A Deep Dive Into IcedID Malware: Part III - Analysis of Child Processes

By Kai Lu

Published: 2019-07-22 · Archived: 2026-04-05 18:07:39 UTC

FortiGuard Labs Threat Analysis Report Series

In Part II of this blog series, we identified three child processes that were created by the IcedID malware. In Part III below, we'll provide a deep analysis of those child processes.

Let's get started!

0x01 Child process A (entry offset: 0x168E)

This first child process is primarily responsible for performing web injection in browsers and acting as a proxy to inspect and manipulate traffic. It can also hook key functions in browsers.

The following is the pseudo code of the entry point.

```
signed int __stdcall func_start(int a1)
{
  BOOL v1; // esi
  signed int result; // eax

  *(_DWORD *)(MEMORY[0x6F098] + 304) = MEMORY[0x6F098];
  v1 = unhook_RtlExitUserProcess();
  result = load_dlls();                         // load dlls
  if ( result )
  {
    result = sub_661A9F();                      // core function
    if ( result )
    {
      while ( 1 )
LABEL_5:
        Sleep(0xFFFFFFFF);
    }
  }
  if ( v1 )
  {
    (*(void (__stdcall **)(_DWORD))(*(_DWORD *)(MEMORY[0x6F098] + 304) + 56))(0);// RtlExitUserProcess
    goto LABEL_5;
  }
  return result;
}
```

Entry point of injected svchost.exe child process(offset:0x168e)

Figure 1. The pseudo code of the entry point in the trampoline code

In this function, the process first unhooks the RtlExitUserProcess API and then loads a number of dynamic libraries. The function sub_0x1A9F() is the core function.

```
signed int sub_661A9F()
{
  HANDLE v0; // eax
  signed int result; // eax

  SetErrorMode(0x8007u);
  dword_66F0B0[1] = *((_DWORD *)&unk_80000 + 215);// RC4 key(Bot ID)
  dword_66F0B0[0] = *((_DWORD *)&unk_80000 + 216);
  *(_DWORD *)key = *(_DWORD *)((char *)&unk_80000 + 869);
  LOBYTE(dword_66F0B0[2]) = *((_BYTE *)&unk_80000 + 868);
  sub_661073((_BYTE *)&unk_80000 + 873);          // build c2 server list
  result = sub_6673FA();                          // create a thread to set IPC with file mapping technique
  if ( result )
  {
    sub_669EE9();                                 // APC related
    if ( !sub_661410() )
      MEMORY[0x6F0C41] = 1;
    result = Init_ProxyServer(dword_66F0C8);      // add self cert to certstore and create a proxy server 127.0.0.1:61420
    if ( result == 1 )
    {
      MEMORY[0x6F0C51] = 1;
    }
    else
    {
      if ( result == 3 )
      {
        MEMORY[0x6F0C6] = 1;
      }
      else if ( result == 4 )
      {
        MEMORY[0x6F0C7] = 1;
      }
      MEMORY[0x6F0BC] = sub_661B7E();
      if ( MEMORY[0x6F0BC] && (v0 = CreateThread(0, 0, (LPTHREAD_START_ROUTINE)browser_injection, 0, 0, 0)) != 0 )// perform code injection into browser
      {
        CloseHandle(v0);
        result = 1;
      }
      else
      {
        result = 0;
      }
    }
  }
  return result;
}
```

Figure 2. The core function sub_0x1A9F()

Here's a list of the key functionalities of this function.

1. Build a C2 server list

2. Create a thread to set IPC with file mapping technique

3. Create a thread and then call the QueueUserAPC function to add a user-mode asynchronous procedure call (APC) object to the APC queue of the specified thread. In APC, it can read the DAT config file, decrypt it with an RC4 key, and then decompress the data as follows.

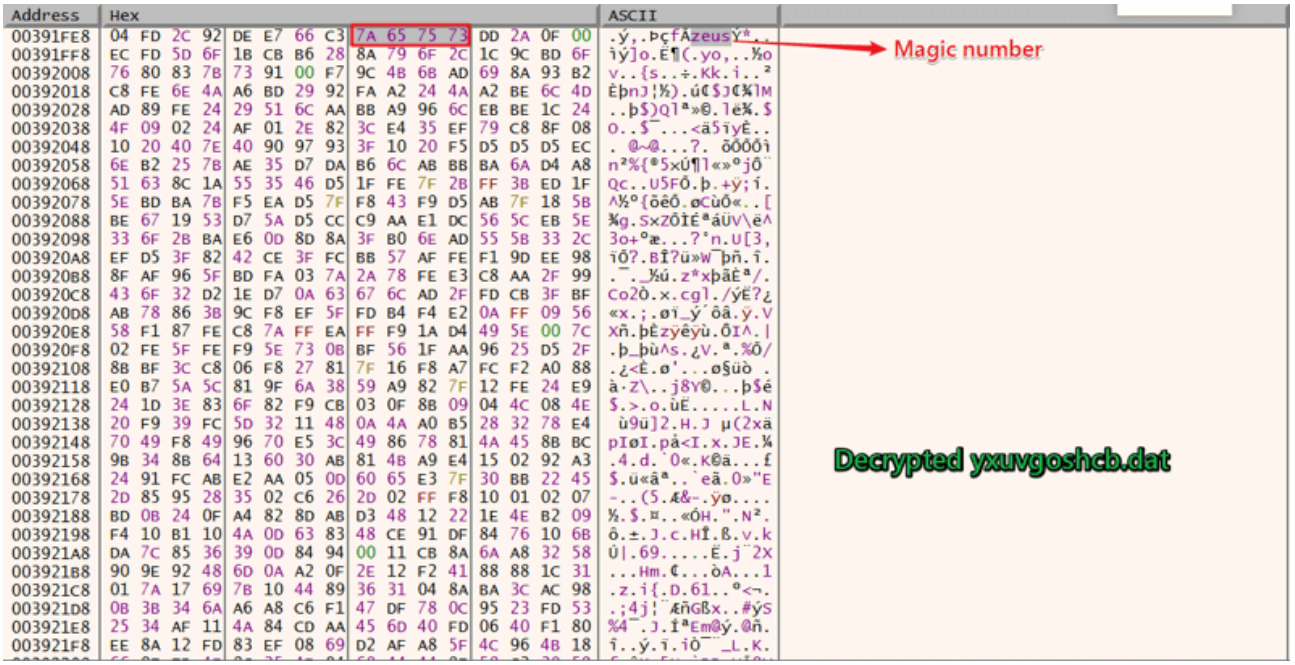

Figure 3. The decrypted web injection DAT config file

This DAT config file is used for performing web injections. It uses a Magic number, “zeus”. IcedID then uses a customized algorithm to decode the content. The following is the decompressed data.

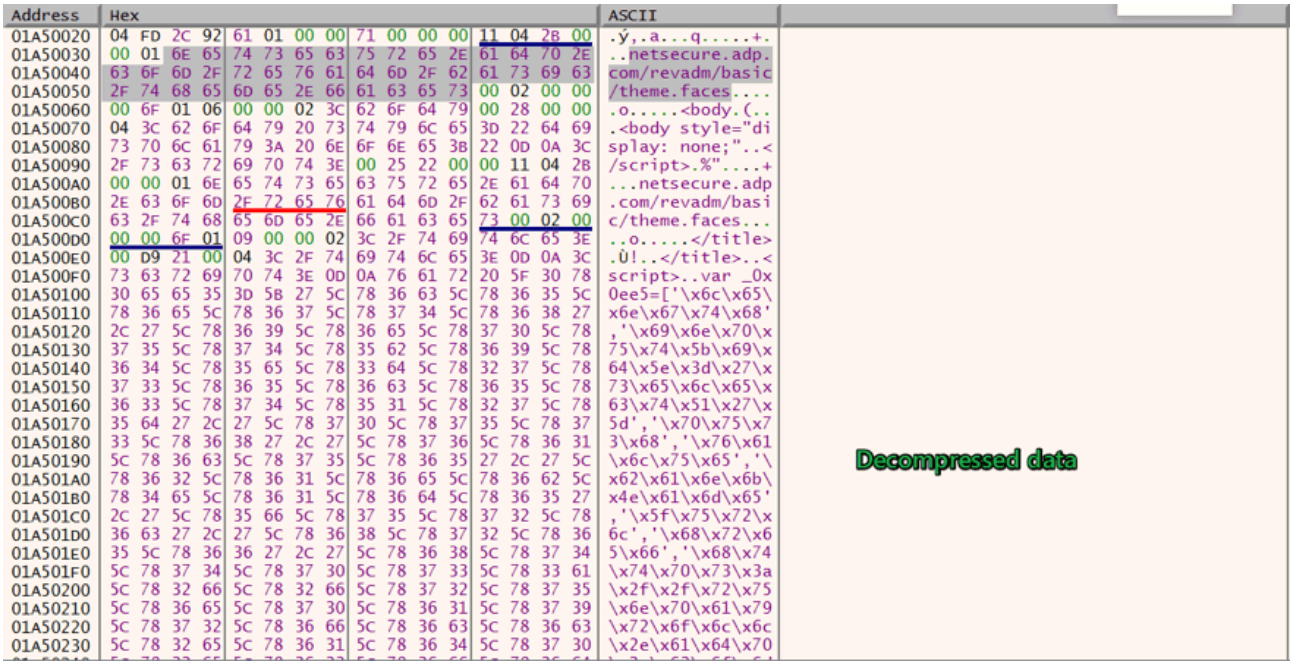

Figure 4. The decompressed data of web injection

4. Add self-signed certificate into the certificate store and then create a proxy server which is bound to 127.0.0.1 on TCP port 61420. Next, it calls the RegisterWaitForSingleObject function to register a WSA (Windows Socket API) event handler, then uses the socket of the initialized proxy server to handle all connect, send, and receive network requests.

```
if ( v2 & 8 )
{
  v3 = accept(*((_DWORD *)lpMem + 10), 0, 0);
  v4 = (void (__stdcall *)(PVOID, SOCKET, _DWORD))*((_DWORD *)lpMem + 18);
  if ( v4 )
    v4(lpMem, v3, *((_DWORD *)lpMem + 20));
  else
    closesocket(v3);
}
else
{
  if ( v2 & 0x10 && NetworkEvents.iErrorCode[4] )
  {
    v2 |= 0x20u;
    NetworkEvents.lNetworkEvents = v2;
  }
  if ( v2 & 2 )
  {
    wrap_send_1((int)lpMem, (char *)&NetworkEvents);
    LOBYTE(v2) = NetworkEvents.lNetworkEvents;
  }
  if ( v2 & 1 )
  {
    wrap_recv_1((SOCKET *)lpMem, &NetworkEvents);
    LOBYTE(v2) = NetworkEvents.lNetworkEvents;
  }
  if ( v2 & 0x20 )
  {
    *((_DWORD *)lpMem + 9) |= 0x1000u;
    v5 = (void (__stdcall *)(PVOID, _DWORD))*((_DWORD *)lpMem + 19);
    if ( v5 )
      v5(lpMem, *((_DWORD *)lpMem + 20));
  }
}
goto LABEL_19;
```

Proxy server handles network requests

Figure 5. Proxy server handles network requests

Additionally, in order to perform a MiTM attack on SSL connections, the proxy server has to generate a certificate and add it into the cert store. The following is that implementation.

```
int __cdecl add_cert2store(void *pvPara)
{
  HCERTSTORE v1; // eax
  const CERT_CONTEXT *v2; // eax
  const CERT_CONTEXT *v3; // esi

  v1 = wrap_CertOpenStore(pvPara, 1);          // pvPara = "C:\\Users\\[redacted]\\AppData\\Local\\Temp\\19AA26AC.tmp"
  dword_66F3C4 = (int)v1;                                                                  Bot ID
  if ( !v1 )
    goto LABEL_5;
  pCertContext = sub_661D05(v1);
  if ( !pCertContext )
  {
    v2 = wrap_CertCreateSelfSignCertificate_1();
    v3 = v2;
    if ( !v2 )
    {
      wrap_CertCloseStore((HCERTSTORE)0x32BDA8);
      dword_66F3C4 = (unsigned int)v3 & 0x32BDA8;
LABEL_5:
      *(&pCertContext + 1) = sub_669E5A();
      return *(&pCertContext + 1) != 0 ? 2 : 0;
    }
    pCertContext = wrap_CertAddCertificateContextToStore((HCERTSTORE)0x32BDA8, v2);
    if ( pCertContext )
      CertFreeCertificateContext(v3);
    else
      pCertContext = v3;                       Add self-signed cert into cert store
  }
  *(&pCertContext + 1) = sub_669E5A();
  return 1;
}
```

Figure 6. Adding a self-signed cert into the cert store

We can also see that this svchost.exe child process is listening on TCP port 61420.

svchost.exe	3980	TCP	WIN-1PCD95NN9GC	61420	WIN-1PCD95NN9GC	0	LISTENING

5. Create a thread to perform code injection into the browser. The following is the thread function of the browser code injection.

```
void __stdcall __noreturn browser_injection(LPVOID lpThreadParameter)
{
  while ( 1 )
  {
    MEMORY[0x6F0D8] = sub_667DE2("#ke");
    sub_665F0F((int (__stdcall *)(CHAR *, char *, int))sub_661A07, 0, 1);// Get all running processes list, and if browser process is found,
                                                                         // it performs code injection and hook ZwWaitForSingleObject.
    Sleep(0x3E8u);
  }
}
```

Figure 7. The browser injection function

It uses the ZwQuerySystemInformation function to gather a list of all current running processes. If a browser process is found, it performs code injection into the browser process and sets up a hook on the ZwWaitForSingleObject function. The following is the function that checks to see if a running process is a browser process. It first generates a hash with the process name using a specified algorithm. Then, it compares the hash with the given hash of four browsers: Firefox, Edge, IE, and Chrome.

```
CHAR *__cdecl check_processhash(LPCSTR pszPath)
{
  int v1; // esi
  CHAR *result; // eax
  CHAR *v3; // edi
  unsigned __int8 v4; // al
  int v5; // ecx
  int v6; // eax
  int v7; // esi
  CHAR *v8; // [esp-4h] [ebp-Ch]

  v1 = 0;
  result = PathFindFileNameA(pszPath);
  v3 = result;
  if ( result )
  {
    CharLowerA(result);
    v4 = *v3;
    v5 = 0;
    while ( v4 )
    {
      v6 = v1 + v5++ + v4;
      v1 = __ROR4__(v6, 3);
      v4 = v3[v5];
    }
    v7 = v1 ^ 0x401056;
    switch ( v7 )
    {
      case 0x1EACD83D:
        v8 = (CHAR *)2;                         // firefox.exe (type=2)
        break;
      case 0x1FC79819:
        v8 = (CHAR *)4;                         // microsoftedgecp.exe (type=4)
        break;
      case (int)0xD31A30C7:
        v8 = (CHAR *)3;                         // iexplore.exe (type=3)
        break;
      default:
        return (CHAR *)(v7 == 0xFA7442ED);      // chrome.exe (type=1)
    }
    result = v8;
  }
  return result;
}
```

Process hash generation algorithm

Figure 8. Checking the hash of the process name

Before performing its code injection, it first checks to see if this process is running on 64 bits by calling the IsWow64Process function. It then performs a code injection into the browser process, and depending on the process bits version, it calls the corresponding hook function to set up a hook on the ZwWaitForSingleObject function.

```
v10 = a1;
v12 = a2;
v11 = a3;
v13 = 0;
v14 = 0;
lpMem = 0;
v16 = 0;
v3 = allocate two memory regions(&v10);          // NtAllocateVirtualMemory
if ( v3 )
{
  v3 = sub_667719(&v10);
  if ( v3 )
  {
    v3 = process injection(&v10);                // ZwWriteVirtualMemory
    if ( v3 )
    {
      if ( v12 & 4 )
        v4 = hook systemAPI 32(&v10);
      else
        v4 = hook systemAPI 64(&v10);
      v3 = v4;
    }
  }
}
```

Process injection and set up hook

Figure 9. Process injection and setting up a hook in a browser

Here we will use Firefox to demonstrate how it performs its process injection and sets up a hook.

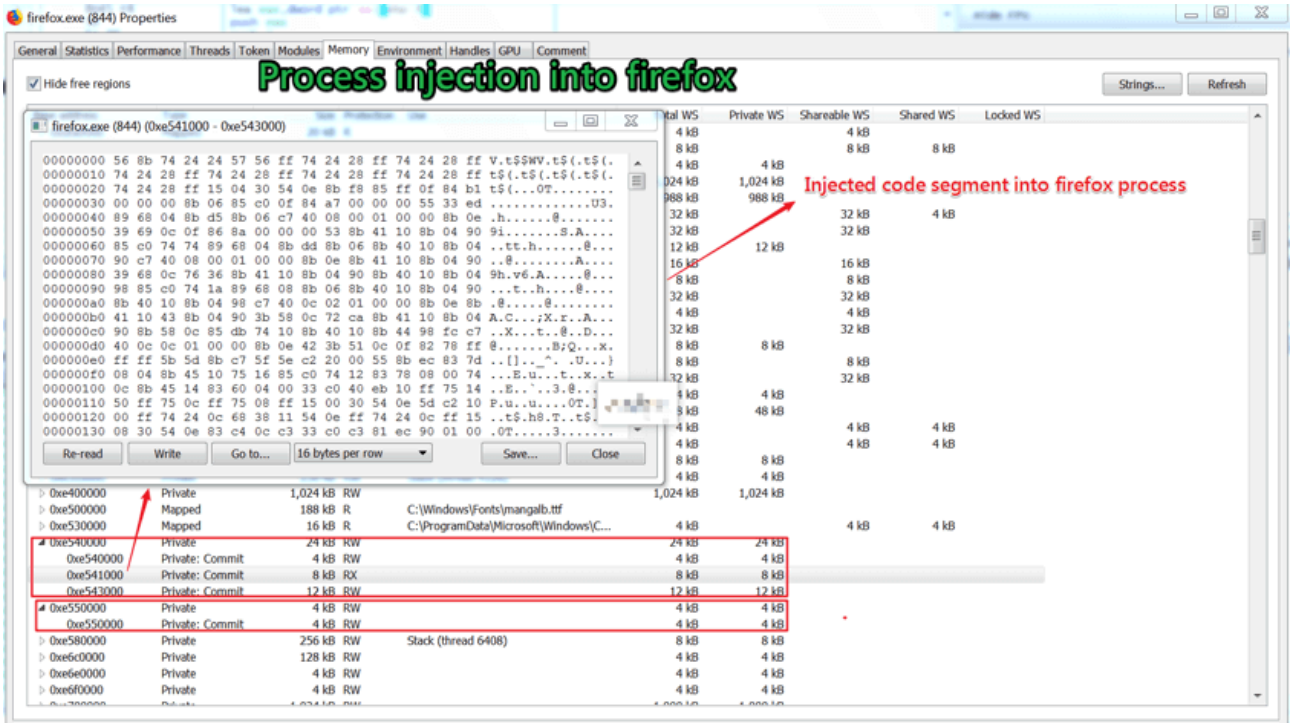

Figure 10. Process injection into Firefox

It sets up a hook on the ZwWaitForSingleObject API in the Firefox process as follows.

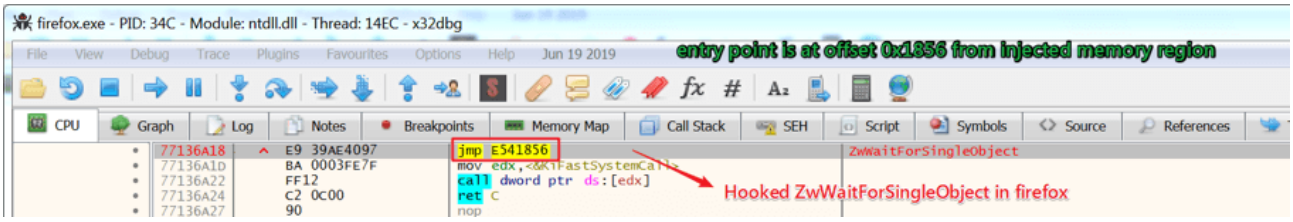

Figure 11. Hooked ZwWaitForSingleObject function

When Firefox calls the ZwWaitForSingleObject function, it jumps to the trampoline code. The entry point of trampoline code is at offset 0x1856 from the injected memory region.

Let's take a closer look at the trampoline code (offset:0x1856).

In this trampoline code, it first unhooks the ZwWaitForSingleObject API. Then it sets up a hook on the SSL_AuthCertificateHook API (in nss3.dll for Firefox.) The nss3.SSL_AuthCertificateHook function specifies a certificate authentication callback function that is called to authenticate an incoming certificate.

The following is the hooked nss3.SSL_AuthCertificateHook function.

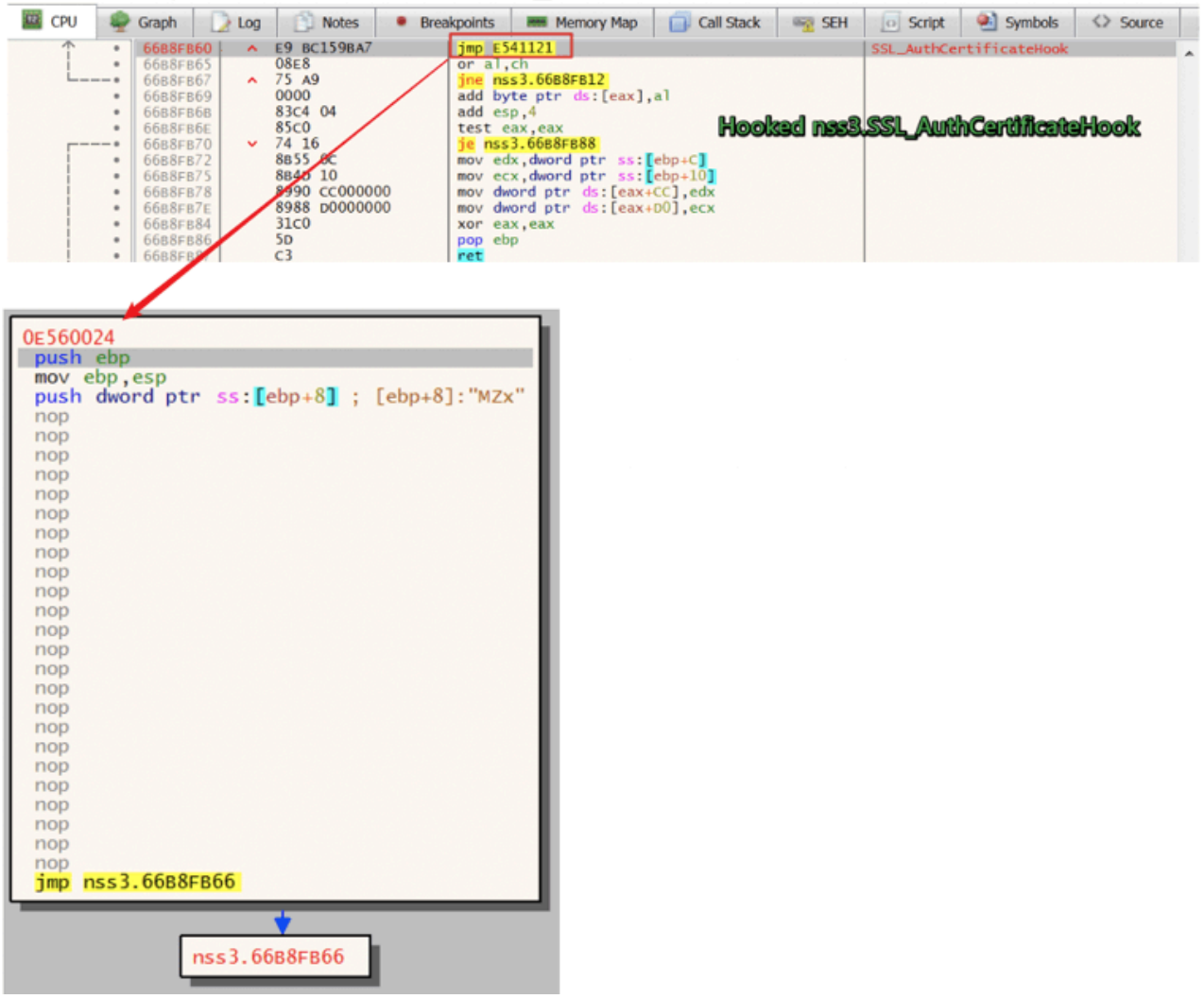

Figure 12. The hooked nss3.SSL_AuthCertificateHook function

It configures the nss3.SSL_AuthCertificateHook function to always return SECSuccess.

Note that it can set up a hook for browser-specific functions depending on the type of browser. However, we won't be providing details for any other browsers in this blog.

Next, it continues to set up a hook on the connect API in ws2_32.dll. The following is the hooked connect API.

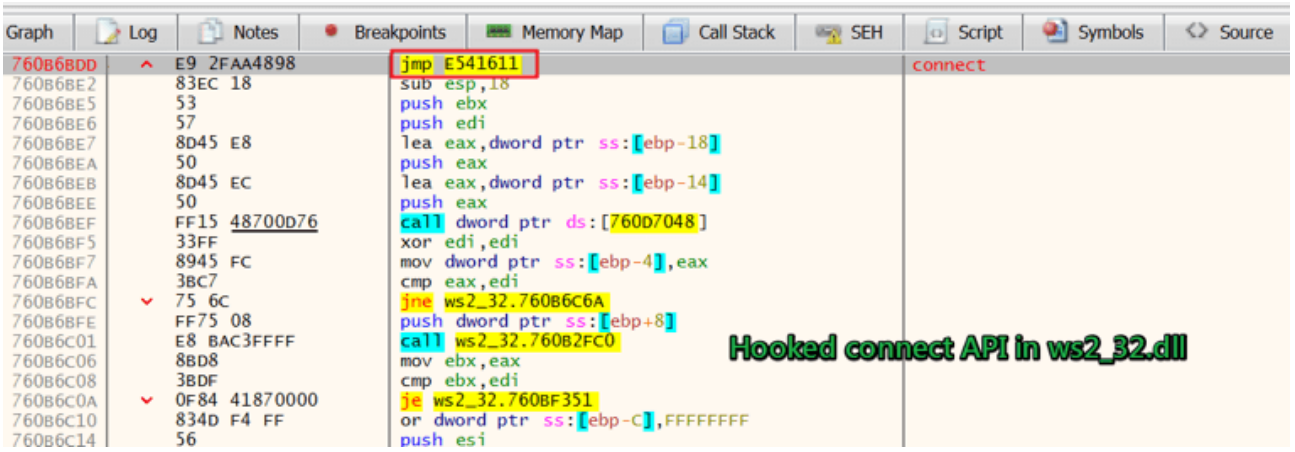

Figure 13. The hooked connect API in ws2_32.dll

The following is the pseudo code of the trampoline code for the hooked ***connect*** API.

```
int __stdcall trampoline_connect_hook(SOCKET s, _WORD *argp, int a3)
{
  _WORD *v3; // esi
  int v4; // edi
  int v6; // [esp+8h] [ebp-1Ch]
  int v7; // [esp+Ch] [ebp-18h]
  int v8; // [esp+10h] [ebp-14h]
  int v9; // [esp+14h] [ebp-10h]
  char buf[4]; // [esp+18h] [ebp-Ch]
  int v11; // [esp+1Ch] [ebp-8h]
  __int16 v12; // [esp+20h] [ebp-4h]
  __int16 v13; // [esp+22h] [ebp-2h]

  v3 = argp;
  if ( *argp != 2 || !sub_B5517FA(2, (unsigned __int16)argp[1]) )
    return MEMORY[0xBED006B](s, v3, a3);
  argp = 0;
  ioctlsocket(s, -2147195266, (u_long *)&argp); // FIONBIO
  v6 = 0;
  v8 = 0;
  v9 = 0;
  LOWORD(v6) = 2;
  v7 = 16777343;
  HIWORD(v6) = *(unsigned __int8 *)(dword_B553018[0] + 853) | (unsigned __int16)(*(_WORD *)(dword_B553018[0] + 852) << 8);
  v4 = MEMORY[0xBED006B](s, &v6, 16);           // execute the rest code of hooked connect
  if ( v4 == -1 )
  {
    WSASetLastError(10061);
  }
  else
  {
    *(_DWORD *)buf = *(_DWORD *)(dword_B553018[0] + 848);
    v11 = *((_DWORD *)v3 + 1);
    v12 = __ROL2__(v3[1], 8);
    v13 = 2;
    send(s, buf, 12, 0);                        // send 12 bytes of data to proxy server 127.0.0.1:61420
    argp = (_WORD *)1;
    ioctlsocket(s, -2147195266, (u_long *)&argp);
    v4 = -1;
    WSASetLastError(10035);
  }
  return v4;
}
```

Figure 14. The pseudo code of the trampoline code for the hooked connect API

Once the ***connect*** function returns 0 (the connection has succeeded), it sends 12 bytes of data to proxy server **127.0.0.1:61420**, which was created in this svchost.exe child process. The captured traffic is shown in Figure 15.

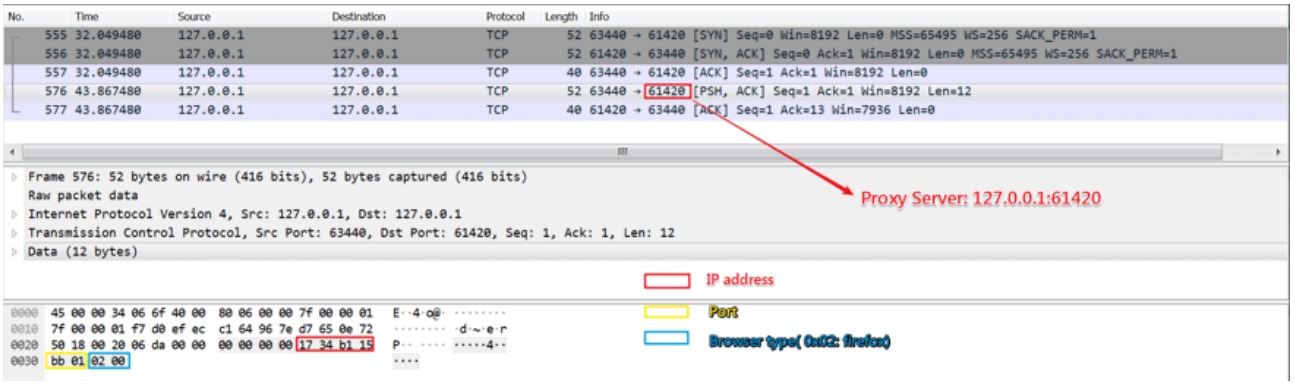

Figure 15. Brower sends 12 bytes of data to proxy server

The structure of these 12 bytes consists of four parts, as follows:

0x00: Unknown

0x04: Target website's IP address

0x08: Port

0x0A: Browser type

0x02 Child Process B (entry offset: 0x1E0A)

This second child process is used to communicate with the C2 server. It will attempt to send an HTTP request to the C2 server via WebSocket, as follows.

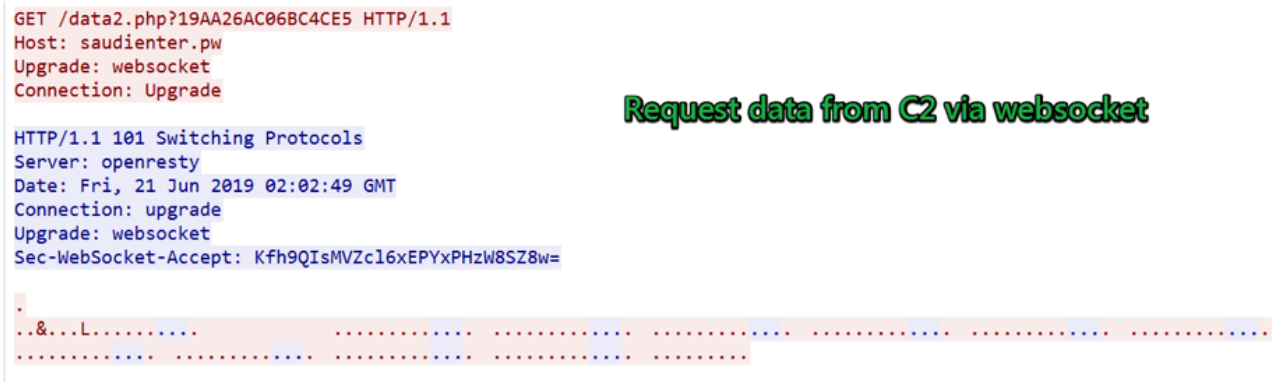

Figure 16. Requesting data from the C2 via WebSocket

It also communicates with the parent svchost.exe process using a mapping file technique. And, depending on the shared info, it may attempt to make network requests to a C2 server over SSL, and then create a new process, perform code injections, and set up a hook on the RtlExitUserProcess function.

0x03 Child Process C (entry offset: 0x10DF)

This process communicates with the parent svchost.exe process by using a mapping file technique. It is also able to perform some registry operations.

0x04 Solution

This malicious PE file has been detected as “W32/Kryptik.GTSU!tr” by the FortiGuard AntiVirus service.

The C2 server list has been rated as “Malicious Websites” by the FortiGuard WebFilter service.

0x05 Conclusion

In this series of posts, I have provided a detailed analysis of a new IcedID malware sample. The entire detailed analysis is divided into three parts. The first two part are available here: Part I: Unpacking, Hooking, and Process Injection and Part II: Analysis of the Core IcedID Payload (Parent Process).

IcedID is a sophisticated and complicated banking trojan that performs web injection in browsers and acts as proxy to inspect and manipulate traffic. It is designed to steal information – such as credentials – from victims and then send that stolen information to attacker-controlled servers. To accomplish this, IcedID uses a large number of hooking and process injection techniques, and it also disguises itself as several svchost.exe processes, which we examined in this deep dive analysis series.

Learn more about FortiGuard Labs and the FortiGuard Security Services portfolio. Sign up for our weekly FortiGuard Threat Brief.

Read about the FortiGuard Security Rating Service, which provides security audits and best practices.
