

[QuickNote] Technical Analysis of recent Pikabot Core Module

 kienmanowar.wordpress.com/2024/01/06/quicknote-technical-analysis-of-recent-pikabot-core-module/

January 6, 2024

1. Overview

In early **February 2023**, cybersecurity experts on Twitter issued a warning about a new malware variant/family being distributed by the #TA577 botnet (associated with the same group from #Qakbot). This malware shares similarities with the **Qakbot** Trojan, including distribution methods, campaigns, and behaviors. It was quickly nicknamed **Pikabot**.

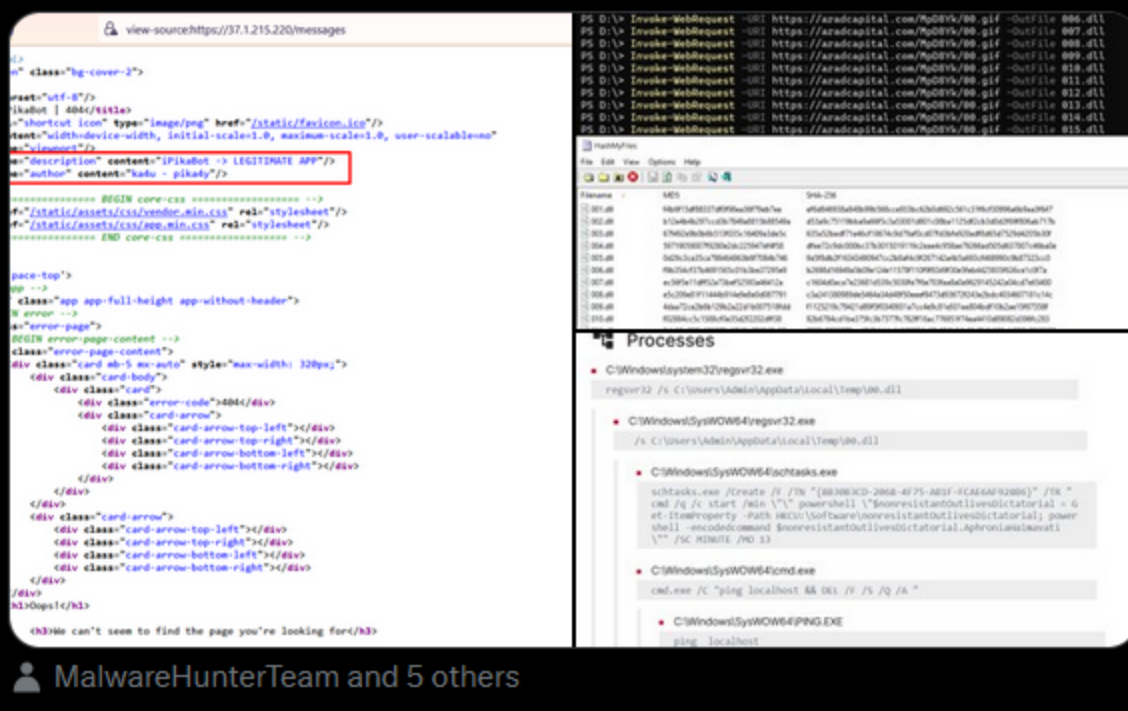
Germán Fernández @1ZRR4H

1/ Heads Up! 📢 @Unit42_Intel and @malware_traffic are reporting a new malware variant/family being distributed by #TA577 / #TR botnet (same guys from #Qakbot BB).

Unique strings in the C2 HTML:

- "iPikaBot -> LEGITIMATE APP"
- "ka4u - pika4y"

Sample: bazaar.abuse.ch/sample/67c61f6...



Pikabot consists of two components: **loader/injector** and **core module**. It utilizes loader/injector to decrypt and inject the core module. Core module then performs malicious behaviors, including gathering information about the victim machine, connecting to command and control server to receive and execute arbitrary commands, downloading and injecting other malware.

Pikabot is continuously upgraded, employing various anti-analysis techniques and different obfuscation methods to make it difficult for analysts to understand its behavior. In the next section of this article, I will focus on analyzing the Pikabot core module, including:

- How Pikabot obfuscates and decrypts strings.
- How Pikabot retrieves API addresses.

- How Pikabot slows down the analysis process.
- How Pikabot generates victim uuid.
- Collecting information from the victim's machine.
- How Pikabot decrypts C2 addresses.
- How Pikabot utilizes Syscall.

Sample hash:

ce742b7cc94a5c668116d343b6a9677523dc13b358294bba3cd248fba8b880da

2. Decrypt string

In some older versions, to decode strings, Pikabot utilizes a **XOR loop** to decode encrypted data stored on the stack:

```

.text:6AF0100A xor     ebx, ebx
.text:6AF0100C mov     dword ptr [ebp+var_50], 7D796E5Fh
.text:6AF01013 mov     [ebp+var_4C], 09517968h
.text:6AF0101A mov     ecx, ebx
.text:6AF0101C mov     [ebp+var_48], 48647968h
; encrypted data

.text:6AF01023
.text:6AF01023   _xor_loop: ; CODE XREF: pika_main_proc+344j
.text:6AF01023     mov     al, [ebp+ecx+var_50]
.text:6AF01027     xor     al, 1Ch
.text:6AF01029     mov     [ebp+ecx+sz_CreateMutexW], al ; Decoded: CreateMutexW
.text:6AF01030     inc     ecx
.text:6AF01031     cmp     ecx, 0Ch
.text:6AF01034     jnl    short _xor_loop
  
```

```

*( _DWORD *)enc_CreateMutexW = 0x7D796E5F;
*( _DWORD *)genc_CreateMutexW[4] = 0x09517968;
idx = 0;
*( _DWORD *)genc_CreateMutexW[8] = 0x48647968;
do
{
    sz_CreateMutexW[idx] = enc_CreateMutexW[idx] ^ 0x1C; // CreateMutexW
    ++idx;
}
while ( idx < 0xC );
  
```

In recent versions of Pikabot, the process of decrypting strings has become more sophisticated.

RC4 is used to decrypt encrypted data stored on stack. Each encrypted data has a corresponding RC4 key.

```

.text:00411131  push  edi
.text:00411132  mov   esi, offset str_currentCont_2 ; "currentContextId"
.text:00411137  mov   dword ptr [ebp+enc_str_1], 1dF4E800h
.text:0041113E  lea   edi, [ebp+str_rc4_key]
.text:00411144  mov   dword ptr [ebp+enc_str_1+4], 5AEE59A6h
.text:0041114B  xor   ebx, ebx
.text:0041114D  mov   dword ptr [ebp+enc_str_1+8], 0F4769308h
.text:00411154  mov   dword ptr [ebp+enc_str_1+8Ch], 261C5DA2h
.text:0041115B  mov   eax, ebx
.text:0041115D  movsd dword ptr [ebp+enc_str_1+10h], 0AA36D275h
.text:0041115E  mov   dword ptr [ebp+enc_str_1+10h], 0A37D38C6h
.text:0041116C  movsd dword ptr [ebp+enc_str_1+10h], 0A37D38C6h
.text:0041116E  movsd dword ptr [ebp+enc_str_1+10h], 0A37D38C6h
.text:0041116F  movsb
.text:00411170  mov   edi, ebx

```

rc4_key

encrypted data

```

.text:00411172  .text:00411172  _rc4_init:
.text:00411172  mov   [ebp+eax+var_288], al
.text:00411179  inc   eax
.text:0041117A  cmp   eax, 100h
.text:0041117F  jnb  short _rc4_init

.text:00411181  mov   esi, ebx

.text:00411183  .text:00411183  _rc4_key_scheduling:
.text:00411183  mov   dl, [ebp+esi+var_288]
.text:0041118A  mov   eax, esi
.text:0041118C  and   eax, 0Fh
.text:0041118F  movzx ecx, dl
.text:00411192  movzx eax, [ebp+eax+str_rc4_key]
.text:0041119A  add   eax, edi
.text:0041119C  add   ecx, eax
.text:0041119E  movzx edi, cl
.text:004111A1  mov   al, [ebp+edi+var_288]
.text:004111A8  mov   [ebp+esi+var_288], al
.text:004111AF  inc   esi
.text:004111B0  mov   [ebp+edi+var_288], dl
.text:004111B7  cmp   esi, 100h
.text:004111BD  jnb  short _rc4_key_scheduling

```

```

*( _DWORD *)enc_str_1 = 0x1BFEBE60;
*( _DWORD *)&enc_str_1[4] = 0x5AEE59A6;
LOBYTE(v0) = 0;
*( _DWORD *)&enc_str_1[8] = 0xF4769308;
*( _DWORD *)&enc_str_1[0xC] = 0x261C5DA2;
idx = 0;
strcpy(str_rc4_key, "currentContextId");
*( _DWORD *)&enc_str_1[0x10] = 0xAA36D275;
*( _DWORD *)&enc_str_1[0x14] = 0xA37D38C6;
LOBYTE(v2) = 0;
// rc4 init
do
{
  *((_BYTE *)&rc4_ksa[8] + idx) = idx;
  ++idx;
}
while ( idx < 0x100 );
// rc4 key scheduling
for ( i = 0; i < 0x100; ++i )
{
  v4 = *((_BYTE *)&rc4_ksa[8] + i);
  v2 = (unsigned __int8)(v2 + str_rc4_key[i & 0xF] + v4);
  *((_BYTE *)&rc4_ksa[8] + i) = *((_BYTE *)&rc4_ksa[8] + v2);
  *((_BYTE *)&rc4_ksa[8] + v2) = v4;
}

```

```

.text:00411102  .text:00411102  _rc4_crypt:
.text:00411102  lea   eax, [esi+1]
.text:0041110D  mov   edx, offset str_onecoreuap_b ; "onecoreuap\base\appmodel\search\com...
.text:00411110  movzx esi, al
.text:00411110  push  esi ; 'e'
.text:00411110  mov   [ebp+str_victim_id], esi
.text:00411112  mov   al, [ebp+esi+var_288]
.text:00411119  mov   [ebp+var_1], al
.text:0041111C  mov   [ebp+var_4c], esi
.text:0041111F  add   eax, ebx
.text:0041111F  movzx ebx, al
.text:0041111F  pop   esi

.text:0041111F  .text:0041111F  loc_41111F:
.text:0041111F  add   eax, 0FFFFFF0h
.text:0041111F  cmp   ax, word ptr [ebp+var_81E1_1]
.text:0041111F  ja   short loc_411217

.text:00411201  .text:00411201  mov   esi, [ebp+delim_char_]
.text:00411206  add   ecx, 2
.text:00411207  movzx eax, word ptr [edx]
.text:00411208  mov   [ebp+var_E8], eax
.text:00411210  test  ax, ax
.text:00411211  jnz  short loc_41111F

.text:00411217  .text:00411217  jmp  short loc_411219

.text:00411217  .text:00411217  loc_411217:
.text:00411217  mov   ecx, esi
.text:00411219  or   ecx, 09CE1A70h
.text:00411223  mov   a2, ecx

.text:00411229  .text:00411229  loc_411229:
.text:00411229  lea   eax, [ecx+7E142B23h]
.text:0041122F  mov   al, eax
.text:00411230  mov   eax, offset str_currentCont_3 ; "currentContextMessage"
.text:00411238  push  ecx
.text:0041123A  mov   edx, eax
.text:0041123C  mov   ecx, eax
.text:0041123E  call  sub_0041123E
.text:00411243  mov   al, [ebp+eax+var_288]
.text:00411244  mov   [ebp+esi+var_288], al
.text:00411251  mov   al, [ebp+var_1]
.text:00411254  mov   [ebp+esi+var_288], al
.text:00411258  movzx eax, [ebp+esi+var_288]
.text:00411266  movzx eax, al
.text:00411269  pop   ecx
.text:0041126A  mov   ecx, a2
.text:00411270  or   ecx, 00000A13h
.text:00411276  mov   al, [ebp+eax+var_288]
.text:00411279  mov   al, [ebp+esi+enc_str_1]
.text:00411281  mov   [ebp+edi+str_enc_createMutex], al
.text:00411288  inc   edi
.text:00411289  mov   a2, ecx
.text:0041128F  cmp   edi, 10h
.text:00411292  jnb  short _rc4_crypt

```

```

// rc4 crypt
i = 0;
do
{
  v0 = "onecoreuap\base\appmodel\search\common\pkmutild\registry.cxx";
  v6 = (char *) (unsigned __int8)((_BYTE)v6 + 1);
  delim_char_ = v6;
  v769 = *((_BYTE *)&rc4_ksa[8] + (_DWORD)v6);
  str_victim_id = (char *)v769;
  v0 = (unsigned __int8)(v0 + v769);
  LOWORD(v9) = 0x6F;
  while ( (unsigned __int16)(v9 - 0x30) <= (unsigned __int16)v_0x1E_1 )
  {
    v6 = delim_char_;
    v9 = ++v8;
    v737 = v9;
    if ( !_DWORD)v9 )
      goto LABEL_x1229;
  }
  v5 = a4 | 0x59FEEA7D;
  ::a2 = a4 | 0x59FEEA7D;
LABEL_x1229:
  a4 = v5 + 0x7E142B23;
  pkb_weird_func_1(v5);
  *((_BYTE *)&rc4_ksa[8] + (_DWORD)v6) = *((_BYTE *)&rc4_ksa[8] + v0);
  *((_BYTE *)&rc4_ksa[8] + v0) = v769;
  v5 = ::a2 | 0x0006A134;
  // a55fxsIEB9MGS1edys0VUQ
  str_enc_createMutex[i] = enc_str_1[i] ^ *((_BYTE *)&rc4_ksa[8]
    + (unsigned __int8)((_BYTE)str_victim_id
    + *((_BYTE *)&rc4_ksa[8] + (_DWORD)v6)));
  ++i;
  ::a2 = v5;
}
while ( i < 0x18 );
v751[0] = (void *)0x91A3D8DE;
LOWORD(v751[1]) = 0x23BF;
v10 = a4 ^ 0x4E5162BE;
str_enc_createMutex[0x18] = 0;

```

- The RC4-decrypted string will be converted to a valid Base64 string (by replacing the character '_' with '=') and then decoded using **Base64**.
- Finally, **AES-CBC** will be used to decrypt the decoded data to return the original string.

```

.text:0041DCCE
.text:0041DCEE loc_41DCEE:
.text:0041DCEE cmp     eax, ecx
.text:0041DCF0 jnb     short loc_41DCF7

.text:0041DCF2 cmp     byte ptr [eax+ebx], 's'
.text:0041DCF6 jnz     short loc_41DCF8

.text:0041DCF8 mov     byte ptr [eax+ebx], 's'

.text:0041DCFC loc_41DCFC:
.text:0041DCFC inc     eax
.text:0041DCFD jmp     short loc_41DCEE

.text:0041DCFF loc_41DCFF:
.text:0041DCFF mov     ecx, ebx ; str_input
.text:0041DD01 call    pkb_strlen
.text:0041DD06 mov     edx, eax ; str_input_len
.text:0041DD08 call    pkb_base64_decode
.text:0041DD0D lea    ecx, [ebp+str_aes_iv]
.text:0041DD10 mov     edx, eax ; str_input_len
.text:0041DD12 push   ecx ; str_aes_iv
.text:0041DD13 lea    ecx, [ebp+str_aes_key]
.text:0041DD19 push   ecx ; str_aes_key
.text:0041DD1A mov     ecx, ebx ; str_input
.text:0041DD1C call    pkb_aes_decrypt
.text:0041DD21 pop     ecx
.text:0041DD22 pop     ecx
.text:0041DD23 pop     edi
.text:0041DD24 pop     esi
.text:0041DD25 mov     byte ptr [eax+ebx], 0
.text:0041DD29 pop     ebx
.text:0041DD2A leave
.text:0041DD2B retn
.text:0041DD2B pkb_base64_decode_n_aes_decrypt_str endp
.text:0041DD2B

```

```

// convert str_input back to base64 format
while ( idx < str_input_len )
{
    if ( str_input[idx] == '\0' )
        str_input[idx] = '\0';
    ++idx;
}
str_base64_len = pkb_strlen(str_input);

// base64 decode
decoded_b64_len = pkb_base64_decode(str_base64_len, str_input);

// AES-CBC decrypt to get plain text
result = pkb_aes_decrypt(decoded_b64_len, str_input, str_aes_key, str_aes_iv);
str_input[result] = 0;
return result;

```

AES Key and AES IV used in this sample are also decrypted using RC4:

```

idx = 0;
do
{
    v12 *= 0x47F0A906;
    v11 = (unsigned __int8)(v11 + 1);
    v14 = str_aes_key[v11 + 0x3C];
    v49 = (unsigned __int8)(v14 + v49);
    str_aes_key[v11 + 0x3C] = str_aes_key[v49 + 0x3C];
    str_aes_key[v49 + 0x3C] = v14;
    // "dVOEz=/e/Xf=0WMiz6uR9cZKe+tyb+VJhSu+tfi0HzT2C0oz25r4+8osEx4"
    str_aes_key[idx] = enc_str_aes_key[idx] ^ str_aes_key[(unsigned __int8)(v14 + str_aes_key[v11 + 0x3C]) + 0x3C];
    ++idx;
}
while ( idx < 0x3B );

```

decrypt aes key

```

v15 = (unsigned __int8)(v15 + 1);
v27 = (v32 - 0x348B7859) & 0xDB0B3ED8;
dword_44D610 = v27;
v33 = *((_BYTE *)&v43 + v15);
v53 = (unsigned __int8)(v33 + v53);
*((_BYTE *)&v43 + v15) = *((_BYTE *)&v43 + v53);
*((_BYTE *)&v43 + v53) = v33;
// nsdA1ANUAH+K1XhVjnsq92tGMNQG=fsgqrqJQ8AtZIacqaYg
str_aes_iv[idx] = v48[idx] ^ *((_BYTE *)&v43 + (unsigned __int8)(v33 + *((_BYTE *)&v43 + v15)));
++idx;
}

```

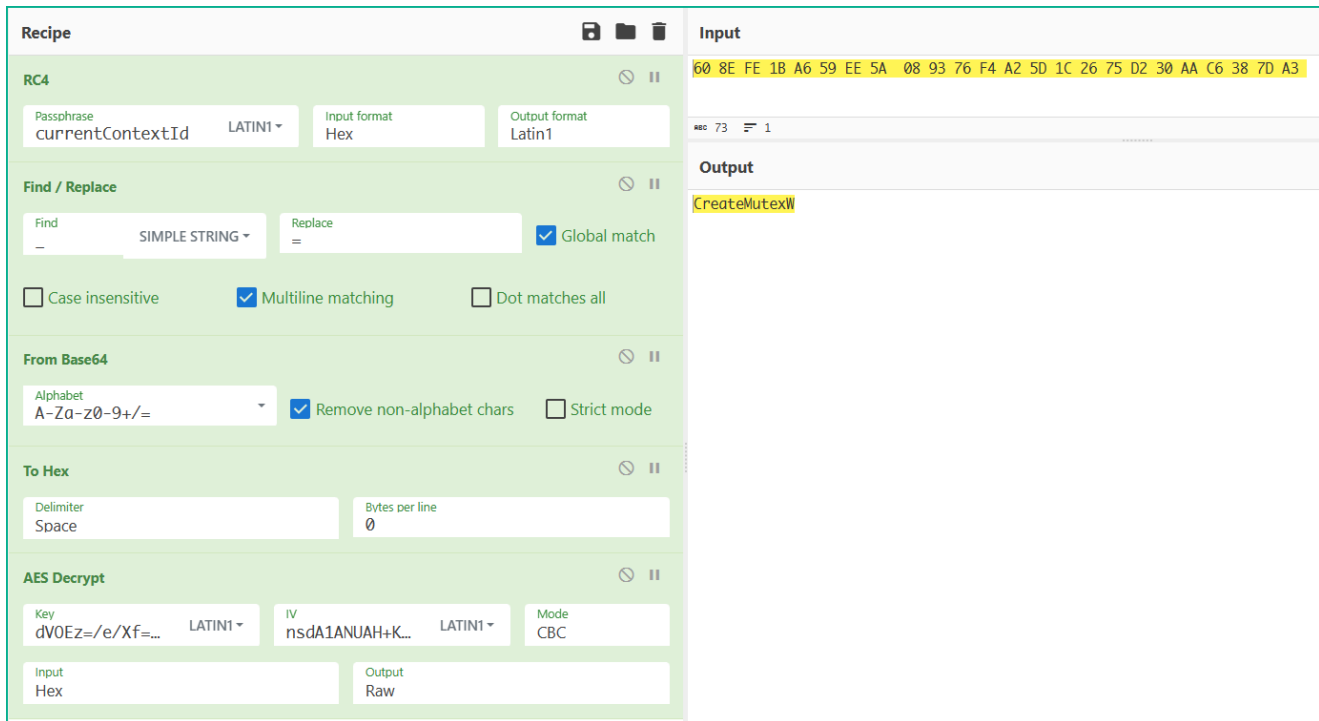
decrypt aes iv

- Decrypted AES Key: "dVOEz=/e/Xf=0WMiz6uR9cZKe+tyb+VJhSu+tfi0HzT2C0oz25r4+8osEx4"
- Decrypted AES IV: "nsdA1ANUAH+K1XhVjnsq92tGMNQG=fsgqrqJQ8AtZIacqaYg"

However, Pikabot only uses **32 bytes** from the decrypted AES Key and **16 bytes** from the decrypted AES IV. Therefore, the final AES Key and IV used for string decryption are:

- AES Key: “dVOEz=/e/Xf=0WMiz6uR9cZKe+tyb+VJ”
- AES IV: “nsdA1ANUAH+K1XhV”

The entire process was simulated using **CyberChef** as follows:



Here is the CyberChef recipe:

```
https://gchq.github.io/CyberChef/#recipe=RC4(%7B'option':'Latin1','string':'currentContextId')To_Hex('Space',0)AES_Decrypt(%7B'option':'Latin1','string':'dVOEz=/e/Xf=0WMiz6uR9cZKe+tyb+VJ',true,false)To_Hex('Space',0)AES_Decrypt(%7B'option':'Latin1','string':'nsdA1ANUAH+K1XhV',true,false)To_Hex('Space',0)
```

3. Retrieve API address

To get the address of API functions, Pikabot does the following:

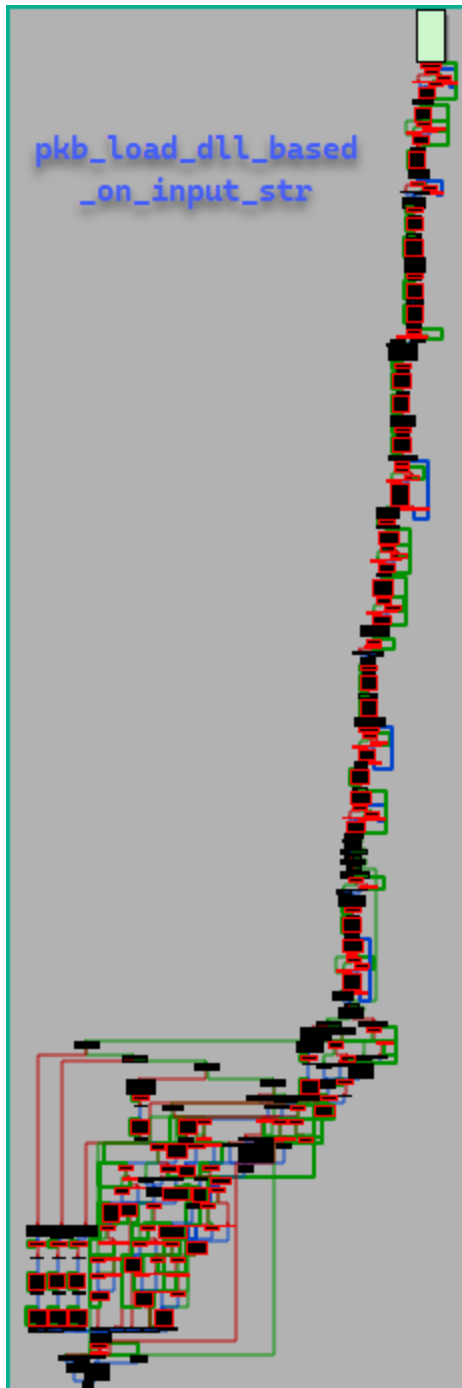
- It gets the base address of the corresponding DLL based on the decrypted input string.
- Decrypts the API function name, then uses **GetProcAddress** to obtain the real address of the API.

```
.text:00413D03 lea   eax, [ebp+wstr_mutex_name]
.text:00413D09 push  eax           ; lpName
.text:00413D0A push  [ebp+bInitialOwner] ; bInitialOwner
.text:00413D10 lea   ecx, [ebp+str_CzYNRd] ; str_input
.text:00413D16 push  [ebp+buffer] ; lpMutexAttributes
.text:00413D1C call  pkb_load_dll_based_on_input_str
.text:00413D21 mov   edx, eax       ; dll_handle
.text:00413D23 lea   ecx, [ebp+str_enc_CreateMutexW] ; str_enc_api_name
.text:00413D29 call  pkb_get_api_addr_by_name_using_GetProcAddress ; CreateMutexW
.text:00413D2E call  eax
```



```
kernel32_hdl = pkb_load_dll_based_on_input_str(str_CzYNRd); // CzYNRd
CreateMutexW = (HANDLE (__stdcall *) (LPSECURITY_ATTRIBUTES, BOOL, LPCWSTR)) pkb_get_api_addr_by_name_using_GetProcAddress(
    str_enc_CreateMutexW,
    kernel32_hdl);
```

The function `pkb_load_dll_based_on_input_str (0x41E657)` has the following code graph:



In this function, Pikabot decrypts relevant strings and compares them to the string passed to the function. If the strings match, Pikabot decrypts the name of the corresponding DLL and loads it using `LoadLibraryA`. Firstly, Pikabot finds the addresses of the `GetProcAddress` and `LoadLibraryA` functions using pre-calculated hash values.


```

.text:0041EFCE    xor     ebx, ebx
.text:0041EFD0    mov     dword_44D6C8, esi
.text:0041EFD6    mov     [ebp+dec_str_8+4], bl
.text:0041EFD9    cmp     g_kernel32_base_addr, ebx
.text:0041EFDf    jnz    short loc_41EFF1

.text:0041EFE1    call   pkb_get_kernel32_base_addr
.text:0041EFE6    mov     esi, dword_44D6C8
.text:0041EFEC    mov     g_kernel32_base_addr, eax

.text:0041EFF1    loc_41EFF1:
.text:0041EFF1    GetProcAddress_0, ebx
.text:0041EFF7    cmp     GetProcAddress_0, ebx
.text:0041EFF7    jnz    short loc_41F00E

.text:0041EFF9    mov     edx, 73F5352Eh ; pre_api_hash
.text:0041EFfE    call   pkb_resolve_kernel32_api_by_hash ; GetProcAddress
.text:0041F003    mov     esi, dword_44D6C8
.text:0041F009    mov     GetProcAddress_0, eax

.text:0041F00E    loc_41F00E:
.text:0041F00E    cmp     LoadLibraryA_0, ebx
.text:0041F014    jnz    short loc_41F02B

.text:0041F016    mov     edx, 0ED86D8FAh ; pre_api_hash
.text:0041F01B    call   pkb_resolve_kernel32_api_by_hash ; LoadLibraryA
.text:0041F020    mov     esi, dword_44D6C8
.text:0041F026    mov     LoadLibraryA_0, eax

```

```

if ( !g_kernel32_base_addr )
{
    kernel32_base_addr = pkb_get_kernel32_base_addr(); // return base address of kernel32.dll
    v87 = dword_44D6C8;
    g_kernel32_base_addr = kernel32_base_addr;
}
if ( !GetProcAddress_0 )
{
    GetProcAddress = (FARPROC (__stdcall *))(HMODULE, LPCSTR)pkb_resolve_kernel32_api_by_hash(0x73F5352E);
    v87 = dword_44D6C8;
    GetProcAddress_0 = GetProcAddress;
}
if ( !LoadLibraryA_0 )
{
    LoadLibraryA = pkb_resolve_kernel32_api_by_hash(0xED86D8EA);
    v87 = dword_44D6C8;
    LoadLibraryA_0 = (HMODULE (__stdcall *))(LPCSTR)LoadLibraryA;
}

```

The pseudo-code for calculating the hash of API functions is as follows:

```

// calc Api name hash
// init hash = 0x2088
do
{
    v_0x5 = 0x25;
    for ( j = 0; j < 2; ++j )
        v_0x5 += tmp_arr[j + 2];
    c = str_input[i];
    lower_c = c + 0x20; // convert to lower case
    if ( (unsigned __int8)(c - 0x41) > 0x19u )
        lower_c = str_input[i];
    hash = lower_c + hash * v_0x5;
    ++i;
}
while ( i < str_input_len );
return hash;

```

Based on the pseudo-code above, we can rewrite it in Python and perform a brute-force to find the API function name corresponding to the pre-calculated hash values:

```
def pika_hash(s):  
    hash = 0x2088 # change this value  
    for c in s.lower():  
        hash = c + 5 * hash  
    return hash & 0xFFFFFFFF
```



```
Trying: C:\Windows\SysWOW64\kernel32.dll  
API hash: 0x73F5352E --> API found: b'GetProcAddress'  
API hash: 0xC280BF30 --> API found: b'HeapFree'  
API hash: 0xED86D0EA --> API found: b'LoadLibraryA'
```

With the API function addresses obtained above, Pikabot will load the corresponding DLL:

```

if ( !pkb_strcmp(str_input, dec_str_1) ) // CzYNRd → decrypt Kernel32.dll
{
    v113 = 0xB77;
    v114 = "HostProcessExecution";
}

```

```

dword_44D6C8 = v129 & 0x70C72B1B;
pkb_weird_func_4();
v123 = str_input;
dec_str_1[v124 + 8] = dec_str_1[(DWORD)str_input + 8];
dec_str_1[(DWORD)v123 + 8] = v256;
// LZsou6QvSXvmXuc6TX7x2A__ → Kernel32.dll
enc_str_kernel32_dll[idx] = dec_str_15[idx] ^ dec_str_1[(unsigned __int8)(v253 + dec_str_1[v124 + 8]) + 8];
++idx;
}
while ( idx < 0x18 );
enc_str_dll_name = enc_str_kernel32_dll;
enc_str_kernel32_dll[0x18] = 0;
goto decrypt_n_load_dll;
}

```

```

decrypt_n_load_dll:
// Decrypt the name of the corresponding dll
dec_str_dll_name = pkb_decrypt_str(enc_str_dll_name);
v212 = 0xB3A;
v213 = (unsigned __int8)str_lbLU[0];
if ( str_lbLU[0] )
{
    v214 = str_lbLU;
    while ( 1 )
    {
        v212 += v213;
        ++v214;
        if ( v212 > 0x14EE )
            break;
        v213 = (unsigned __int8)*v214;
        if ( !*v214 )
            goto LABEL_xFE48;
    }
    pkb_weird_func_5();
    v215 = 0x2AA03443 * (dword_44D6C8 & 0x5525FBAF);
}
else
{
LABEL_xFE48:
    v215 = dword_44D6C8 | 0x2D2B9DA6;
}
dword_44D6C8 = v215 ^ 0x11F78E1F;

// Loads the specified dll
dll_handle = LoadLibraryA_0(dec_str_dll_name);
pkb_free_heap_region(dec_str_dll_name);
return dll_handle;
}

```

Here is the list of DLLs that Pikabot will load during execution:

Input string	Dll to load
CzYNRd	Kernel32.dll
osPFU	User32.dll
QJJniV	Shell32.dll
MIT3nE	Ole32.dll
fWHur	Wininet.dll
YgeYS	Advapi32.dll
ss6HQA	NetApi32.dll
olOo	ntdll.dll

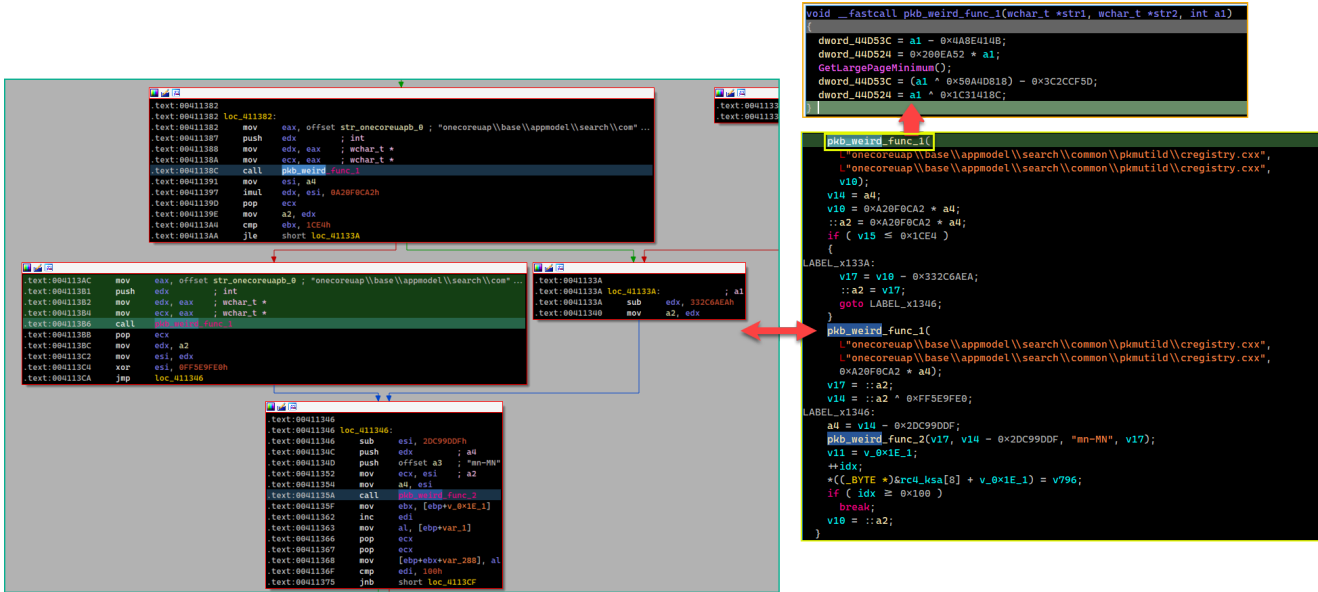
The function `pkb_get_api_addr_by_name_using_GetProcAddress (0x41E636)` will decrypt the API function name and call `GetProcAddress` to retrieve the function address:

```
.text:0041E636 ; FARPROC __fastcall pkb_get_api_addr_by_name_using_GetProcAddress(char *str_enc_api_name, HMODULE dll_handle)
.text:0041E636 pkb_get_api_addr_by_name_using_GetProcAddress proc near
.text:0041E636     push     esi
.text:0041E637     push     edi
.text:0041E638     mov      esi, edx
.text:0041E63A     call    pkb_decrypt_str
.text:0041E63F     mov      edi, eax
.text:0041E641     push    edi          ; lpProcName
.text:0041E642     push    esi          ; hModule
.text:0041E643     call   GetProcAddress_0 ←
.text:0041E649     mov     ecx, edi     ; buffer
.text:0041E64B     mov     esi, eax
.text:0041E64D     call   pkb_free_heap_region
.text:0041E652     pop     edi
.text:0041E653     mov     eax, esi
.text:0041E655     pop     esi
.text:0041E656     retn
.text:0041E656 pkb_get_api_addr_by_name_using_GetProcAddress endp
```

```
str_dec_api_name = pkb_decrypt_str(str_enc_api_name);
api_addr = GetProcAddress_0(dll_handle, str_dec_api_name);
pkb_free_heap_region(str_dec_api_name);
return api_addr;
```

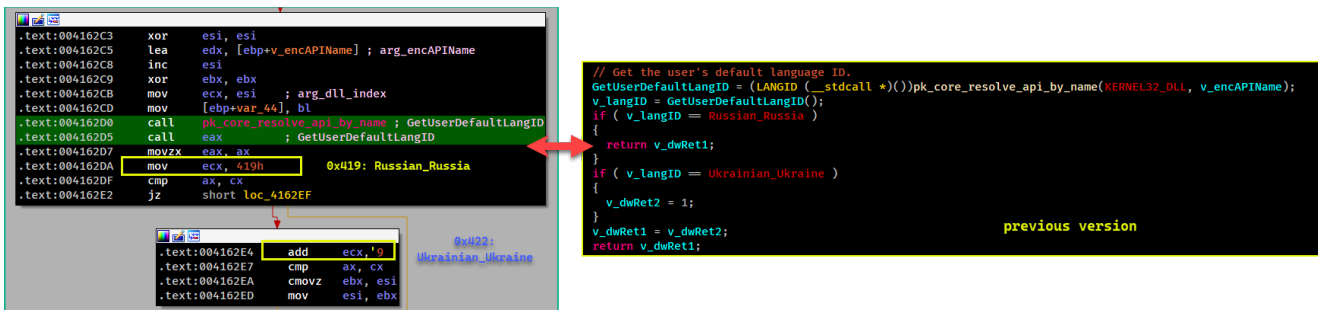
4. Slowing down the analysis process

In order to slow down the code analysis, Pikabot inserts a large number of meaningless junk functions into the execution flow. These functions typically do nothing. This can make it much more time-consuming for analysts to understand the code and identify its malicious behavior.

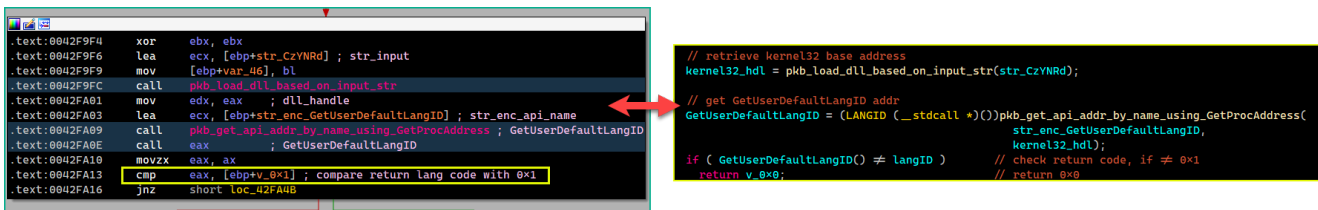


5. System language check

Pikabot checks the system language code of the victim's machine before executing its main task by using API function `GetUserDefaultLangID`. In the previous version, if the result returned a region code for a country such as `Russia` or `Ukraine`, the malware would immediately exit without any further activity.



However, in the version I am analyzing, Pikabot simply checks the return code if it is different from `0x1`, the function `pkb_check_default_lang (0x0042F7A0)` will return `0x0`:



6. Create Mutex

When the result of the function `pkb_check_default_lang (0x42F7A0)` return `0x0`, Pikabot will continue executing, with the sample I am analyzing it uses the hardcoded mutex name (after decrypting): `{F0B9756B-5D50-4696-A969-4C9AF7B69188}` to prevent reinfection on

the victim's machine.

```
.text:00413CF3 mov [ebp+al+10h], ecx
.text:00413CF6 call pkb_check_default_lang
.text:00413CFB test eax, eax
.text:00413CFD jnz loc_415423

.text:00413D03 lea eax, [ebp+wstr_mutex_name]
.text:00413D09 push eax ; lpName
.text:00413D0A push [ebp+bInitialOwner] ; bInitialOwner
.text:00413D10 lea ecx, [ebp+str_CzYNRd] ; str_input
.text:00413D16 push [ebp+buffer] ; lpMutexAttributes
.text:00413D1C call pkb_load_dll_based_on_input_str
.text:00413D21 mov edx, eax ; dll_handle
.text:00413D23 lea ecx, [ebp+str_enc_CreateMutexW] ; str_enc_api_name
.text:00413D29 call pkb_get_api_addr_by_name_using_GetProcAddress ; CreateMutexW
.text:00413D2E call eax
.text:00413D30 lea ecx, [ebp+str_CzYNRd_2] ; str_input
.text:00413D36 call pkb_load_dll_based_on_input_str
.text:00413D3B mov edx, eax ; dll_handle
.text:00413D3D lea ecx, [ebp+str_enc_GetLastError] ; str_enc_api_name
.text:00413D43 call pkb_get_api_addr_by_name_using_GetProcAddress ; GetLastError
.text:00413D48 call eax
.text:00413D4A cmp eax, ERROR_ALREADY_EXISTS
.text:00413D4F jz loc_415423
```

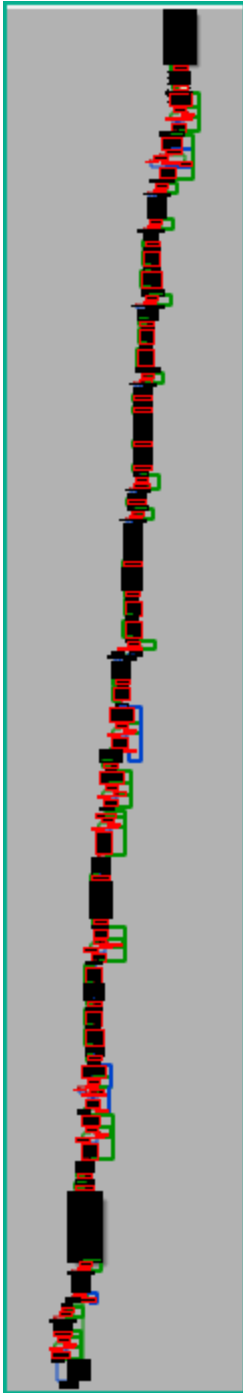
```
if ( !pkb_check_default_lang(v_0x1) )
{
    tmp_str_var.wstr_mutex_name = wstr_mutex_name;
    tmp_var.bInitialOwner = bInitialOwner;
    tmp_stru_var.lpMutexAttributes = ppMutexAttributes;

    // create mutex: {F8B97568-5D50-4696-A969-4C9AF7B69188}
    kernel32_hdl = pkb_load_dll_based_on_input_str(str_CzYNRd); // CzYNRd
    CreateMutexW = pkb_get_api_addr_by_name_using_GetProcAddress(str_enc_CreateMutexW, kernel32_hdl);
    CreateMutexW(tmp_stru_var.lpMutexAttributes, tmp_var.bInitialOwner, tmp_str_var.wstr_mutex_name);

    // check mutex exists to avoid reinfecting the host
    kernel32_hdl = pkb_load_dll_based_on_input_str(str_CzYNRd_2);
    GetLastError = pkb_get_api_addr_by_name_using_GetProcAddress(str_enc_GetLastError, kernel32_hdl);
    if ( GetLastError() != ERROR_ALREADY_EXISTS )
    {
```

7. Create victim uuid

After creating the Mutex as described above, Pikabot creates the victim **uuid** using the function **pkb_collect_victim_info_n_gen_victim_uuid (0x42E233)**. The graph code for this function is as follows:



The **uuid** string is generated based on the information collected from the victim machine, including:

- **Volume serial number** by using API function **GetVolumeInformationW**. This is a unique identifier assigned to each physical volume on a computer.
- **computer name** by using API function **GetComputerNameW**. This is the name of the computer that the malware is running on.
- **user name** by using API function **GetUserNameW**. This is the name of the user who is currently logged on to the computer.
- **OS product type** by using API function **GetProductInfo**.

```

// "qlAHqqa26dEIg9V7IxxJMA_" -> "C:\"
wstr_C_drive = pkb_decrypt_n_convert_wstr(str_enc_root_drive);
lpRootPathName = wstr_C_drive;

// Retrieves information about volume serial number associated with C:\ drive
kernel32_hdl = pkb_load_dll_based_on_input_str(str_CzYNRd);
GetVolumeInformationW = pkb_get_api_addr_by_name_using_GetProcAddress(str_enc_GetVolumeInformationW, kernel32_hdl);
GetVolumeInformationW(
    lpRootPathName,
    lpVolumeNameBuffer,
    nVolumeNameSize,
    &lpVolumeSerialNumber,           // receives the volume serial number
    lpMaximumComponentLength,
    lpFileSystemFlags,
    lpFileSystemNameBuffer,
    nFileSystemNameSize);

```



```

// get computer name by using GetComputerNameW
pkb_get_computer_name(wstr_computer_name);

// get user name by using GetUserNameW
pkb_get_user_name(wstr_user_name);

// Retrieves the product type for the operating system by using GetProductInfo
pkb_get_os_type(wstr_os_type);

// "dbDz8sCH2TCG7QYaEIvt3Q_" -> %s\%s|%s
wstr_format = pkb_decrypt_n_convert_wstr(str_enc_format);

user32_hdl = pkb_load_dll_based_on_input_str(str_osPFU);
_wsprintfW = pkb_get_api_addr_by_name_using_GetProcAddress(str_enc_wsprintfW, user32_hdl);
// format collected inf: "<computer_name>\<user_name>|\<os_type>"
_wsprintfW(wstr_collected_info, wstr_format, wstr_computer_name, wstr_user_name, wstr_os_type);

```

The information collected above will be formatted as follows: “<computer_name>\<user_name>|\<os_type>”. This information will then be hashed using the algorithm mentioned in **3. Retrieve API address** with the hash value will be initialized to the value of **VolumeSerialNumber**.


```

do
{
    v_0x5 = 0x2D;
    v21 = 0x45;
    v15 = 0;
    *&v22 = 0x6400000005Bi64;
    *(&v22 + 1) = 0xFFFFFEEB000000039ui64;
    do
        v_0x5 -= *(&v21 + v15++);
    while ( v15 < 5 );

    c = str_input[idx];
    c_lower = c + 0x20; // conver to lower char
    if ( (c - 0x41) > 0x19u )
        c_lower = c;
    hash_data = volume_ser_num * v_0x5 + c_lower;
    ++idx;
    volume_ser_num = hash_data;
}
while ( idx < len );
return hash_data;

```

The hash value calculated for the collected information along with the **VolumeSerialNumber** will be further calculate by using function **pkb_calc_hash_2 (0x42E123)** below:

```


.text:0042E123 ; int __thiscall pkb_calc_hash_2(_DWORD *pdwNum)
.text:0042E123 pkb_calc_hash_2 proc near
.text:0042E123     mov     eax, [ecx]
.text:0042E125     inc     eax
.text:0042E126     imul   eax, 75BCD15h
.text:0042E12C     mov     [ecx], eax
.text:0042E12E     retn
.text:0042E12E pkb_calc_hash_2 endp

```

```

hash = 0x75BCD15 * (*pdwNum + 1);
*pdwNum = hash;
return hash;

```



Finally, use the API function **wsprintfW** to format the **uuid** string in the format **%07lX%09lX%lu**:

```

str_len = pkb_wstrlen(wstr_collected_info);
pkb_copy_wstr_to_str(wstr_collected_info, str_collected_info, str_len);

// hash collected data
str_len = pkb_strlen(str_collected_info);
hashed_data = pkb_hash_collect_data(str_len, str_collected_info, lpVolumeSerialNumber);

```



```

pkb_uuid.dwPart1 = pkb_calc_hash_2(&hashed_data);
pkb_uuid.wPart2 = pkb_calc_hash_2(&lpVolumeSerialNumber);
v218 = pkb_calc_hash_2(lpVolumeSerialNumber);
pkb_uuid.wPart3 = v218;

```

```

dword_44D90C *= 0x2FEF02D2;

```

```

while ( v_count < v_0x8 )
{
    bVal = pkb_calc_hash_2(&lpVolumeSerialNumber);
    pkb_uuid.next_8bytes[v222] = bVal;
    v_count = v222 + 1;
}

```

```

victim_uuid_final = pkb_alloc_heap_region(2 * v_0x104_1);
if ( victim_uuid_final )
{
    // "mYp5nzADAiCTaixeSeKDUA_" -> %07LX%09LX%lu
    wstr_format_1 = pkb_decrypt_n_convert_wstr(str_enc_format_2);
    dwLast4Bytes = * &pkb_uuid.next_8bytes[v_0x4]; // get the last 4bytes as dword
    wPart3 = pkb_uuid.wPart3;
    dwPart1 = pkb_uuid.dwPart1;

    // format victim_uuid
    user32_hdl = pkb_load_dll_based_on_input_str(str_input);
    _wsprintfW = pkb_get_api_addr_by_name_using_GetProcAddress(str_enc_wsprintfW_2, user32_hdl);
    // %07LX%09LX%lu
    _wsprintfW(victim_uuid_final, wstr_format_1, dwPart1, wPart3, dwLast4Bytes);
}

```

8. Collecting victim machine information

Before connecting to the C2 server, Pikabot will collect some information about the victim machine. The function `pkb_collect_victim_system_info (0x410E37)` performs the following collection tasks:

- Retrieves the `PEB`, gather operating system information, including (`OSMajorVersion`, `OSMinorVersion`, `OSBuildNumber`), determines whether it is running on a `64-bit` operating system or not through the API function `IsWow64Process`.
- Collects the operating system type by using the `GetProductInfo`.
- Gathers the computer name and username by calling the `GetComputerNameW` and `GetUserNameW`.
- Collects CPU information by employing `cpuid` with the initial value of `EAX = 0x80000000`.
- Obtains information about display devices on the machine through the API `EnumDisplayDevicesW`.
- Retrieves the RAM capacity of the victim's machine using `GlobalMemoryStatusEx`.

- Gets the system uptime by utilizing the API function `GetTickCount`.
- Checks if its process is running in admin privileges or not through the `GetCurrentProcess`, `OpenProcessToken`, `GetTokenInformation`.
- Retrieves information about screen resolution using the `GetDesktopWindow` and `GetWindowRect`.
- Collects the domain name using the API `GetComputerNameExW` with `NameType` is `ComputerNameDnsDomain`.
- Gathers `DomainControllerName`, `DomainControllerAddress` using `DsGetDcNameW`. If no information is available, Pikabot will assign it as “`unknown`”.

```

result = pkb_alloc_heap_region(0x22A4);
victim_system_info = result;
if ( result )
{

    // Gather OS info (OSMajorVersion, OSMinorVersion, OSBuildNumber ) from PEB, and
    // determines pkb process is running under WOW64.
    result->victim_os_info = *pkb_get_victim_os_info(&os_info);

    // Retrieves the product type for the operating system by using GetProductInfo.
    // Return value is number that can be mapping with Windows Product (ex:
    // PRODUCT_EDUCATION (0x00000079):Windows 10 Education
    pkb_get_os_type(&victim_system_info->wstr_os_type);

    // Get user name by using GetUserNameW
    pkb_get_user_name(&victim_system_info->wstr_user_name);

    // Get computer name by using GetComputerNameW
    pkb_get_computer_name(&victim_system_info->wstr_computer_name);

    // Gather CPU info by using cpuid with initial value EAX = 0x80000000
    pkb_get_cpu_name(&victim_system_info->wstr_cpu_name);

    // Gather the display adapter name by using EnumDisplayDevicesW
    pkb_get_display_adapter_name(&victim_system_info->wstr_display_adapter);

    // Gather physical ram amount by using GlobalMemoryStatusEx
    victim_system_info->total_physical_mem = pkb_get_physical_mem_info();

```



```

// Gather system uptime by using GetTickCount
victim_system_info->dwUpTime = pkb_get_system_uptime();

// Check current process has elevated priv by using GetTokenInformation
victim_system_info->bisElevated = pkb_check_curr_process_has_elevated_privs();

// Retrieve screen_resolution by using GetDesktopWindow and GetWindowRect
pkb_get_desktop_resolution(&victim_system_info->screen_resolution);

// Gather domain name by using GetComputerNameExW
pkb_get_domain_name(&victim_system_info->domain_name);

// Get domain_controller_address and name by using DsGetDcNameW
pkb_get_domain_controller_name(
    &victim_system_info->domainControllerAddress,
    &victim_system_info->domainControllerName,
    &victim_system_info->domain_name);
return victim_system_info;

```

Next, Pikabot decrypts information related to pikabot `version` and `stream`, my sample has respectively info “`1.1.17-ghost`” and “`GG13TH@T@f0adda360d2b4ccda11468e026526576`”.

Then, the information about the victim collected above will be constructed into a JSON string with the following format:

```
{
  "Xtt2VRnA": "%s",
  "qleNiC": "%s",
  "LPLLXuTl2": " Win %d.%d %d ",
  "0RbIhQuDq": %s,
  "6bw35n": "%s",
  "FQkA0G": "%s",
  "bFFqxURzx": "%s",
  "a0xIcXZI": %d,
  "LkLMKwP1": "%s",
  "R8N3ujt": %d,
  "2sIw0rUG": "%s",
  "UTrXReY": "%s",
  "YoViBQC": "%s",
  "QeMM8": "%s",
  "VLsFyV4d": "%s",
  "EcZbr": %d,
  "XKb5WP": %d
}
```

```

// → {"Xtt2VRnA": "%s", "qleNiC": "%s", "LPLLXuTL2": "Win %d.%d %d", "0RbIhQuDq": %s, "6bw35n": "%s", "FQkA0G": "%s"}
wstr_json_format = pkb_decrypt_n_convert_wstr(enc_wstr_json_format);

// "GK71G23BXn/DLVKDHRbptTsuXMhspwJ/KixyLrcGO/P9soqi70HuvTQnjpkTs4B" → "GG13TH@T@f0adda360d2b4ccda11468e026526576"
pkb_stream_info = pkb_decrypt_n_convert_wstr(enc_stream_info);
victim_system_info_json = pkb_alloc_heap_region(2 * v_0x800);
bisElevated = victim_system_info→bisElevated;
p_domainControllerAddress = &victim_system_info→domainControllerAddress;
p_domainControllerName = &victim_system_info→domainControllerName;
p_domain_name = &victim_system_info→domain_name;
p_screen_resolution = &victim_system_info→screen_resolution;
total_physical_mem = victim_system_info→total_physical_mem;
p_wstr_display_adapter = &victim_system_info→wstr_display_adapter;
dwUpTime = victim_system_info→dwUpTime;
p_wstr_cpu_name = &victim_system_info→wstr_cpu_name;
p_wstr_computer_name = &victim_system_info→wstr_computer_name;
p_wstr_user_name = &victim_system_info→wstr_user_name;
p_wstr_os_type = &victim_system_info→wstr_os_type;
OSBuildNumber = victim_system_info→victim_os_info.OSBuildNumber;
OSMinorVersion = victim_system_info→victim_os_info.OSMinorVersion;
OSMajorVersion = victim_system_info→victim_os_info.OSMajorVersion;

user32_base_addr = pkb_load_dll_based_on_input_str(&pStr[4]);
// gnrBQgCFmVhYOvgDugnqNQ_ → wsprintfW
_wsprintfW = pkb_get_api_addr_by_name_using_GetProcAddress(str_enc_api_name, user32_base_addr);
_wsprintfW(
    victim_system_info_json,
    wstr_json_format,
    wstr_victim_uuid,
    pkb_stream_info,
    OSMajorVersion,
    OSMinorVersion,
    OSBuildNumber,
    p_wstr_os_type,
    p_wstr_user_name,
    p_wstr_computer_name,
    p_wstr_cpu_name,
    dwUpTime,
    p_wstr_display_adapter,
    total_physical_mem,
    p_screen_resolution,
    pkb_version,
    p_domain_name,
    p_domainControllerName,
    p_domainControllerAddress,
    time_seed,
    bisElevated);

```

All information after being formatted into a JSON string will be encrypted. The encryption process is as follows:

- Call the function `pkb_gen_random_chars(0x41BC4A)` to generate the session key: `aes_key` (32 bytes) and `aes_iv` (16 bytes).
- Call the function `pkb_gen_rndom_chars(0x41BC4A)` for generating 3 random characters, which was used as a marker. I will temporarily call it `marker`.
- Call the function `pkb_aes_crypt_data(0x40A97A)` to encrypt the JSON string with the generated `aes_key` and `iv`.
- Call the function `pkb_base64_encode(0x0040B4DD)` to encode the encrypted data above.
- Then all information will be stored in the following format: `<marker (rand_3_chars)><aes_key (first 16 bytes)><aes_iv><encoded data><aes_key (last 16 bytes)>`.

- Finally, use a loop to iterate through the entire buffer to replace the character '=' with '_'.

Here is the code flow:

```
// generate an AES session iv:
// ex: "5hbB5hbB5hbB5hbB"
rand_session_aes_iv_16bytes = pkb_gen_random_chars(v_0x10);
```

```
// generate an AES session iv:
// ex: "5hbB5hbB5hbB5hbB"
rand_session_aes_iv_16bytes = pkb_gen_random_chars(v_0x10);
```

```
// generate random 3 chars, using it as marker
// ex: "IEI"
rand_3_chars = pkb_gen_random_chars(v_0x3);
```

```
// perform AES-CBC to encrypt and base64 to encode data
if ( rand_session_aes_key_32bytes && rand_session_aes_iv_16bytes && rand_3_chars )
{
    aes_data_len = pkb_aes_crypt_data(
        dst_buf,                // dst_buf: store original data
        src_buf,                // src_buf: overwrite by encrypted data
        src_buf_size,
        rand_session_aes_key_32bytes,
        rand_session_aes_iv_16bytes);
    base64_data_len = pkb_base64_encode(dst_buf, src_buf, aes_data_len); // dst_buf store encoded base64

    total_len = v211 + v212 + base64_data_len + v213;

    // save data to buffer int the following format
    // <rand_3_chars><aes_key(first 16 bytes)><aes_iv><encoded data><aes_key(last 16 bytes)>
    pkb_strcpy(src_buf, rand_3_chars, str_input_len);
    pkb_strcpy(&src_buf[v_0x3], rand_session_aes_key_32bytes, v_0x20 / v_0x2);
    pkb_strcpy(&src_buf[v_0x3 + v_0x20 / v_0x2], rand_session_aes_iv_16bytes, v_0x10);
    GetMessageExtraInfo();

    // save encoded data to buffer
    pkb_strcpy(&src_buf[v_0x20 + v_0x3], dst_buf, base64_data_len);

    // save the last 16 bytes of aes key to buffer
    pkb_strcpy(
        &src_buf[v_0x3 + base64_data_len + v_0x20],
        &rand_session_aes_key_32bytes[v_0x20 / v_0x2],
        v_0x20 / v_0x2);
    src_buf[total_len + v_0x1] = 0;

    // replace '=' with '_'
    for ( m = v_0x0; m < total_len; ++m )
    {
        if ( src_buf[m] == '=' )
            src_buf[m] = '_';
    }
}
```

9. Information gathering with other commands

In addition to the information collected as mentioned above, Pikabot also executes the following commands to gather additional information from the victim's machine:

- `netstat.exe -aon`
- `ipconfig.exe /all`
- `whoami.exe /all`

```
// - Pikabot executes commands to collect information or is configured not to execute (DISABLED)
// - command result is encrypted and store into following format:
// &yi5SX=<random_3chars><aes_key(first 16bytes)><aes_iv(16bytes)><encrypted data><aes_key(last 16bytes)>
pkb_exec_command_n_encrypt_result(
    tmp_buffer_1,
    enc_collected_data,
    tmp_buffer,
    enc_str_yi5SX,
    str_enc_netstat_aon); // netstat.exe -aon
pkb_memcpy_n_concat(encrypted_victim_info_final, enc_collected_data);
pkb_memset(enc_collected_data, v_0x0, 2 * v_0x400 * v_0x32);

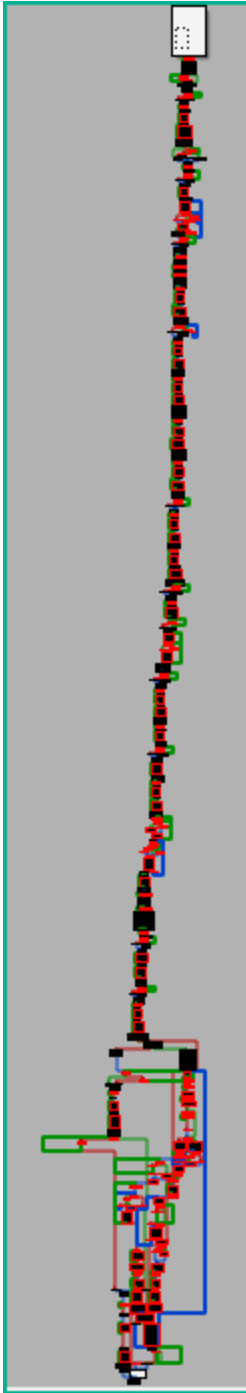
// &n68lp=
pkb_exec_command_n_encrypt_result(
    tmp_buffer_1,
    enc_collected_data,
    tmp_buffer,
    en_str_n68lp,
    str_enc_ipconfig_all); // ipconfig.exe /all
pkb_memcpy_n_concat(encrypted_victim_info_final, enc_collected_data);
pkb_memset(enc_collected_data, v_0x0, 2 * v_0x400 * v_0x32);

// &jv6KH=
// whoami.exe /all
pkb_exec_command_n_encrypt_result(tmp_buffer_1, enc_collected_data, tmp_buffer, en_str_jv6KH, str_enc_whoami_all);
pkb_memcpy_n_concat(encrypted_victim_info_final, enc_collected_data);
pkb_memset(enc_collected_data, v_0x0, 2 * v_0x400 * v_0x32);
```

The results of these commands are also encrypted and stored in the same way as above. However, the sample that I am analyzing is configured as **DISABLED**.

10. Collect running processes

Pikabot call the function `pkb_enum_n_collect_all_running_processes (0x415BAF)` to gather information about running processes on the victim's machine by employing the API functions `CreateToolhe132Snashot`, `Process32FirstW` và `Process32NextW`. The graph code of this function is as follows:



The information collected will be compiled in the following format:

```
00000020 ["[System Process]:0:0:0:0:1:0", "System:4:0:8:0:0:0", "Registry:108:4:8:0:0:0",
"sms.exe:376:4:11:0:0:0", "csrss.exe:468:460:13:0:0:0", "wininit.exe:568:460:13:0:0:0",
"csrss.exe:576:560:13:0:0:0", "winlogon.exe:664:560:13:0:0:0", "services.exe:712:568:9:0:0:0",
"lsass.exe:732:568:9:0:0:0", "svchost.exe:856:712:8:0:0:0", "fontdrvhost.exe:884:664:8:0:0:0",
"fontdrvhost.exe:892:568:8:0:0:0", "svchost.exe:972:712:8:0:0:0", "svchost.exe:304:712:8:0:0:0",
"dwm.exe:460:664:13:0:0:0", "svchost.exe:1060:712:8:0:0:0", "svchost.exe:1096:712:8:0:0:0",
"svchost.exe:1180:712:8:0:0:0", "svchost.exe:1188:712:8:0:0:0", "svchost.exe:1216:712:8:0:0:0",
"svchost.exe:1240:712:8:0:0:0", "svchost.exe:1268:712:8:0:0:0", "svchost.exe:1396:712:8:0:0:0",
"svchost.exe:1452:712:8:0:0:0", "svchost.exe:1536:712:8:0:0:0", "svchost.exe:1544:712:8:0:0:0",
"svchost.exe:1552:712:8:0:0:0", "svchost.exe:1568:712:8:0:0:0", "svchost.exe:1576:712:8:0:0:0"]
```

Then, the information will also be encrypted and encoded in the same way as described above:

```
// "pAiJP3YU4EDKK1eYLh0Dkw_" -> "&sTICX="
str_STICX = pkb_decrypt_str(enc_str_var);

// take a snapshot of all the running processes of the victim machine using CreateToolhel32Snapshot, Process32FirstW and Process32NextW
collected_data_len = pkb_enum_n_collect_all_running_processes(tmp_buffer1, v10, v_0x200, v_0x400);

// copy and convert data from wstr to str
if ( collected_data_len != v_0x0 )
    pkb_copy_wstr_to_str(tmp_buffer1, tmp_buffer2, collected_data_len);

// encrypt and encode collected data
pkb_aes_crypt_n_base64_data_n_store_into_buff(encrypt_collected_data, tmp_buffer2, collected_data_len);

dec_str_var_len = pkb_strlen(str_STICX);
pkb_strcpy(data_out_final, str_STICX, dec_str_var_len);

// copy the final encrypted data to buffer
pkb_memcpy_n_concat(data_out_final, tmp_buffer2);

// free all buffer
collected_data_len = pkb_strlen(str_STICX);
pkb_memset(str_STICX, v_0x0, collected_data_len);
pkb_free_heap_region(str_STICX);
pkb_memset(tmp_buffer1, v_0x0, 2 * v59 * v60);
pkb_memset(encrypt_collected_data, v_0x0, 2 * v_0x400 * v_0x200);
return pkb_memset(tmp_buffer2, v_0x0, 2 * v_0x400 * v_0x200);
```

11. Decrypt C2 configuration

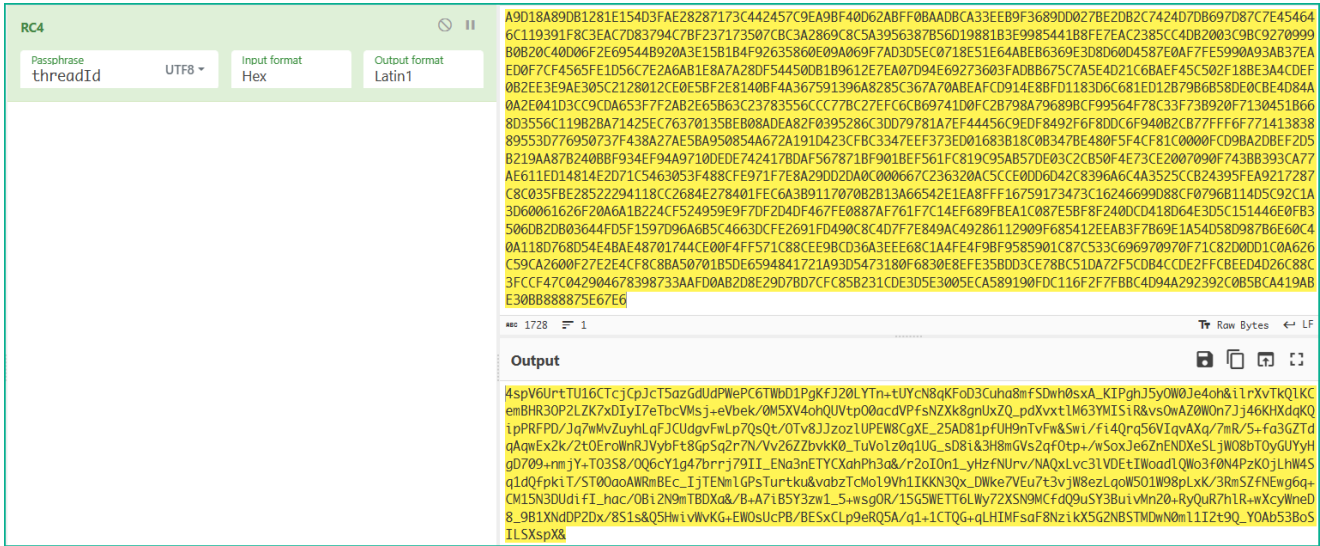
The C2 addresses (IP and port) will be decrypted by Pikabot during execution. First, Pikabot performs the decryption of C2 encrypted data using RC4, with the decryption key in this sample being “**threadId**”:

```
// copy encrypted c2 config to buffer
memcpy(enc_c2_config_data, g_enc_c2_config, sizeof(enc_c2_config_data));

do
{
    v157 = 0;
    v154 = (v154 + 1);
    v775 = *(&rc4_ksa[8] + v154);
    v_0x1E_1 = v775;
    v152 = (v152 + v775);
    v158 = L"oncoreuap\\base\\appmodel\\search\\common\\pkmutild\\cregistry.cxx";
    do
    {
        ++v157;
        ++v158;
    }
    while ( v157 >= 0x536 && v158 );
    ::a2 = a4 & 0x812DAC11;
    a4 ^= 0xF8CDB105;
    IsZoomed(0x12F2);
    *(&rc4_ksa[8] + v154) = *(&rc4_ksa[8] + v152);
    *(&rc4_ksa[8] + v152) = v775;
    p2_config_data[idx] = enc_c2_config_data[idx] ^ *(&rc4_ksa[8] + (v_0x1E_1 + *(&rc4_ksa[8] + v154)));
    ++idx;
}
while ( idx < 0x360 );
```

```
.text:00403A88 g_enc_c2_config dd 898AD1A9h, 0E18112D0h, 0E2FAD354h, 3C178782h, 0C9572444h, 0DF49BEAh
.text:00403A88 ; DATA XREF: pkb_main_proc+EFC+o
.text:00403A88 dd 0BFFAB62h, 33CADBAAh, 68F3B9EEh, 0BE27D09Dh, 42C7B22Dh, 97B67D4Dh
.text:00403A88 dd 457E7CD8h, 19C14646h, 3E8C1F39h, 79837DACH, 37F27B4Ch, 0CB073517h
.text:00403A88 dd 9C86A2C3h, 56395A8Ch, 0D1567B38h, 0E9B38198h, 0B8415498h, 23AC7EFEh
.text:00403A88 dd 0B24DCC85h, 0C99B3C00h, 0B0990927h, 0D040CB2h, 54692E6Fh, 3E0A9248h
```

Here is the result with **CyberChef**:



Then, Pikabot decrypts the character “&” and uses it as delimiter to extract the decrypted string above into sub base64 strings:

```
// "u/63I5wcol7eCGNohOX1kQ_" → "&"
delim_char = pkb_decrypt_str(enc_str_delim);
pkb_extracted_enc_c2_arr = pkb_alloc_heap_region(4 * v_0x1E); // 0x78
```

```
// find position of delim char and fill null byte, then return extracted string.
enc_sub_str_c2_extracted = pkb_extract_str_based_on_delim(str_c2_config_data_, delim_char);
```

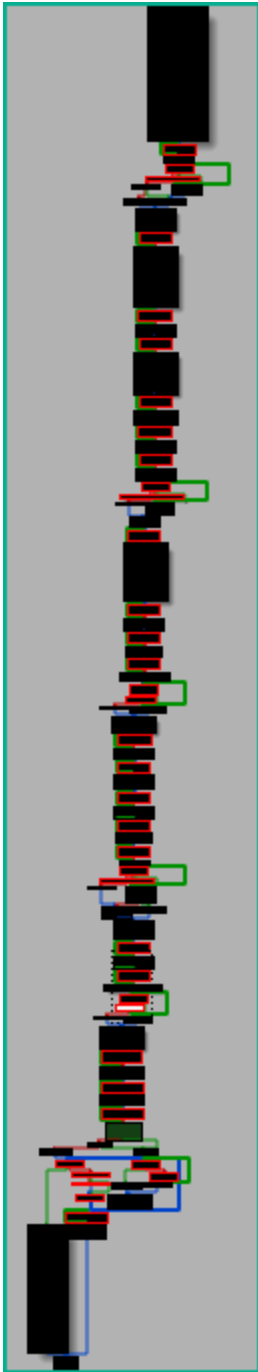
```
// copy extracted enc c2 config to allocated buffer
len = pkb_strlen(enc_sub_str_c2_extracted);
pkb_strcpy(pkb_extracted_enc_c2_arr[count], enc_sub_str_c2_extracted, len);
}
}
++count;
}
while ( count < v_0x1E );
}
```

Result of the above process when debugged with x32dbg:

Address	Hex	ASCII	Address	Hex	ASCII
007DC6F8	34 73 70 56 36 55 72 74 54 55 31 36 43 54 63 6A	4spV6UrtTU16CTcj	007DCB00	69 6C 72 58 76 54 6B 51 6C 4B 43 65 60 42 48 52	1lRxvTkQlKcembHR
007DC708	43 70 4A 63 54 35 61 7A 47 64 55 64 50 57 65 50	CpJcT5azGdUdPwEP	007DCB10	33 4F 50 32 4C 5A 4B 37 78 44 49 79 49 37 65 54	30P2LZK7xDIyI7eT
007DC718	43 36 54 57 62 44 31 50 67 4B 66 4A 32 30 4C 59	C6TWbD1PgKfJ20LY	007DCB20	62 63 56 4D 73 6A 2B 65 56 62 65 6B 2F 30 4D 35	bcVmsj+eVbek/0M5
007DC728	54 6E 2B 74 55 59 63 4E 38 71 4B 46 6F 44 33 43	Tn+TUYcN8qF0D3C	007DCB30	58 56 34 6F 68 51 55 56 74 70 4F 30 61 63 64 56	XV4ohQUVtp00acdV
007DC738	75 68 61 38 6D 66 53 44 77 68 30 73 78 41 5F 4B	uhamfSDwh0sxA_K	007DCB40	50 66 73 4E 5A 58 6B 38 67 6E 55 78 5A 51 5F 70	PfsNzXk8gnUxZQ_p
007DC748	49 50 67 68 4A 35 79 4F 57 30 4A 65 34 6F 68 00	iPghJ5y0W0Jea4oh.	007DCB50	64 58 76 78 74 6C 4D 36 33 59 4D 49 53 69 52 00	dXvxtlM63YmISiR.

Address	Hex	ASCII
007D4E88	F8 C6 7D 00 00 CB 7D 00 70 30 7D 00 78 34 7D 00	øR{.E}.p0}.x4}.
007D4EF8	80 38 7D 00 38 3C 7D 00 90 40 7D 00 08 CF 7D 00	.8}.<}.~}.I}.
007D4F08	10 D3 7D 00 18 D7 7D 00 20 DB 7D 00 28 DF 7D 00	.0}.*}.0}.(8}.
007D4F18	30 E3 7D 00 38 E7 7D 00 40 EB 7D 00 48 EF 7D 00	0â}.8ç}.@è}.Hi}.
007D4F28	E0 27 7E 00 E8 2B 7E 00 C8 1B 7E 00 A0 07 7E 00	â'~.è+-E..-.-.
007D4F38	A8 0B 7E 00 D8 23 7E 00 78 F3 7D 00 30 F7 7D 00	'..0#~.x0}..+}.
007D4F48	88 FB 7D 00 90 FF 7D 00 98 03 7E 00 80 0F 7E 00	.0}.y}...~.~.
007D4F58	B8 13 7E 00 D0 1F 7E 00 79 B5 57 85 C0 B1 00 08	'..0..~.yµW.A+..
007D4F68	00 00 00 00 FF FF FF FF 68 01 00 00 40 49 00 00	...yÿÿÿh...@I..

Next, Pikabot calls function `pkb_decrypt_data (0x41D07B)` to perform the task of decrypting the C2 address. The graph code of this function is as follows:



The entire decrypting process is as follows:

- Allocate buffers to store the `AES key` and `iv`.
- Convert the string to the valid `Base64` string by replacing the character “_” with “=”.
- Discard first 3 characters of string, take the next 16 characters (bytes) and store them to the buffer to create the first part of the `AES key`.
- Take the next 16 characters (bytes) and store them to the buffer to use as `AES iv`.

- Take the last 16 characters (bytes) to make the second part of the **AES key**, combine it with the first part to create the complete **AES key**.
- Get the string to be decoded after obtaining the **AES key** and **iv**.
- Perform **Base64** decode.
- Use **AES-CBC** with **AES key** and **iv** above to decrypt the final C2 data.

Pseudocode of the entire process is as follows:

```
// alloc buffer to store aes key
aes_key = pkb_alloc_heap_region(v_0x20 + v_0x1); // size = 0x21

// alloc buffer to store aes iv
aes_iv = pkb_alloc_heap_region(v_0x1 + v_0x10); // size = 0x11
if ( aes_key && aes_iv )
{
    // Convert back to a valid base64 string by replace "_" by "="
    for ( n = v_0x0; n < enc_data_len; ++n )
    {
        if ( enc_data[n] == '_' )
            enc_data[n] = '=';
    }

    // Skip the first 3 bytes,
    // take the next 16 characters (bytes) and store them in the buffer to become the first part of the AES Key.
    pkb_strcpy(aes_key, &enc_data[v_0x3], v_0x20 / v_0x2);

    // Take the next 16 characters (bytes) and store them in the buffer to become AES IV.
    pkb_strcpy(aes_iv, &enc_data[v_0x3 + v_0x20 / v_0x2], iv_len_0x10);
    dword_44D60C = dword_44D5C0 & 0x834C544;
    pkb_weird_func_19(dword_44D5C0 & 0x834C544);

    // Take the last 16 characters (bytes) to make the second part of the AES Key,
    // combine it with the first part to create a complete AES Key.
    pkb_strcpy(&aes_key->aes_key[v_0x20 / v_0x2], &enc_data[enc_data_len - v_0x20 / v_0x2], aes_len_0x10);

    // Get the string that needs to be decrypted after obtaining the AES key and IV.
    pkb_strcpy(enc_data, &enc_data[v_0x3 + v_0x20], enc_data_len - v_0x3 - v_0x10 - v_0x20);
    pkb_memset(&enc_data[enc_data_len - v_0x20], value, nBytes);
    len = pkb_strlen(enc_data);

    // Base64 decode on the encrypted c2 config
    b64_decoded_data_len = pkb_base64_decode(len, enc_data);

    // AES CBC with Key and IV above to decrypt and return the final C2 config
    len = pkb_aes_crypt(b64_decoded_data_len, enc_data, aes_key->aes_key, aes_iv);

    // release allocated memory
    pkb_memset(aes_key, v160, v127 + v126);
    enc_data[len] = 0;
    pkb_free_heap_region(aes_iv);
    pkb_free_heap_region(aes_key);
    return len;
}
```

AES Key part 1	AES IV	Encrypted C2 config	AES Key part 2
4spV6UrtTU16CTcjCpJ	cT5azGdUdPWPC6T	WbD1PgKfJ20LYTn+tUYcN8qKFoD3Cuha8mfSDwh0sxA	_KIPghJ5yOW0Je4oh

Using **CyberChef**, we get the following results:

Recipe

From Base64

Alphabet
A-Za-z0-9+/=

Remove non-alphabet chars

Strict mode

AES Decrypt

Key: V6Ur+tTU16CTcjC... UTF8

IV: cT5azGdUdPWeP... LATIN1

Mode: CBC Input: Raw Output: Raw

Input
WbD1PgKfJ20LYTn+tUYcN8qKFoD3Cuha8mf5Dwh0sxA=

Output
45.32.188.56:2967

We can write a Python script to decrypt all the C2 addresses that Pikabot will use:

```
λ python pikabot_decrypt_c2_ip_addr.py
Decrypted c2: 45.32.188.56:2967

Decrypted c2: 154.221.30.136:13724
Decrypted c2: 78.141.222.198:13786
Decrypted c2: 216.128.136.231:13786
Decrypted c2: 108.61.224.209:2967
Decrypted c2: 139.84.235.8:2225
Decrypted c2: 45.32.235.46:5242
Decrypted c2: 210.243.8.247:23399
Decrypted c2: 192.248.151.140:23399
```

12. Pikabot uses Syscall

During the analysis, we will encounter the following functions:

68	BF8D2F82	push	0x822F8DBF
E8	55FFFFFF	call	<pkb_retrieve_ntdll_api_by_hash_n_exec>
68	5346CA40	push	0x40CA4653
E8	4BFFFFFF	call	<pkb_retrieve_ntdll_api_by_hash_n_exec>
68	15079F05	push	0x59F0715
E8	41FFFFFF	call	<pkb_retrieve_ntdll_api_by_hash_n_exec>
68	4179DD48	push	0x48DD7941
E8	37FFFFFF	call	<pkb_retrieve_ntdll_api_by_hash_n_exec>
68	320EA30F	push	0xFA30E32
E8	2DFFFFFF	call	<pkb_retrieve_ntdll_api_by_hash_n_exec>
68	58F0A686	push	0x86A6F058
E8	23FFFFFF	call	<pkb_retrieve_ntdll_api_by_hash_n_exec>

The above function will perform the following tasks:

```
int __usercall pkb_retrieve_ntdll_api_by_hash_n_exec@<eax>(NTDLL_API_HASHES a1)
{
    // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

    g_ZwAPI_ctx.ret_addr_of_caller = ret_addr_of_caller; // save return address of caller (ex: 0x0044C78B)
    g_ZwAPI_ctx.ret_addr_of_caller_to_caller = ret_addr_of_caller_to_caller; // save return address of caller to caller (ex: 0x0415B95)
    g_ZwAPI_ctx.pStackArgs = &a1; // stack arguments
    // ex:
    // 1: [esp] FFFFFFFF
    // 2: [esp+4] 00000000
    // 3: [esp+8] 0019DF08
    // 4: [esp+C] 00000018
    // 5: [esp+10] 00000000

    // returns the index of the function whose
    // hash value is equal to the precomputed hash value
    g_ZwAPI_ctx.g_ZwAPI_idx = pkb_get_idx_of_ZwAPI_func_by_hash(v3);
    g_ZwAPI_ctx.g_rand_syscall_stub = pkb_retrieve_rand_syscall_stub(NtCurrentTeb() -> WOW32Reserved != 0);

    // jump to "call edx" in that random stub and then the syscall is performed.
    (g_ZwAPI_ctx.g_rand_syscall_stub)();
    return (g_ZwAPI_ctx.ret_addr_of_caller_to_caller)();
}
```

Iterate over the PEB, check if the loaded dll is **ntdll.dll**

```
.text:0044186C mov     eax, [ebp-90h]
.text:00441872 add     esi, 18996197h
.text:00441878 mov     ecx, [ebp-60h]
.text:0044187B mov     ecx, [eax+ecx*8+IMAGE_NT_HEADERS.OptionalHeader.DataDirectory.VirtualAddress]
.text:0044187F cmp     ecx, [ebp-64h]
.text:00441882 jz      short loc_4418B1
.text:00441882
.text:00441884 mov     eax, [ebp-8]
.text:00441887 lea    edx, [ecx+eax]
.text:0044188A mov     ecx, [edx+IMAGE_EXPORT_DIRECTORY.Name]
.text:0044188D add     ecx, eax
.text:0044188F mov     [ebp-24h], edx
.text:00441892 mov     edx, ' '
.text:00441897 mov     eax, [ecx]
.text:00441899 or     eax, edx
.text:0044189B cmp     eax, 'ldtn'
.text:004418A0 jnz     short loc_4418B1
.text:004418A0
.text:004418A2 mov     eax, [ebp-3Ch]
.text:004418A5 mov     eax, [ecx+eax]
.text:004418A8 or     eax, edx
.text:004418AA cmp     eax, 'ld.L'
.text:004418AF jz      short loc_4418C9

// Check if the dll name is ntdll.dll or not?
export_dir_rva = pNtHeaders->OptionalHeader.DataDirectory[dir_idx].VirtualAddress;
if ( export_dir_rva != v_0x0 )
{
    export_dir_va = (export_dir_rva + dllBaseAddr);
    str_dll_name = (dllBaseAddr + *(sexport_dir_rva->Name + dllBaseAddr));
    if ( (*str_dll_name | ' ') == 'ldtn' && (*(str_dll_name+160 | ' ') == 'ld.L' )
        break;
}
pLdrEntry = *(&pLdrEntry->InLoadOrderLinks.Flink + v158);
while ( pLdrEntry->DllBase != v_0x0 );
```

If yes, proceed to find API functions starting with “Zw” exported by **ntdll.dll**.

```

.text:0044196A mov     eax, [ebp-24h]
.text:0044196D add     esi, 73716732h
.text:00441973 sub     eax, edx
.text:00441975 mov     [ebp-4], edx
.text:00441978 mov     edx, [ebp-40h]
.text:0044197B mov     [ebp-28h], esi
.text:0044197E mov     dword_44ED84, esi
.text:00441984 mov     edx, [edx+eax*4]
.text:00441987 mov     eax, 'wZ'
.text:0044198C add     edx, ecx
.text:0044198E cmp     [edx], ax
.text:00441991 jnz     _continue

```

```

// Searching for API functions starting with "Zw" in order from the bottom up.
ntdll_ZwAPIName = (dllBaseAddr + pNamesAddrTbl[nAPINames - v_0x1]);
if ( *ntdll_ZwAPIName != 'wZ' )
{
    nCalcedZwAPIsHashes = v_0x0;
    goto _continue;
}

```

The found functions will be hashed, and the result will be stored in the format:

<calced_hash><api_func_RVA>

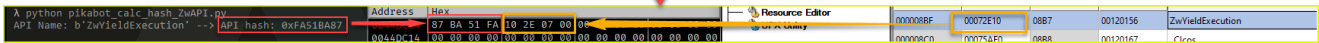
```

// If found, calculate the hash with the found function name.
calced_hash = pkb_calc_hash_of_ntdll_api(ntdll_ZwAPIName);
arr_idx = v171;

// save calced hash to global array
g_ZwAPI_arr.g_ZwAPI_calced_hash_arr[2 * count] = calced_hash;

// save corresponding API RVA addr to global array
// the final table is: <calced_hash><api_func_RVA>
g_ZwAPI_arr.g_ZwAPIFuncRVA_arr[2 * arr_idx] = pFunctionsAddrTbl[pNameOrdinalsAddrTbl[nAPINames - v_0x1]];
nCalcedZwAPIsHashes = arr_idx + 1;
count = nCalcedZwAPIsHashes;
if ( nCalcedZwAPIsHashes == _ESI + 0x157C ) // 0x226
    break;
v79 = v_0x0;
_continue:
--nAPINames;
}
while ( nAPINames );

```



The calculated table will be then sorted by Function RVA in ascending order:

Address	Hex	ASCII
0044DC04	0A 13 A7 08 90 29 07 00 1D 2D B8 01 A0 29 07 00	..§..)...)..
0044DC14	5E 7F F1 64 B0 29 07 00 39 25 94 7F C0 29 07 00	^..nd°)..9%..Ä)..
0044DC24	2B 08 97 38 D0 29 07 00 26 5F A8 22 E0 29 07 00	+..8ð)..ö_""à)..
0044DC34	AD 61 18 B1 F0 29 07 00 46 3A F4 62 00 2A 07 00	.a.±ð)..F:ðb.*..
0044DC44	50 DC C3 D4 10 2A 07 00 DF 20 8A C2 20 2A 07 00	PÛÃÔ.*..ß .Â *..
0044DC54	48 7C DA 48 30 2A 07 00 60 79 F0 64 40 2A 07 00	H ÚH0*..`yðd@*..
0044DC64	50 63 88 26 50 2A 07 00 99 3A 0F 30 60 2A 07 00	Pc.ðP*...:0`*..
0044DC74	94 AB C2 68 70 2A 07 00 41 79 DD 48 80 2A 07 00	.«Âhp*..AyÝH.*..
0044DC84	65 12 F8 18 90 2A 07 00 70 A5 F6 5E A0 2A 07 00	e.ø..*..p%ö^ *..
0044DC94	5C D2 02 05 B0 2A 07 00 38 A2 9B 8F C0 2A 07 00	\ð..°*..8ç..À*..
0044DCA4	08 75 DA 4A D0 2A 07 00 FF 78 2F 42 E0 2A 07 00	.uÚJð*..ÿx/Bâ*..
0044DCB4	6C B4 EA 55 F0 2A 07 00 6A 70 09 9D 00 2A 07 00	l`êUð*..jp...+..
0044DCC4	23 1B AD 1D 10 2B 07 00 BF 8D 2F 82 20 2B 07 00	#...+...¿./.. +..
0044DCD4	5C F3 93 72 50 2B 07 00 BF EB 15 83 60 2B 07 00	\ð.rP+...ë..`+..
0044DCE4	0C C3 53 E1 70 2B 07 00 C3 D9 25 B6 80 2B 07 00	.ASáp+..ÄÛ%¶..+..
0044DCF4	47 54 2B B1 90 2B 07 00 39 A6 06 D5 A0 2B 07 00	GT±+. ...!..Ö +..
0044DD04	00 F2 96 38 B0 2B 07 00 A2 78 92 A7 C0 2B 07 00	ðð.8°+..çx.§Â+..
0044DD14	12 3F B1 20 D0 2B 07 00 39 93 1E 8D E0 2B 07 00	.?± ð+... ..â+..
0044DD24	E6 09 A2 23 F0 2B 07 00 8E 44 3D 1A 00 2C 07 00	æ.ç#ð+...D=... ..
0044DD34	32 0E A3 0F 10 2C 07 00 31 3F C4 E5 20 2C 07 00	2.E... ..?Ââ ..
0044DD44	17 F6 8D D2 30 2C 07 00 07 10 4A CF 40 2C 07 00	.ö.ðð, ...JÏ@,..
0044DD54	53 46 CA 40 50 2C 07 00 C5 CE 91 13 60 2C 07 00	SFÊ@P, ...ÄÏ..`... ..

Finally, compare the pre-calculated hash value with the table containing the calculated hash values above, if equal, return the function ID. This ID value is stored in the EAX register:


```
if ( pkb_gen_ntdll_Zw_api_hashes_tbl(&savedregs) )
{
    if ( count ≥ g_nCalcedZwAPIsHashes )
    {
        return -v_0x1;
    }
    else
    {
        while ( pre_api_hash ≠ g_ZwAPI_calced_hash.ZwAPI_calced_hash_arr[2 * count] )
        {
            if ( ++count ≥ g_nCalcedZwAPIsHashes )
                return -v_0x1;
        }
        return count; // return the index of the corresponding function
    }
}
```

Hide FPU	
EAX	00000019
EBX	006274B8
ECX	822F8DBF

syscall ID

Based on the hash algorithm, we can find out the API functions that Pikabot will use as follows:

```
λ python pikabot_brute_api_funcs_of_ntdll.py
API hash: 0x1DAD1B23 --> API found: b'ZwAllocateVirtualMemory'
API hash: 0x48DD7941 --> API found: b'ZwClose'
API hash: 0x86A6F058 --> API found: b'ZwCreateThreadEx'
API hash: 0x19BA1F1B --> API found: b'ZwGetContextThread'
API hash: 0xFA30E32 --> API found: b'ZwOpenProcess'
API hash: 0x59F0715 --> API found: b'ZwProtectVirtualMemory'
API hash: 0x822F8DBF --> API found: b'ZwQueryInformationProcess'
API hash: 0xE0C7AF3 --> API found: b'ZwReadVirtualMemory'
API hash: 0x1C30891E --> API found: b'ZwResumeThread'
API hash: 0x1BBE470E --> API found: b'ZwSetContextThread'
API hash: 0x40CA4653 --> API found: b'ZwUnmapViewOfSection'
API hash: 0xDD95C91E --> API found: b'ZwWriteVirtualMemory'
```

13. References

End.

m4n0w4r