

# GitHub - Tomasuh/retefe-unpacker: Retefe static unpacker

By Tomasuh

Archived: 2026-04-05 16:26:56 UTC

<b>layout</b>	post
<b>title</b>	Retefe unpacker
<b>date</b>	2018-12-28
<b>comments</b>	true
<b>categories</b>	

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

```

Copy data

Buffer size

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.

```

decoder:
mov    dword [rsp+0x10 {buffer_size-1}], edx
mov    qword [rsp+0x8 {the_buffer}], rcx
sub    rsp, 0x58
mov    edx, 0x6
mov    ecx, dword [rsp+0x68 {buffer_size-1}]
call   _1ndarg_to_the_power_of_2ndarg
shl   eax, 0x3
mov    ecx, 0x7fffffff
sub    ecx, eax
mov    eax, ecx
mov    dword [rsp+0x2c {xor_key}], eax

```

More actions on result

$(\text{Buffer length} - 1) \wedge 6$

Store result

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:

Address	Hex	ASCII
00000000002E1690	28 66 75 6E 63 74 69 6F 6E 28 65 2C 72 29 7B 22	{function(e,r){"
00000000002E16A0	6F 62 6A 65 63 74 22 3D 3D 74 79 70 65 6F 66 20	object"==typeof
00000000002E16B0	65 78 70 6F 72 74 73 3F 6D 6F 64 75 6C 65 2E 65	exports?module.e
00000000002E16C0	78 70 6F 72 74 73 3D 72 28 29 3A 22 66 75 6E 63	xports=r():"func
00000000002E16D0	74 69 6F 6E 22 3D 3D 74 79 70 65 6F 66 20 64 65	tion"==typeof de
00000000002E16E0	66 69 6E 65 26 26 64 65 66 69 6E 65 2E 61 6D 64	fine&&define.amd
00000000002E16F0	3F 64 65 66 69 6E 65 28 72 29 3A 65 2E 52 78 4A	?define(r):e.RxJ
00000000002E1700	70 4A 52 77 52 79 3D 72 28 29 7D 29 28 74 68 69	pJRWry=()){thi
00000000002E1710	73 2C 66 75 6E 63 74 69 6F 6E 28 29 7B 22 75 73	s,function(){"us
00000000002E1720	65 20 73 74 72 69 63 74 22 3B 76 61 72 20 65 3D	e strict";var e
00000000002E1730	31 34 2C 72 3D 38 2C 6E 3D 21 31 2C 66 3D 66 75	14,r=8,n=1,f=fun
00000000002E1740	6E 63 74 69 6F 6E 28 65 29 7B 74 72 79 7B 72 65	ction(e){try{re
00000000002E1750	74 75 72 6E 20 75 6E 65 73 63 61 70 65 28 65 6E	turn unescape(en
00000000002E1760	63 6F 64 65 55 52 49 43 6F 6D 70 6F 6E 65 6E 74	codeURIComponent
00000000002E1770	28 65 29 29 7D 63 61 74 63 68 28 72 29 7B 74 68	(e)}catch(r){th
00000000002E1780	72 6F 77 22 45 72 72 6F 72 20 6F 6E 20 55 54 46	row"Error on UTF
00000000002E1790	20 38 20 65 6E 63 6F 64 65 22 7D 7D 2C 63 3D 66	-8 encode"}};c=f
00000000002E17A0	75 6E 63 74 69 6F 6E 28 65 29 7B 74 72 79 7B 72	unction(e){try{r
00000000002E17B0	65 74 75 72 6E 20 64 65 63 6F 64 65 55 52 49 43	eturn decodeURIC

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
1a3f25f4067e50aa113dfd9349fc4bdcf346d2e589ed6b4ceb9c0a33e9eea50d
```

Example run:

```
crunch:retefe tom$ python3 exporter.py 1a3f25f4067e50aa113dfd9349fc4bdcf346d2e589ed6b4ceb9c0a33e9eea50d
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 1a3f25f4067e50aa113dfd9349fc4bdcf346d2e589ed6b4ceb9c0a33e9eea50d.extracted
crunch:retefe tom$
crunch:retefe tom$
crunch:retefe tom$ cut -c-100 1a3f25f4067e50aa113dfd9349fc4bdcf346d2e589ed6b4ceb9c0a33e9eea50d.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

```
1a3f25f4067e50aa113dfd9349fc4bdcf346d2e589ed6b4ceb9c0a33e9eea50d .
```

---

Source: <https://github.com/Tomasuh/retefe-unpacker>