Dissecting CoreBot, From Loader to Config

Jason Reaves

December 10, 2015

Abstract

While analyzing the CoreBot malware I hope to show my work reverse engineering a number of the routines used by Corebot as well as break down most of the encryption used to conceal various portions of data used by the bot. By doing so we can hope to not only help security researchers gain insight into the workings of this bot but also help incident response personnel with being able to understand the bot in hopes that they can better detect or prevent infections in their respective environments.

Keywords - Reverse Engineering, Malware Analysis, CoreBot, Banking Trojan

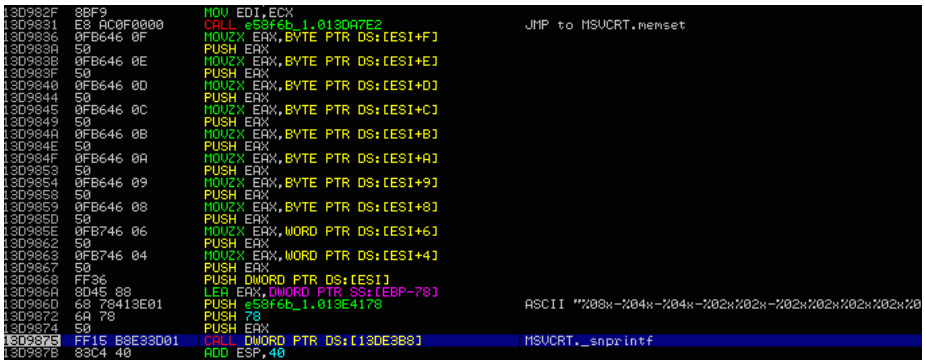

Figure 1: FolderName Generate

```
: 05 06 07 08 09 0A 0B 0C 0D 0E 0F
  C4 19 DC 61 82 0C 68 32 BE 7A 56   :Š*æ.Ä.Üa‚.h2¾zV
  1D 5F 92 2B EA 62 53 A4 42 09 DD   "Ò}XŒ._'+êbS¤B.Ý
  FE A6 81 E4 92 30 68 9A B9 36 19   8Tü}™þ¦.ä'0hš¹6.
  39 FD 11 58 30 85 9F B3 F8 2F 9C   1øÀZ\9ý.XO…Ÿ³ø/œ
  7A 27 E3 82 57 ED E0 59 79 EF DA   ‹Þ8€.z'ã‚WíàYyïÚ
  7F F7 B7 3B B3 3E EF 73 21 03 B6   i6ï'Ì.÷·;³>ïs!.¶
  25 DD C4 59 2A 9F 35 D7 56 1C D5   o±Ýå¾%ÝÄY*Ÿ5×V.Õ
  E7 45 C7 2A C8 1F E1 38 50 AA A7   ÷Æ.ßQçEÇ*È.á8P³§
  0C 1D 1B 30 C5 B1 E3 8D 4E CB 44   j=ò„"...0Å±ã.NËD
  52 CD 9F 3E 02 6F 19 B0 91 9F 89   ÊâKñ.RÍŸ>.o.°'Ÿ‰
  B9 45 77 11 26 5E 11 E9 2B D9 2A   XZ³Äf¹Ew.&^.é+Ù*
```

Figure 2: Encrypted File

1 Introduction

Corebot is a relatively new bot commonly found being delivered by a newer Kovter variant at the time of this writing. The goal of Corebot appears to be in data and credential theft, coming with modules commonly found in most of the bigger banking trojans such as a keylogger module, a stealer(pony) module and a module for performing mitm with the browser in order to facilitate credential theft for targeted institutions found in a zeus-like config.

2 Loader

Most of the relevant data stored on disk was found in `APPDATA\local\Microsoft` in a series of sub folders. The subfolder names are generated by XORing the computers volume serial number against a few sets of 16 byte hex strings and

```
from Crypto.Cipher import AES
import binascii
import sys
#AES key for main corebot component
#11bff5d5cfad69e638e73f00e35a2b3c77ab5710692c42dfe4e5e7aa6e54a75e
#AES key for corebot modules
#1473c09a3ce86f1f5d2bf8e69e5eec2598e380d740cc68a7c58019b9c5799388


data = open(sys.argv[1],'rb').read()

key = binascii.unhexlify('11bff5d5cfad69e638e73f00e35a2b3c77ab5710692c42dfe4e5e7
#Modules
#key = binascii.unhexlify('1473c09a3ce86f1f5d2bf8e69e5eec2598e380d740cc68a7c5801

iv = binascii.unhexlify('00'*16)

aes = AES.new(key, AES.MODE_CBC, iv)

decrypted = aes.decrypt(data)

open(sys.argv[1]+'.decr', 'wb').write(decrypted)
```

Figure 3: CoreBot component decryption

then sprintf the hex data into the format string `%08x-%04x-%04x-%02x%02x-%02x%02x%02x%02x%02x%02x` (Figure 1). The loader finds the encrypted corebot module on disk by recreating the pathnames, at this point it decrypts the module using an hardcoded AES key and a zeroed 16 byte IV(Figure 2). A POC for decrypting both the main CoreBot component and its modules can be seen in Figure 3. This main module is spun up into a hollowed out svchost.exe.

3 Main Module

The main module of CoreBot uses a similar routine for path names, registry keys and encryption keys. I've pulled out some of the hardcoded hex data used to generate this data and written a small proof of concept for some of the items in Figure 4, in the script you can see the hardcoded hex data used to produce multiple folder names, file names, the registry persistence key and the RC4 key used to encrypt the CoreBot settings file. The settings file is constantly updated during CoreBots activity, it's not only used by the loader, the main CoreBot module but also used by the downloaded modules themselves.

4 Watchdog

The svchost.exe spins up a dllhost.exe and connects to a named pipe 'core_ps', the process writes it's processid to the named pipe while the watchdog process also communicates with a heartbeat pipe(Figure 5). Should the watchdog not receive a proper response in the proper time frame then it will interrupt the main process tearing it down and then spinning up the loader again. Using WinDBGs child process ability however we can attach in during the loader process, finding relevant code sections that look interesting for exploration in IDA we can set our breakpoints and then pause the watchdog to easily circumvent this system.

5 Modules

The decrypted modules can be seen in Figure 6. All modules communicate with the main CoreBot component over the core_ps pipe, the stealer component works similar to Pony malware except that it packages up all the data and sends it to the main component for delivery to the C2. The keylogger will log keystrokes to a text file which gets GZIP compressed and also sent back for delivery to the C2. The mitm module is a bit more complex in that it has references in the settings to it's own config and an interval on when it should be updated.

6 Settings

The settings file stored on disk is in the core.work_dir which is the same directory where the encrypted main module is kept. To generate the RC4 key needed to decrypt this file you take the hardcoded hex data stream and the volume serial number(VSN). The VSN will be XORd with the hardcoded dword 0x42ab3122 and then this value will be XORd against every dword in the hardcoded hex stream while rolling the key to the right 2 bytes every iteration. A proof of concept script for decrypting a settings fail can be found in Figure 7. In this decrypted settings file you can find the RC4 key used to decrypt C2 traffic which is called core.server_key.

7 C2 Traffic

C2 traffic is encrypted with RC4 using the core.server_key from the settings file(Figure 8). Aside from checkins and messages there are a few big areas that most people will be interested in when it comes to banking trojans, the modules and the config(Figure 9). In both cases the data is RC4 encrypted on top of custom structured data, in the case of the modules there is extra header data on top describing the module(Figure 10). In both cases however it boils down to the same structure of a single byte, dword and blob of zlib compressed data(Figure 11).
Going through getting the config from the C2 traffic, we start with the encrypted

data(Figure 12) from there we simply RC4 decrypt the data and then pull out the length of the compressed blob that follows as seen in Figure 13. A quick decompress and we can see the beginnings of an unparsed config(Figure 14).

8 Conclusions

Sample SHA256: 781c6743230918c591b20121ab34ff639968ac14c958f6207effbba465b71ee0

8.1 Signatures

ET: ET TROJAN Corebot Checkin

Yara:

```
rule CoreBot_Scan_Mem {
strings:
    $s0 = "core.dga"
    $s1 = "core.server_key"
    $s2 = "core_ps"
    $m0 = "keylogger.dll"
    $m1 = "stealer.dll"
    $m2 = "m3.dll"
    $m3 = "mitm"
condition:
    any of ($s*) and any of ($m*)
}
```

References

[1] Hex-Rays Decompiler, http://www.hex-rays.com/products/decompiler/index.shtml.

[2] Python, https://www.python.org/

[3] Python and Bitwise Rotation, http://www.falatic.com/index.php/108/python-and-bitwise-rotation

[4] Emerging Threats, http://www.emergingthreats.net/

```
import ctypes
import binascii
import struct

filekey1 = binascii.unhexlify('74c16ccc7a2928459911b21a2fd4d8be')
regkey1 = binascii.unhexlify('3aa7af21467cf0429694560492b893a0')
folderkey1 = binascii.unhexlify('40cd38f54df3ec43854bccb8b9931c48')
folderkey2 = binascii.unhexlify('b94afd7ee333414399d916b60956beaa')
rc4key = binascii.unhexlify('64306743d52625f694d4e2d3654d63d81beac3a4a4bd4675')

def gen_name(vsn, fkey):
        (f1, f2, f3, f4) = struct.unpack('<IIII', fkey)
        new = struct.pack('<IIII', f1^vsn, f2^rol(vsn, 2), f3^rol(vsn,4), f4^rol
        print "%08x-%04x-%04x-%02x%02x-%02x%02x%02X%02X%02x%02x" % struct.unpack

def gen_key(vsn, hkey):
        temp = vsn ^ int('42ab3122',16)
        (f1, f2, f3, f4, f5, f6) = struct.unpack('<IIIIII', hkey)
        new = struct.pack('<IIIIII', f1^temp, f2^ror(temp, 2), f3^ror(temp,4), f
        print(binascii.hexlify(new))


rol = lambda val, r_bits, max_bits=32: \
(val << r_bits%max_bits) & (2 ** max_bits-1) | \
((val & (2**max_bits-1)) >> (max_bits-(r_bits%max_bits)))

ror = lambda val, r_bits, max_bits=32: \
((val & (2**max_bits-1)) >> r_bits%max_bits) | \
(val << (max_bits-(r_bits%max_bits)) & (2**max_bits-1))

kernel32 = ctypes.windll.kernel32

volumesn = ctypes.c_ulong()
kernel32.GetVolumeInformationW(ctypes.c_wchar_p("C:\\"), None, 0, ctypes.byref(v

vsn = volumesn.value

gen_name(vsn, folderkey1)
gen_name(vsn, filekey1)
gen_key(vsn, rc4key)
gen_name(vsn, folderkey2)
gen_name(vsn, regkey1)
```

Figure 4: CoreBot Folder/FileName and Key Generation POC

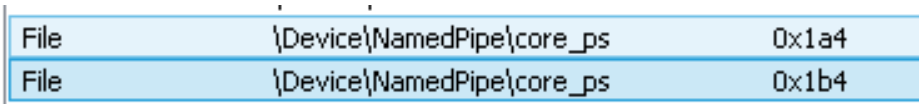

Figure 5: Main named pipe

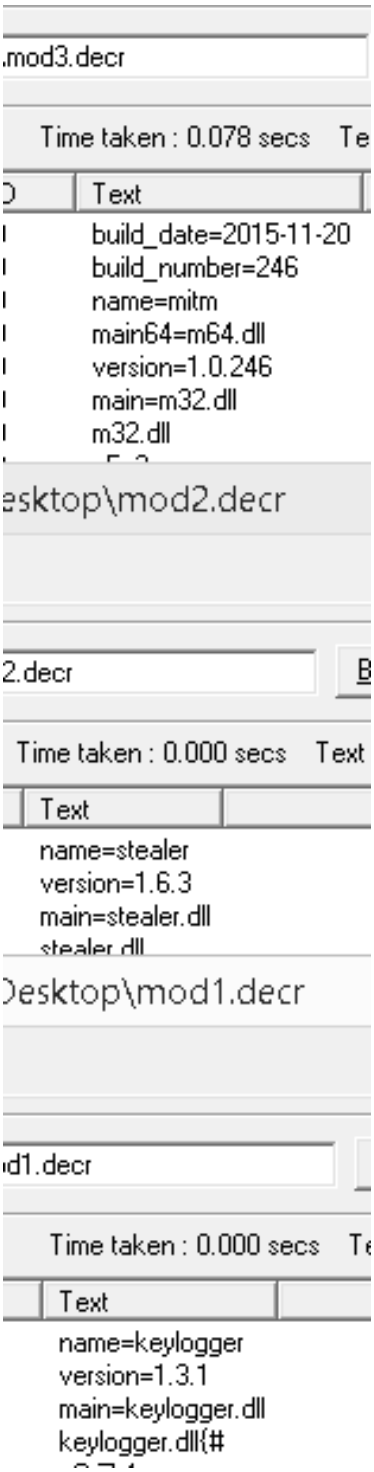

Figure 6: CoreBot Modules

```
import ctypes
from Crypto.Cipher import ARC4
import binascii
import struct
import sys

rc4key = binascii.unhexlify('64306743d52625f694d4e2d3654d63d81beac3a4a4bd4675')

def gen_key(vsn, hkey):
        temp = vsn ^ int('42ab3122',16)
        (f1, f2, f3, f4, f5, f6) = struct.unpack('<IIIIII', hkey)
        new = struct.pack('<IIIIII', f1^temp, f2^ror(temp, 2), f3^ror(temp,4), f
        return(new)

ror = lambda val, r_bits, max_bits=32: \
((val & (2**max_bits-1)) >> r_bits%max_bits) | \
(val << (max_bits-(r_bits%max_bits)) & (2**max_bits-1))


kernel32 = ctypes.windll.kernel32

volumesn = ctypes.c_ulong()
kernel32.GetVolumeInformationW(ctypes.c_wchar_p("C:\\"), None, 0, ctypes.byref(v

vsn = volumesn.value

key = gen_key(vsn, rc4key)

rc4 = ARC4.new(key)
data = open(sys.argv[1], 'rb').read()
open(sys.argv[1]+'.decr','wb').write(rc4.decrypt(data))
```

Figure 7: CoreBot Settings File Decryption POC

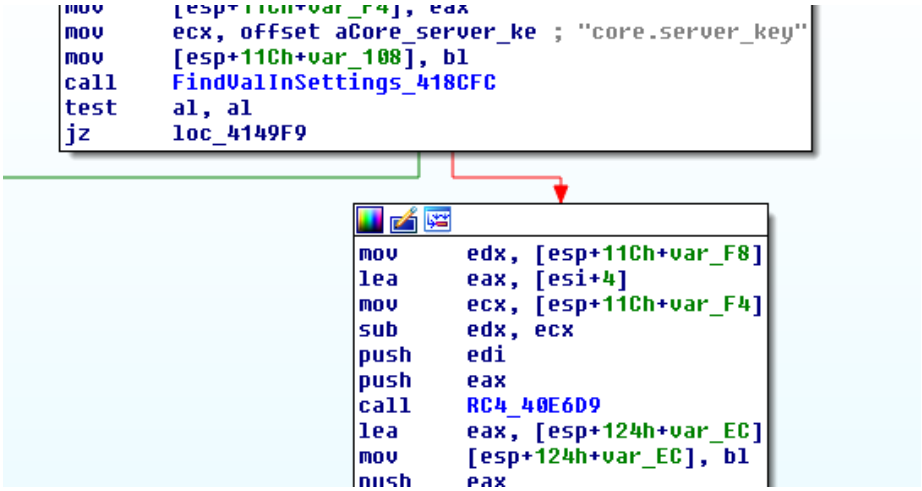

Figure 8: CoreBot C2 Data Process

haloadoxy.com	/gate/	61,359	r
haloadoxy.com	/gate/	0	
haloadoxy.com	/gate/	0	
haloadoxy.com	/gate/	166	r
haloadoxy.com	/gate/	5	r
haloadoxy.com	/gate/	5	r
haloadoxy.com	/gate/	5	r
haloadoxy.com	/gate/	5	r
haloadoxy.com	/gate/	9,184	r
haloadoxy.com	/gate/	0	
haloadoxy.com	/gate/	0	
haloadoxy.com	/gate/	110	r
haloadoxy.com	/gate/	5	r
haloadoxy.com	/gate/	5	r
haloadoxy.com	/gate/	5	r
haloadoxy.com	/gate/	564,533	r
haloadoxy.com	/gate/	5	r
haloadoxy.com	/gate/	5	r
haloadoxy.com	/gate/	14	r
haloadoxy.com	/gate/	5	r
haloadoxy.com	/gate/	377,322	r

Figure 9: CoreBot Traffic

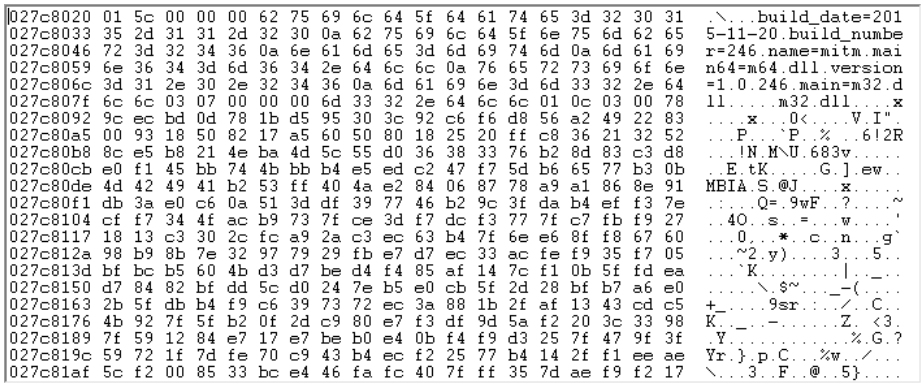

Figure 10: CoreBot Decrypted Module

Figure 11: CoreBot Data Header

Figure 12: CoreBot Encrypted Config

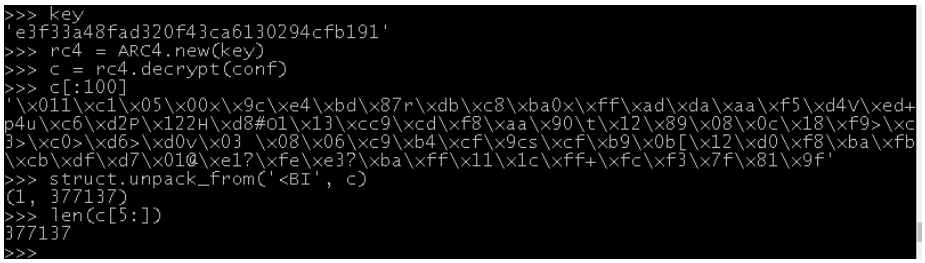

Figure 13: CoreBot Decrypted Config Data Structure

Figure 14: CoreBot Decompressed Config