# Knowledge Fragment: Unwrapping Fobber

byte-atlas.blogspot.ch/2015/08/knowledge-fragment-unwrapping-fobber.html

F Functions window		I	- 8 ×
Function name	Segment	Start	Length
f sub_95112A	seg000	0095112A	0000005
f sub_951928	seg000	00951928	0000001
f sub_951C69	seg000	00951C69	0000003
<u>f</u> sub_952B5F	seg000	00952B5F	0000001
<u>f</u> sub_952E3C	seg000	00952E3C	0000001
<u>f</u> sub_952F9E	seg000	00952F9E	0000002
<u>f</u> sub_953213	seg000	00953213	000000C
f sub_953475	seg000	00953475	0000002

About two weeks ago I came across an interesting sample using an interesting anti-analysis pattern.

The anti-analysis technique can be best described as "runtime-only code decryption". This means prior to execution of a function, the code is decrypted, then executed and finally encrypted again, but with a different key.

Malwarebytes has already published an <u>analysis</u> on this family they called "Fobber". However, in this blog post I wanted to share how to "unwrap" the sample's encrypted functions for easier analysis. There is also another blog post detailing how to work around the string encryption.

The sample and code related to this blog post can be found on bitbucket.

#### **Fobber's Function Encryption Scheme**

First off, let's have a look how Fobber looks in memory, visualized by IDA's function analysis:

Functions window         E           Function name         Segment         Stat         Length           sub_95112A         seg000         0095112A         000000           f sub_951928         seg000         00951269         000000           f sub_95265F         seg000         0095285F         0000000           f sub_95287E         seg000         0095283C         0000000           f sub_95287E         seg000         0095283C         0000000           f sub_95287E         seg000         0095287E         0000000           f sub_952879E         seg000         0095287E         0000000           f sub_953213         seg000         00953213         0000000           f sub_953475         seg000         00953475         0000000	
Image: second	
Image: sub_95112A         seg000         0095112A         0000005           Image: sub_951928         seg000         00951928         0000000           Image: sub_951928         seg000         00951928         0000000           Image: sub_951928         seg000         00951928         0000000           Image: sub_951928         seg000         0095285F         0000000           Image: sub_95283C         seg000         0095283C         0000000           Image: sub_95289E         seg000         0095289E         0000000           Image: sub_95213         seg000         0095213         0000000	
f         sub_951928         seg000         00951928         0000001           f         sub_951C69         seg000         00951C69         0000000           f         sub_95285F         seg000         0095285F         0000000           f         sub_95283C         seg000         0095283C         00000000           f         sub_952F9E         seg000         00952F9E         0000000           f         sub_952F9E         seg000         00952F9E         0000000           f         sub_952F13         seg000         00952F32         0000000	
f         sub_951C69         seg000         00951C69         0000000           f         sub_95285F         seg000         0095285F         0000001           f         sub_95283C         seg000         0095283C         0000001           f         sub_952F9E         seg000         00952F9E         000000000           f         sub_952F9E         seg000         00952F9E         0000000           f         sub_95213         seg000         0095213         0000000	
Image: Section of the sectio	
Image: Sub_952E3C         seg000         00952E3C         0000001           Image: Sub_952F9E         seg000         00952F9E         0000002           Image: Sub_953213         seg000         00953213         0000000	
Image: Sub_952F9E         seg000         00952F9E         0000002           Image: Sub_953213         seg000         00953213         0000000	
sub_953213 seg000 00953213 0000000 sub_953475 seg000 00953475 0000000	
j	
×	

IDA's first view on Fobber.

IDA only recognizes a handful of functions. Among these is the actual code decryption routine, as well as some code handling translating relevant addresses of the position independent code into absolute offsets.

Next, a closer look at how the on-demand decryption/encryption of functions works:

seg000:00951125 seg000:00951126 seg000:00951127 seg000:00951128 seg000:00951128	db 7Bh; { db46h db29h; > db 8 db 8	
	; END OF FUNCTION CHUNK FOR	sub_95112A
		U T I N E
seg000:0095112A	sub_95112A proc near	; CODE XREF: sub_952F9E+Cip
seg000:0075112A seg000:0075112A seg000:0075112A seg000:0075112A	; FUNCTION CHUNK AT seg000:0 ; FUNCTION CHUNK AT seg000:0	00951117 SIZE 00000006 BYTES 00951122 SIZE 00000008 BYTES
seg000:0095112A	call decr	yptFunctionCode
seg000:0095112A seg000:0095112F	db 7Ch	
seg000:00951130 seg000:00951131	db 0F1h ; 📓 db 87h	
seg000:00951132	db ØC8h	
seg000:00951133 seg000:00951134	db 10h db 90h ; m	
seg000:00951135	db ØF3h	
seg000:00751136 seg000:00751137	db 25h : ∦ db 0FBh : ⊍ db 0A2h : ≦	
seg000:00951138	db ØA2h ; ≤	
seg000:00951139 seg000:0095113A	db 8Eh db 16h	
seg000:0095113B	db ØF3h	
seg000:0095113C seg000:0095113D	db ØF3h db 3Dh ; = db 20h	
seg000:0095113E	db 78h i k	
seg000:0095113F seg000:00951140	db 62h ; b db 16h	
seg000:00951141	db 81h	
seguuu:00751142	<b>db <mark>9Ch</mark> ; մ</b>	

The Fobber-encrypted function sub\_95112A, starting with call to decryptFunctionCode.

We can see that function sub\_95112A starts with a call to what I renamed "decryptFunctionCode":

₩ ¥ ≌ 60 8B 74 24 20	decryptFunctionCode proc near pusha mov esi, [esp+20h]
	<b></b> • <b>`</b> •
₩ №4 12 66 B8 96 69 ØF BØ 66 F5 75 F6	loc_951C6E: mov ax, 6996h cmpxchg [esi-ØBh], ah jnz short loc_951C6E
₩ №4 ፻ 66 B8 Ø1 ØØ 66 ØF C1 46 F9 66 85 CØ 75 15	mov ax, 1 xadd [esi-7], ax test ax, ax jnz short loc_951C9B
🖬 🕰 📼	
0F B6 46 F8 50 0F B7 46 F6 35 1F 46 00 00 50 56 E8 8D FC FF FF	movzx eax, byte ptr [esi-8] push eax movzx eax, word ptr [esi-0Ah] xor eax, 461Fh push eax push esi call cryptXorCode
🖬 🕰 🖭	
C6 46 F5 96 61 C3	loc_951C9B: mov byte ptr [esi-0Bh], 96h popa retn decryptFunctionCode endp

Fobber's on-demand decryption code for functions, revealing the parameter offsets neccessary for decryption.

This function does not make use of the stack, thus it is surrounded by a simple pushad/popad prologue/epilogue. We can see that some references are made relative to the return address (initially put into esi by copying from [esp+20h]):

- Field [esi-7] contains a flag indicating whether or not the function is already decrypted.
- Field [esi-8h] contains the single byte key for encryption, while
- field [esi-Ah] contains the length of the encrypted function, stored xor'ed with 0x461F.

The actual cryptXorCode takes those values as parameters and then loops over the

encrypted function body, xor'ing with the current key and then updating the key by rotating 3bit and adding 0x53.

	; Attributes: bp-based fram
	cryptXorCode proc near
	arg_0= dword ptr  8 arg_4= dword ptr  0Ch arg_8= dword ptr  10h
55 89 E5	push ebp mov ebp, esp
51 8B 45 10 8B 55 08 8B 4D 0C	push ecx mov eax, [ebp+arg_8] mov edx, [ebp+arg_0] mov ecx, [ebp+arg_4]
	loc 951935:
30 02 ca ca ac	loc_951935: xor [edx], al
CØ C8 Ø3 Ø4 53	xor [edx], al ror al, 3 add al, 53h
CØ C8 Ø3	xor [edx], al ror al, 3
CØ C8 Ø3 Ø4 53 42	xor [edx], al ror al, 3 add al, 53h inc edx
CØ C8 Ø3 Ø4 53 42	xor [edx], al ror al, 3 add al, 53h inc edx
CØ C8 Ø3 Ø4 53 42 E2 F6	xor [edx], al ror al, 3 add al, 53h inc edx

Function for decrypting one function, given the neccessary parameters.

After decryption, our function makes a lot more sense and we can see the default function prologue (push ebp; mov ebp, esp) among other things.

seg000:00951125	db 7Bh	
seg000:00951126	db 46h	
seg000:00951127	db 29h ; )	
seg000:00951128	db 1	
seg000:00951129	db Ø	
seg000:0095112A		
	C II D D O	U T I N E
seg000:0095112A		01102
seg000:0095112A		
seg000:0095112A sub_95112A	proc near	; CODE XREF: seg000:009505FE1p
seg000:0095112A		; sub_952F9E+C1p
seg000:0095112A		
seg000:0095112A ; FUNCTION CH	UNK AT seg000:	00951117 SIZE 00000008 BYTES
seg000:0095112A		
seg000:0095112A	call dec	ryptFunctionCode
seg000:0095112F	push ebp	
seg000:00951130	nov ebu	, esp
seg000:00951132	push edi	
seg000:00951133	push esi	
seg000:00951134		, [ebp+0Ch]
seg000:00951137		. 11h
seg000:0095113A	push eax	
seg000:0095113B		951ACB
seg000:00951140		, cax
seg000:00951142		, [ebp+8]
seg000:00951145		, eax
seg000:00951147	nov cax	, [ebp+0Ch]
seg000:0095114A		, ODh
seg000:0095114D	stosd	
seg000:0095114E	nov cax	, [ebx+3F6D98Ah]
seg000:00951154	stosd	
seg000:00951154 sub_95112A	endp ; sp-a	nalysis failed
seg000:00951154		
2 zeg888:88951155	nov eax	, [ebx+3F6D98Eh]
2 pog866:8695115B	stosd	
RegRIA: 0095115C		, [ebp+10h]
acg808:8095115F	stosb	

The decrypted equivalent of function sub\_95112A, revealing some "real" code.

Also note the parameters:

- 0x951125 key: 0x7B
- 0x951126 length: 0x4629^0x461F -> 0x36 bytes
- 0x951128 encryption flag: 0x01

So far so good. Now let's decrypt all of those functions automatically.

### **Decrypt All The Things**

First, we want to find our decryption function. For all Fobber samples I looked at, the regex r"\x60\x8B.\x24\x20\x66" was delivering unique results for locating the decryption function.

Next, we want to find all calls to this decryption function. For this we can use the regex r"\xE8" to find all potential "call rel\_offset" instructions.

Then we just need to do some address math and check if the call destination (calculated as: image\_base + call\_origin + relative\_call\_offset + 5) is equal to the address of our decryption function.

Should this be the case, we can extract the parameters as described above and decrypt the code.

We then only need to exchange the respective bytes in our binary with the decrypted bytes. In the following code I also set the decryption flag and fix the function ending with a "retn" (0xC3) instruction to ease IDA's job of identifying functions afterwards. Otherwise, rinse/repeat until all functions are decrypted.

Code:

```
#!/usr/bin/env python
import re
import struct
def decrypt(buf, key):
  decrypted = ""
  for char in buf:
     decrypted += chr(ord(char) ^ key)
     # rotate 3 bits
     key = ((key >> 3) | (key << (8 - 3))) & 0xFF
     key = (key + 0x53) \& 0xFF
  return decrypted
def replace bytes(buf, offset, bytes):
  return buf[:offset] + bytes + buf[offset + len(bytes):]
def decrypt all(binary, image base):
  # locate decryption function
  decrypt function offset = re.search(r"x60x8B.x24x20x66", binary).start()
  # locate all calls to decryption function
  regex call = r'' \approx 8(P < rel call > \{4\})''
  for match in re.finditer(regex call, binary):
     call origin = match.start()
     packed call = binary[call origin + 1:call origin + 1 + 4]
     rel call = struct.unpack("I", packed call)[0]
     call destination = (image base + call origin + rel call + 5) & 0xFFFFFFF
     if call destination == image base + decrypt function offset:
       # decrypt function and replace/fix
       decrypted flag = ord(binary[call origin - 0x2])
       if decrypted flag == 0x0:
          key = ord(binary[call origin - 0x3])
          size = struct.unpack("H", binary[call origin - 0x5:call origin - 0x3])[0] ^ 0x461F
          buf = binary[call origin + 0x5:call origin + 0x5 + size]
          decrypted function = decrypt(buf, key)
          binary = replace bytes(binary, call origin + 0x5, decrypted function)
          binary = replace bytes(binary, call origin + len(decrypted function), "\xC3")
```

```
binary = replace_bytes(binary, call_origin - 0x2, "\x01")
return binary
```

## [...]

#### IDA likes this this already better:

Functions window	🗆 🗗 🗙 💽	] IDA Yiew-A
Function name	Segment	
7 SUD_95055E	seguto	
f sub_950781	seg000	
f sub_9507F8	seg000	
f sub_950837	seg000	
1 sub_95089A	seg000	
f sub_9508D0	seg000	
f sub_950945	seg000	
f sub_950806	seg000	
f sub_950875	seg000	
f sub_950DSA 7 sub_950E0E	seg000	
	seg000	
f_sub_95105C f_sub_9510DB	seg000	
f sub_951066	seg000	
f sub_95112A	seg000	
	seg000 seg000	e1
f sub_951489 f sub_9514E8	seg000	
f sub_951507	seg000	
f sub_951515	seg000	
7 sub_95153E	seg000	
1 sub_951571	seg000	
f sub_951778	seg000	
f sub_951847	seg000	
f sub_951882	seg000	
f sub_951928	seg000	
f sub_95194B	seg000	
7 encryptCode	seg000	
/ sub_951ACB	seg000	-
f sub_951AE2	seg000	
f sub_951818	seg000	
f decryptCode	seg000	
f sub_951CA7	seg000	
f sub_951CFD	seg000	
7 sub_951E50	seg000	
7 sub_951E6A	seg000	
f sub_951EC8	seg000	
f sub_9524F1	seg000	
f sub_952589	seg000	
f cryptRc4	seg000	
7 sub_952680	seg000	
7 sub_9526C7	seg000	
	· •	

IDA's view on a code-decrypted Fobber sample.

However, we are not quite done yet, as IDA still barfs on a couple of functions.

### Conclusion

After decrypting all functions, we can already start analyzing the sample effectively. But we are not quite done yet, and the <u>second post</u> looks closer at the inline usage of encrypted strings.

sample used:

md5: 49974f869f8f5d32620685bc1818c957 sha256: 93508580e84d3291f55a1f2cb15f27666238add9831fd20736a3c5e6a73a2cb4

<u>Repository with memdump + extraction code</u>