# breaking the ice a deep dive into the IcedID banking trojans new major version release

By Nir Somech, Limor Kessem

Published: 2020-04-01 · Archived: 2026-04-05 13:01:38 UTC

Nir Somech

Malware Researcher – Trusteer IBM

Limor Kessem

X-Force Cyber Crisis Management Global Lead

IBM

The IcedID banking Trojan was discovered by IBM X-Force researchers in 2017. At that time, it targeted banks, payment card providers, mobile services providers, payroll, webmail and e-commerce sites, mainly in the U.S. IcedID has since continued to evolve, and while one of its more recent versions became active in late-2019, X-Force researchers have identified a new major version release that emerged in 2020 with some substantial changes.

This post will delve into the technical details of IcedID version 12 (0xC in hexadecimal). Before we delve into the technical details, here are the components that saw changes applied in this new version:

- New anti-debugging and anti-VM checks.
- Modified infection routine and file location on disk.
- Hiding encrypted payload in .png file (steganography).
- Modification of code injection tactics.
- Modified cryptographic functions.

In this post, you will also find information on IcedID's naming algorithms that are used for creating names for its various files, events, and resources. We also mention how to find and extract the malware's internal version number.

## IcedID's entry point – Targeted maldocs via other malware

IcedID is spread via malspam emails typically containing Office file attachments. The files are boobytrapped with malicious macros that launch the infection routine, fetch and run the payload.

In February 2020 campaigns, maldocs spread in spam first dropped the OStap malware, which then dropped IcedID. OStap was also a vehicle for TrickBot infections in recent months. IcedID has a connection to the Emotet gang, having been dropped by Emotet in the past.

## Attack turf

IcedID's targeting has been consistent since it emerged. Its focus remains on the North American financial sector, e-commerce, and social media. IcedID targets business users and business banking services.

## IcedID's loader's behavior

Upon the first time running the loader in our labs, we observed that it performs a few actions before deciding whether to infect the machine or not. It begins by loading encrypted code from the resource section, decrypts it and executes.

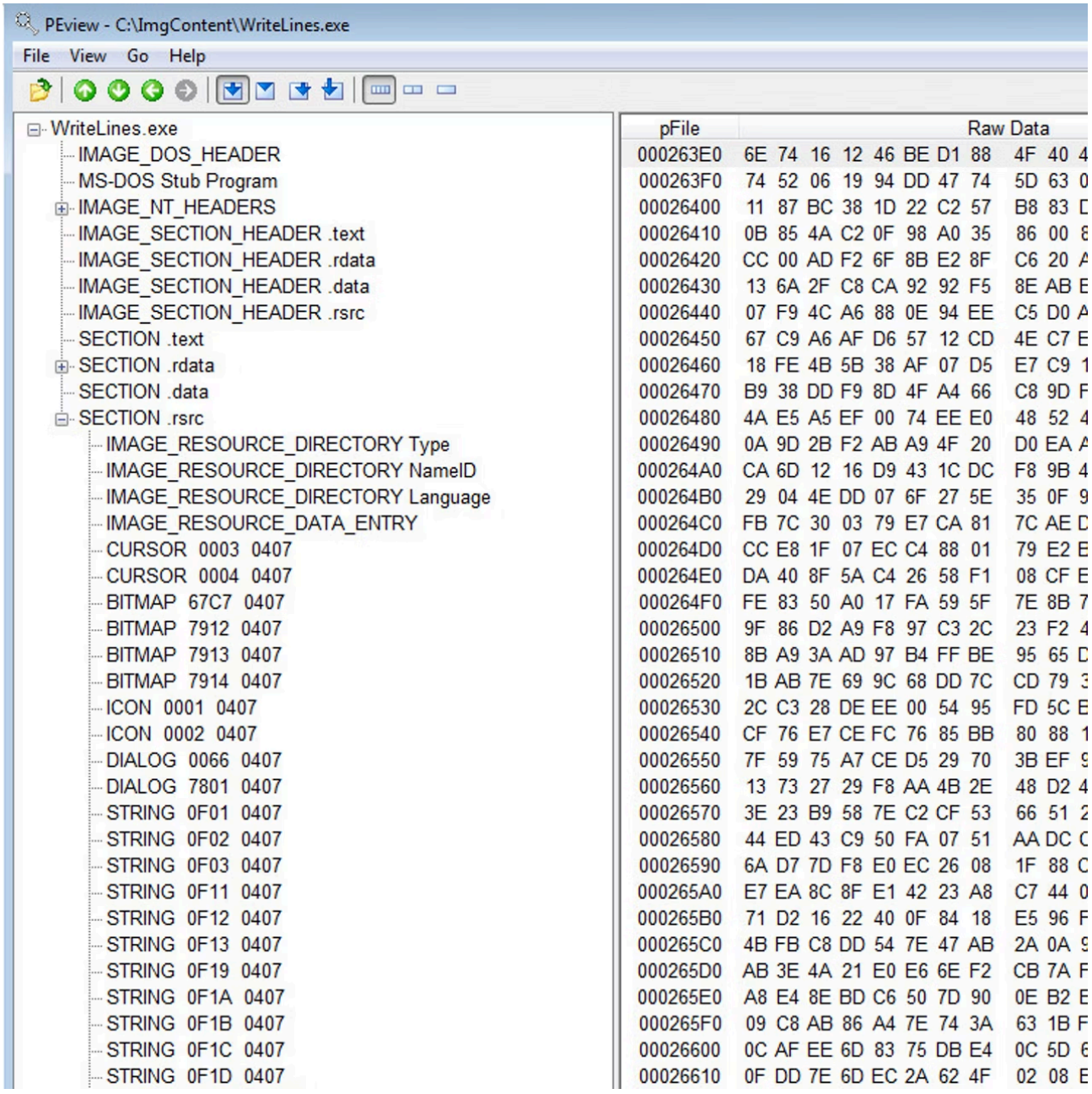



*Figure 1: IcedID loader running encrypted code*

1. As part of the code being executed, the loader uses some newly added anti-VM & anti-debugging techniques in order to check if it's being run in a VM or a sandbox. The output of these checks, alongside other parameters, is then sent to the attacker's command and control (C&C/C2) server to determine if the victim's machine should be infected or not.If the malware is to proceed to infect the machine, the C2 server sends back the necessary files to execute, like the malware's configuration file, the actual malicious payload (saved as encrypted and packed .png file) and a few other accompanying files.The loader then copies itself into two locations:
    1. C:\ImgContent\

    Hiding code inside an image is a technique known as steganography.

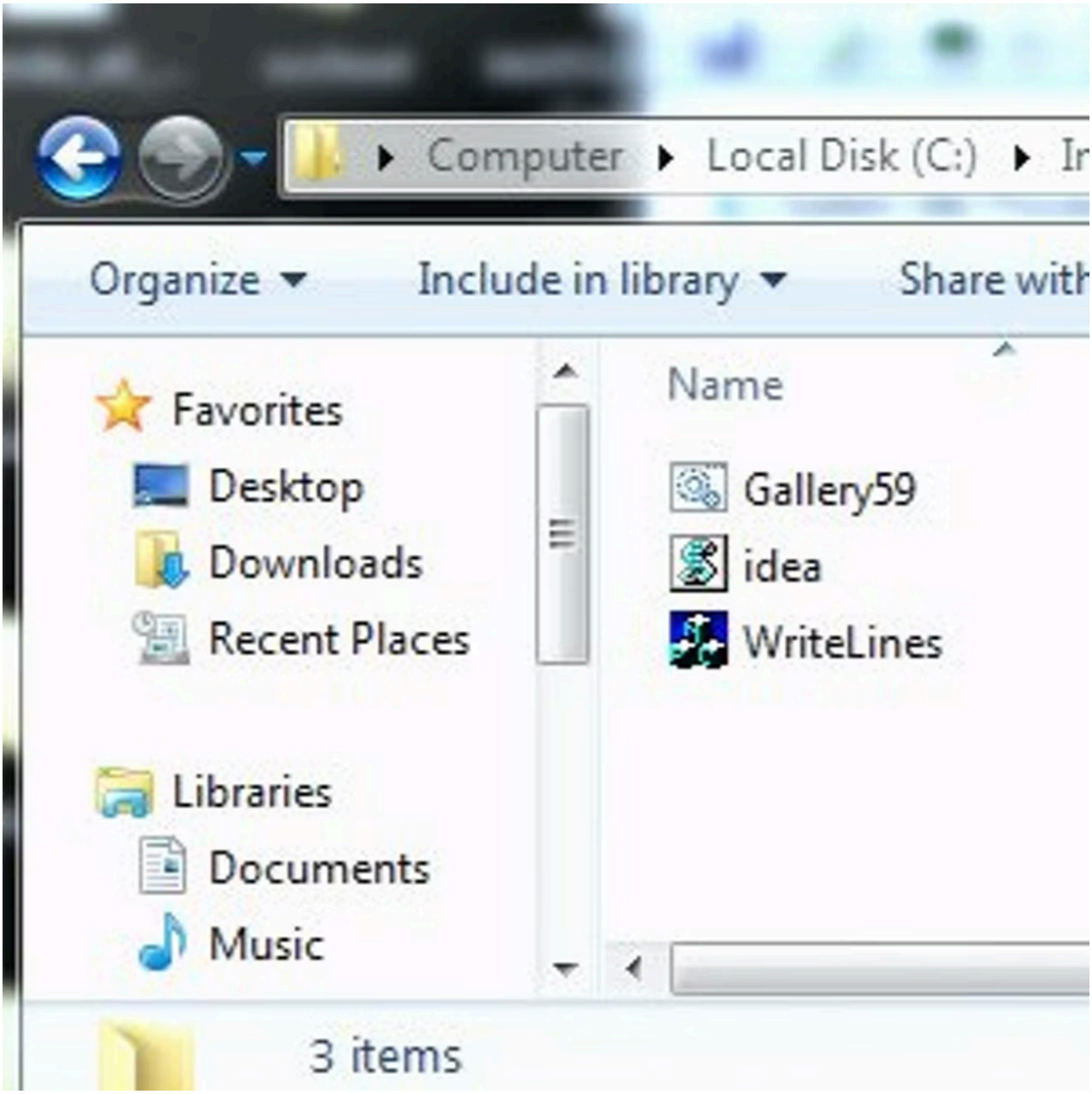



*Figure 2: IcedID loader copies itself to local disk*

1. %appdata%\local\[user]\[generated_name]\

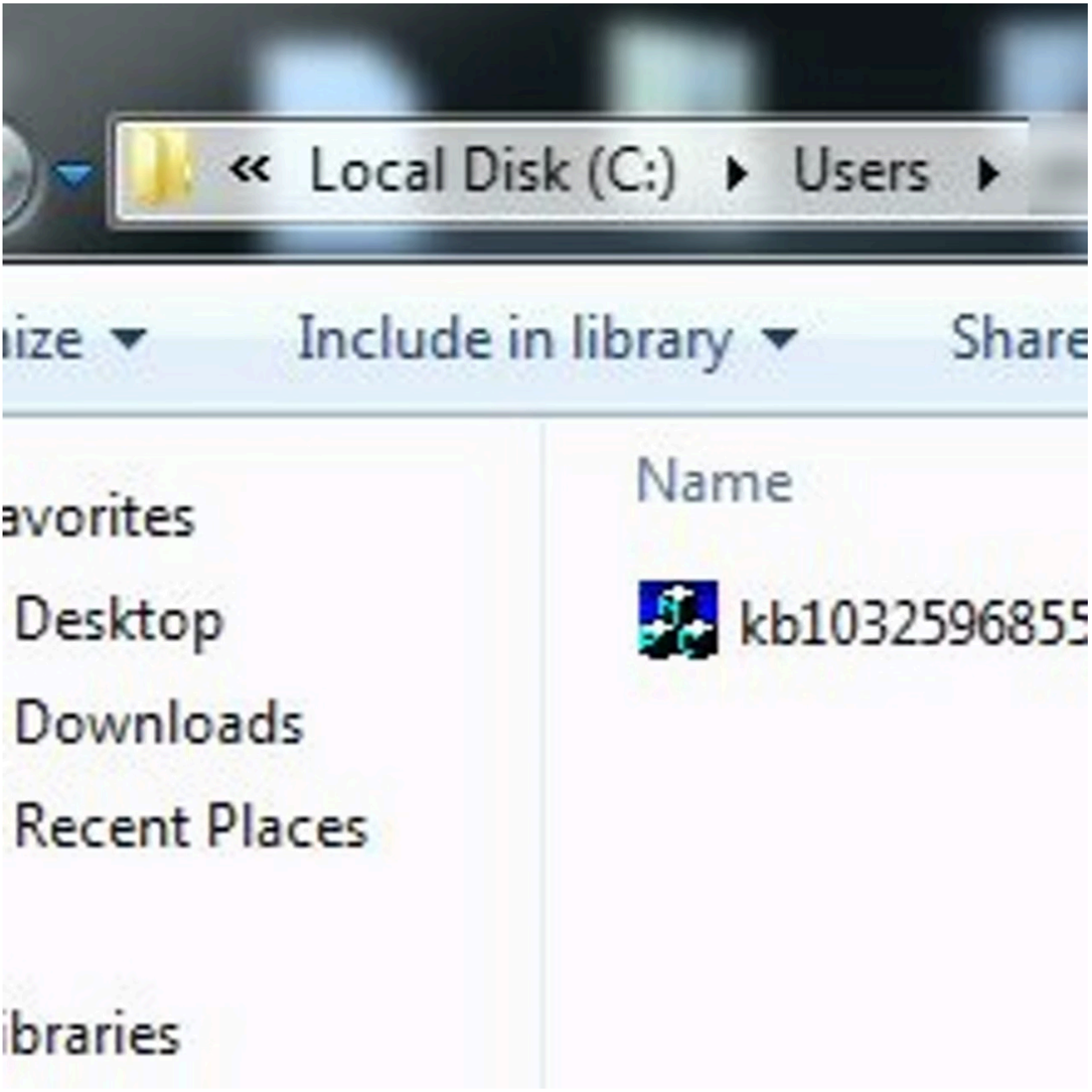



*Figure 3: IcedID loader copies itself to a second location on the local disk*

Next, in order to maintain persistency on the victim's machine, the malware creates a task in the task scheduler triggering a run of the loader at log on and every hour.

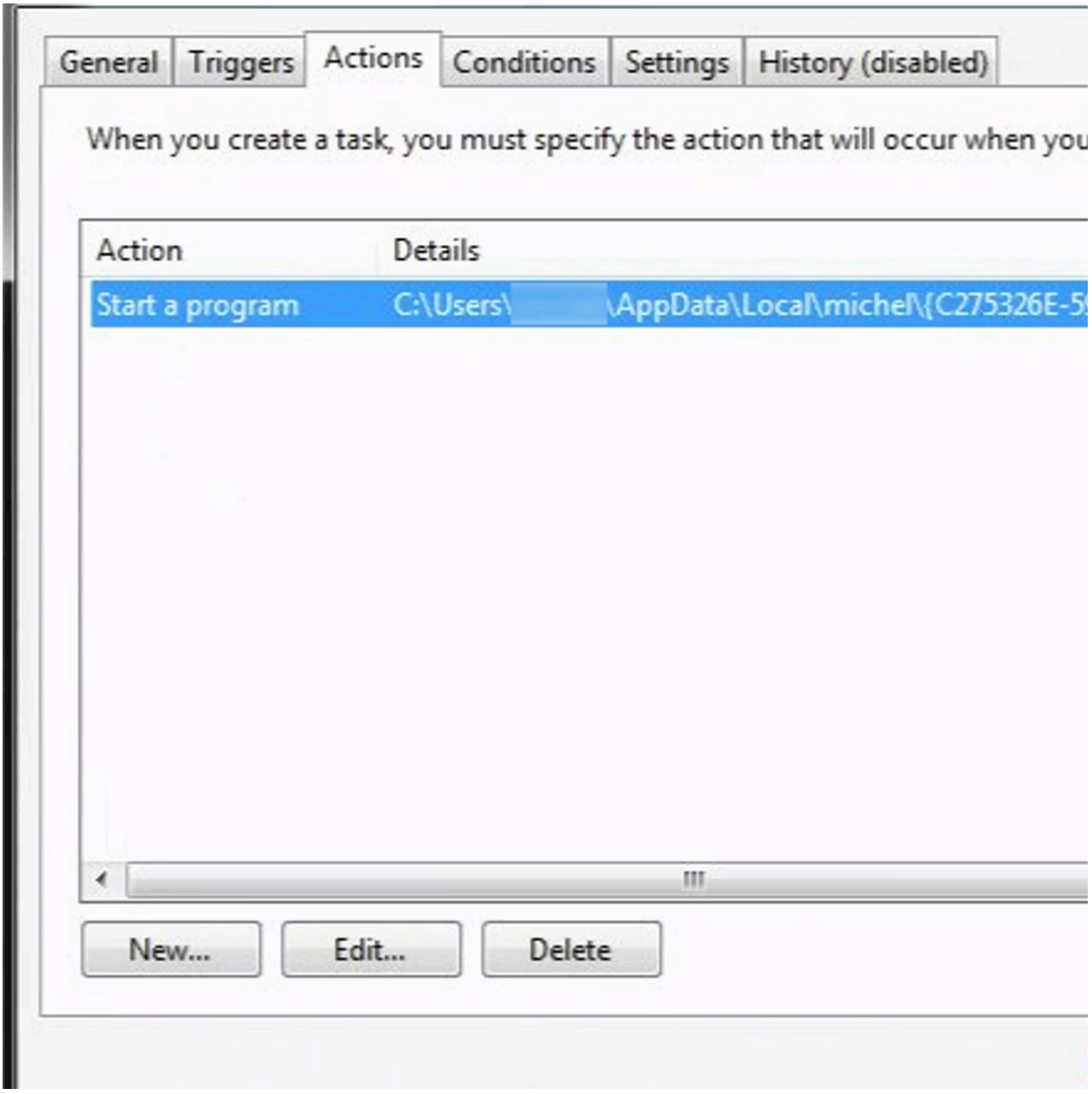



*Figure 4: IcedID task scheduler*

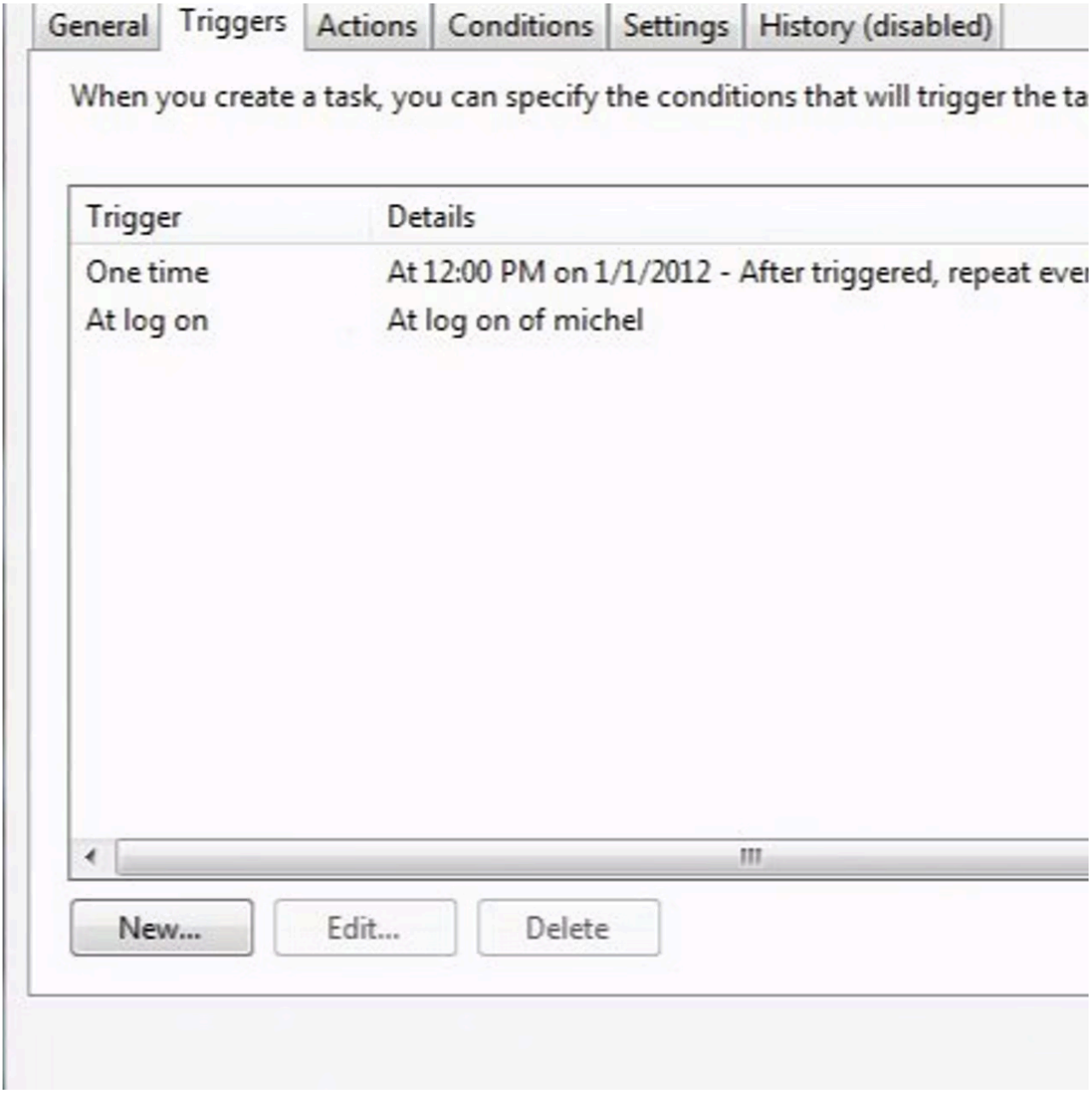



*Figure 5: IcedID task scheduler*

After copying itself to these two locations and after creating the persistency mechanism, it executes couple of anti-VM and anti-debugging checks to allow the C2 to ‘approve’ proceeding to infect the machine. The loader downloads all the necessary files in order to execute.

In the next section, we will go over the anti-VM and anti-debugging checks the loader performs.

One of the important files the loader fetches is the malicious payload that contains all of IcedID’s logic and which is actually the main module of the malware. It saves this core module under the path “%appdata%\local\user_name\photo.png“. This file is initially encrypted and packed.

The path and the file name of the downloaded payload are both hardcoded.

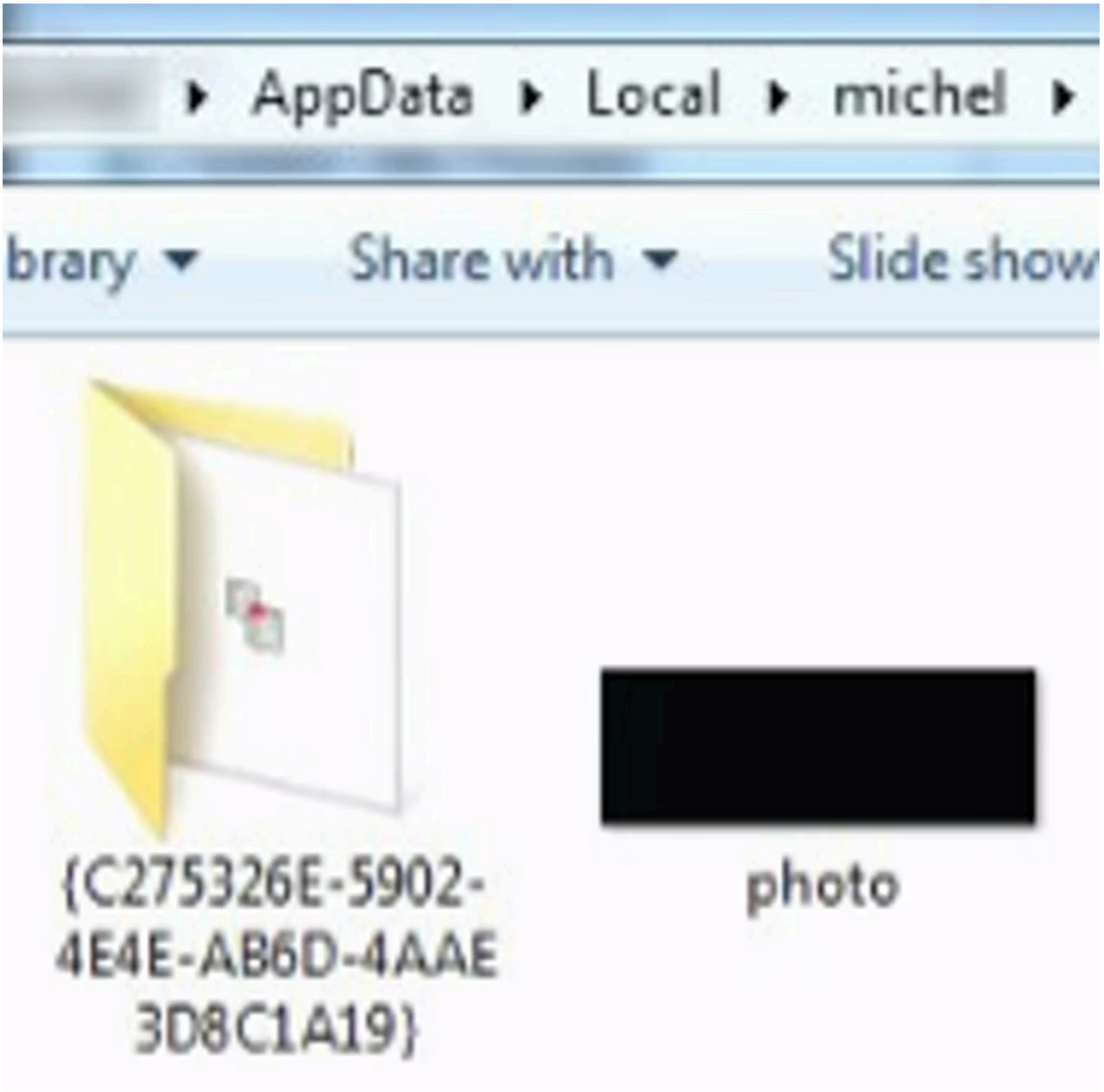



*Figure 6: IcedID is saved as a packed, encrypted image*

The loader reads the downloaded payload, which is saved as an encrypted .png file, then decrypts it, and will later inject it to a newly created svchost process.


*Figure 7: Loader reads IcedID payload*

Next, the loader creates a new process, svchost, and injects the decrypted IcedID payload into it.

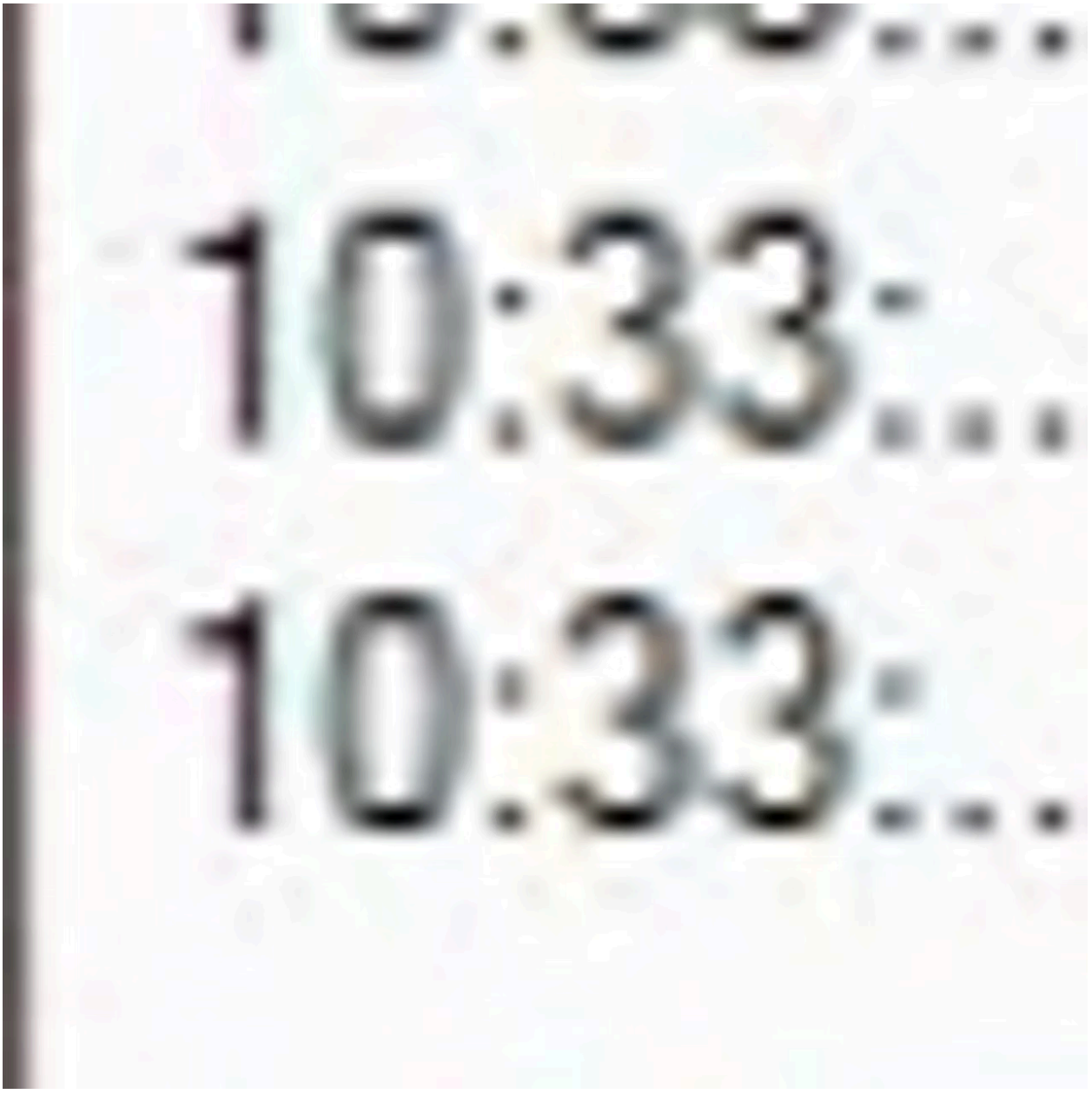



*Figure 8: Process injection into a svchost process*

## IcedID's injected payload

IcedID's actual payload starts executing once it is injected to a newly created svchost process.

Since the injected IcedID payload is originally packed, it begins by unpacking itself.

After unpacking itself, the core IcedID module performs some initialization steps in order to run properly. It downloads and reads different files necessary for the execution flow, like the configuration file (chercrgnaa.dat in this case), certificate file (3D8C2D77.tmp in this case), and other supporting resources.

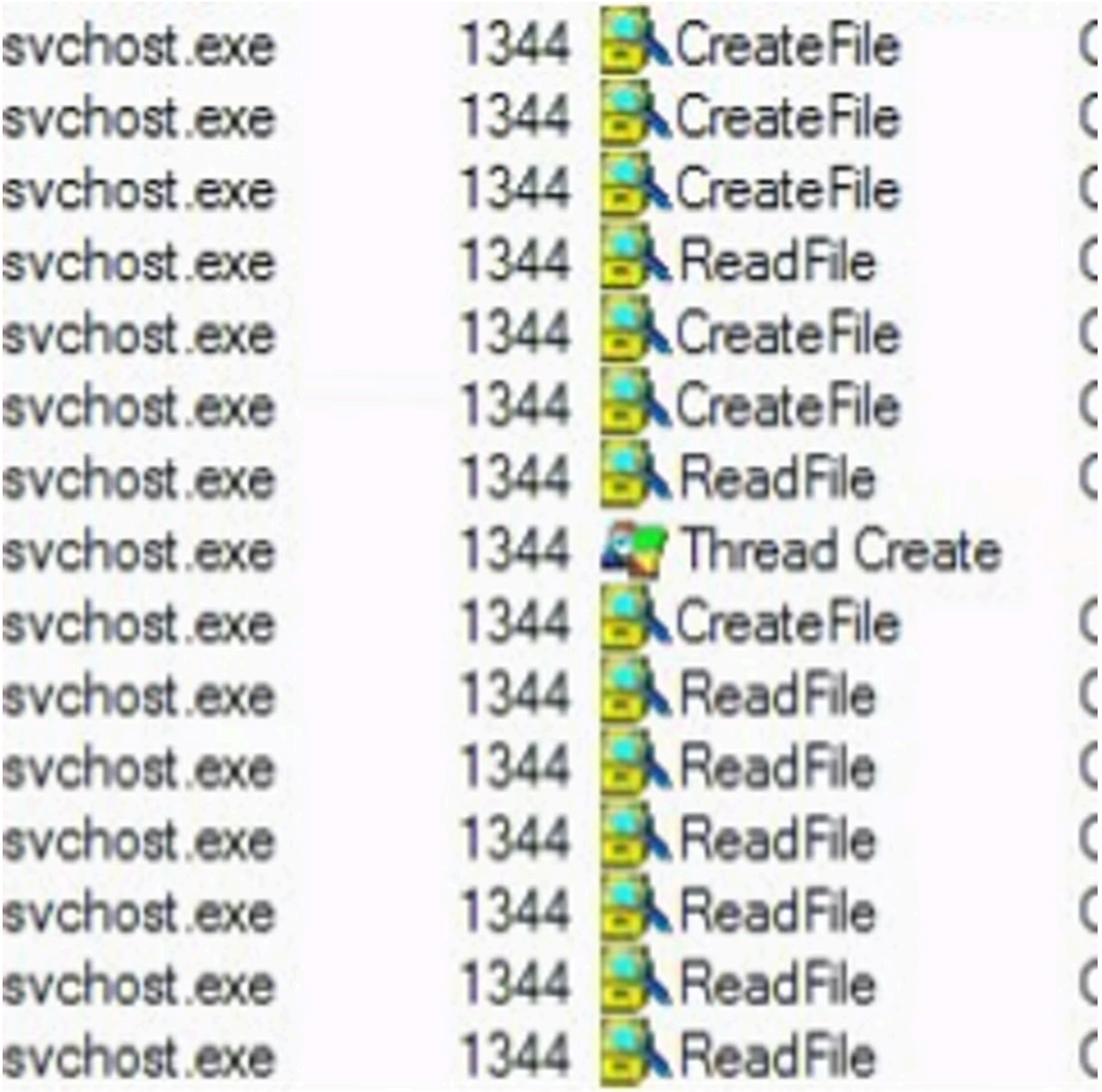



*Figure 9: IcedID payload launched into action*

The IcedID module checks to make sure there is no other instance of that same file and that it is not already running, gets some information about the victim's system, checks the versioning, sends some information to the C2 and starts scanning for running browser processes in order to inject its payload into them and hook them. The browser hooking will allow IcedID to detect targeted URLs and deploy web-injections accordingly.

Infection process via loader and IcedID payload:

1. The loader communicates with the C2 to fetch the IcedID payload
2. The downloader downloads the malicious payload – a file named photo.png*
3. The loader runs photo.png
4. The loader creates a new instance of svchost.exe
5. Loader injects the malicious payload into the new svchost.exe process

The second step in the list above happens only the first time the malware is run, or when it updates and downloads a newer version of the malicious payload, photo.png. That happens because after downloading the photo.png file, it saves it to a disk so that every time the malware is launched, it simply has to load the file from the disk into memory.

See the naming algorithms section – svchost mutex name – for more information.

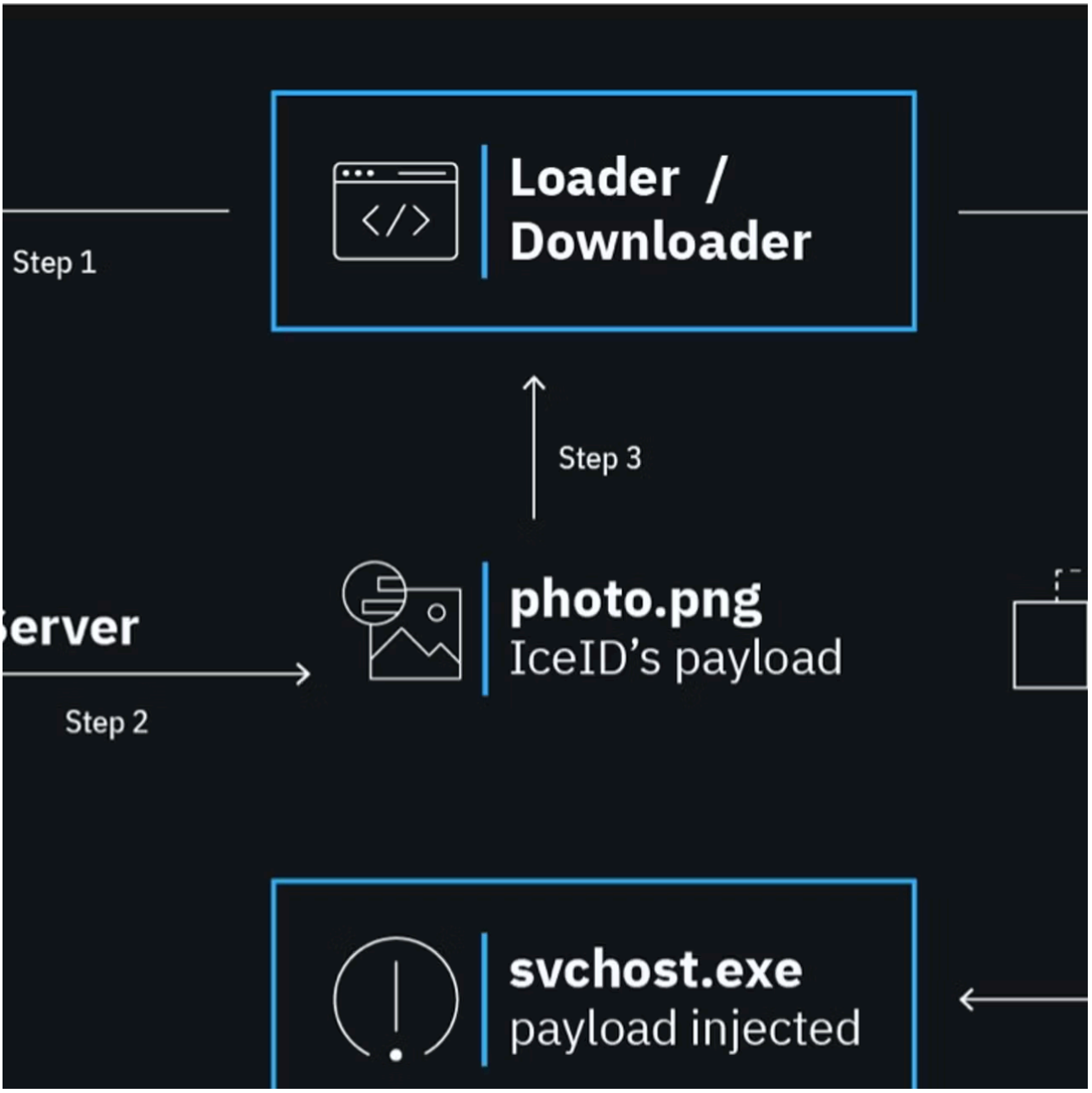



*Figure 10: Infection process via loader and IcedID payload*

## Anti-VM and anti-debugging tactics

IcedID's new version has been upgraded with additional abilities to hide itself and detect when it is being run in a virtual environment or in debug mode. **In previous versions, this malware did not feature these evasion techniques.**

The checks start with the loader that's programmed to identify if it is running in a virtualized environment or under a debugger. In the images below, we can see the function named "anti_vm" that checks whether the malware is running in an emulator.

The loader also uses some anti-debugging and anti-VM instructions such as the Read-Time-Stamp-Counter instruction (RDTSC), which can help it detect if a researcher is pausing the debugger in different steps., if it is being run under a debugger. It uses CPUID with 0x40000000 as a parameter looking for hypervisor brands, CPUID and more.

We can see in the image below that the loader runs in a loop 255 times. Inside the loop, it executes the RDSTC instruction at both the beginning and at the end of the loop, and in between it executes the command CPUID with 0x01 as a parameter.

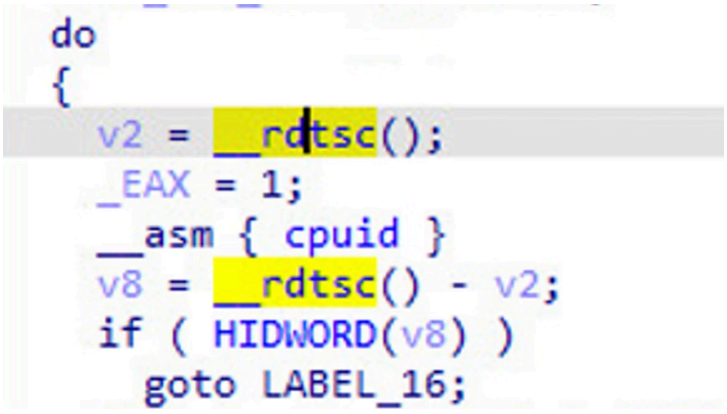



*Figure 11: Using the RDTSC instruction to detect running under debug mode*

Usually, the output of the CPUID instruction with 0x01 as a parameter is used to detect VMs, but here it ignores the output and only calculates the time difference between the first and the second calls of RDSTC.

Depending on the calculated delta, the loader increments the relevant counter:

0 < Difference < 250? ++less_250_miliseconds

250 < Difference < 500? ++less_500_miliseconds

500 < Difference < 750? ++less_750_miliseconds

750 < Difference < 1000? ++less_1000_miliseconds

else? ++Big_Gap

Next, the malware performs yet another test in order to validate if it is running in a VM or on a physical machine:

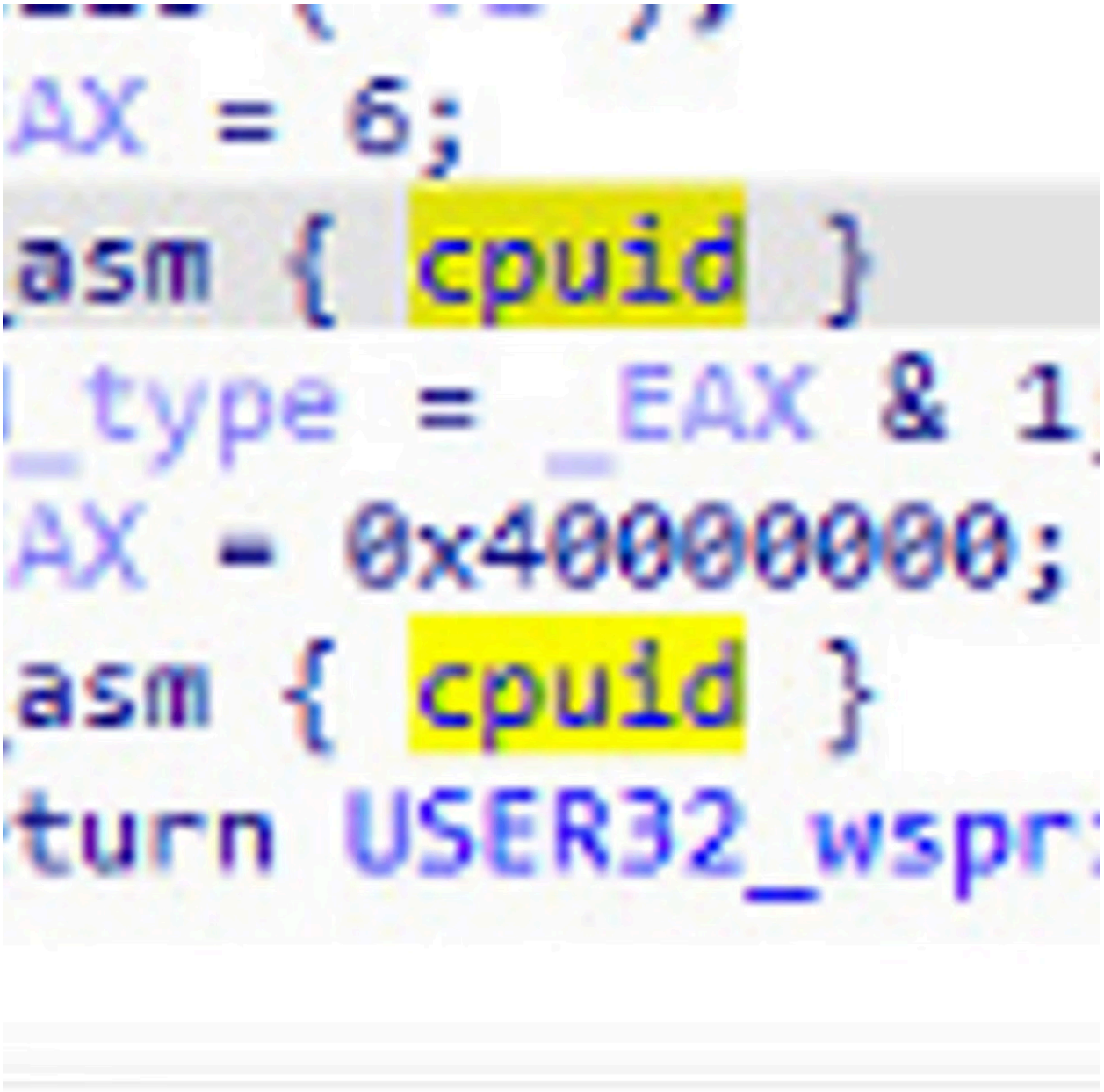



Figure 12: Malware checking if it is being run in a test environment

It calls CPUID with 0x40000000 as a parameter and the output is a value that indicates the VM vendor brand and type when it is run inside a virtual machine:

VMM_XEN: ebx = 0x566e6558 and ecx = 0x65584d4d and edx = 0x4d4d566e

VMM_HYPER_V: ebx = 0x7263694D and ecx = 0x666F736F and edx = 0x76482074

VMM_VMWARE: ebx = 0x61774d56 and ecx = 0x4d566572 and edx = 0x65726177

VMM_KVM: ebx = 0x4b4d564b and ecx = 0x564b4d56 and edx = 0x0000004d

All this collected data is later sent to the C2 server to help determine if the malware is being run in a VM or debugged environment. According to the output, the C2 server decides whether to send all the files necessary to infect the machine and run the IcedID core module.

## Code injection techniques

IcedID has two code injection methods.

The first injection method is used when the malware is launched for the first time or at system start-up. This first method injects code into a spawned svchost.exe process.

The second injection method takes place when the malware detects that a browser process is running in the background and injects its shellcode to that running browser process.

IcedID uses a slight code obfuscation to make it difficult to analyze. The code obfuscation works by calling Windows API functions indirectly instead of calling the functions directly using dynamically loading the Windows API functions it calls into Registers.

Let’s look at these two techniques more visually. IcedID has two code injection methods.

### IcedID’s svchost process injection method

In the first code injection type, the malware begins by creating a svchost process in suspend state.

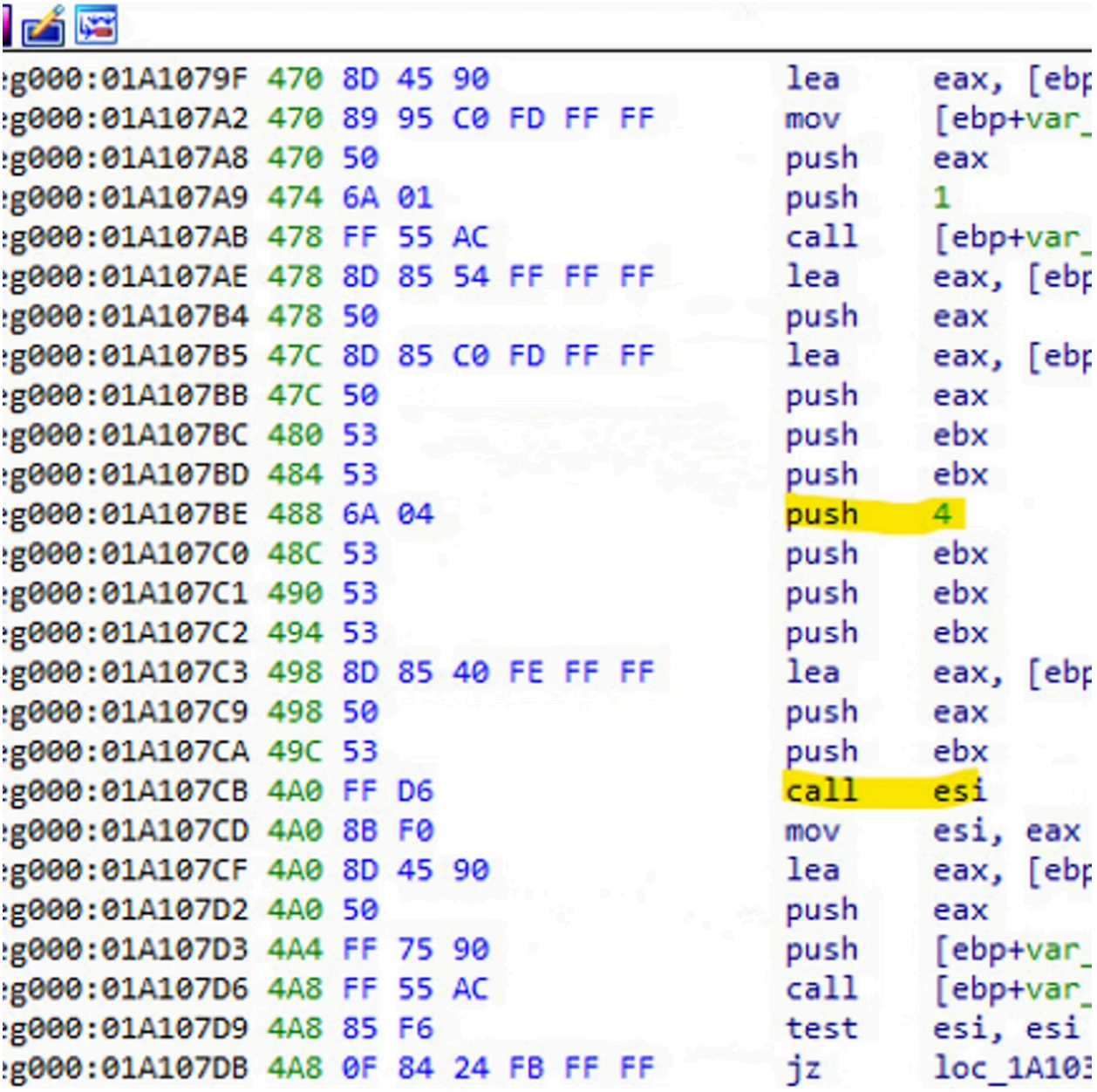



*Figure 13: IcedID code injection process*

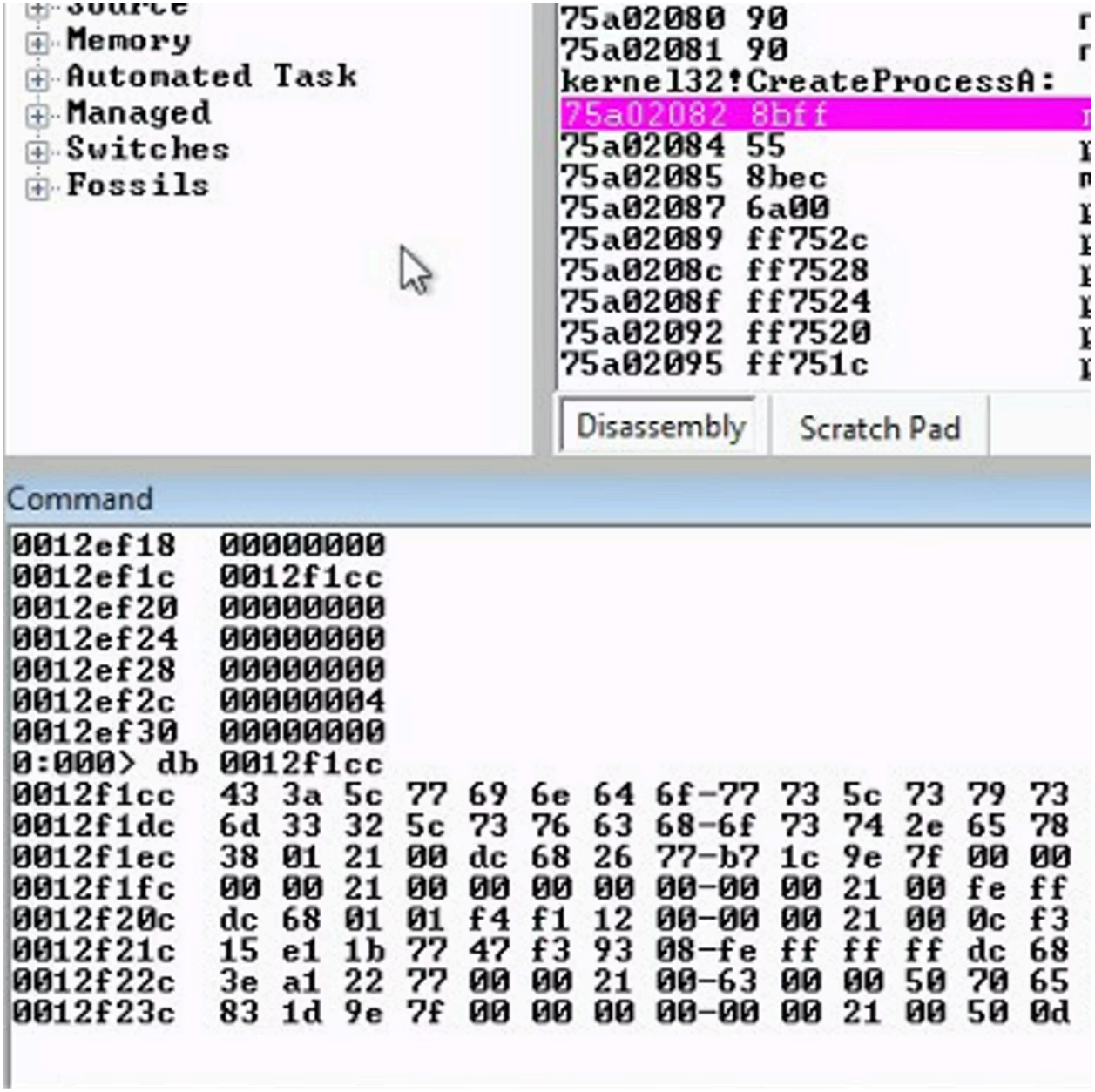



*Figure 14: IcedID code injection process*

Next, it allocates memory in the target process (svchost.exe) and injects its shellcode into the allocated memory. It then changes the protection flag of the allocated memory space to PAGE_READ_EXECUTE.

Next, it calls the function “NtQueueApcThread” with the main thread of the injected process and the entry point address in the injected shellcode.

At the end of the injection process, it calls “NtResumeThread” in order to run its own code in the now injected process (svchost.exe).

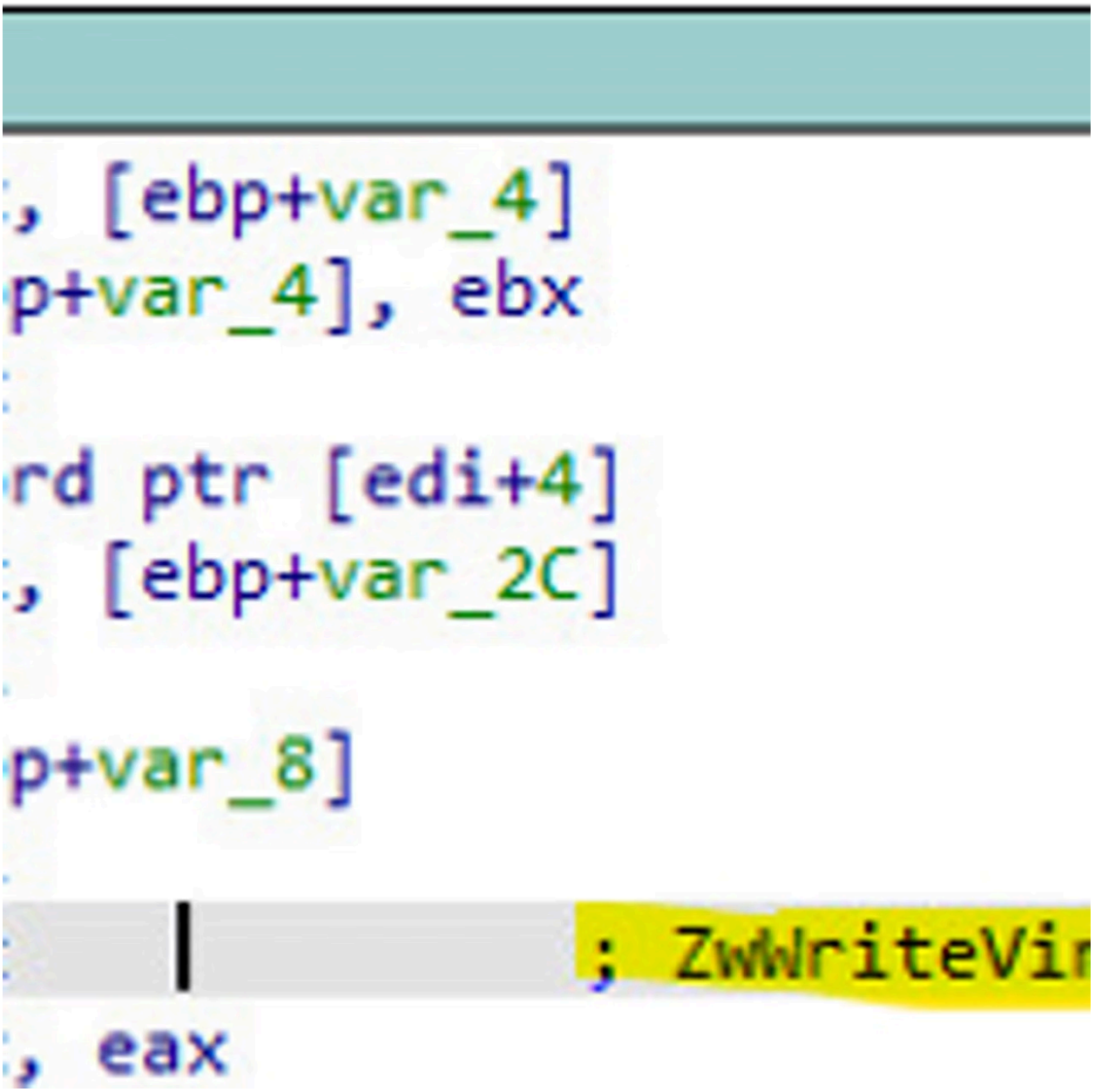



*Figure 15: IcedID code injection process*

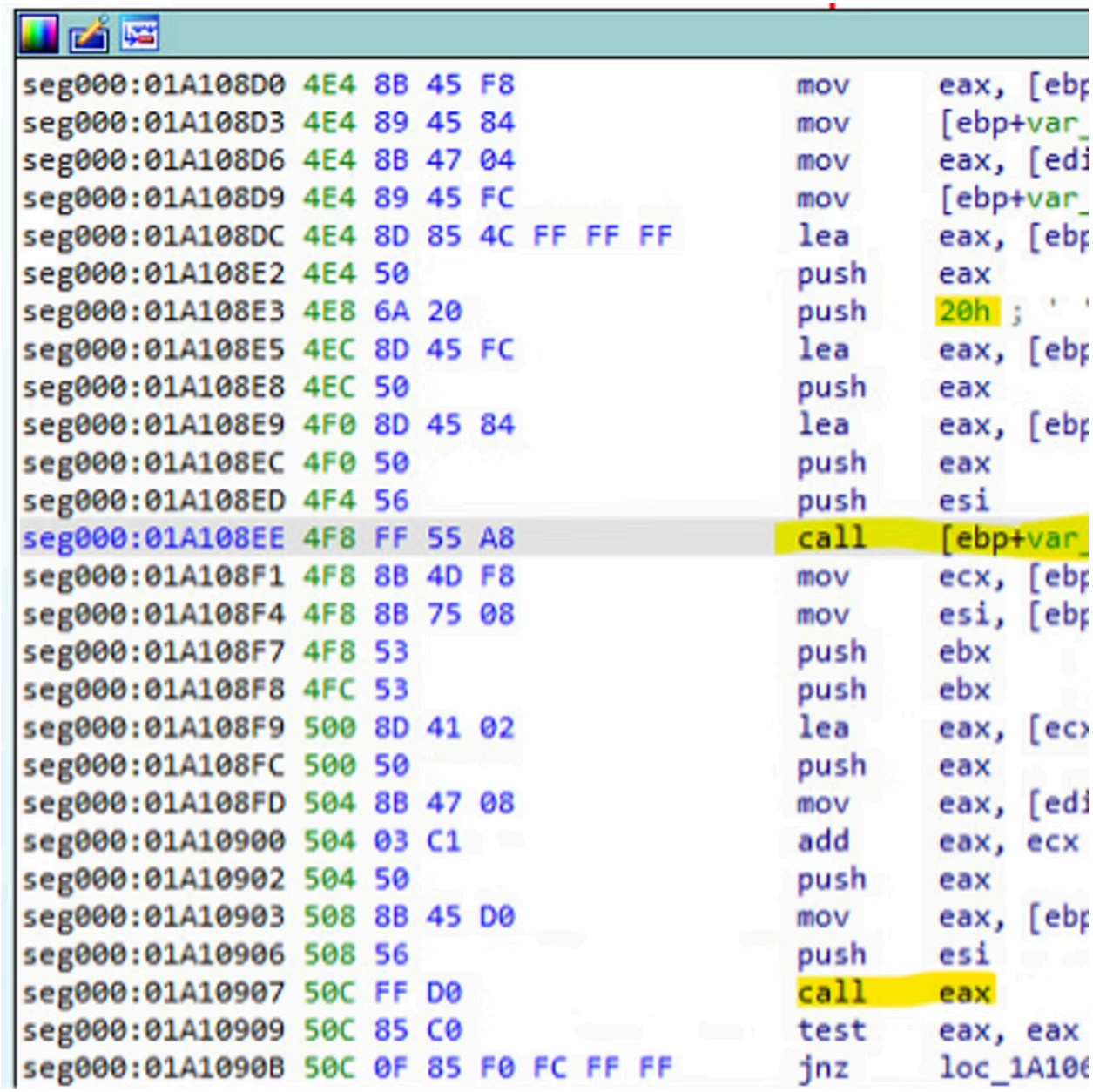



*Figure 16: IcedID code injection process*

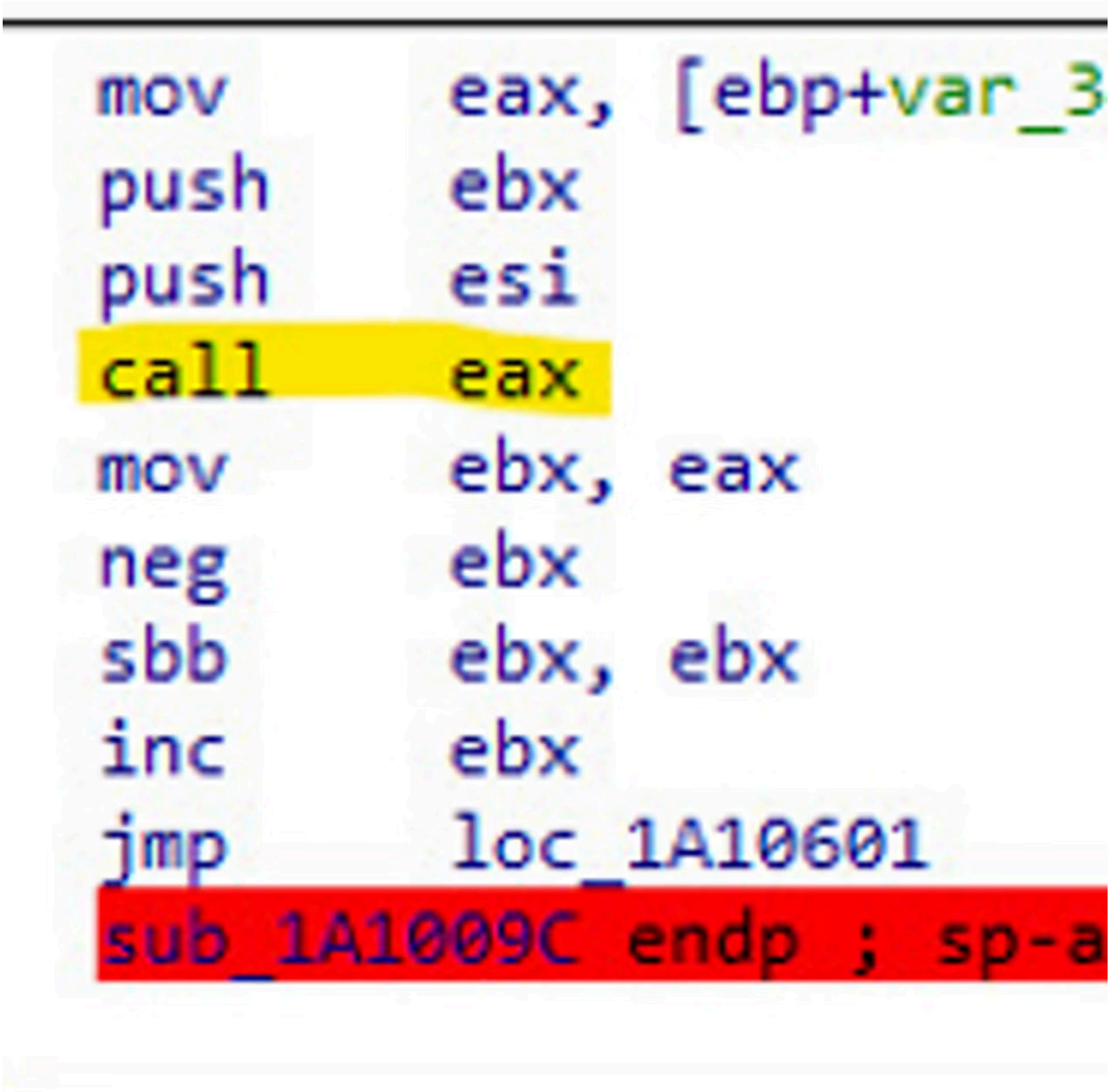



*Figure 17: IcedID code injection process*

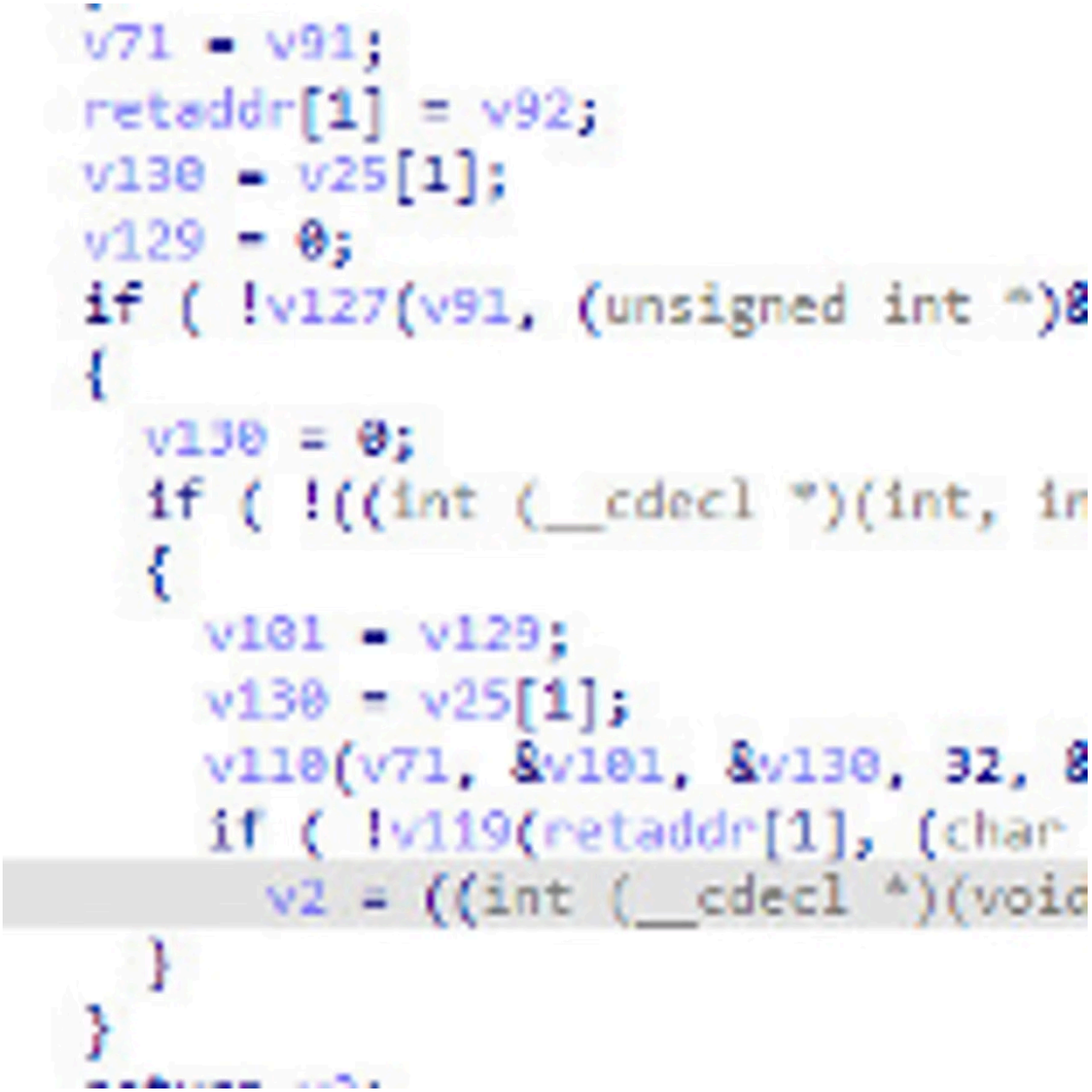



*Figure 18: IcedID code injection process*

**Svchost –> Browser injection method**

This second case takes place when the malicious svchost process detects that a browser process was launched. It gets a handle to the process and calls the function “Inject_browser” as shown in the figure below.

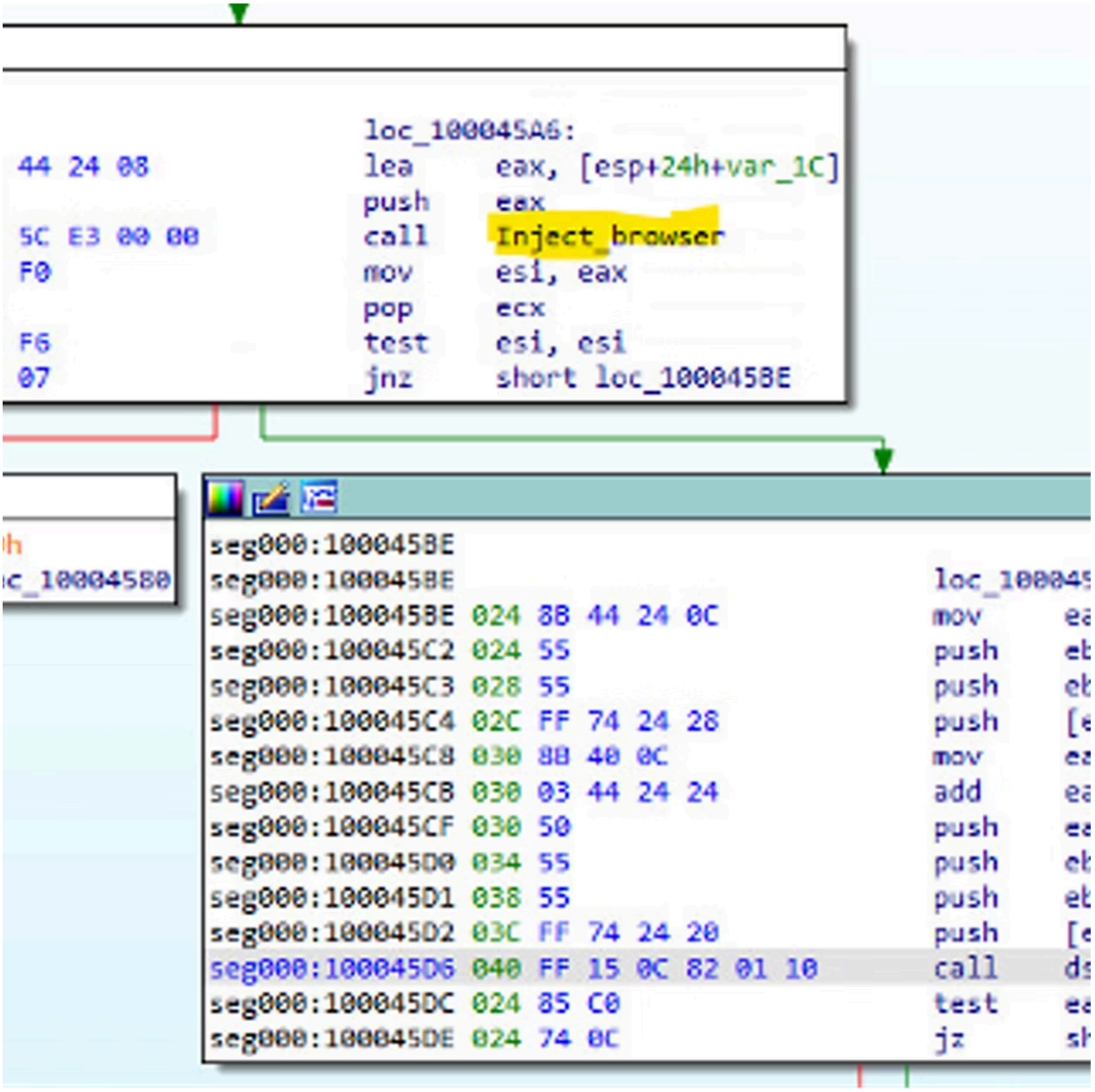
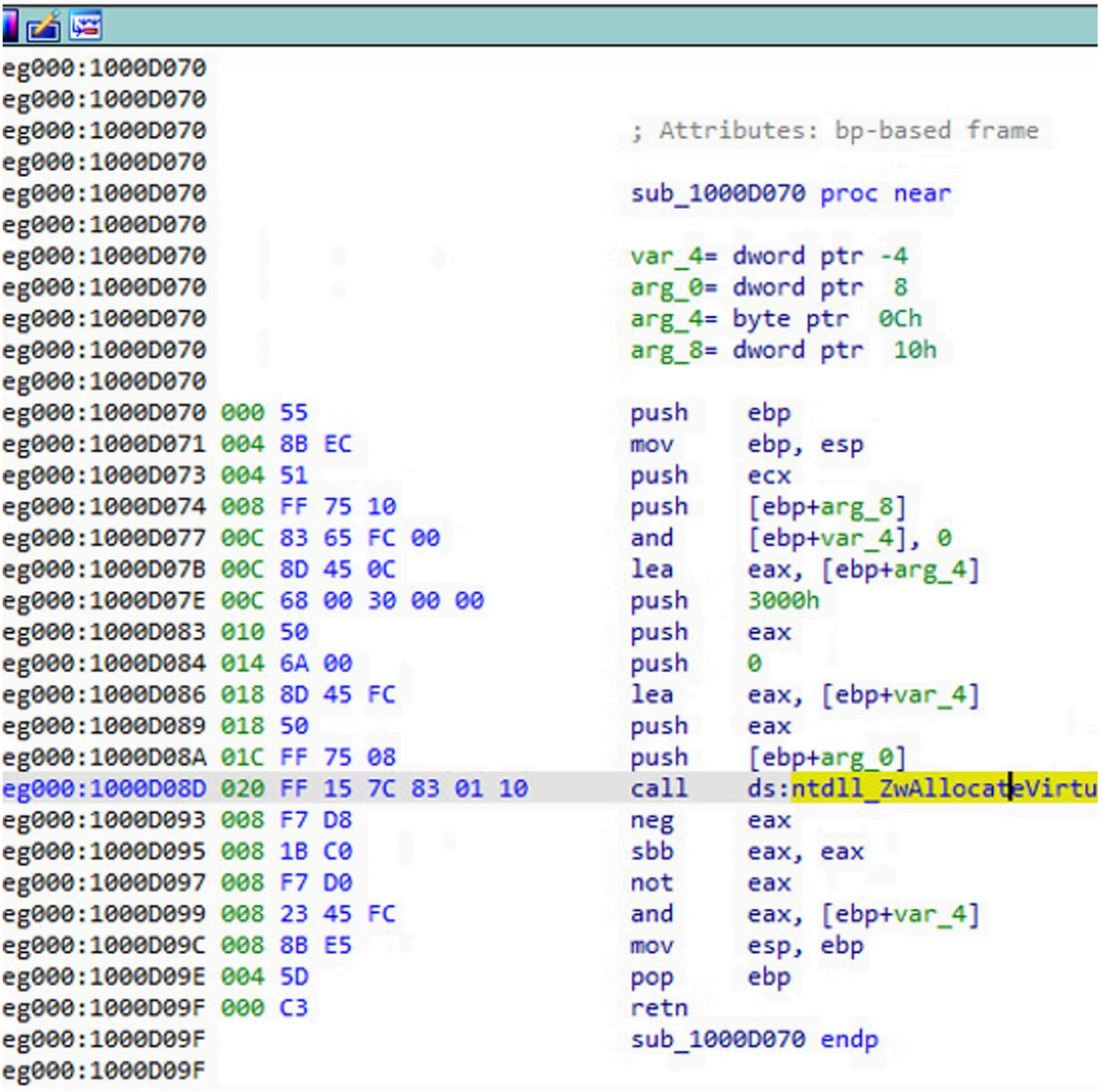



*Figure 19: IcedID browser injection process*

Inside “Inject_browser” it calls three functions — “ZwWritevirtualMemory”, “NtProtectVirtualMemory” and “ZwAllocatevirtualMemory” — in order to inject its shellcode into the browser’s process.

After injecting the shellcode into the browser process, it calls “CreateRemoteThread” with the entry point address in the injected shellcode.



```
eg000:1000D070
eg000:1000D070
eg000:1000D070                          ; Attributes: bp-based frame
eg000:1000D070
eg000:1000D070                          sub_1000D070 proc near
eg000:1000D070
eg000:1000D070                          var_4= dword ptr -4
eg000:1000D070                          arg_0= dword ptr  8
eg000:1000D070                          arg_4= byte ptr  0Ch
eg000:1000D070                          arg_8= dword ptr  10h
eg000:1000D070
eg000:1000D070 000 55                   push    ebp
eg000:1000D071 004 8B EC                mov     ebp, esp
eg000:1000D073 004 51                   push    ecx
eg000:1000D074 008 FF 75 10             push    [ebp+arg_8]
eg000:1000D077 00C 83 65 FC 00          and     [ebp+var_4], 0
eg000:1000D07B 00C 8D 45 0C             lea     eax, [ebp+arg_4]
eg000:1000D07E 00C 68 00 30 00 00       push    3000h
eg000:1000D083 010 50                   push    eax
eg000:1000D084 014 6A 00                push    0
eg000:1000D086 018 8D 45 FC             lea     eax, [ebp+var_4]
eg000:1000D089 018 50                   push    eax
eg000:1000D08A 01C FF 75 08             push    [ebp+arg_0]
eg000:1000D08D 020 FF 15 7C 83 01 10    call    ds:ntdll_ZwAllocateVirtu
eg000:1000D093 008 F7 D8                neg     eax
eg000:1000D095 008 1B C0                sbb     eax, eax
eg000:1000D097 008 F7 D0                not     eax
eg000:1000D099 008 23 45 FC             and     eax, [ebp+var_4]
eg000:1000D09C 008 8B E5                mov     esp, ebp
eg000:1000D09E 004 5D                   pop     ebp
eg000:1000D09F 000 C3                   retn
eg000:1000D09F                          sub_1000D070 endp
eg000:1000D09F
```

*Figure 20: IcedID browser injection process*

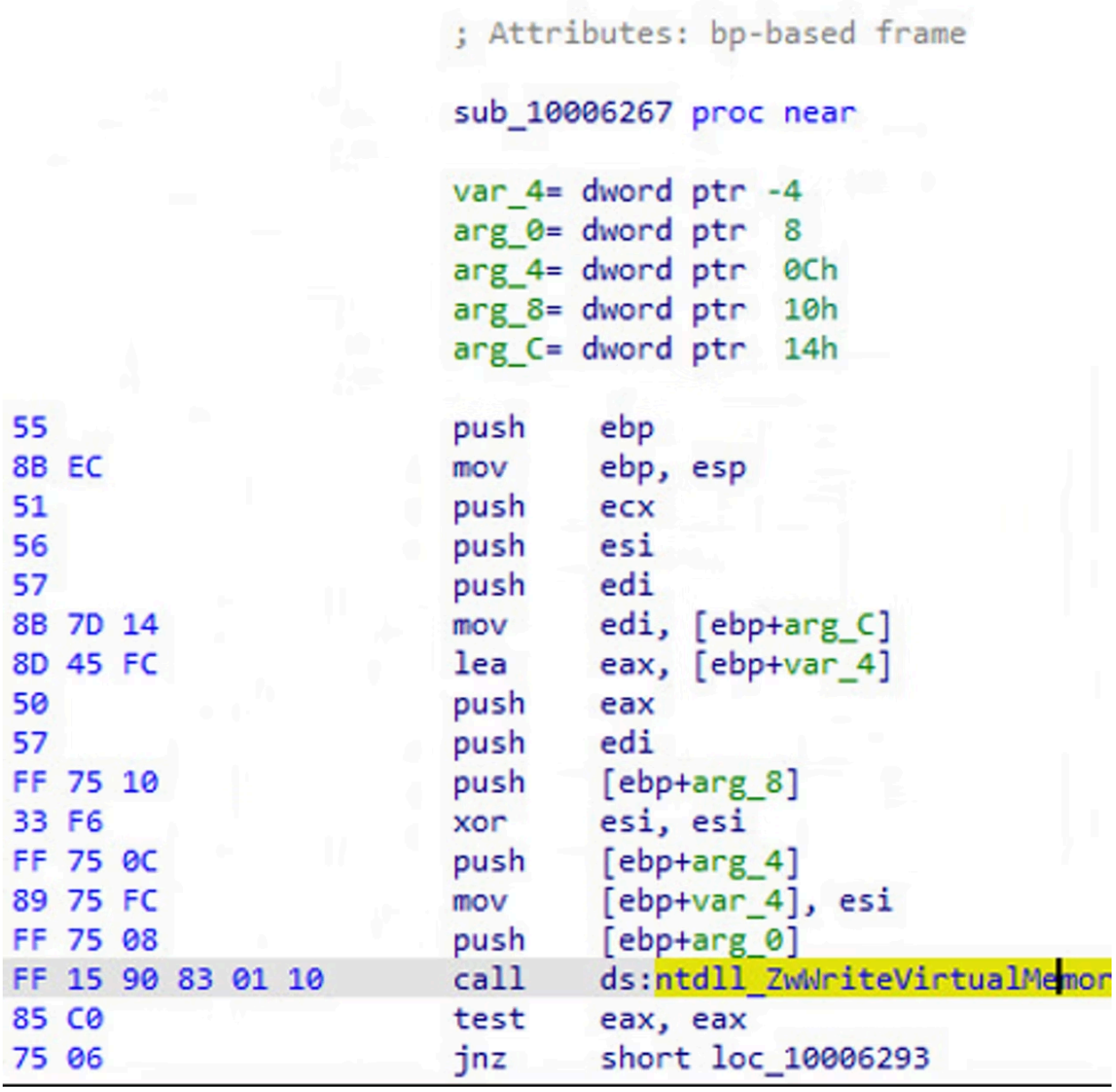



*Figure 21: IcedID browser injection process*

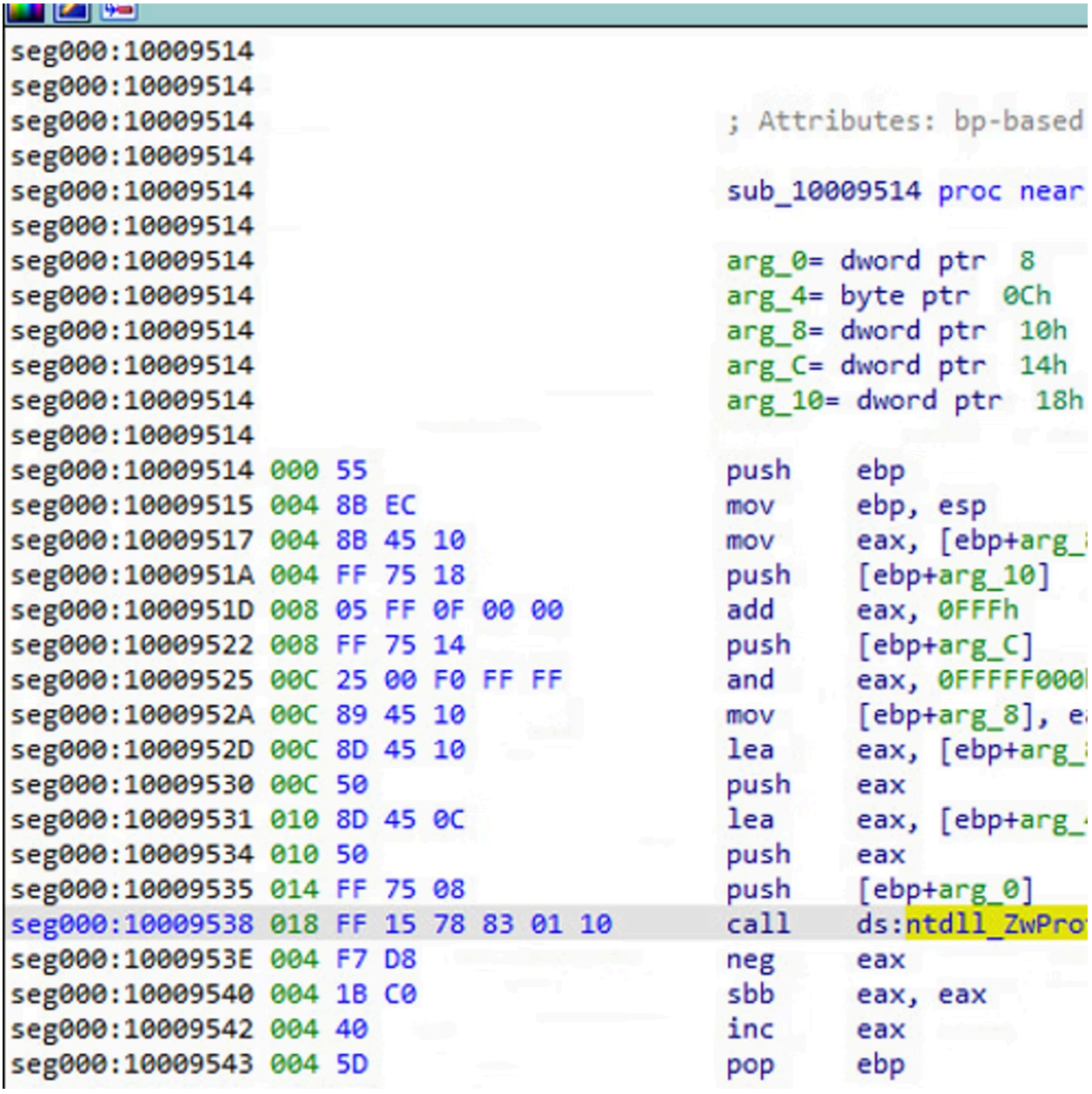



*Figure 22: IcedID browser injection process*

## IcedID's cryptographic choices and techniques

IcedID uses a few different decryption and decoding algorithms to hide some artefacts that can help malware researchers understand the context of its functions and operational flow.

The cryptographic functions are being used in all processes that IcedID creates or injects, namely svchost and web browser processes.

IcedID's cryptography functions are as follows:

**1. Decode (int a1)**

Usually used with other decryption algorithms — for example, the algorithms that decode browser event name or mutex in svchost process. The function gets one argument, an integer, and runs a few bitwise operations on this parameter.

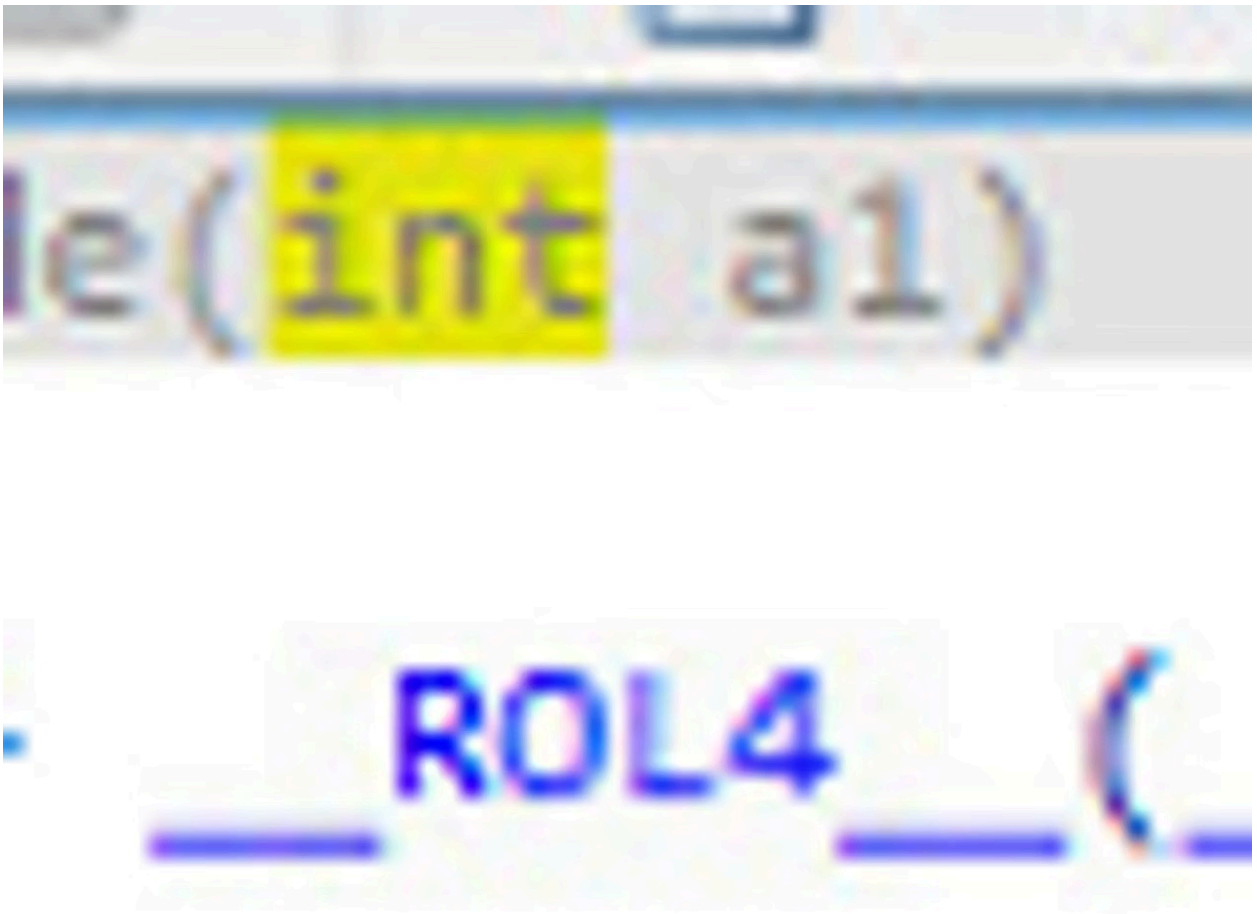



*Figure 23: IcedID decoding function Decode (int a1)*

**2. Decrypt (int key, string encrypted_string)**

Usually used to decrypt encrypted strings and artefacts in the code in order to harden the reverse-engineering process. The function gets two arguments: key (integer) and string to decrypt (string). This function remains unchanged from the last version of IcedID.

```
{
  unsigned int v3; // [esp+0h] [ebp-Ch
  unsigned __int16 v4; // [esp+4h] [eb
  unsigned __int16 i; // [esp+8h] [ebp
  int k; // [esp+14h] [ebp+8h]

  v3 = *key;
  v4 = *key ^ *(key + 4);
  k = key + 6;
  for ( i = 0; i < v4; ++i )
  {
    v3 = i + ((v3 << 29) | (v3 >> 3));
    *(encrypted_string + i) = v3 ^ *(k
  }
  return encrypted_string;
}
```

*Figure 24: IcedID decryption function*

**3. Decode_by_constant (int key, char[] arr)**

Responsible for generating event names for browser processes that IcedID manages to infect. The function gets two arguments: constant\key (integer) and array (char[]).

```
unsigned int v2; // ecx
unsigned int counter; // edi
unsigned int v4; // esi

v2 = decode(key + init_key);
counter = 0;
v4 = (v2 & 7) + 5;
if ( (v2 & 7) != -5 )
{
  do
  {
    v2 = decode(v2);
    *(_WORD *)(arr + 2 * counter++) = v2 %
  }
  while ( counter < v4 );
}
*(_WORD *)(arr + 2 * v4) = 0;
return v4;
```

*Figure 25: IcedID Decode_by_constant function*

**4. CreateGlobalKey (string sid)**

Creates a global key that is then used for other encryption and decryption algorithms and for naming algorithms. This function gets one argument: the system ID (SID) of the infected user's device. The algorithm being used here is the Fowler–Noll–Vo hash.

```
{
  char *v1; // ecx
  int i; // edx
  char v3; // al

  v1 = sid;
  for ( i = 0x811C9DC5; ; i = 0x10
  {
    v3 = *v1;
    if ( !*v1 )
      break;
    ++v1;
  }
  return i;
}
```

*Figure 26: IcedID using the Fowler–Noll–Vo hash non-cryptographic hash function*

**5. RC4_variant (int[] arr)**

This function is responsible for decrypting encrypted files, such as configuration files, the encrypted payload downloaded from the C2, the code loaded and injected to the svchost process, and more.

This encryption/decryption algorithm is an RC4 cipher *variant* that continues to use the string ‘zeus’ as the keyword like it does in previous versions.

There were slight changes in the RC4 cipher in this version which means that standard RC4 decryption algorithms in Python libraries will not work against only the RC4-encrypted data because it has been customized by IcedID’s developers. A custom or modified RC4 decryptor would have to be used to decrypt new configurations.

The function gets one argument – an array – that contains the encrypted payload to decrypt.

```
int __cdecl encrypt_decrypt(_DWORD *arr)
{
  unsigned __int8 v1; // bl
  int v2; // eax
  int v3; // eax
  int v4; // eax
  unsigned int i; // ebp
  char v6; // dl
  unsigned __int8 v8; // [esp+Dh] [ebp-101h]
  char key[256]; // [esp+Eh] [ebp-100h]

  v1 = 0;
  v8 = 0;
  if ( !arr )
    return 0;
  if ( !*arr )                                  // GlobalKey
    return 0;
  if ( !arr[1] )                                // const = 4
    return 0;
  if ( !arr[2] )
    return 0;
  v2 = arr[3];                                  // length
  if ( !v2 )
    return 0;
  if ( !arr[4] )
  {
    v3 = kernel32_GetProcessHeap(8, v2 + 1);
    v4 = ntdll_RtlAllocateHeap(v3);
    arr[4] = v4;
    if ( !v4 )
      return 0;
  }
  initKey(*arr, arr[1], (int)key);
  for ( i = 0; i < arr[3]; ++i )
  {
    v6 = key[++v1];
    v8 += v6;
    key[v1] = key[v8];
    key[v8] = v6;
```

*Figure 27: IcedID using the RC4 stream cipher to encrypt/decrypt resources*

**6. initKey (int GlobalKey, int constant, int[] key)**

Initializes the key for the RC4 variant decryption algorithm. It gets three parameters: the GlobalKey (integer), constant (integer) and an array of integer/chars.

```
unsigned int j; // edx
char v7; // dl
unsigned __int8 v8; // ah
unsigned __int8 v9; // [esp+13h] [ebp-1h]

v3 = 0;
for ( i = 0; i < 256; ++i )
  *(_BYTE *)(i + key) = i;
BYTE1(result) = 0;
for ( j = 0; j < 256; ++j )
{
  v7 = *(_BYTE *)(j + key);
  v8 = v7 + *(_BYTE *)(v3 + GlobalKey) + BYTE1(r
  v9 = v8;
  *(_BYTE *)(j + key) = *(_BYTE *)(v8 + key);
  *(_BYTE *)(v8 + key) = v7;
  result = ((unsigned int)v3 + 1) / constant;
  BYTE1(result) = v9;
  v3 = ((unsigned int)v3 + 1) % constant;
}
return result;
}
```

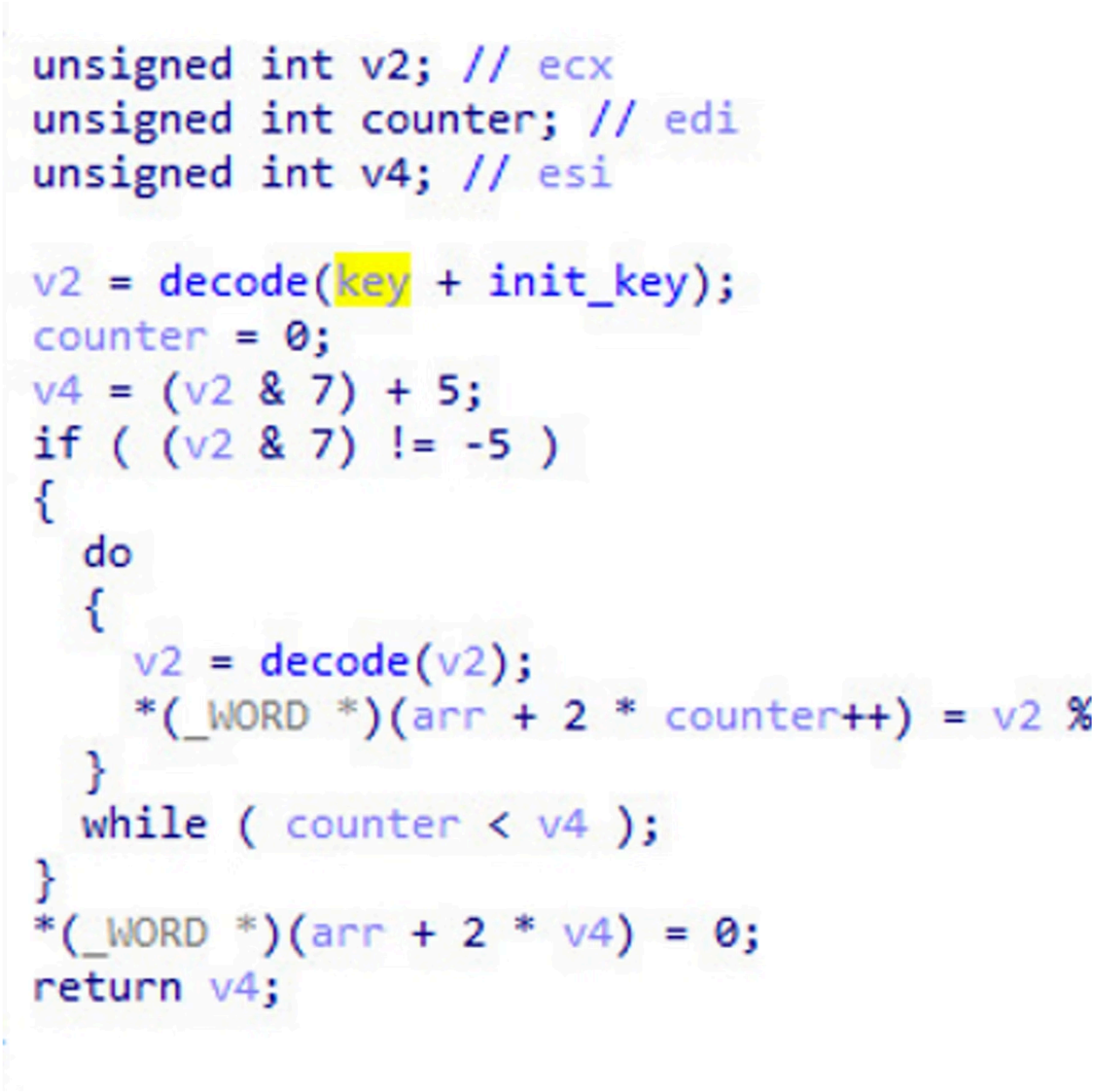
*Figure 28: IcedID initKey function*

## IcedID's naming algorithms

IcedID uses a few naming algorithms in order to generate names for its files, directories, mutexes, events, etc. Here are some of the different uses of IcedID's naming algorithms:

**1. Browser event name**

When the malware injects itself into a new browser process, it creates an event in order to know that this browser process has already been injected. It uses the function "decode_by_constant(int key, char[] arr)" in order to generate the event name. The event name is similar for all types of browsers. For the event name created in the browser, the key is hardcoded with the value 6 (key = 6).



```
unsigned int v2; // ecx
unsigned int counter; // edi
unsigned int v4; // esi

v2 = decode(key + init_key);
counter = 0;
v4 = (v2 & 7) + 5;
if ( (v2 & 7) != -5 )
{
  do
  {
    v2 = decode(v2);
    *(_WORD *)(arr + 2 * counter++) = v2 %
  }
  while ( counter < v4 );
}
*(_WORD *)(arr + 2 * v4) = 0;
return v4;
```

*Figure 29: IcedID uses a naming algorithm to generate names for events*

**2. Svchost mutex name**

```
1 int create_mutex_svchost()
2 {
3   unsigned int i; // esi
4   int v1; // edi
5   int result; // eax
6   int arr; // [esp+8h] [ebp-40h]
7
8   generate_mutex_name(7, (int)&arr);
9   for ( i = 0; i < 3; ++i )
0   {
1     v1 = kernel32_CreateMutexA(0, 0, &arr);
2     if ( v1 )
3     {
4       result = kernel32_GetLastError();
5       if ( result != 183 )
6         return result;
7       kernel32_CloseHandle(v1);
8     }
9     kernel32_Sleep(5000);
0   }
1   sub_10007955(2, 1, 268542552);
2   return kernel32_ExitProcess(0);
3 }
```

*Figure 30: IcedID uses a naming algorithm to name mutexes it creates*

When the malware is injected into a svchost process that it created, it also creates a mutex in order to know that this svchost process has already been injected.

In the image below, we can see the “create_mutex_svchost” function. At the beginning of the function, it calls the function “generate_mutex_name” and generates the name of the mutex that will be created. The function has two parameters: key, which is the hardcoded value 7, and array, which will contain the generated name.

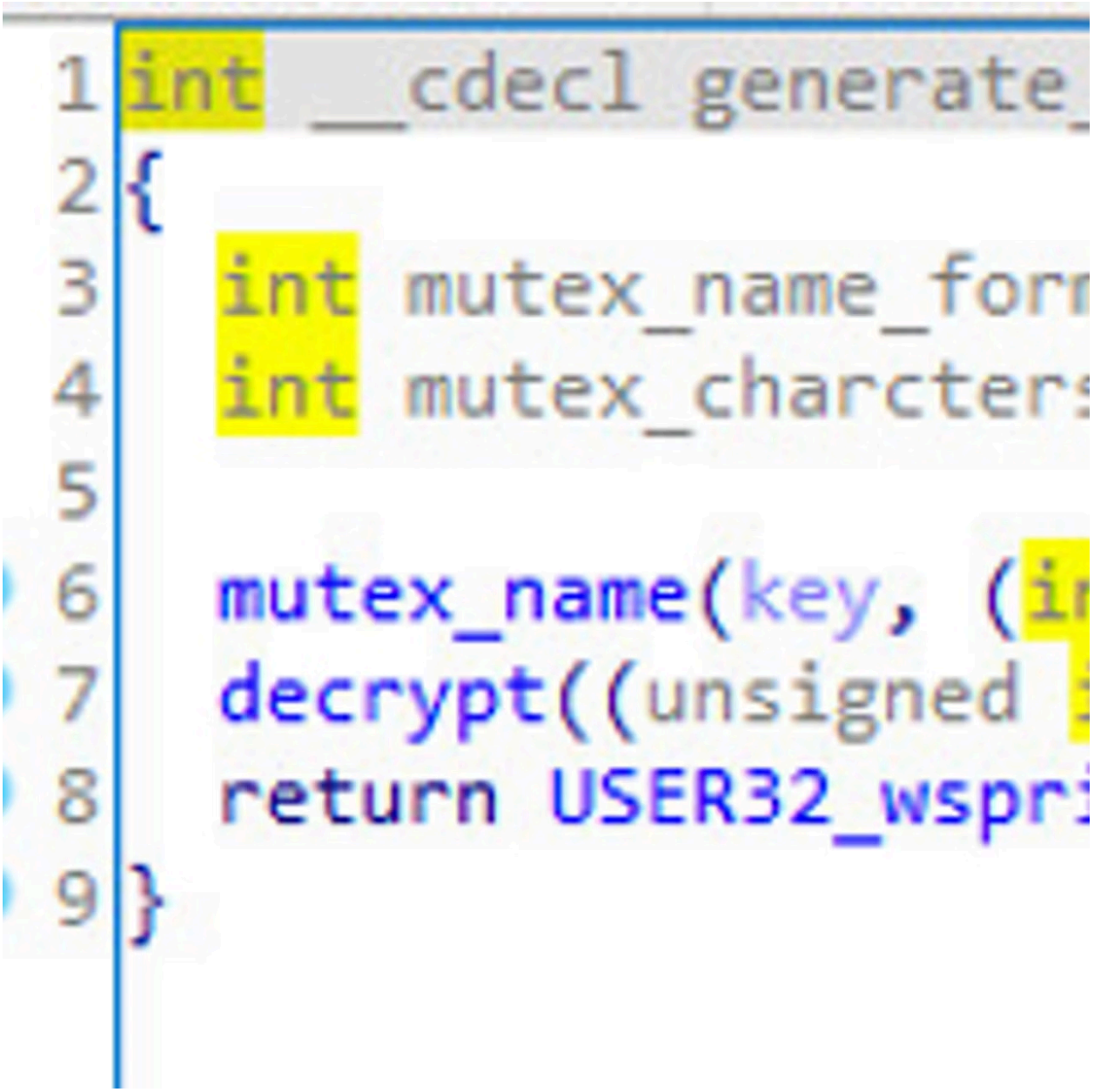



*Figure 31: IcedID mutex naming technique*

The function “generate_mutex_name” first calls the function “mutex_name” with key=7 and an empty array that will contain the characters used for the mutex name.

Next, it calls the function “decrypt” with the encrypted “mutex_name_format” string. Finally, it prints and returns the mutex name to the array it got as an argument.

The function “mutex_name” gets as arguments a key (key=7) and array that will contain the resulting mutex name. It runs a few bitwise operations on the argument key and the globalkey/init_key.

The result of this bitwise operation is the svchost mutex name.

```
3  int v2; // eax
4  int v3; // eax
5  int v4; // eax
6  unsigned int v5; // eax
7  int v6; // eax
8  int result; // eax
9
10 v2 = decode(key + init_key);
11 *(_DWORD *)arr = v2;
12 *(_BYTE *)(arr + 4) = key;
13 v3 = decode(v2);
14 *(_BYTE *)(arr + 5) = v3;
15 v4 = decode(v3);
16 *(_WORD *)(arr + 6) = v4 & 0xFFF | 0x4000;
17 v5 = decode(v4);
18 *(_WORD *)(arr + 8) = v5 % 0x3FFF + 0x8000;
19 v6 = decode(v5);
20 *(_WORD *)(arr + 10) = v6;
21 result = decode(v6);
22 *(_DWORD *)(arr + 12) = result;
23 return result;
24 }
```

*Figure 32: IcedID mutex naming technique*

**3. Configuration file paths**

First, the malware retrieves the “appdata\local\” path by calling the function SHGetFolderPathW with 0x1c as a parameter, as pictured below in Figure 33 (circled in black).

Next, it calls the function Decode_by_constant(key, arr) with key=4. The return value is the name of the folder in “appdata\local\”, as pictured below in Figure 33 (circled in gray).

There are two configuration files in the configuration folder, both with .dat extensions.

In order to generate the file names, IcedID performs the following steps:

1. Each of the files has an initial constant value (0 and 1) and the malware runs some bitwise operations on them and gets an integer as a result (5 and 261), as pictured below in Figure 33 (circled in red).
2. Next, it takes the generated value and calls the function Decode_by_constant(key, arr) with key=generated_value. The returned value is the first X letters in X+2 letters of the T configuration file, as pictured below in Figure 33 (circled in red).

3. The last two characters of the configuration file's name are generated by executing bitwise operations on the initial value related to the file, as pictured below in Figure 33 (circled in yellow).

```
int __cdecl Build_ConfigFilePath(unsigned __int8 a1,
{
  void (__stdcall *v2)(int, void *); // esi
  int arr; // [esp+Ch] [ebp-48h]
  int v5; // [esp+4Ch] [ebp-8h]
  __int16 v6; // [esp+50h] [ebp-4h]

  if ( SHELL32_SHGetFolderPathW(0, 28, 0, 0, a2) < 0
    decrypt((unsigned int *)&unk_1001D378, a2);
  decode_by_constant(4, (int)&arr);
  v2 = (void (__stdcall *)(int, void *))kernel32_lstr
  kernel32_lstrcatW(a2, &unk_1001A98C);
  v2(a2, &arr);
  kernel32_CreateDirectoryW(a2, &unk_100452D4);
  LOWORD(v5) = (a1 & 0xF) + 0x61;
  HIWORD(v5) = (a1 >> 4) + 0x61;
  v6 = 0;
  decode_by_constant((a1 << 8) | 5, (int)&arr);
  v2(a2, &unk_1001A98C);
  v2(a2, &arr);
  v2(a2, &v5);
  return ((int (__stdcall *)(int, __int16 *))v2)(a2,
}
```

*Figure 33: IcedID configuration files path definition*

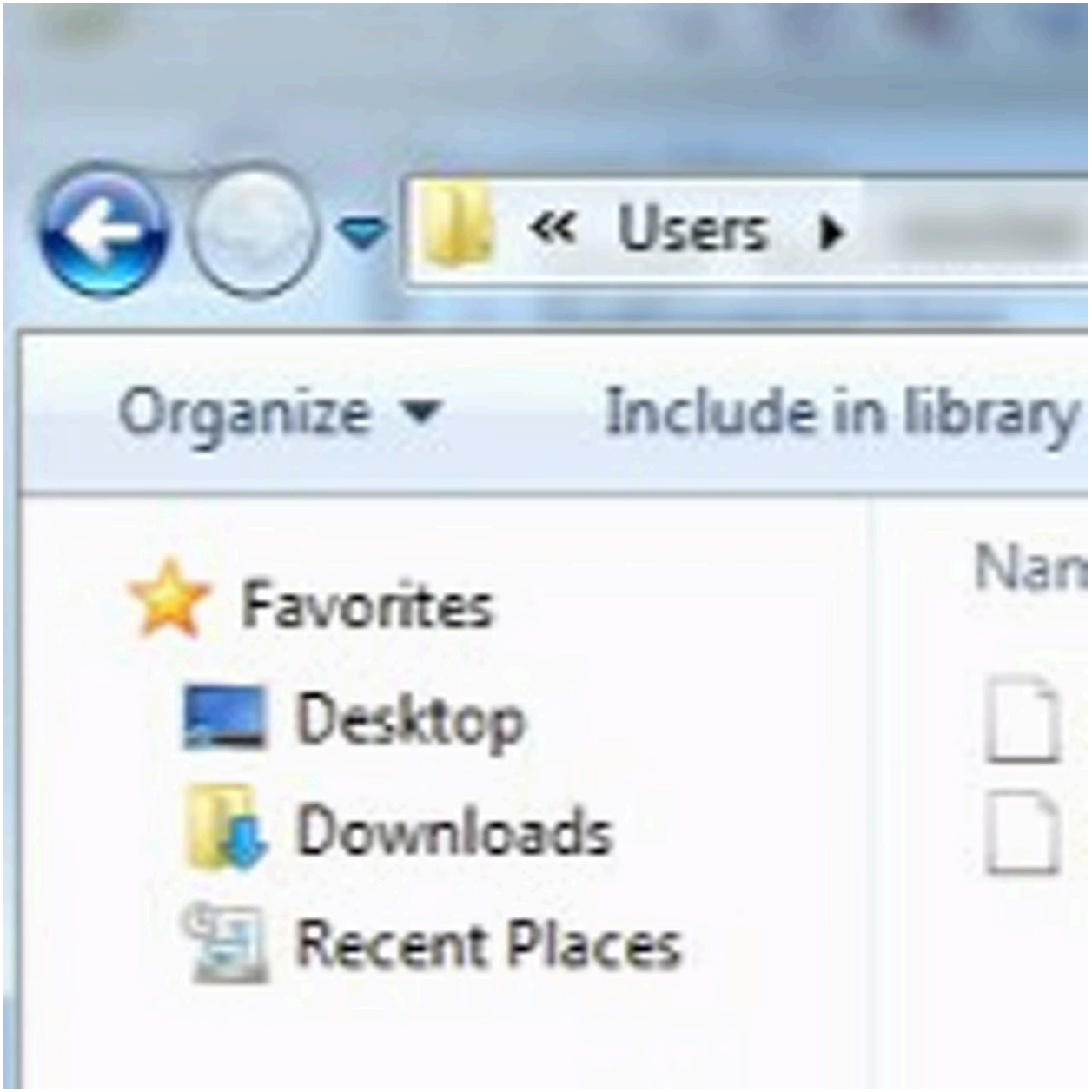



*Figure 34: IcedID configuration files*

## IcedID's certificate files

The IcedID version we examined stores certificate files related to the malware under "appdata\local\Temp" with the same name as the global key the malware generates. The suffix of the certificate file is .tmp.

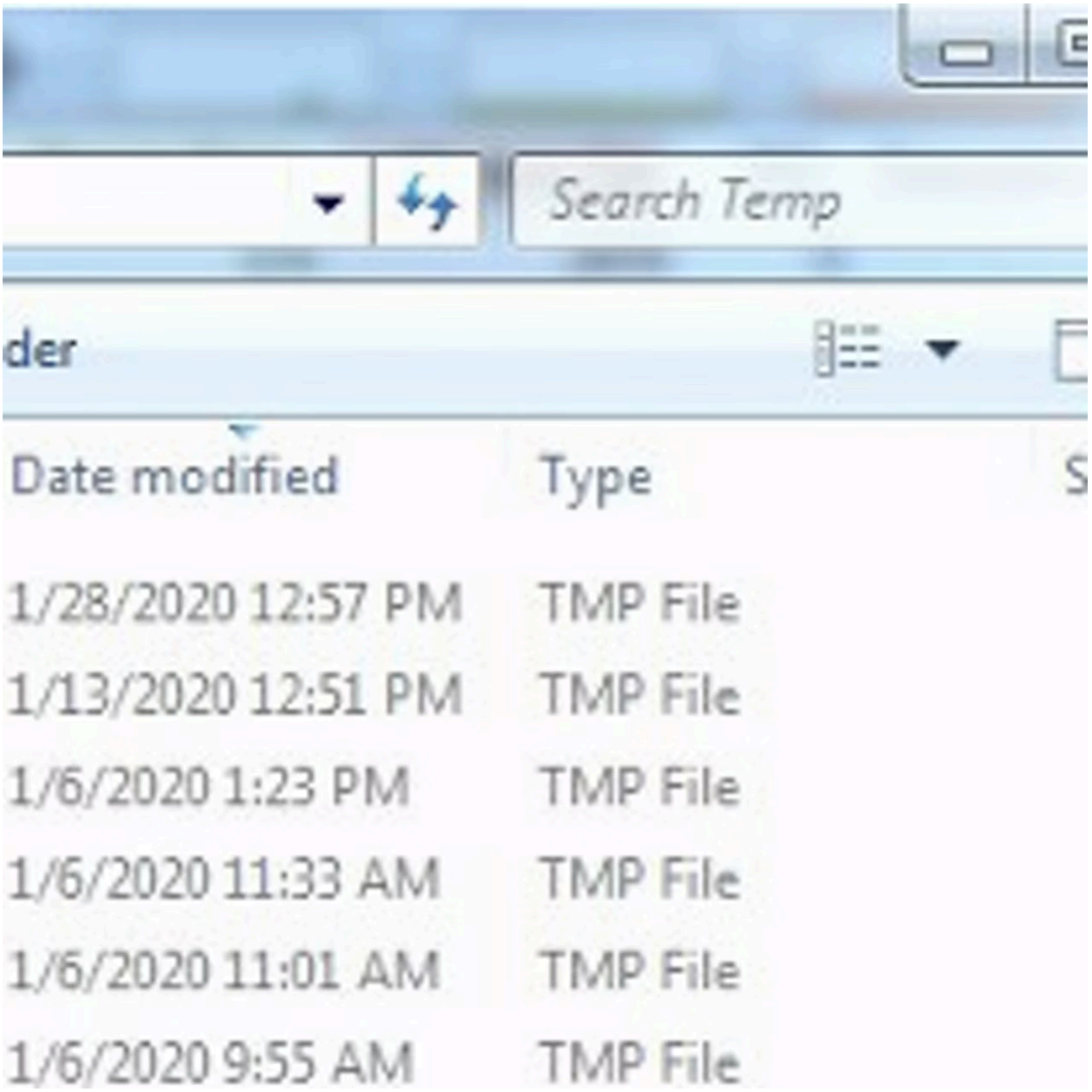



*Figure 35: IcedID certificate files stored as .tmp files*

## IcedID's configuration files

IcedID downloads configuration files from the C&C. The configuration files contain targeted banks and retailers, and the payload injected into the browser processes.

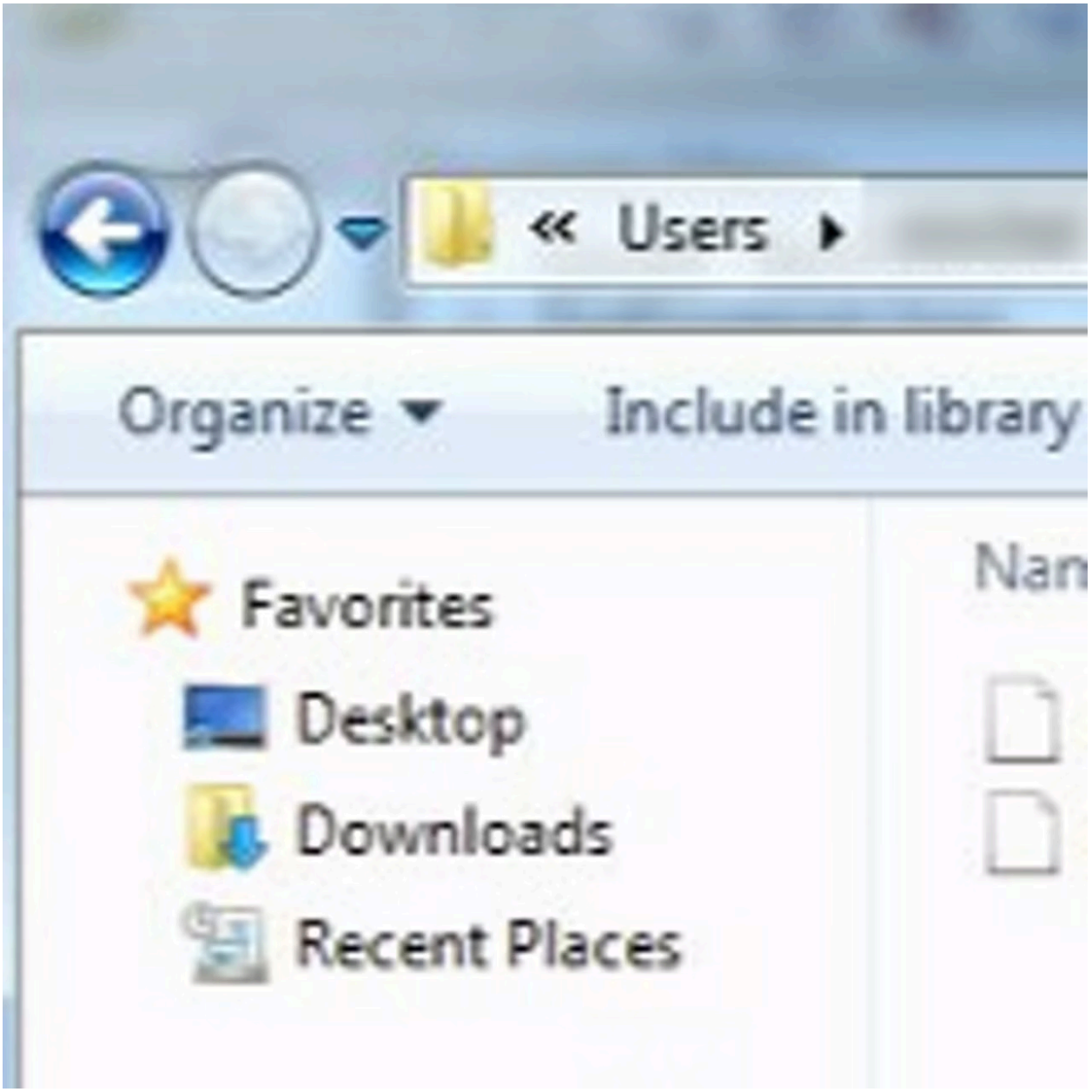



*Figure 36: IcedID configuration files*

## IcedID’s code versioning

IcedID’s developers continuously update its code over time. While bug fixes are naturally more frequent, we also see the occasional release of a new major version. This malware’s version number is hardcoded and resides in the encrypted photo.png file.

We can see the version number in the malicious svchost process by looking at the function that contains the main logic of the malware.

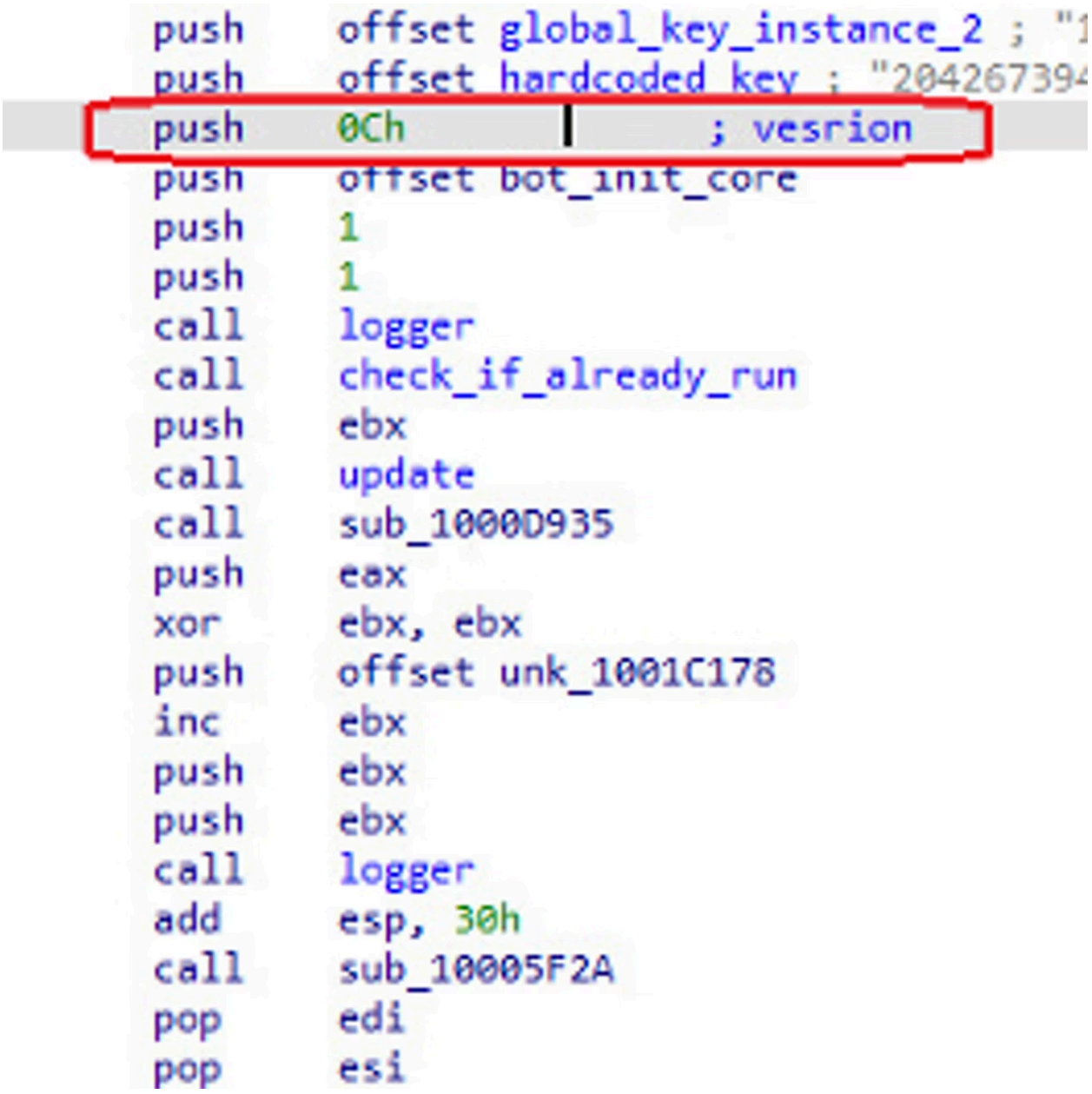



*Figure 37: IcedID version number is a part of its code*

Circled in red in Figure 37 above, we can see the sample version: 12 (0xC in hexadecimal).

## Browser hooking

When IcedID detects a browser process running in the system, it injects its malicious payload into the browser process and hooks relevant functions. Some of the targeted functions are Windows API functions, such as those from the kernel32, crypt32, WinINet and ws2_32 dynamic link libraries and some related to the browser vendor.


rome.exe-->ws2_32.dll-->WSAEventSelect
rome.exe-->ws2_32.dll-->connect
rome.exe-->crypt32.dll-->CertGetCertificateChai
rome.exe-->crypt32.dll-->CertVerifyCertificateCh
rome.exe-->kernel32.dll-->LoadLibraryA
rome.exe-->ws2_32.dll-->WSAEventSelect
rome.exe-->ws2_32.dll-->connect
rome.exe-->crypt32.dll-->CertGetCertificateChai
rome.exe-->crypt32.dll-->CertVerifyCertificateCh
rome.exe-->kernel32.dll-->LoadLibraryA
rome.exe-->ws2_32.dll-->WSAEventSelect
rome.exe-->ws2_32.dll-->connect
rome.exe-->crypt32.dll-->CertGetCertificateChai
rome.exe-->crypt32.dll-->CertVerifyCertificateCh

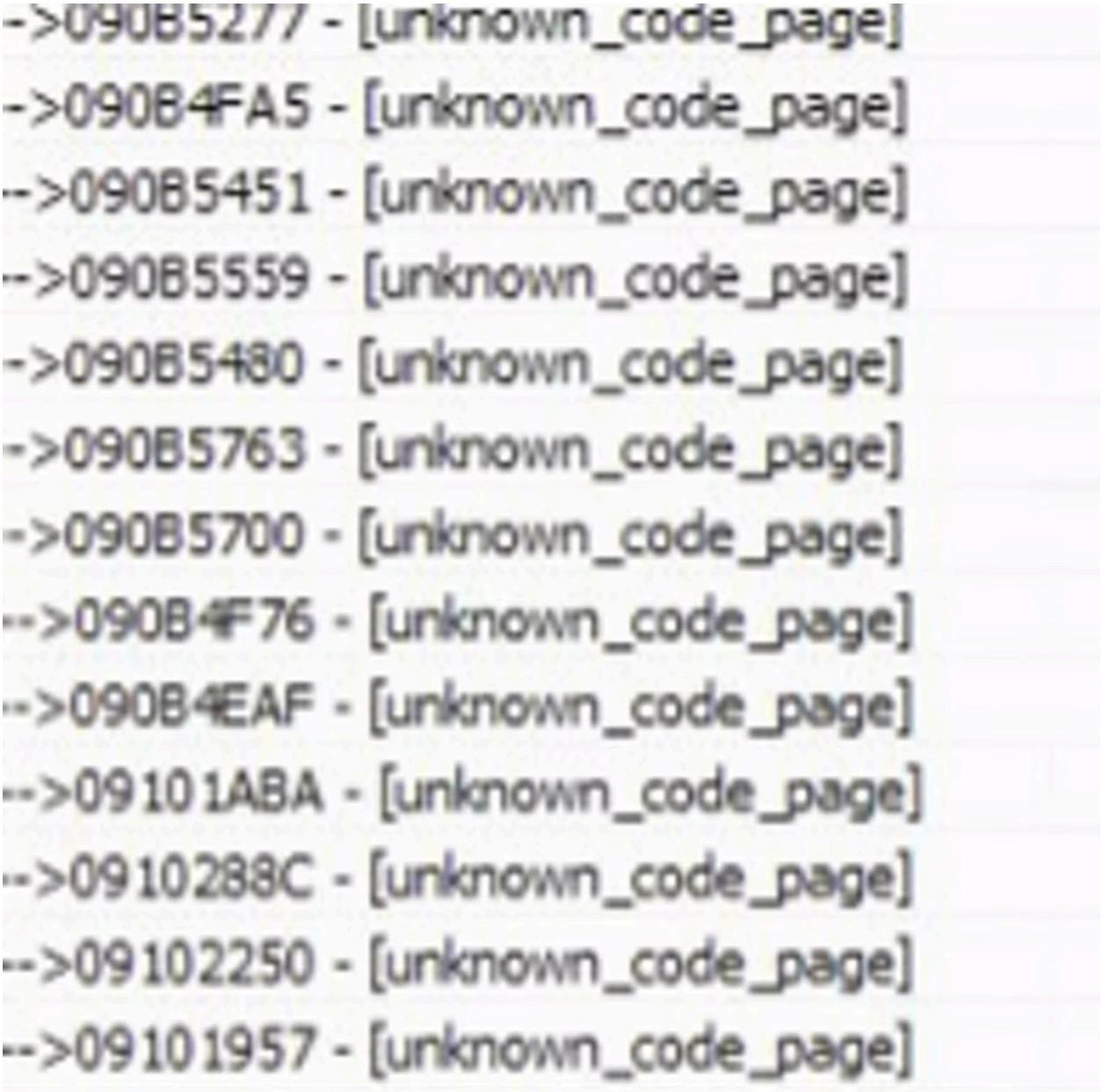



*Figure 38: Browser hooked functions*

## IcedID still in the cyber crime game

IcedID emerged in 2017 and continues to evolve its capabilities. While it has not been the most active malware over the past three years, its low activity volumes are mostly attributed to its continued focus on the same attack turf: North America. And while it has not topped the charts of 2019's most prevalent banking Trojans, it has consistently been targeting banks and retailers, and receives updates to the mechanisms that allow it to keep working on devices that get updated over time.

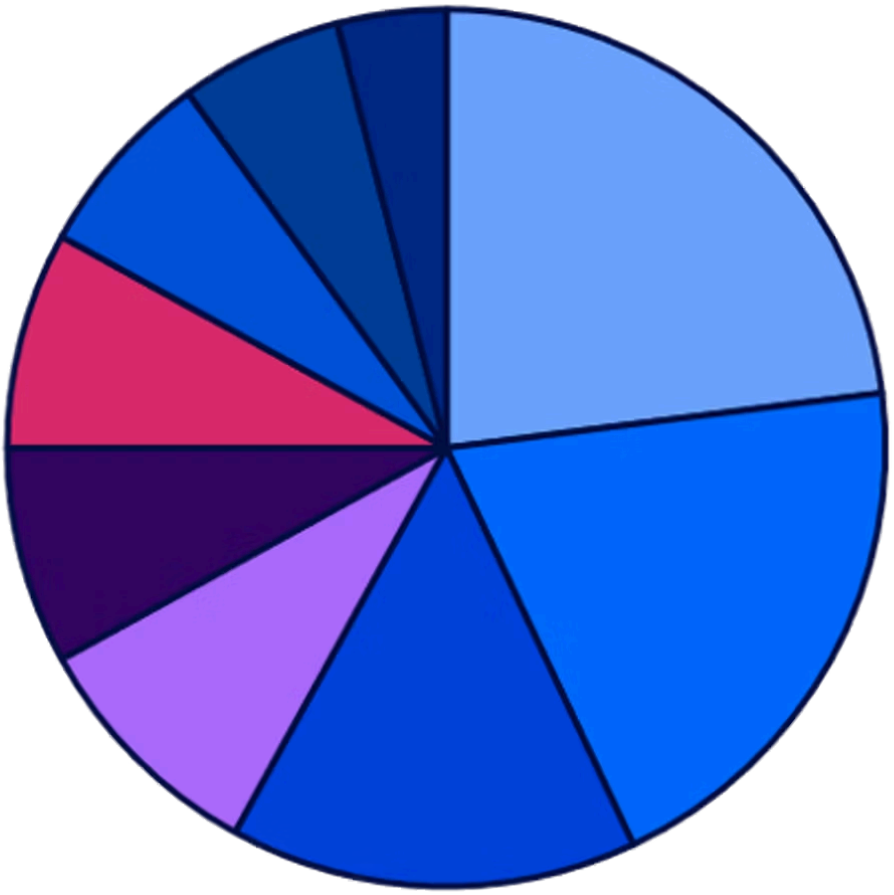


*Figure 39: Top banking Trojans in 2019 (Source: IBM X-Force)*

With its known connections to elite cybercrime gangs originating in Russia and other parts of Eastern Europe, the IcedID Trojan, and the group that operates it, are likely to continue to be part of the cybercrime arena. Interestingly, while we have been observing different gangs in this sphere diversify their revenue models to include high-stakes ransomware attacks, IcedID has not been part of the same trend so far. Will it follow in the footsteps of its counterparts? We will be following the trend as it evolves.

To stay up to date on IcedID and other malware and receive technical updates on the threat landscape, read IBM X-Force research blogs on Security Intelligence and visit X-Force Exchange.

## IOCs

MD5s:
Loader: 9C8EF9CCE8C2B7EDA0F8B1CCDBE84A14
Payload: 925689582023518E6AA8948C3F5E7FFE

Connections:
hxxp://www.hiperdom[.]top
hxxp://www.gertuko[.]top

File Paths:
C:\Users\[user]\AppData\Local\”generated name”
C:\Users\[user]\AppData\Local\”user name”
C:\Users\[user]\AppData\Local\Temp

---