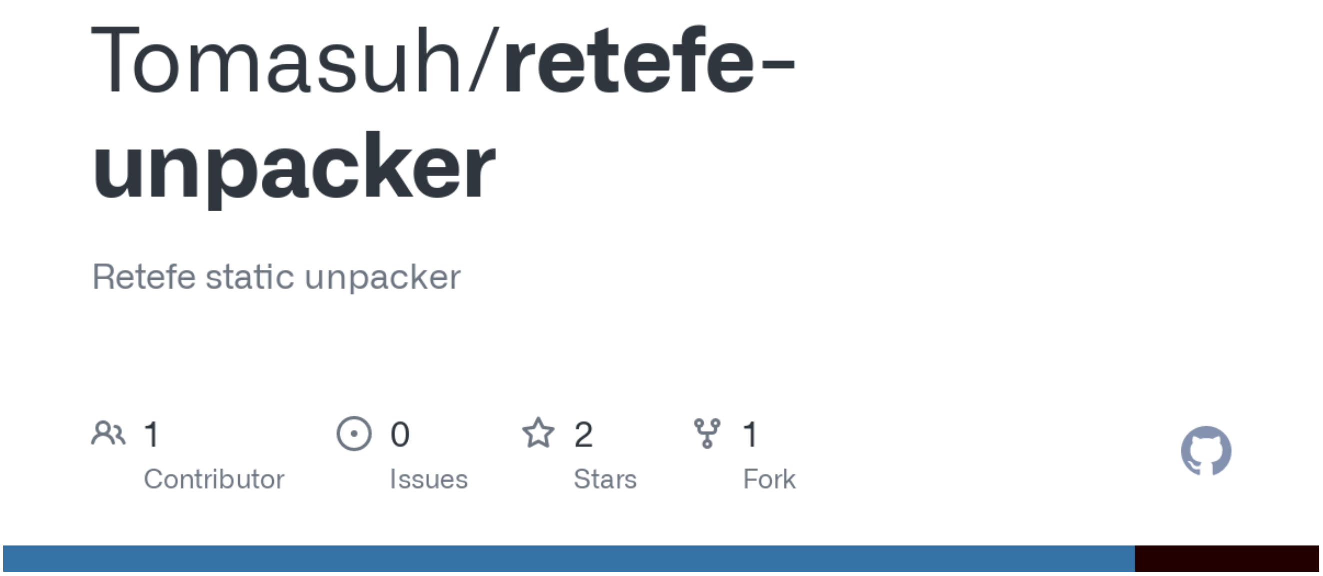
# Retefe unpacker

github.com/Tomasuh/retefe-unpacker

Tomasuh




| layout | title | date | comments | categories |
|---|---|---|---|---|
| post | Retefe unpacker | 2018-12-28 | true | |

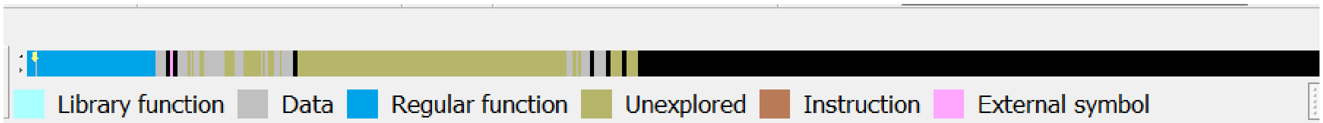
This is a writeup on how to implement an unpacker for current versions (at the time of publication) of the banking malware Retefe.

Resources about the threat:

- Retefe banking Trojan leverages EternalBlue exploit in Swiss campaigns
- The Retefe Saga
- Reversing Retefe
- New version of Retefe Banking Trojan Uses EternalBlue

Historically there seems to be some variance of ways the malware has stored it's Javascript payload. Some sources mentions self extracting ZIP files and other XORed data. The current version makes use of a 4 byte XOR key which is generated based on the scripts length and a few mathematical operations performed on it. The post Reversing Retefe from about two months back (2018-11-08) shows use of a one byte XOR key which indicates that the threat actor has changed its code base after the release of that post. This post is made with the intention to shed some light on the current way the threat Retefe stores its payload.

Looking at the mapped binary image with IDA shows a large amount of unexplored data that is in the `.data` segment.




Browsing the `.data` segment with Binary Ninja shows a large segment of data whose top is referenced in a copy instruction:

```
mov     rcx, qword [rsp+0x20 {write_encoded_buffer_to}]
lea     rdx, [rel 0x140050960]
mov     rdi, qword [rcx]
mov     rsi, rdx  {0x140050960}
mov     ecx, eax
rep movsb byte [rdi], [rsi]  {0x0}
```

The copy instruction is part of a function that passes the address of this copied data as an argument to a decoding function together with the length of the buffer:

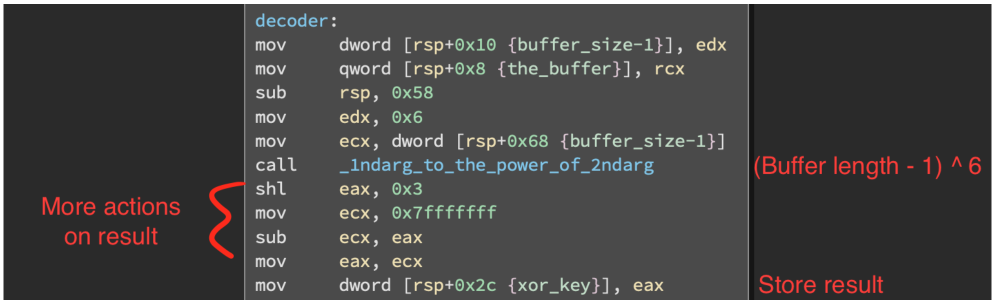
```
sub_140005b70:
mov     dword [rsp+0x20 {arg_20}], r9d
mov     qword [rsp+0x18 {arg_18}], r8
mov     qword [rsp+0x10 {arg_10}], rdx
mov     qword [rsp+0x8 {arg_8}], rcx
push    rsi {__saved_rsi}
push    rdi {__saved_rdi}
sub     rsp, 0x68
mov     ecx, 0x10
call    sub_140006110
mov     qword [rsp+0x38 {var_40}], rax
mov     rax, qword [rsp+0x38 {var_40}]
mov     qword [rsp+0x20 {write_encoded_buffer_to}], rax
mov     rax, qword [rsp+0x20 {write_encoded_buffer_to}]
mov     dword [rax+0x8], 0x1691f  // hard coded buffer size
mov     rax, qword [rsp+0x20 {write_encoded_buffer_to}]
mov     eax, dword [rax+0x8]
mov     ecx, eax
call    sub_1400066a0
mov     qword [rsp+0x40 {var_38}], rax
mov     rax, qword [rsp+0x20 {write_encoded_buffer_to}]
mov     rcx, qword [rsp+0x40 {var_38}]
mov     qword [rax], rcx
mov     rax, qword [rsp+0x20 {write_encoded_buffer_to}]
mov     eax, dword [rax+0x8]
mov     rcx, qword [rsp+0x20 {write_encoded_buffer_to}]
lea     rdx, [rel 0x140050960]
mov     rdi, qword [rcx]
mov     rsi, rdx  {0x140050960}
mov     ecx, eax
rep movsb byte [rdi], [rsi]  {0x0}
mov     rax, qword [rsp+0x20 {write_encoded_buffer_to}]
mov     eax, dword [rax+0x8]
dec     eax
mov     edx, eax  // buffer length argument
mov     rax, qword [rsp+0x20 {write_encoded_buffer_to}]
mov     rcx, qword [rax]  // encoded buffer address value
call    decoder
```

Buffer size

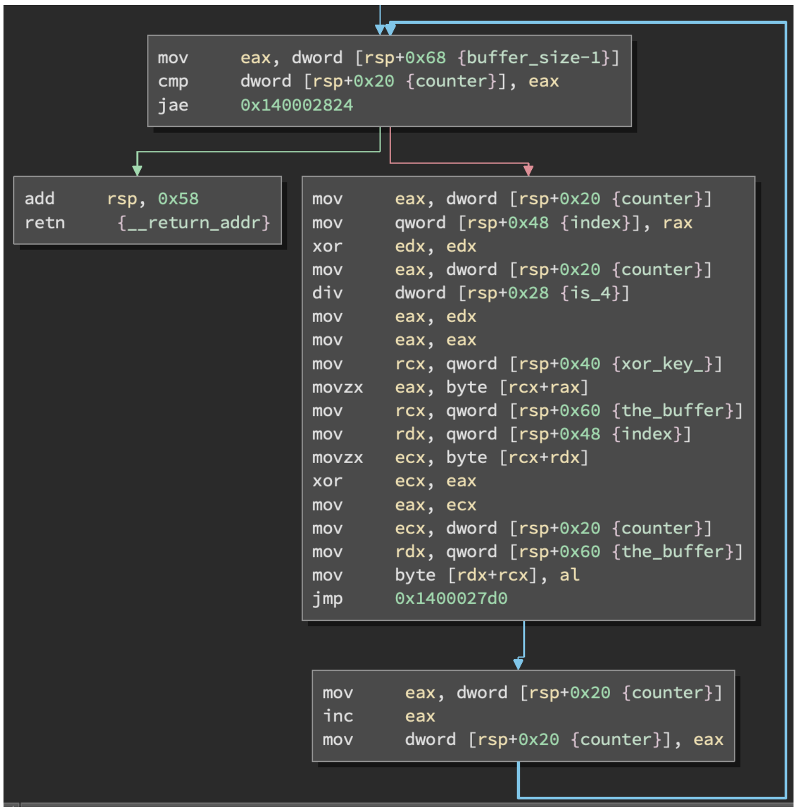
Copy data

Calling decoder

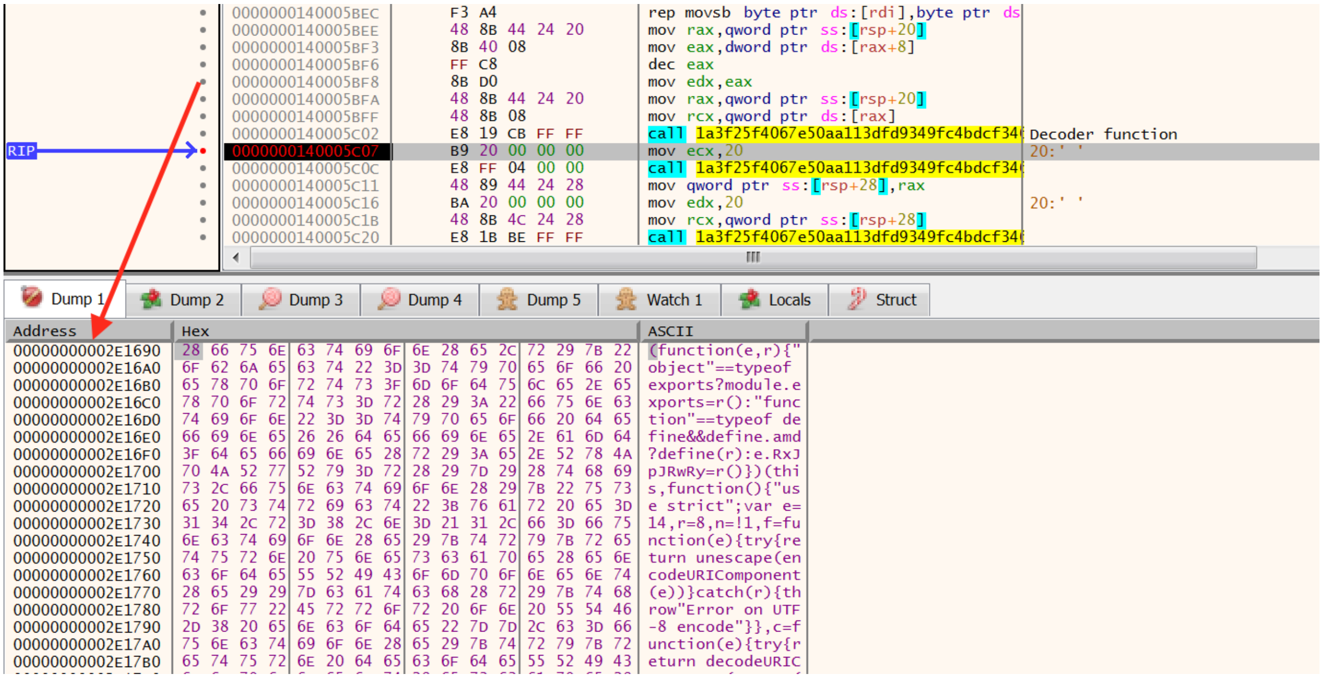
The `decoder` function passes the `buffer length` and another `int` to a function that takes `buffer length` to the power of that `int`. Then a shift and subtraction is performed. The result is the XOR key that is used to decode the buffer.




Later on the decode operation is performed:




That the data actually becomes decoded can be verified with a debugger, watching the memory of the buffer after the decoder function has ran:



With the above research its possible to write an unpacker.

The actions performed by the unpacker:

- Use yara rules to find buffer location buffer length, number of shifts, subtraction value and power to value of it.
- Calculate the buffer RVA as the extracted location is relative to the LEA instruction that references it
- Calculate XOR array based on values extracted with the help of the yara rules
- Extract and decode the script

The sourcecode to do this is available in this github repo.

Recent hashes that it has been confirmed to work on:

```
352b78b8ed38be7ada1d9f4d82352da5015a853bf3c3bdb8982e4977d98f981c
5c548447203104e9a26c355beaf2367a8fa4793a1b0d3668701ee9ba120b9a7b
1a3f25f4067e50aa113dfd9349fc4bdcf346d2e589ed6b4cebbc0a33e9eea50d
```

Example run:

```
crunch:retefe tom$ python3 exporter.py 1a3f25f4067e50aa113dfd9349fc4bdcf346d2e589ed6b4cebbc0a33e9eea50d
Found seed (and buffer size) value 0x1691e
Found power to value 0x6
Found shift left value 0x3
Found subtract value 0x7fffffff
Found buffer place arg 0x4ad90
XOR array that will be used for decryption [255, 189, 120, 68]
Calculated RVA for encoded buffer is 0x50960
Extracted file written to 1a3f25f4067e50aa113dfd9349fc4bdcf346d2e589ed6b4cebbc0a33e9eea50d.extracted
crunch:retefe tom$
crunch:retefe tom$
crunch:retefe tom$ cut -c-100 1a3f25f4067e50aa113dfd9349fc4bdcf346d2e589ed6b4cebbc0a33e9eea50d.extracted
(function(e,r){"object"==typeof exports?module.exports=r():"function"==typeof define&&define.amd?def
crunch:retefe tom$
crunch:retefe tom$
crunch:retefe tom$ 
```

Screenshots in this post are based on the sample `1a3f25f4067e50aa113dfd9349fc4bdcf346d2e589ed6b4cebbc0a33e9eea50d`.