Flare-On 10 Challenge 8: AmongRust

By Chuong Dong (@cPeterr)

Overview

The file `infector.exe` is a Windows executable that executes and terminates silently. After the executable is run, 64-bit executable files in the user folder appear to be infected with their icons changed to an Among Us image.

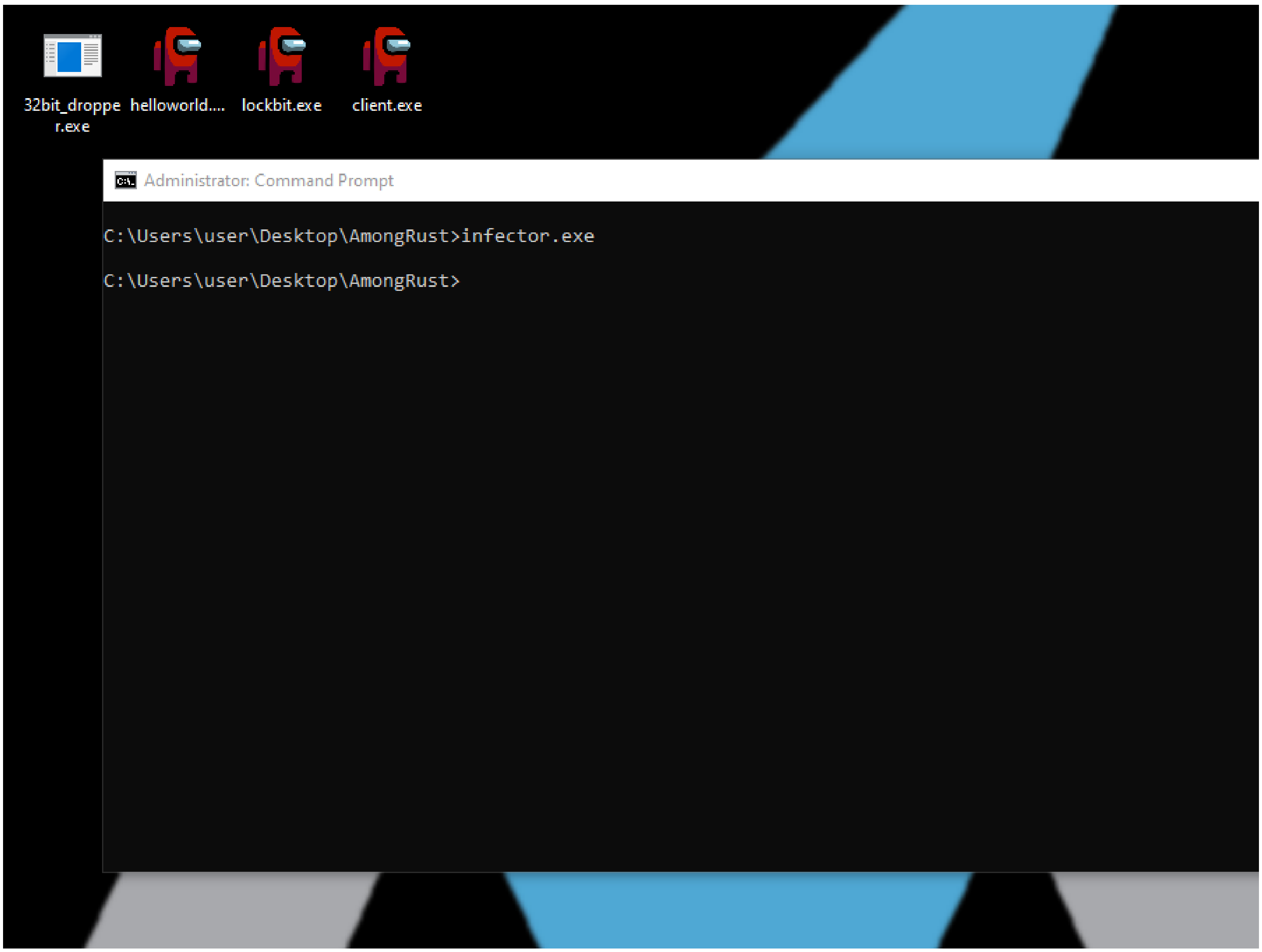

Figure 1: Executing Infector

When running in the Command Prompt, most infected executables print the following link.

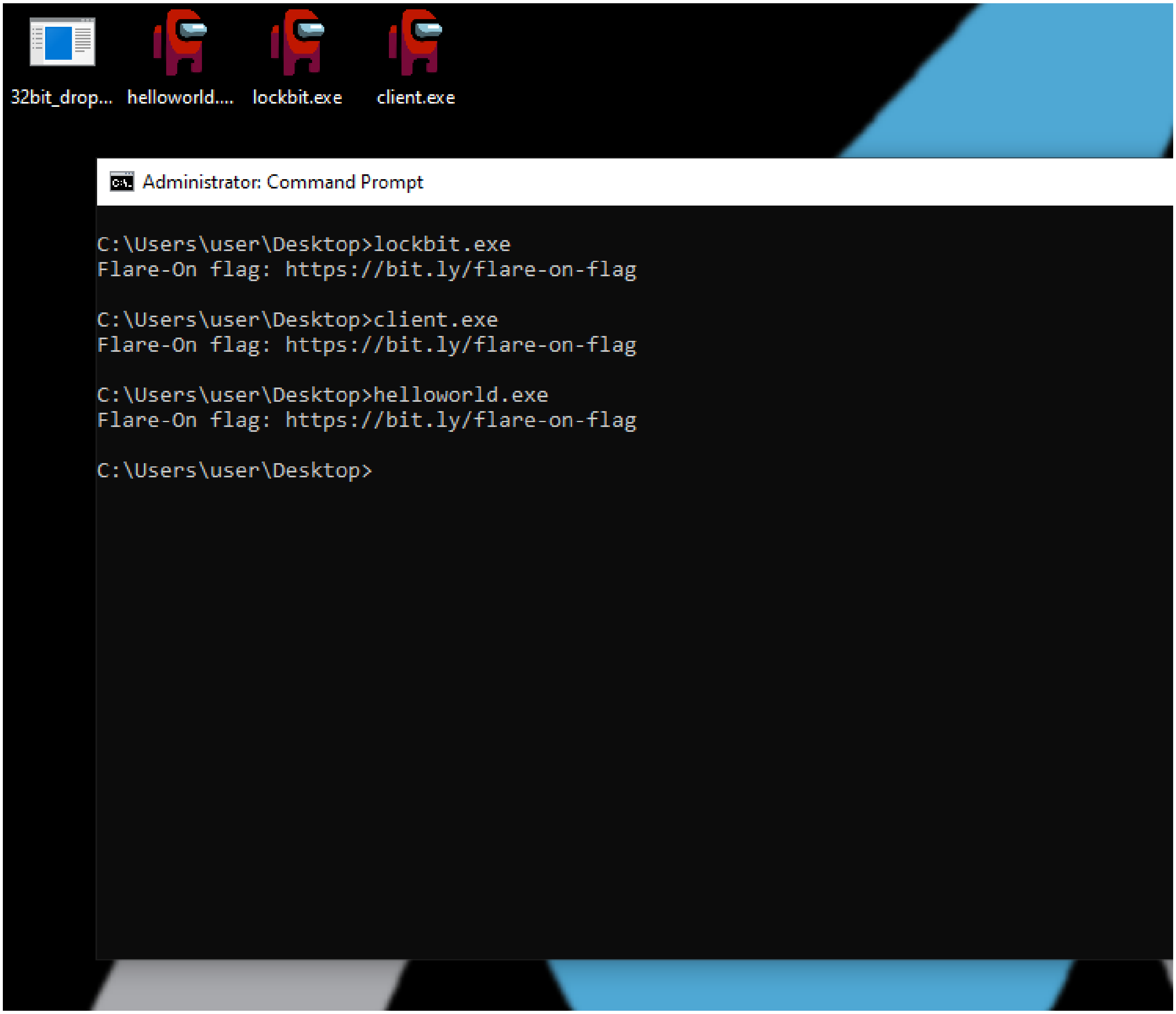

Figure 2: Executing Infected Files

The link, of course, just rickrolls whoever clicks on them instead of giving them the flag.

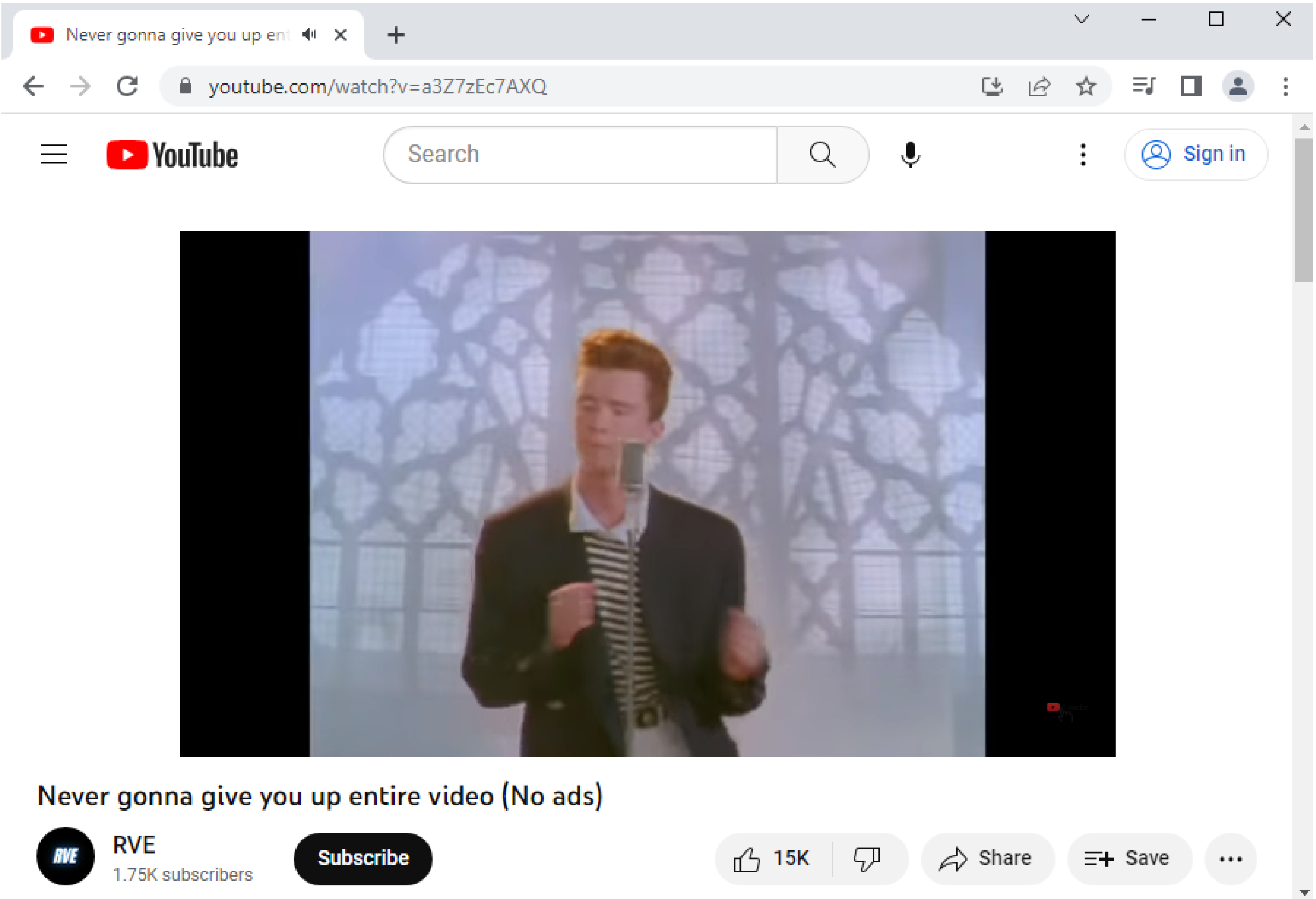

Figure 3: Rickrolling

Infector Basic Analysis

In the spirit of Among Us, the game this challenge is inspired by, the player is meant to search for an "imposter" executable that is infected with a different payload from the rest. There are different ways to accomplish this. One way is to analyze the 64-bit Rust infector executable statically in IDA/Ghidra and check to see how 64-bit executables on the machine are infected. If going this route, the player will find that the payloads are XOR-encoded with the key "`@cPeterr`" to prevent them being carved out directly from the infector executable. The infector utilizes multithreading to walk through directories in the user folder to find all 64-bit executables and randomly infects one of them with the second stage payload.

The method above is left as an exercise for the reader as fully analyzing this statically is not the intended solution for this first stage. A much simpler approach would be to assume that the infector, as most dropper malware, must somehow execute the second stage executable after infecting it. With this in mind, we can set up ProcMon to monitor the Process Create operations performed by the infector.

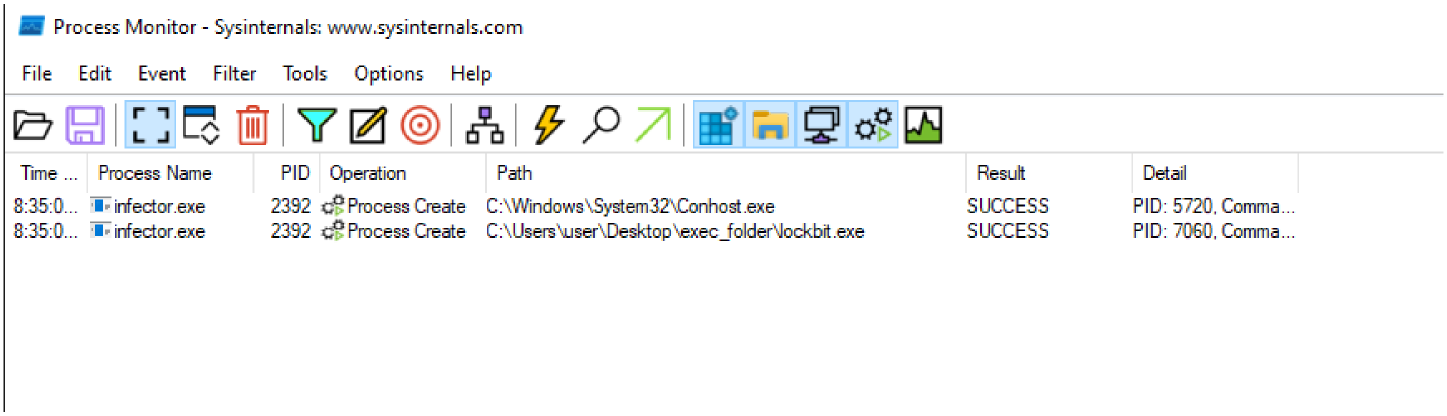

Figure 4: Finding Second Stage Executable with ProcMon

From the ProcMon result, we see that out of all the infected executables, the malware only creates a process to launch one of them. This is how we can find our second stage executable. An even simpler solution to find this is through Task Manager. The list of background processes contains a very suspicious-looking process whose properties also point us to the infected executable.

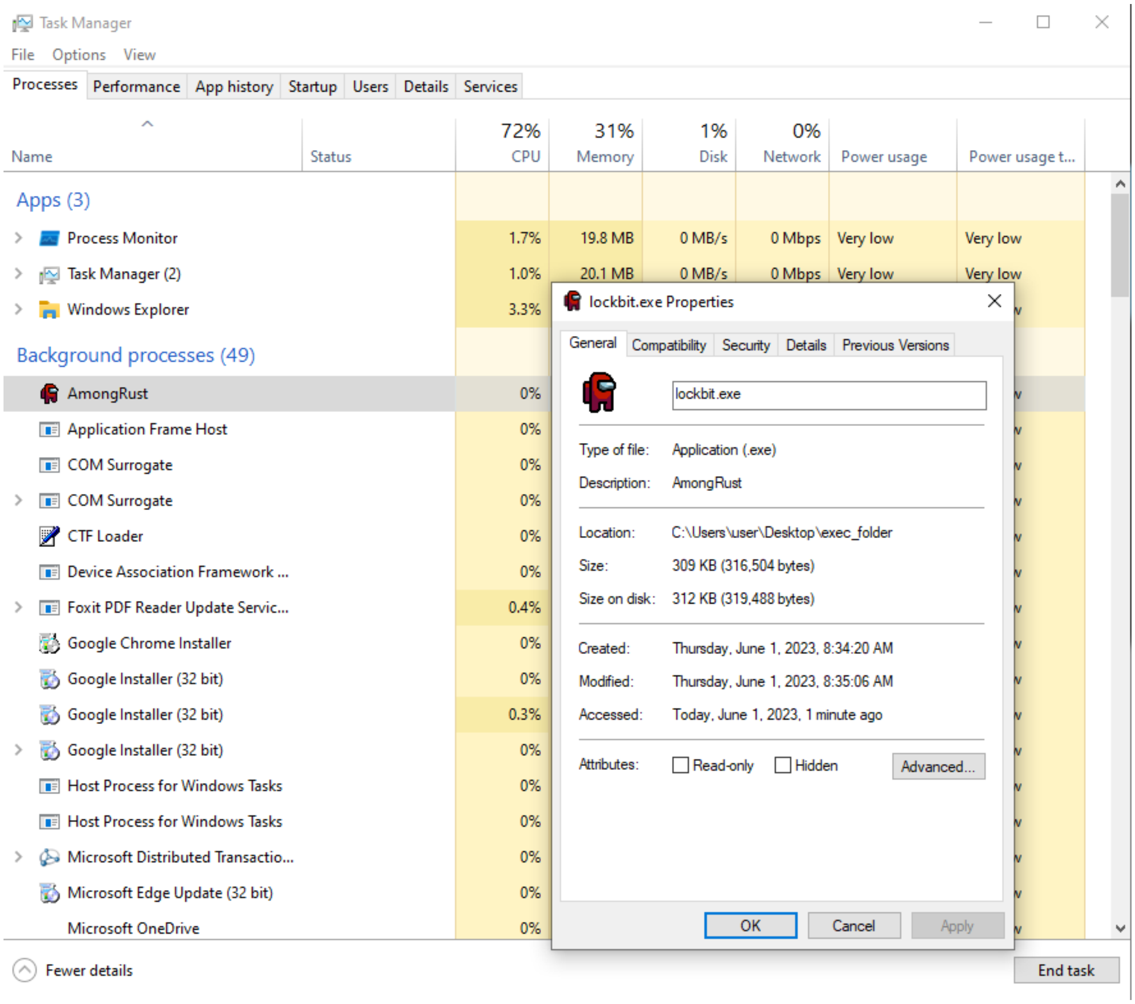

Figure 5: Finding Second Stage Executable with Task Manager

Second Stage Analysis

As seen in Task Manager, the AmongRust application just runs silently in the background. Once the infected file is located, we can begin analyzing it in IDA. A quick look in IDA will show us that similar to the infector, the malware is a 64-bit executable written in Rust. It sets up a TCP server on the host machine on port 8345. For each connection established with a client, the malware spawns a separate thread to handle it.

```
int sub_1400038D0()
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  v14[1] = 0xFFFFFFFFFFFFFFFEi64;
  server_bind(&v5 + 0x10000000Bi64, "0.0.0.0:8345Could not bind", 0xCi64);
  if ( hObject )                                // bind to host IP on port 8345
  {
    hObject = v13;
    core::result::unwrap_failed::hd18e5b485cc9c5ed("Could not bind", 0xE, &hObject, &o
  }
  server_socket = v13;
  server_socket_1 = sub_1400134B0(&server_socket);
  while ( 1 )
  {
    accept_connection(&v6, &server_socket_1);   // spin & accept client connected
    if ( v6 == 2 )
      break;
    if ( v6 )
    {
      if ( (v7 & 3) == 1 )
      {
        v9 = (v7 - 1);
        v0 = *(v7 - 1);
        v10 = v7;
        (**(v7 + 7))(v0);
        v1 = *(v10 + 7);
        v2 = *(v1 + 8);
        v3 = v9;
        if ( v2 )
          j___rdl_dealloc(*v9, v2, *(v1 + 0x10));
        j___rdl_dealloc(v3, 0x18i64, 8i64);
      }
    }
    else
    {
      spawn_thread_to_handle_client(&hObject, v7);// spawn thread to handle
      CloseHandle(hObject);
      if ( !_InterlockedDecrement64(v13) )
        sub_140006520(&v13);
      if ( !_InterlockedDecrement64(v14[0]) )
        sub_1400065D0(v14);
    }
  }
  return closesocket(server_socket);
}
```

Figure 6: TCP Server Setup

In the function to handle the client's communication, the malware's server thread first receives a 32-byte key and a 32-byte nonce from the client. The server sends back "`ACK_K\r`" after receiving the key and "`ACK_N\r`" after receiving the nonce. This implies that the key and nonce are potentially used to encrypt network communication with a symmetric cryptographic algorithm.

```
memset(v140, 0, sizeof(v140));
v194 = 0;
w_recv(v143, &client_socket, key_buffer, 0x20i64);// receive 32-byte key
v1 = *(&v143[0] + 1);
if ( !*&v143[0] )
{
  if ( *(&v143[0] + 1) != 0x20i64 )
  {
    invalid_key_size_str = j___rdl_alloc(0x10ui64, 1ui64);
    if...
    v3 = invalid_key_size_str;
    *invalid_key_size_str = *"Invalid key sizeInvalid nonce sizeCan't parse response into stringsrc\\server.rs";
    v184 = invalid_key_size_str;
    v183 = 0x10i64;
    v194 = 1;
    w_send(v143, &client_socket, invalid_key_size_str, 0x10i64);// send "Invalid key size"
    v1 = *&v143[0];
    if...
    v4 = 0x10i64;
    goto LABEL_20;
  }
  ACK_K_str = j___rdl_alloc(6ui64, 1ui64);
  if...
  v6 = ACK_K_str;
  *(ACK_K_str + 2) = '\rK';
  *ACK_K_str = '_KCA';
  v184 = ACK_K_str;
  v183 = 6i64;
  v194 = 1;
  v4 = 6i64;
  w_send(v143, &client_socket, ACK_K_str, 6i64);// server send "ACK_K\r"
  if ( *&v143[0] )
```

Figure 7: Server Receiving Key

```
w_recv(v143, &client_socket, nonce_buffer, 0x20i64);// receive 32-byte nonce
v1 = *(&v143[0] + 1);
if ( !*&v143[0] )
{
  if ( *(&v143[0] + 1) != 0x20i64 )
  {
    v7 = j___rdl_alloc(0x12ui64, 1ui64);
    if...
    v2 = v7;
    *v7 = *"Invalid nonce sizeCan't parse response into stringsrc\\server.rs";
    *(v7 + 8) = 0x657A;
    j___rdl_dealloc(v6, 6i64, 1i64);
    v184 = v2;
    v183 = 0x12i64;
    v194 = 1;
    w_send(v143, &client_socket, v2, 0x12i64);// send "Invalid nonce size"
    v1 = *&v143[0];
    if ( *&v143[0] )
      v1 = *(&v143[0] + 1);
    v4 = 0x12i64;
    goto LABEL_20;
  }
  v8 = j___rdl_alloc(6ui64, 1ui64);
  if...
  v2 = v8;
  *(v8 + 2) = '\rN';
  *v8 = '_KCA';
  v4 = 6i64;
  j___rdl_dealloc(v6, 6i64, 1i64);
  v184 = v2;
  v183 = 6i64;
  v194 = 1;
  w_send(v143, &client_socket, v2, 6i64); // send "ACK_N\r"
  if ( *&v143[0] )
```

Figure 8: Server Receiving Nonce

Next, the server spins and waits to receive commands from the client. The server can accept three different commands: `exit`, `exec`, and `upload`.

```
INVALID_COMMAND:
            w_send(v143, &client_socket, "Invalid backdoor command\r\nNot a valid size", 26i64);
            v3 = v185;
            if ( !*&v143[0] )
              goto LABEL_37;
            v1 = *(&v143[0] + 1);
            goto LABEL_281;
          }
          if ( **(&backdoor_command + 1) == 'tixe' )// exit
          {
            v1 = 0i64;
            goto LABEL_281;
          }
          if ( *(&v151[1] + 7) < 5ui64 )
            goto INVALID_COMMAND;
          if ( !(**(&backdoor_command + 1) ^ 'cexe' | *(*(&backdoor_command + 1) + 4i64) ^ 0x20) )
            break;                              // exec
          if ( *(&v151[1] + 7) < 7ui64
            || **(&backdoor_command + 1) ^ 'olpu' | *(*(&backdoor_command + 1) + 3i64) ^ ' dao' )
          {                                     // upload
            goto INVALID_COMMAND;
          }
          w_string_clone(&v152, &backdoor_command);
```

Figure 9: Supported Backdoor Commands

The `exec` command is followed by a Windows command to execute on the system. The malware crafts the string "`cmd /c` *<Windows command>*", creates a process to execute it, and sends the output back to the client.

```
w_memcpy(received_buffer, "cmd/Cfailed to execute process", 3i64);
memcpy(v151, received_buffer, sizeof(v151));
w_concat(v151, "/Cfailed to execute process", 2i64);
w_concat(v151, v19, v17);
create_process_to_execute(received_buffer, v151);
if ( !*(&received_buffer[0] + 1) )
{
  *&v175 = *&received_buffer[0];
  core::result::unwrap_failed::hd18e5b485cc9c5ed(
    "failed to execute process",
    25,
    &v175,
    &off_140038568,
    &off_1400386F0);
}
```

Figure 10: exec Backdoor Command

The exit command simply terminates the connection between the client and the server thread. The upload command is used to upload a file from the client to the host machine that the server is running on. The command is followed by the path to upload the file to and the size of the file. After receiving an upload command, the server responds with the string "`ACK_UPLOAD\r`" and starts receiving file data from the client. Data is read 512 bytes at a time and written to the specified file path until the entire file has been delivered.

```
w_send(received_buffer, &client_socket, "ACK_UPLOAD\rACK_UPLOAD_FIN\r", 11i64);
if ( *&received_buffer[0] )        // server send "ACK_UPLOAD/r"
  goto LABEL_274;
v122 = *(&v192 + 1);
v187 = *(&v192 + 1);
v186 = v120;
std::fs::OpenOptions::new::h8dea3ac55a4035c6(received_buffer);
v187 = v122;
v186 = v120;
LOBYTE(v123) = 1;
v124 = sub_140010250(received_buffer, v123);
v187 = v122;
v186 = v120;
LOBYTE(v125) = 1;
v126 = sub_140010270(v124, v125);
v187 = v122;
v186 = v120;
LOBYTE(v127) = 1;
file_path = sub_140010260(v126, v127);
v187 = v122;
v186 = v120;
v129 = file_open(file_path, v116, v117);
v1 = file_handle_1;
if ( !v129 )
{
  file_handle[0] = file_handle_1;
  memset(received_buffer, 0, sizeof(received_buffer));
  v174 = v120;
  if...
  do
  {                                // start receiving 512-byte of file data at a time
    if ( received_len < 0x200 )
    {
      w_recv(v151, &client_socket, received_buffer, received_len);
      v1 = *(&v151[0] + 1);
      if ( *&v151[0] )
        goto LABEL_273;
      file_write(v151, file_handle, received_buffer, received_len);
      if ( *&v151[0] )             // write to disk
      {

        v1 = *(&v151[0] + 1);
        goto LABEL_273;
      }
    }
    else
    {
      w_recv(v151, &client_socket, received_buffer, 512i64);
      v1 = *(&v151[0] + 1);
      if ( *&v151[0] )
        goto LABEL_273;
      file_write(v151, file_handle, received_buffer, 512i64);
      if ( *&v151[0] )
        goto LABEL_271;
    }
```

Figure 11: upload Backdoor Command

Finally, we see the file path is passed into a function along with the received key and nonce to be decrypted. After decrypting the file, the server responds to the client with the string “ACK_UPLOAD_FIN\r”.

```
v187 = v132;
v186 = v121;
v1 = decrypt_file(file_path, file_path_len, key_buffer, 0x20ui64, nonce_buffer, 0x20ui64);
if ( !v1 )
{
  v187 = *(&v192 + 1);
  v186 = v121;
  w_send(received_buffer, &client_socket, "ACK_UPLOAD_FIN\r", 15i64);
  if ( !*&received_buffer[0] )
  {
    if ( v121 )
      j___rdl_dealloc(*(&v192 + 1), 16 * v121, 8i64);
    if ( v82 )
      j___rdl_dealloc(v181, v82, (v82 & 0x8000000000000000ui64) == 0i64);
    v3 = v185;
    v4 = 6i64;
    goto LABEL_37;
  }
```

Figure 12: Decrypting Uploaded File

The decryption function generates an HC-256 XOR stream using the received key and nonce to decrypt the file.

```
v67 = a1;
v59 = (v64[31] << 24) | *(v64 + 14) | (v64[30] << 16);
for ( i = 16i64; i != 2560; ++i )
{
  v19 = *(&v45[1018] + i + 1);
  *(&v46 + i) = i
              + (__ROL4__(v19, 25) ^ __ROL4__(v19, 14) ^ (v19 >> 3))
              + (__ROL4__(*(&v45[0x401] + i), 15) ^ __ROL4__(*(&v45[0x401] + i), 13) ^ (*(&v45[0x401] + i) >> 10))
              + *(&v45[1022] + i + 1)
              + v17;
  v17 = v19;
}
memcpy(v45, &v60[1984], 0x1000ui64);          // populate HC-256 ptable
memcpy(&v45[512], &v60[6080], 0x1000ui64);    // populate HC-256 qtable
LODWORD(v45[1025]) = 0;
v20 = 4096;
do
{
  gen_word_keystream_generation(v45);
  --v20;
}
while ( v20 );
```

Figure 13: HC-256 ptable & qtable Population

```
__int64 __fastcall gen_word_keystream_generation(__int64 a1)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  v1 = *(a1 + 8200);
  v2 = v1 & 0x3FF;
  *(a1 + 8200) = (v1 + 1) & 0x7FF;
  if ( v1 > 0x3FF )
  {
    v8 = *(a1 + 4i64 * (((v1 & 0x3FF) + 1021) & 0x3FF) + 4096);
    v4 = (__ROL4__(v8, 22) ^ __ROL4__(*(a1 + 4i64 * ((v2 + 1) & 0x3FF) + 4096), 9))
       + *(a1 + 4i64 * ((v8 ^ *(a1 + 4i64 * ((v2 + 1) & 0x3FF) + 4096)) & 0x3FF))
       + *(a1 + 4 * v2 + 4096)
       + *(a1 + 4i64 * ((v2 + 1014) & 0x3FF) + 4096);
    *(a1 + 4 * v2 + 4096) = v4;
    v9 = *(a1 + 4i64 * (((v1 & 0x3FF) + 1012) & 0x3FF) + 4096);
    v6 = *(a1 + ((v9 >> 14) & 0x3FC) + 2048) + *(a1 + 4i64 * v9) + *(a1 + 4i64 * BYTE1(v9) + 1024);
    v7 = (a1 + 4 * (v9 >> 24) + 3072);
  }
  else
  {
    v3 = *(a1 + 4i64 * (((v1 & 0x3FF) + 1021) & 0x3FF));
    v4 = (__ROL4__(v3, 22) ^ __ROL4__(*(a1 + 4i64 * ((v2 + 1) & 0x3FF)), 9))
       + *(a1 + 4i64 * ((v3 ^ *(a1 + 4i64 * ((v2 + 1) & 0x3FF))) & 0x3FF) + 4096)
       + *(a1 + 4 * v2)
       + *(a1 + 4i64 * ((v2 + 1014) & 0x3FF));
    *(a1 + 4 * v2) = v4;
    v5 = *(a1 + 4i64 * (((v1 & 0x3FF) + 1012) & 0x3FF));
    v6 = *(a1 + ((v5 >> 14) & 0x3FC) + 6144) + *(a1 + 4i64 * v5 + 4096) + *(a1 + 4i64 * BYTE1(v5) + 5120);
    v7 = (a1 + 4 * (v5 >> 24) + 7168);
  }
  return v4 ^ (*v7 + v6);
}
```

Figure 14: HC256 Keystream Generation

The flag image for the challenge is one of the files sent through an upload command that is captured in the provided PCAP file.

Network Traffic Analysis

Knowing that the initial communication contains the visible strings “ACK_K\r” and “ACK_N\r” sent from the client side, we can use Wireshark to filter for the packets containing these strings.

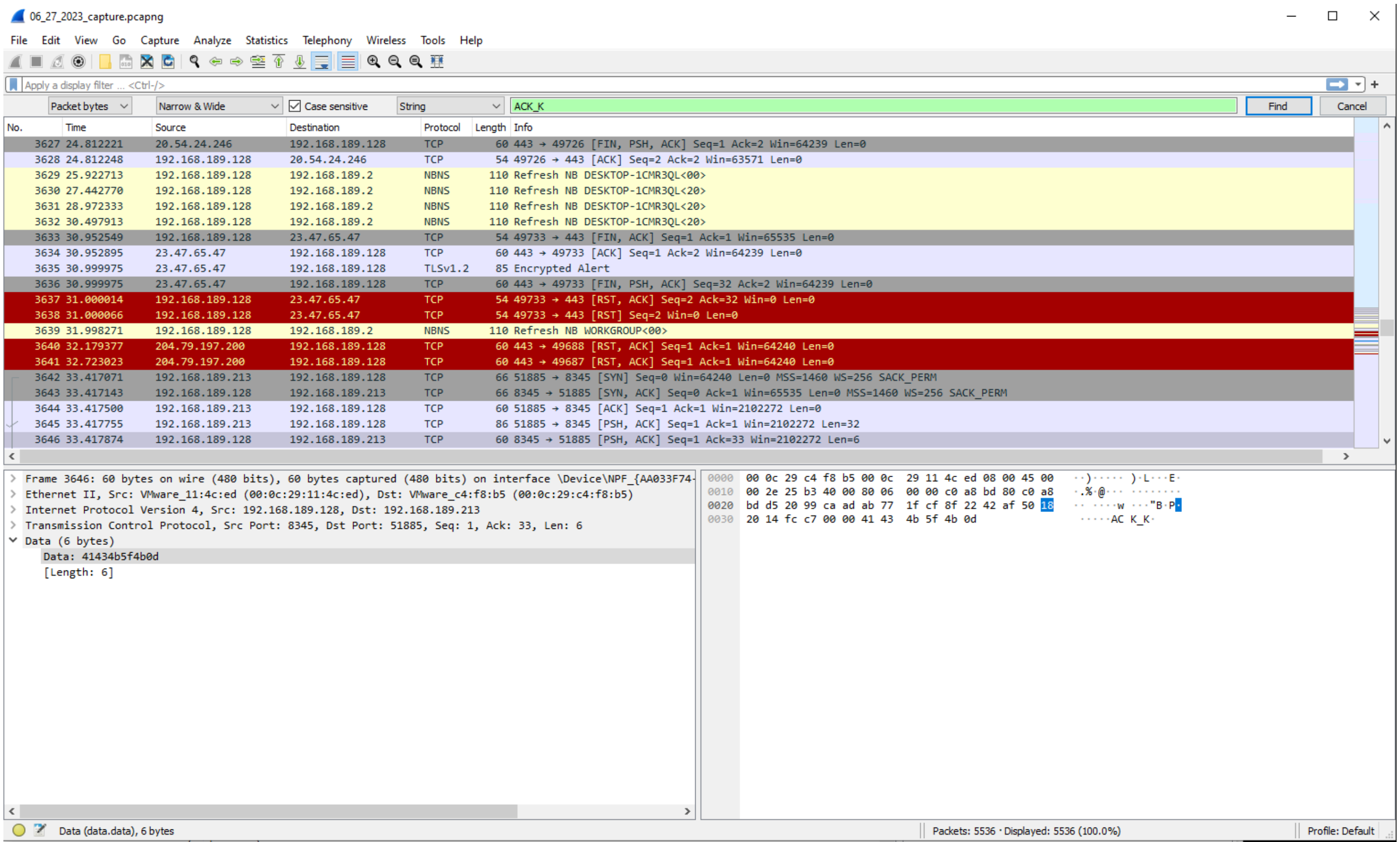

Figure 15: Searching for Malicious Traffic in Wireshark

By following the TCP stream from frame 3646, we can observe the following TCP stream of communication between a client and the malware. We see the 32-byte key and 32-byte nonce sent along with the `exec` and `upload` commands in here.

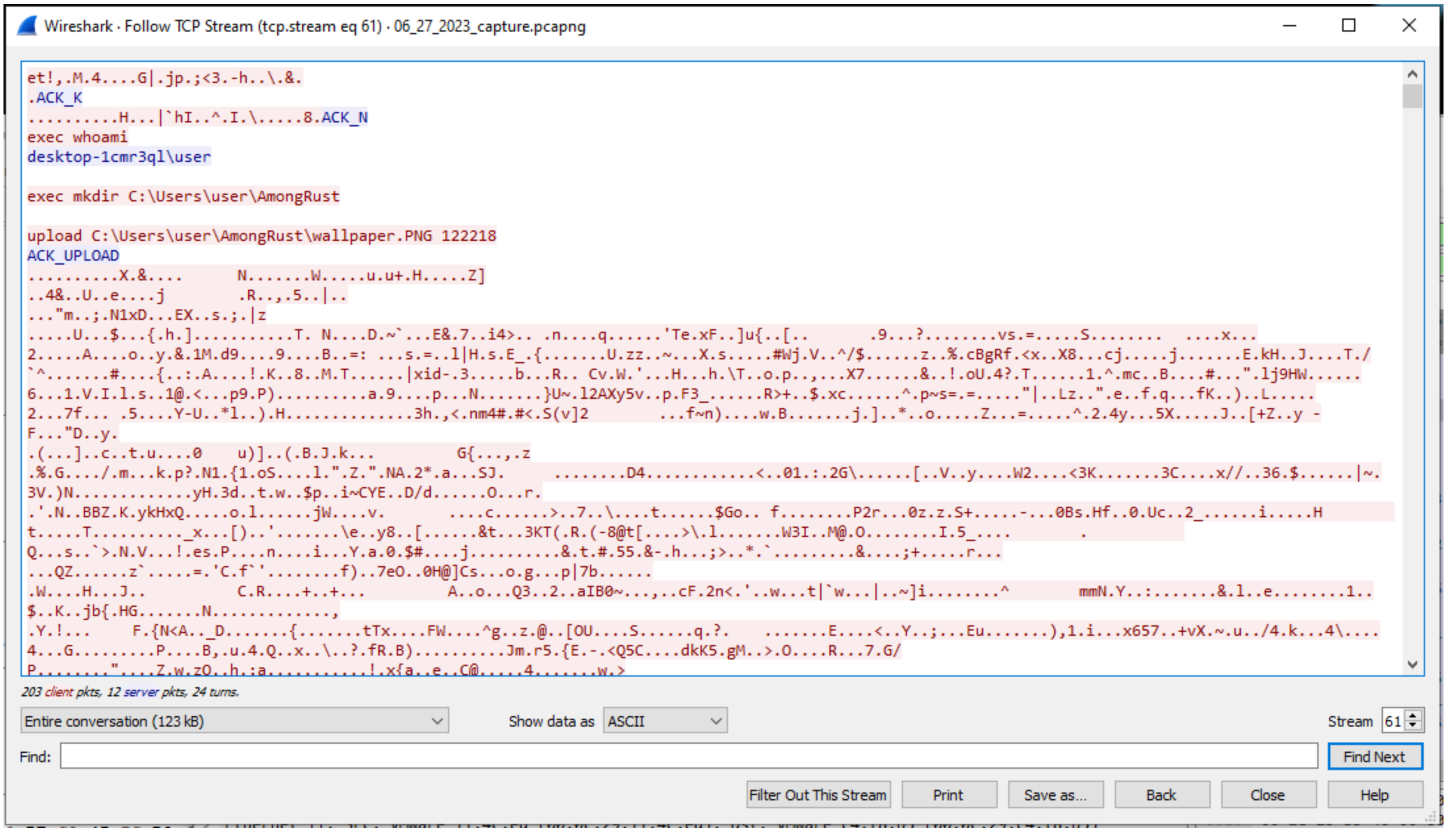

Figure 16: Malicious TCP Stream

Dumping the TCP stream allows us to recover the HC-256 key and nonce being used to decrypt the uploaded files.

Through the commands sent by the client, the malware creates the folder AmongRust in the user folder. It then receives and writes the files `wallpaper.PNG` and `wallpaper.ps1` in there. Finally, the malware is instructed to execute the PowerShell script and delete everything afterward.

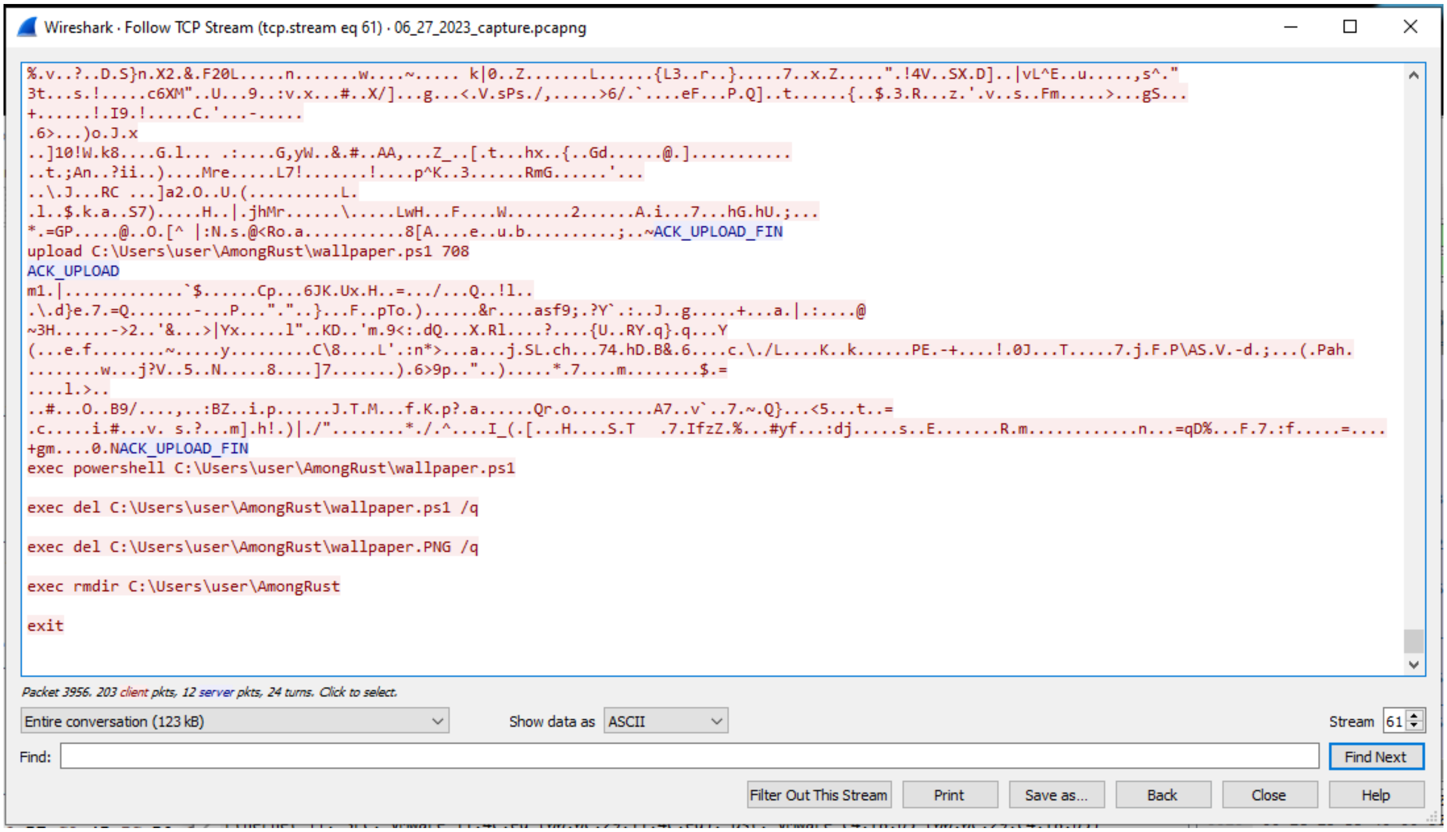

Figure 17: Captured Backdoor Commands

Solving The Challenge

Dynamically Decrypt Files in Debugger

In Figure 11, we see that the HC-256 file decrypting function takes in the encrypted file path, key, nonce, and their lengths as parameters. With this, we can solve the challenge by first dumping the encrypted file to disk and debugging the sample in x64dbg or any other debugger of your choice.

```
0140002BE6    mov     [rsp+750h+nonce], rax ; nonce
0140002BEB    mov     [rsp+750h+nonce_len], 20h ; ' ' ; nonce_len
0140002BF4    mov     r9d, 20h ; ' '   ; key_len
0140002BFA    mov     rcx, r14         ; file_path
0140002BFD    mov     rdx, r15         ; file_path_len
0140002C00    lea     r8, [rbp+6D0h+key_buffer] ; key
0140002C07    call    decrypt_file
```

Figure 18: Parameter Setup for Decrypting Function

By jumping directly to the address 140002C07, we can start patching the register values to decrypt the file. First, we can write the extracted key and nonce into memory. The rax value can then be changed to the address of the nonce buffer, r14 to the address of the encrypted file path, r15 to the length of the file path, and r8 to the address of the key.
After setting this up in the debugger, executing the file decrypting function will result in the file being decrypted on disk.

Programmatically Decrypt Files in Rust

Since we know the malware uses HC-256 cryptographic algorithm to decrypt the files using the provided key and nonce, we can write a Rust script to programmatically decrypt the flag. I recommend using RustCrypto's HC256 crate for this.

```
use std::{
    fs::File,
    io::{Read, Write},
};

use hc_256::{
    cipher::{KeyIvInit, StreamCipher},
    Hc256,
};

fn main() {
    let key: Vec<u8> = vec![
        0x65, 0x74, 0x21, 0x2c, 0x9b, 0x4d, 0x93, 0x34, 0xd8, 0x93, 0xbe, 0xc2, 0x47, 0x7c, 0xb8,
        0x6a, 0x70, 0x98, 0x3b, 0x3c, 0x33, 0x95, 0x2d, 0x68, 0xa8, 0xcc, 0x5c, 0x02, 0x26, 0x07,
        0x0a, 0xbf,
    ];

    let nonce: Vec<u8> = vec![
        0x0e, 0x02, 0xf4, 0xa9, 0xa8, 0xb5, 0xbe, 0xea, 0xba, 0x83, 0x48, 0xd6, 0xd2, 0xf8, 0x7c,
        0x60, 0x68, 0x49, 0xdf, 0x9a, 0x5e, 0xef, 0x49, 0xa6, 0x5c, 0x98, 0xcf, 0x07, 0xd4, 0xc2,
        0x38, 0xa6,
    ];

    let mut cipher = Hc256::new(key.as_slice().into(), nonce.as_slice().into());

    let mut contents = Vec::new();

    let mut file = File::open("<encrypted flag path").unwrap();

    file.read_to_end(&mut contents).unwrap();

    cipher.apply_keystream(&mut contents);

    let mut flag_file = File::create("flag.PNG").unwrap();

    flag_file.write(&contents).unwrap();
}
```

Writing a Client in Python

Another approach to solving the challenge would be writing a client in a language of your choice to communicate with the server and instructing it to decrypt the flag for us. This requires our client to initially send the correct key and nonce to the server and an upload command with a file path to write the flag to. Then, the client can send the encrypted flag to the server to be decrypted.
Instead of having to write the Python script ourselves, Google Bard AI chatbot can be used to quickly generate the script for us. Below are the instructions I give to Bard.

```
Hey Bard. Write me a Python script to perform the following tasks.

1. Create a client TCP socket and connect to the server 192.168.189.128 on port 8345.

2. Convert the following list [0x65, 0x74, 0x21, 0x2c, 0x9b, 0x4d, 0x93, 0x34, 0xd8, 0x93,
0xbe, 0xc2, 0x47, 0x7c, 0xb8, 0x6a, 0x70, 0x98, 0x3b, 0x3c, 0x33, 0x95, 0x2d, 0x68, 0xa8, 0xcc,
0x5c, 0x02, 0x26, 0x07, 0x0a, 0xbf] to a bytes object and send it to the server.

3. Receive 6 bytes from the server and check if it's 'ACK_K\r'. If it's not, terminate the
connection.

4. Convert the following list [0x0e, 0x02, 0xf4, 0xa9, 0xa8, 0xb5, 0xbe, 0xea, 0xba, 0x83,
0x48, 0xd6, 0xd2, 0xf8, 0x7c, 0x60, 0x68, 0x49, 0xdf, 0x9a, 0x5e, 0xef, 0x49, 0xa6, 0x5c, 0x98,
0xcf, 0x07, 0xd4, 0xc2, 0x38, 0xa6] to a bytes object and send it to the server.

5. Receive 6 bytes from the server and check if it's 'ACK_N\r'. If it's not, terminate the
connection.

6. Send the bytes array of the following string "upload
C:\Users\user\Desktop\decrypted_flag.PNG 122218\r" to the server.

7. Receive 11 bytes from the server and check if it's 'ACK_UPLOAD\r'. If it's not, terminate
the connection.

8. Open the file 'encrypted_flag.bin' and read its content into a variable named
encrypted_flag_data.

9. Send all of encrypted_flag_data data to the server.

10. Receive 15 bytes from the server and check if it's 'ACK_UPLOAD_FIN\r'. If it's not,
terminate the connection.

11. Send the bytes array of the string "exit" to the server.

12. Close the socket and terminate the Python program.
```

View other drafts

Sure, here is the Python script to perform the task:

Python

```python
import socket

def main():
    # Create a client TCP socket and connect to the server
    host = "192.168.189.128"
    port = 8345
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect((host, port))

    # Convert the list to a bytes object and send it to the server
    data = bytes([0x65, 0x74, 0x21, 0x2c, 0x9b, 0x4d, 0x93, 0x34, 0xd8, 0x93, 0xbe, 0xc2, 0x47, 0x7c,
    s.sendall(data)

    # Receive 6 bytes from the server and check if it's 'ACK_K\r'
    ack = s.recv(6)
    if ack != b"ACK_K\r":
        print("Invalid response from server")
        s.close()
        return

    # Convert the list to a bytes object and send it to the server
    data = bytes([0x0e, 0x02, 0xf4, 0xa9, 0xa8, 0xb5, 0xbe, 0xea, 0xba, 0x83, 0x48, 0xd6, 0xd2, 0xf8,
    s.sendall(data)

    # Receive 6 bytes from the server and check if it's 'ACK_N\r'
    ack = s.recv(6)
    if ack != b"ACK_N\r":
        print("Invalid response from server")
        s.close()
        return

    # Send the bytes array of the following string to the server
    data = b"upload C:\Users\user\Desktop\decrypted_flag.PNG 122218\r"
    s.sendall(data)

    # Receive 11 bytes from the server and check if it's 'ACK_UPLOAD\r'
    ack = s.recv(11)
    if ack != b"ACK_UPLOAD\r":
        print("Invalid response from server")
        s.close()
        return

    # Open the file 'encrypted_flag.bin' and read its content into a variable named encrypted_flag_dat
    with open("encrypted_flag.bin", "rb") as f:
        encrypted_flag_data = f.read()

    # Send all of encrypted_flag_data data to the server
    s.sendall(encrypted_flag_data)

    # Receive 15 bytes from the server and check if it's 'ACK_UPLOAD_FIN\r'
    ack = s.recv(15)
    if ack != b"ACK_UPLOAD_FIN\r":
        print("Invalid response from server")
        s.close()
        return

    # Send the bytes array of the string "exit" to the server
    data = b"exit"
    s.sendall(data)

    # Close the socket and terminate the Python program
    s.close()
```

Use code with caution. Learn more

Figure 19: Bard Python Solution

Flag

The `wallpaper.PNG` image contains the flag for the challenge, and the `wallpaper.ps1` file is a script to set the system's wallpaper to the flag image. The decrypted versions of them are shown below.

```
Add-Type -TypeDefinition @'
using System.Runtime.InteropServices;
public class Wallpaper {
    public const uint SPI_SETDESKWALLPAPER = 0x0014;
    public const uint SPIF_UPDATEINIFILE = 0x01;
    public const uint SPIF_SENDWININICHANGE = 0x02;
    [DllImport("user32.dll", SetLastError = true, CharSet = CharSet.Auto)]
    private static extern int SystemParametersInfo (uint uAction, uint uParam, string lpvParam,
uint fuWinIni);
    public static void SetWallpaper (string path) {
        SystemParametersInfo(SPI_SETDESKWALLPAPER, 0, path, SPIF_UPDATEINIFILE |
SPIF_SENDWININICHANGE);
    }
}
'@


$wallpaper = 'C:\Users\user\AmongRust\wallpaper.PNG'  # absolute path to the image file
[Wallpaper]::SetWallpaper($wallpaper)
```

Figure 20: Decrypted wallpaper.PNG - Challenge Flag