RedDelta PlugX Undergoing Changes and Overlapping Again with Mustang Panda PlugX Infrastructure

blog.xorhex.com/blog/reddeltaplugxchangeup/

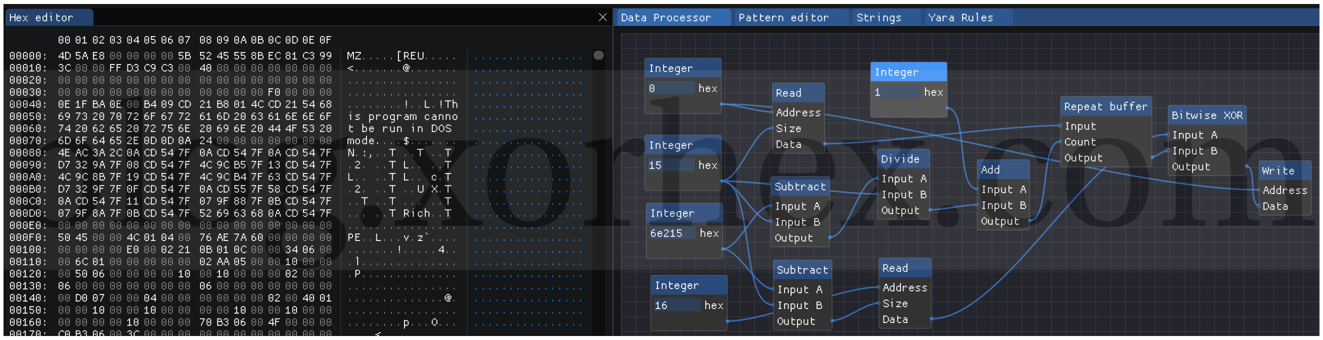
xorhex

Focus on Threat Research through malware reverse engineering

New RedDelta PlugX variant undergoes revisions to slow down analysis. Extracted C2s link back to two known Mustang Panda command and control servers.

June 2, 2021
xorhex

9-Minute Read



```
       00 01 02 03 04 05 06 07  08 09 0A 0B 0C 0D 0E 0F
00000: 4D 5A E8 00 00 00 00 5B  52 45 55 8B EC 81 C3 99  MZ.....[REU.....
00010: 3C 00 00 FF D3 C9 C3 00  40 00 00 00 00 00 00 00  <.......@.......
00020: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00030: 00 00 00 00 00 00 00 00  00 00 00 00 F0 00 00 00  ................
00040: 0E 1F BA 0E 00 B4 09 CD  21 B8 01 4C CD 21 54 68  ........!..L.!Th
00050: 69 73 20 70 72 6F 67 72  61 6D 20 63 61 6E 6E 6F  is program canno
00060: 74 20 62 65 20 72 75 6E  20 69 6E 20 44 4F 53 20  t be run in DOS 
00070: 6D 6F 64 65 2E 0D 0D 0A  24 00 00 00 00 00 00 00  mode....$.......
00080: 4E AC 3A 2C 0A CD 54 7F  0A CD 54 7F 0A CD 54 7F  N.:,..T...T...T.
00090: D7 32 9A 7F 08 CD 54 7F  4C 9C B5 7F 13 CD 54 7F  .2....T.L.....T.
000A0: 4C 9C 8B 7F 19 CD 54 7F  4C 9C B4 7F 63 CD 54 7F  L.....T.L...c.T.
000B0: D7 32 9F 7F 0F CD 54 7F  0A CD 55 7F 58 CD 54 7F  .2....T...U.X.T.
000C0: 0A CD 54 7F 11 CD 54 7F  07 9F 88 7F 0B CD 54 7F  ..T...T.......T.
000D0: 07 9F 8A 7F 0B CD 54 7F  52 69 63 68 0A CD 54 7F  ......T.Rich..T.
000E0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
000F0: 50 45 00 00 4C 01 04 00  76 AE 7A 60 00 00 00 00  PE..L...v.z`....
00100: 00 00 00 00 E0 00 02 21  0B 01 0C 00 00 34 06 00  .......!.....4..
00110: 00 6C 01 00 00 00 00 00  02 AA 05 00 00 10 00 00  .l..............
00120: 00 50 06 00 00 00 00 10  00 10 00 00 00 02 00 00  .P..............
00130: 06 00 00 00 00 00 00 00  06 00 00 00 00 00 00 00  ................
00140: 00 D0 07 00 00 04 00 00  00 00 00 00 02 00 40 01  ..............@.
00150: 00 00 10 00 00 10 00 00  00 00 10 00 00 10 00 00  ................
00160: 00 00 00 00 10 00 00 00  70 B3 06 00 4F 00 00 00  ........p...O...
```

Family	PlugX - RedDelta Variant
Threat Actor	Mustang Panda
Encrypted	1c7897a902b35570a9620c64a2926cd5d594d4ff5a033e28a400981d14516600
Decryption Key	0x78, 0x61, 0x6c, 0x72, 0x45, 0x5a, 0x6f, 0x78, 0x43, 0x59, 0x73, 0x71, 0x6c, 0x52, 0x6e, 0x77, 0x78, 0x46, 0x63, 0x43, 0x46
Key Length	21

Decrypted	ec1c29cb6674ffce989576c51413a6f9cbb4a8a41cbd30ec628182485a937160
Config C2	101.36.125.203:965
Config C2	101.36.125.203:110
Config C2	vitedannews.com:965
Config C2	vitedannews.com:110

Summary

Mustang Panda (aka RedDelta, BRONZE PRESIDENT) is striving to make their PlugX variant more challenging to reverse statically. This RedDelta PlugX variant overlaps with instrastructure tied to Mustang Panda's PlugX variant, something we've seen before. Mustang Panda is believed to be a Chinese nation-sponsored espionage group. Public reporting shows MustangPanda targeting non-government organizations (NGOs), including religious entities. They appear to focus locations in close proximity like Mongolia, Hong Kong, and Vietnam. Mustang Panda is known for making use of PlugX, Posion Ivy, and Cobalt Strike. The PlugX sample covered in this blog demonstrates how this group is continuing to evolve their toolset in a likely attempt to slow down researchers and avoid security automation tools.

Key Findings

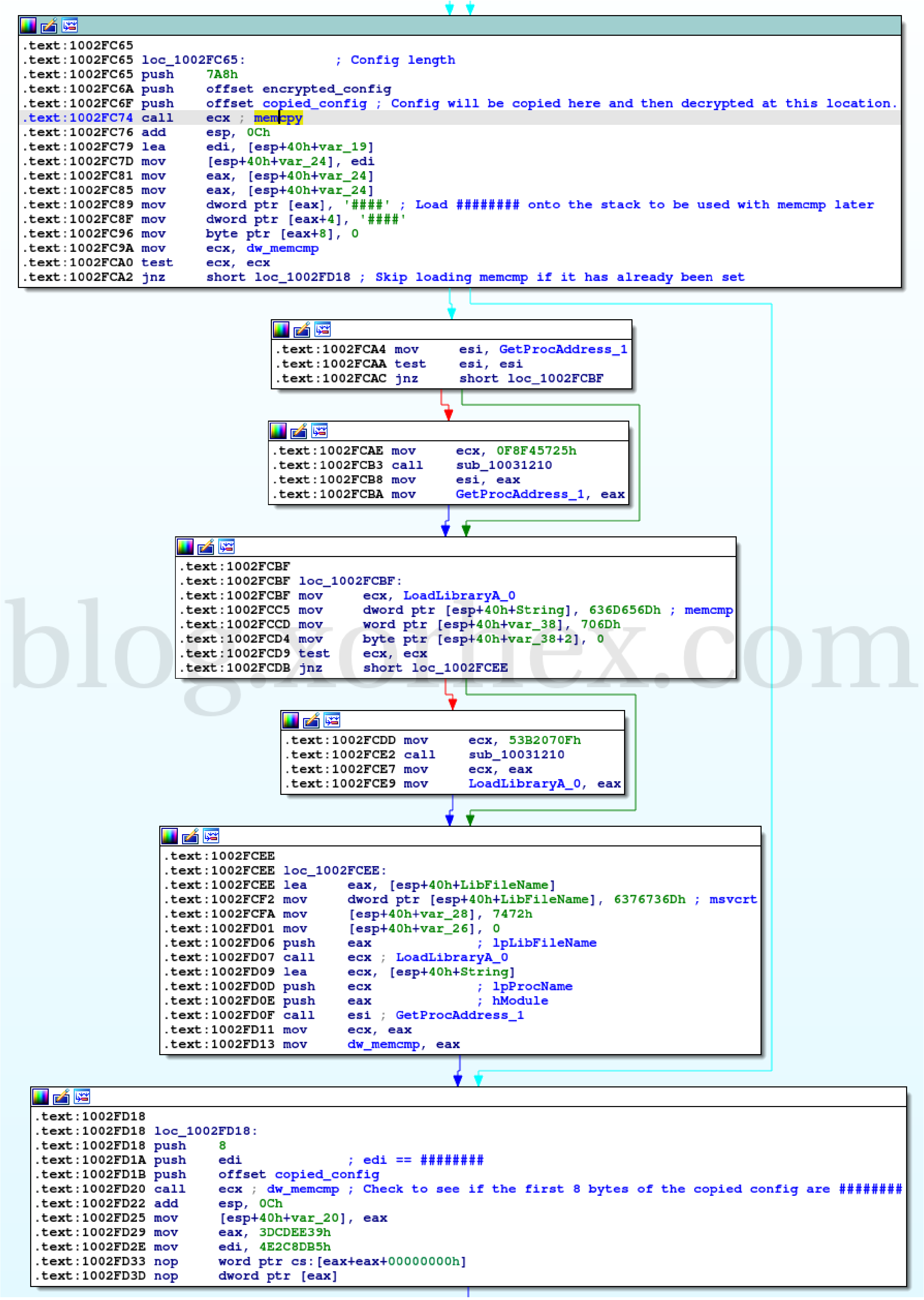
- Shares command and control infrastructure with other Mustang Panda PlugX binaries
- Decryption key length increased to 21 characters
- Control flow obfuscation added
- Code variations in the dynamic Windows API resolutions

Mustang Panda / RedDelta Connection

When Recorded Future reported on RedDelta last year they differentiated between Mustang Panda and RedDelta. They both make use of PlugX binaries, and due to binary similarities and overlapping infrastructure, we track them as the same group. The key binary differences in the RedDelta PlugX version are:

- Config block check for `########`
- RC4 Encryption

> The config block check is how we primarily distinguish between the Mustang Panda variant and the RedDelta variant. The "original" Mustang Panda variant uses `XXXXXXXX`.



```
.text:1002FC65
.text:1002FC65 loc_1002FC65:            ; Config length
.text:1002FC65 push    7A8h
.text:1002FC6A push    offset encrypted_config
.text:1002FC6F push    offset copied_config ; Config will be copied here and then decrypted at this location.
.text:1002FC74 call    ecx ; memcpy
.text:1002FC76 add     esp, 0Ch
.text:1002FC79 lea     edi, [esp+40h+var_19]
.text:1002FC7D mov     [esp+40h+var_24], edi
.text:1002FC81 mov     eax, [esp+40h+var_24]
.text:1002FC85 mov     eax, [esp+40h+var_24]
.text:1002FC89 mov     dword ptr [eax], '####' ; Load ######## onto the stack to be used with memcmp later
.text:1002FC8F mov     dword ptr [eax+4], '####'
.text:1002FC96 mov     byte ptr [eax+8], 0
.text:1002FC9A mov     ecx, dw_memcmp
.text:1002FCA0 test    ecx, ecx
.text:1002FCA2 jnz     short loc_1002FD18 ; Skip loading memcmp if it has already been set
```

```
.text:1002FCA4 mov     esi, GetProcAddress_1
.text:1002FCAA test    esi, esi
.text:1002FCAC jnz     short loc_1002FCBF
```

```
.text:1002FCAE mov     ecx, 0F8F45725h
.text:1002FCB3 call    sub_10031210
.text:1002FCB8 mov     esi, eax
.text:1002FCBA mov     GetProcAddress_1, eax
```

```
.text:1002FCBF
.text:1002FCBF loc_1002FCBF:
.text:1002FCBF mov     ecx, LoadLibraryA_0
.text:1002FCC5 mov     dword ptr [esp+40h+String], 636D656Dh ; memcmp
.text:1002FCCD mov     word ptr [esp+40h+var_38], 706Dh
.text:1002FCD4 mov     byte ptr [esp+40h+var_38+2], 0
.text:1002FCD9 test    ecx, ecx
.text:1002FCDB jnz     short loc_1002FCEE
```

```
.text:1002FCDD mov     ecx, 53B2070Fh
.text:1002FCE2 call    sub_10031210
.text:1002FCE7 mov     ecx, eax
.text:1002FCE9 mov     LoadLibraryA_0, eax
```

```
.text:1002FCEE
.text:1002FCEE loc_1002FCEE:
.text:1002FCEE lea     eax, [esp+40h+LibFileName]
.text:1002FCF2 mov     dword ptr [esp+40h+LibFileName], 6376736Dh ; msvcrt
.text:1002FCFA mov     [esp+40h+var_28], 7472h
.text:1002FD01 mov     [esp+40h+var_26], 0
.text:1002FD06 push    eax              ; lpLibFileName
.text:1002FD07 call    ecx ; LoadLibraryA_0
.text:1002FD09 lea     ecx, [esp+40h+String]
.text:1002FD0D push    ecx              ; lpProcName
.text:1002FD0E push    eax              ; hModule
.text:1002FD0F call    esi ; GetProcAddress_1
.text:1002FD11 mov     ecx, eax
.text:1002FD13 mov     dw_memcmp, eax
```

```
.text:1002FD18
.text:1002FD18 loc_1002FD18:
.text:1002FD18 push    8
.text:1002FD1A push    edi              ; edi == ########
.text:1002FD1B push    offset copied_config
.text:1002FD20 call    ecx ; dw_memcmp ; Check to see if the first 8 bytes of the copied config are ########
.text:1002FD22 add     esp, 0Ch
.text:1002FD25 mov     [esp+40h+var_20], eax
.text:1002FD29 mov     eax, 3DCDEE39h
.text:1002FD2E mov     edi, 4E2C8DB5h
.text:1002FD33 nop     word ptr cs:[eax+eax+00000000h]
.text:1002FD3D nop     dword ptr [eax]
```

Figure 1: RedDelta Variant Config Check (ec1c29cb6674ffce989576c51413a6f9cbb4a8a41cbd30ec628182485a937160)

Overlapping features between both of these variants include:

- Prepended XOR key
- Shellcode in the MZ header
- Stack Strings
- Rolling Config XOR decryption key: `123456789`

This sample contains all of these features including the RedDelta PlugX ones.

We believe with moderate confidence that this sample is tied to the Mustang Panda/RedDelta threat actor group.

Similar Yet Different

Encrypted DAT File

On May 24 2021 an encrypted DAT file was uploaded to VirusTotal from Vietnam. The file was uploaded with the name `SmadDB.dat` and is encypted with a 21 byte XOR key prepended to the binary.

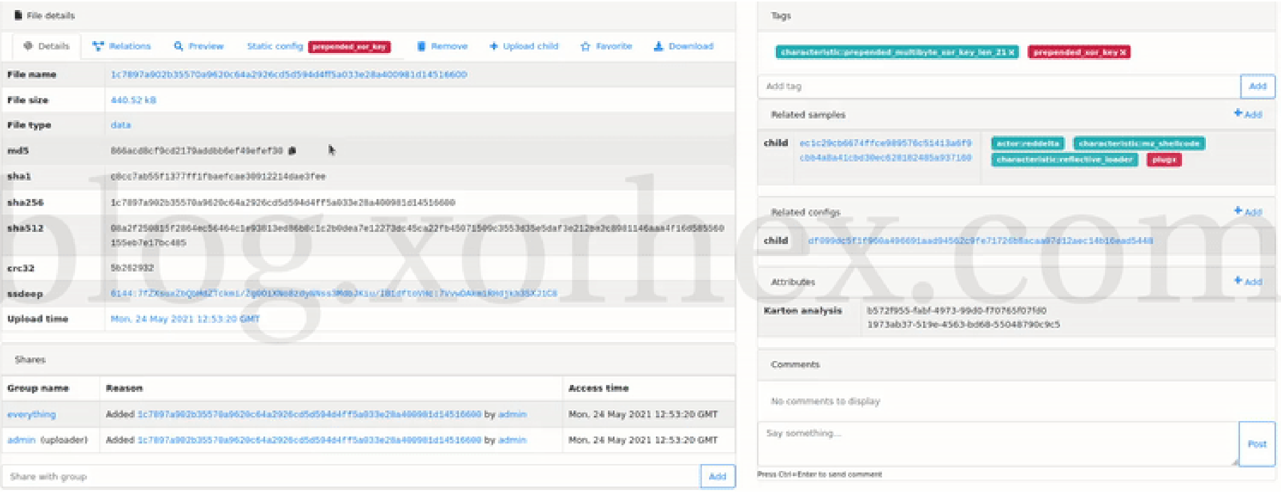


Figure 2: MWDB

While the majority of the RedDelta PlugX variants we have seen use a 10 byte prepended XOR key; this is not the first deviation. There are three others in our collection that have a prepended XOR key longer than 10 bytes.

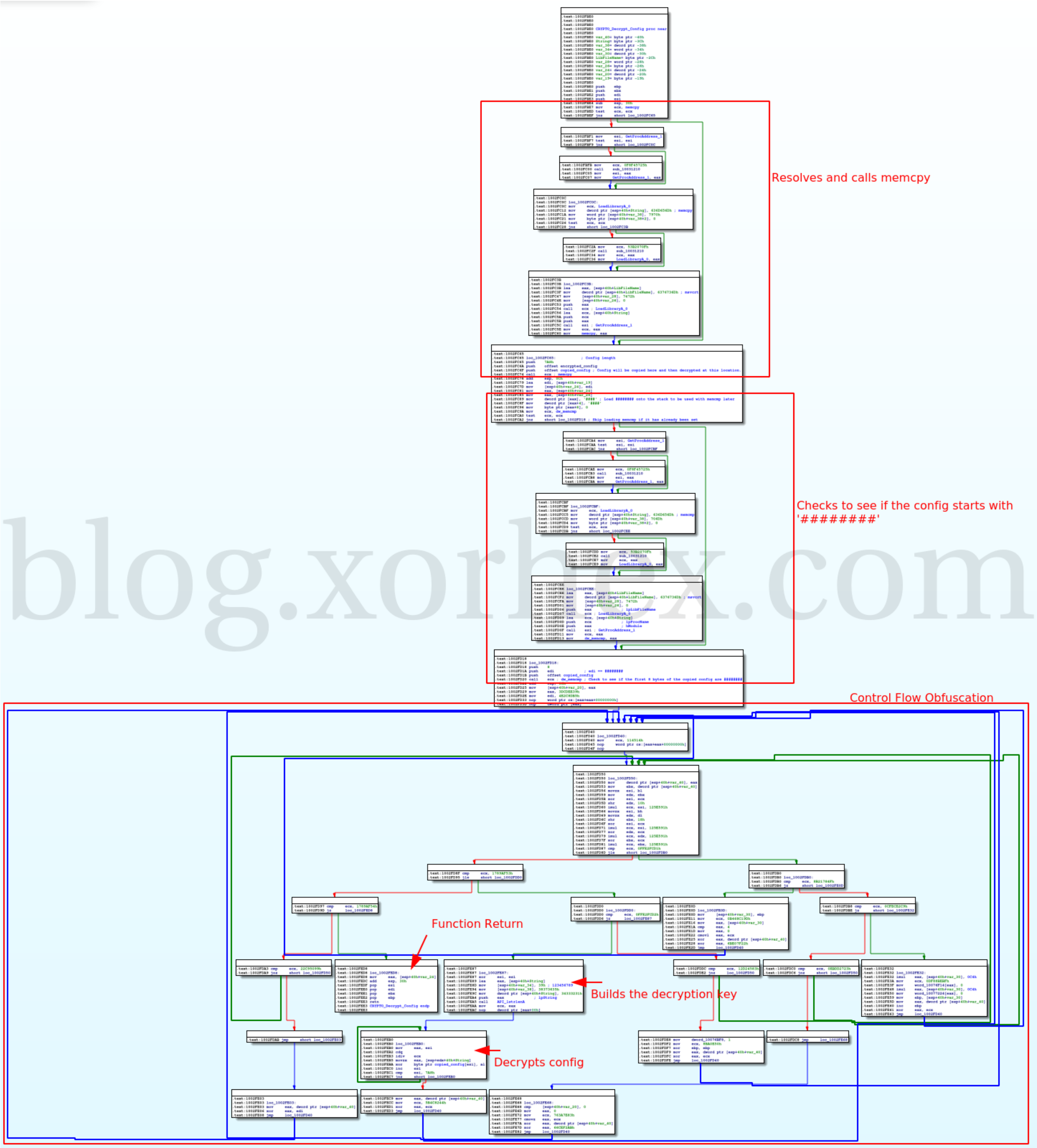
XOR Key Length	SHA256 (Encrypted File)
13	dba437c9030b5f857ce9820a0c9e2c252fd8aeda71c2101024d3576c446972a0
15	a1eb4ce6eaa0c35ca4e8285c32b59cd0dfb34018b3f454d4fa4cebe9906534d8

XOR Key Length	SHA256 (Encrypted File)
17	2304891f176a92c62f43d9fd30cae943f1521394dce792c6de0e097d10103d45
21	1c7897a902b35570a9620c64a2926cd5d594d4ff5a033e28a400981d14516600 (most recent sample)

This is not the only change; control flow obfuscation is also being added to the malware.

Control Flow Obfuscation

Mustang Panda is working on adding control flow obfuscation to their PlugX variant. This first example shows control flow obfuscation added to the config decrypting routine.



Resolves and calls memcpy

Checks to see if the config starts with '########'

Control Flow Obfuscation

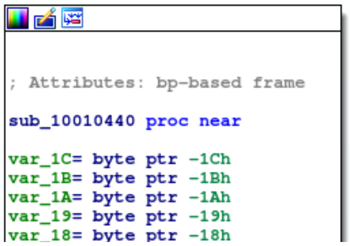
Function Return

Builds the decryption key

Decrypts config

Figure 3:1c7897a902b35570a9620c64a2926cd5d594d4ff5a033e28a400981d14516600

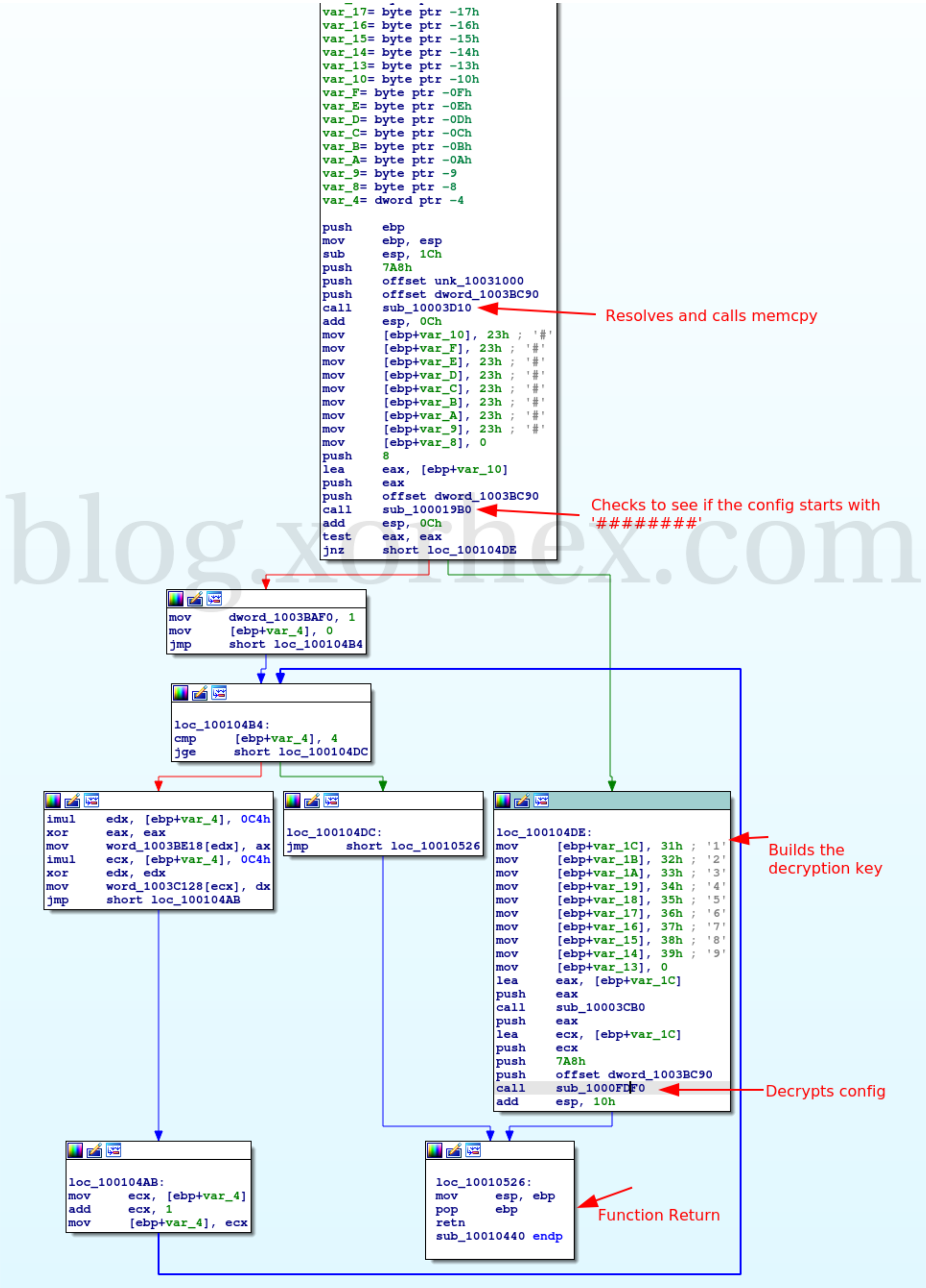
We don’t see this when comparing it to a prior sample (Figure 4).



```
; Attributes: bp-based frame

sub_10010440 proc near

var_1C= byte ptr -1Ch
var_1B= byte ptr -1Bh
var_1A= byte ptr -1Ah
var_19= byte ptr -19h
var_18= byte ptr -18h
```



```
var_17= byte ptr -17h
var_16= byte ptr -16h
var_15= byte ptr -15h
var_14= byte ptr -14h
var_13= byte ptr -13h
var_10= byte ptr -10h
var_F= byte ptr -0Fh
var_E= byte ptr -0Eh
var_D= byte ptr -0Dh
var_C= byte ptr -0Ch
var_B= byte ptr -0Bh
var_A= byte ptr -0Ah
var_9= byte ptr -9
var_8= byte ptr -8
var_4= dword ptr -4

push    ebp
mov     ebp, esp
sub     esp, 1Ch
push    7A8h
push    offset unk_10031000
push    offset dword_1003BC90
call    sub_10003D10
add     esp, 0Ch
mov     [ebp+var_10], 23h ; '#'
mov     [ebp+var_F], 23h ; '#'
mov     [ebp+var_E], 23h ; '#'
mov     [ebp+var_D], 23h ; '#'
mov     [ebp+var_C], 23h ; '#'
mov     [ebp+var_B], 23h ; '#'
mov     [ebp+var_A], 23h ; '#'
mov     [ebp+var_9], 23h ; '#'
mov     [ebp+var_8], 0
push    8
lea     eax, [ebp+var_10]
push    eax
push    offset dword_1003BC90
call    sub_100019B0
add     esp, 0Ch
test    eax, eax
jnz     short loc_100104DE
```

Resolves and calls memcpy

Checks to see if the config starts with '########'

```
mov     dword_1003BAF0, 1
mov     [ebp+var_4], 0
jmp     short loc_100104B4
```

```
loc_100104B4:
cmp     [ebp+var_4], 4
jge     short loc_100104DC
```

```
imul    edx, [ebp+var_4], 0C4h
xor     eax, eax
mov     word_1003BE18[edx], ax
imul    ecx, [ebp+var_4], 0C4h
xor     edx, edx
mov     word_1003C128[ecx], dx
jmp     short loc_100104AB
```

```
loc_100104DC:
jmp     short loc_10010526
```

```
loc_100104DE:
mov     [ebp+var_1C], 31h ; '1'
mov     [ebp+var_1B], 32h ; '2'
mov     [ebp+var_1A], 33h ; '3'
mov     [ebp+var_19], 34h ; '4'
mov     [ebp+var_18], 35h ; '5'
mov     [ebp+var_17], 36h ; '6'
mov     [ebp+var_16], 37h ; '7'
mov     [ebp+var_15], 38h ; '8'
mov     [ebp+var_14], 39h ; '9'
mov     [ebp+var_13], 0
lea     eax, [ebp+var_1C]
push    eax
call    sub_10003CB0
push    eax
lea     ecx, [ebp+var_1C]
push    ecx
push    7A8h
push    offset dword_1003BC90
call    sub_1000FDF0
add     esp, 10h
```

Builds the decryption key

Decrypts config

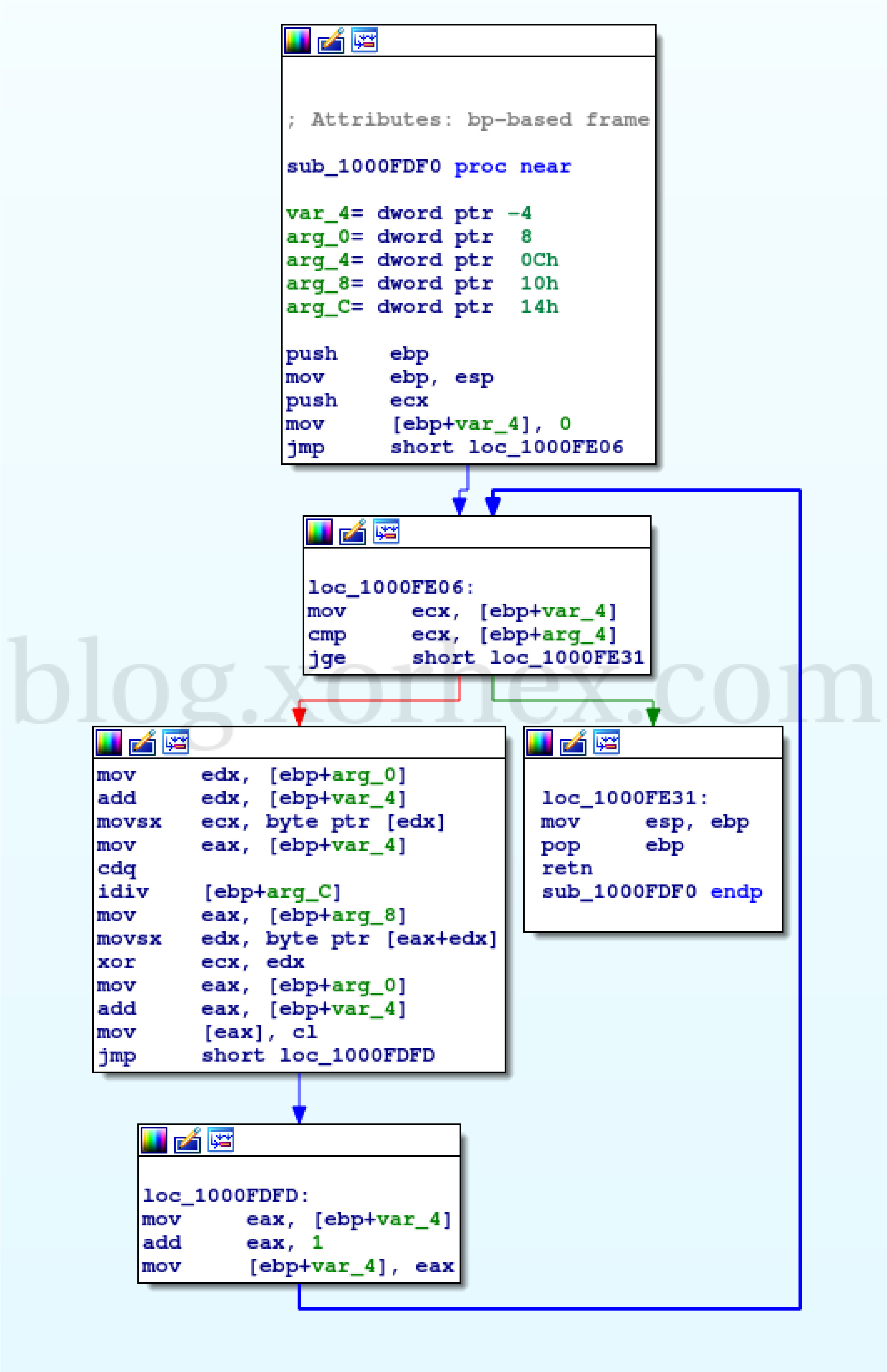
```
loc_100104AB:
mov     ecx, [ebp+var_4]
add     ecx, 1
mov     [ebp+var_4], ecx
```

```
loc_10010526:
mov     esp, ebp
pop     ebp
retn
sub_10010440 endp
```

Function Return

Figure 4: dba437c9030b5f857ce9820a0c9e2c252fd8aeda71c2101024d3576c446972a0

In addition to control flow obfuscating being added, the newest sample's function (Figure 3) is updated to directly house the decryption routine versus it being in a separate function, as seen in the prior sample (Figure 5).



```
; Attributes: bp-based frame

sub_1000FDF0 proc near

var_4= dword ptr -4
arg_0= dword ptr  8
arg_4= dword ptr  0Ch
arg_8= dword ptr  10h
arg_C= dword ptr  14h

push    ebp
mov     ebp, esp
push    ecx
mov     [ebp+var_4], 0
jmp     short loc_1000FE06
```

```
loc_1000FE06:
mov     ecx, [ebp+var_4]
cmp     ecx, [ebp+arg_4]
jge     short loc_1000FE31
```

```
mov     edx, [ebp+arg_0]
add     edx, [ebp+var_4]
movsx   ecx, byte ptr [edx]
mov     eax, [ebp+var_4]
cdq
idiv    [ebp+arg_C]
mov     eax, [ebp+arg_8]
movsx   edx, byte ptr [eax+edx]
xor     ecx, edx
mov     eax, [ebp+arg_0]
add     eax, [ebp+var_4]
mov     [eax], cl
jmp     short loc_1000FDFD
```

```
loc_1000FE31:
mov     esp, ebp
pop     ebp
retn
sub_1000FDF0 endp
```

```
loc_1000FDFD:
mov     eax, [ebp+var_4]
add     eax, 1
mov     [ebp+var_4], eax
```

Figure 5: XOR Routine - dba437c9030b5f857ce9820a0c9e2c252fd8aeda71c2101024d3576c446972a0

Our second control flow obfuscation example in this binary is pulled from an API hashing algorithm.

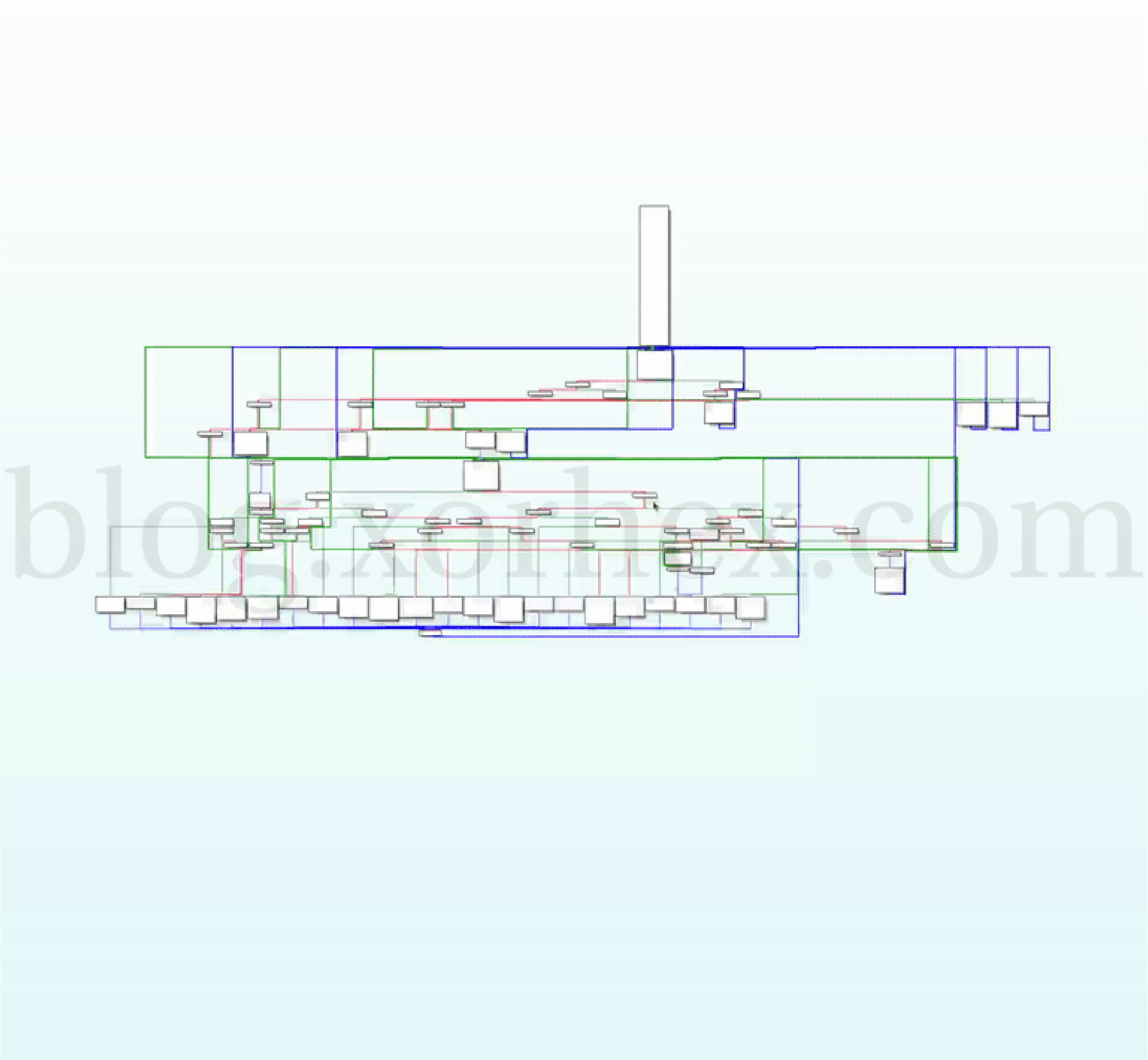


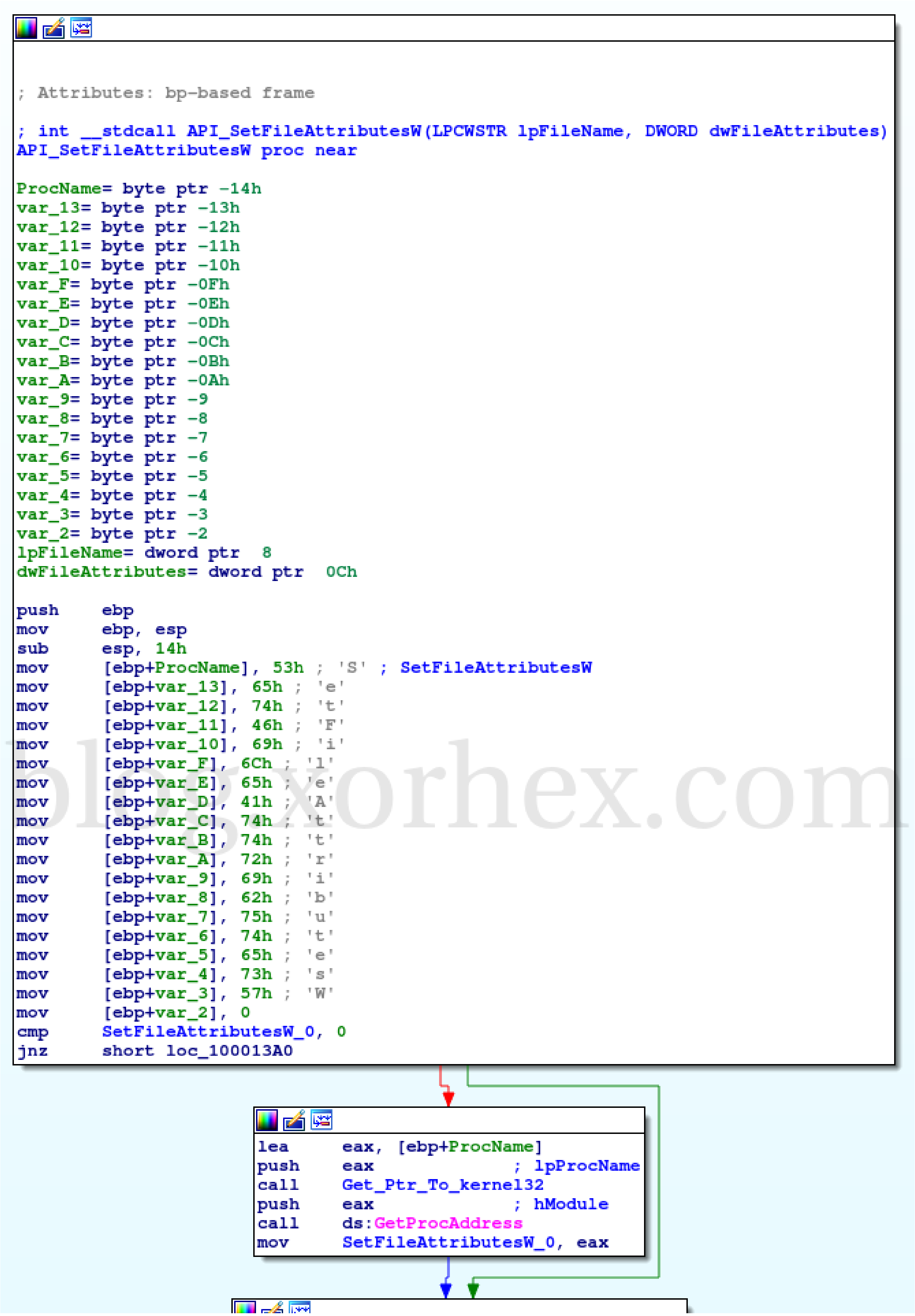
Figure 6: Control Flow Obfuscation - API Hashing Function

Notice the multiple comparison statements controlling which branches are taken (Figure 6), making it harder to follow the exection flow.

Mustang Panda appears to be adding control flow obfuscation to parts of their code but it does not exist in all of the functions yet. They don’t stop with control flow obfuscation; they are also modifing how the dynamic Windows API lookup is being performed.

Resolving Windows API Calls

The binary also contains deviations in how it dynamically resolves Windows API functions. Figure 7 shows how the previous samples did this.



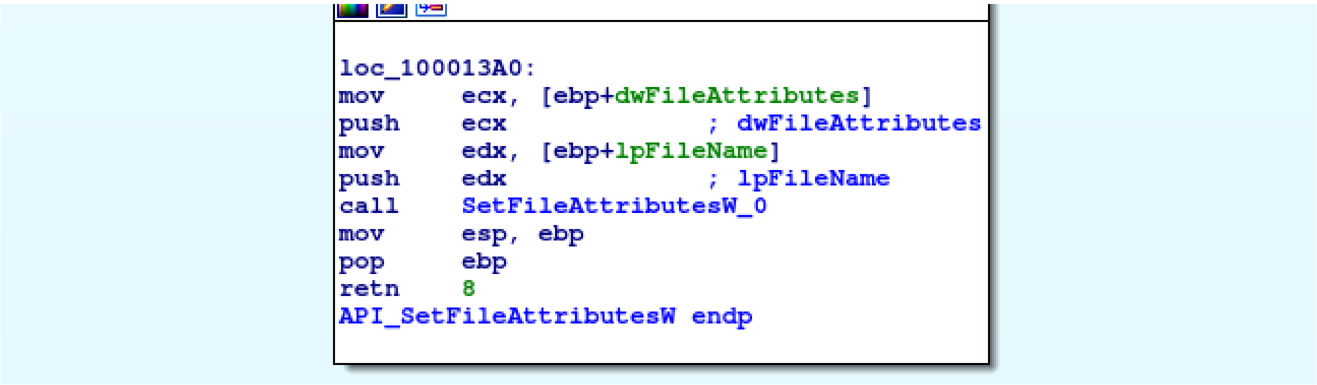
```
; Attributes: bp-based frame

; int __stdcall API_SetFileAttributesW(LPCWSTR lpFileName, DWORD dwFileAttributes)
API_SetFileAttributesW proc near

ProcName= byte ptr -14h
var_13= byte ptr -13h
var_12= byte ptr -12h
var_11= byte ptr -11h
var_10= byte ptr -10h
var_F= byte ptr -0Fh
var_E= byte ptr -0Eh
var_D= byte ptr -0Dh
var_C= byte ptr -0Ch
var_B= byte ptr -0Bh
var_A= byte ptr -0Ah
var_9= byte ptr -9
var_8= byte ptr -8
var_7= byte ptr -7
var_6= byte ptr -6
var_5= byte ptr -5
var_4= byte ptr -4
var_3= byte ptr -3
var_2= byte ptr -2
lpFileName= dword ptr  8
dwFileAttributes= dword ptr  0Ch

push    ebp
mov     ebp, esp
sub     esp, 14h
mov     [ebp+ProcName], 53h ; 'S' ; SetFileAttributesW
mov     [ebp+var_13], 65h ; 'e'
mov     [ebp+var_12], 74h ; 't'
mov     [ebp+var_11], 46h ; 'F'
mov     [ebp+var_10], 69h ; 'i'
mov     [ebp+var_F], 6Ch ; 'l'
mov     [ebp+var_E], 65h ; 'e'
mov     [ebp+var_D], 41h ; 'A'
mov     [ebp+var_C], 74h ; 't'
mov     [ebp+var_B], 74h ; 't'
mov     [ebp+var_A], 72h ; 'r'
mov     [ebp+var_9], 69h ; 'i'
mov     [ebp+var_8], 62h ; 'b'
mov     [ebp+var_7], 75h ; 'u'
mov     [ebp+var_6], 74h ; 't'
mov     [ebp+var_5], 65h ; 'e'
mov     [ebp+var_4], 73h ; 's'
mov     [ebp+var_3], 57h ; 'W'
mov     [ebp+var_2], 0
cmp     SetFileAttributesW_0, 0
jnz     short loc_100013A0
```

```
lea     eax, [ebp+ProcName]
push    eax             ; lpProcName
call    Get_Ptr_To_kernel32
push    eax             ; hModule
call    ds:GetProcAddress
mov     SetFileAttributesW_0, eax
```



```
loc_100013A0:
mov     ecx, [ebp+dwFileAttributes]
push    ecx             ; dwFileAttributes
mov     edx, [ebp+lpFileName]
push    edx             ; lpFileName
call    SetFileAttributesW_0
mov     esp, ebp
pop     ebp
retn    8
API_SetFileAttributesW endp
```

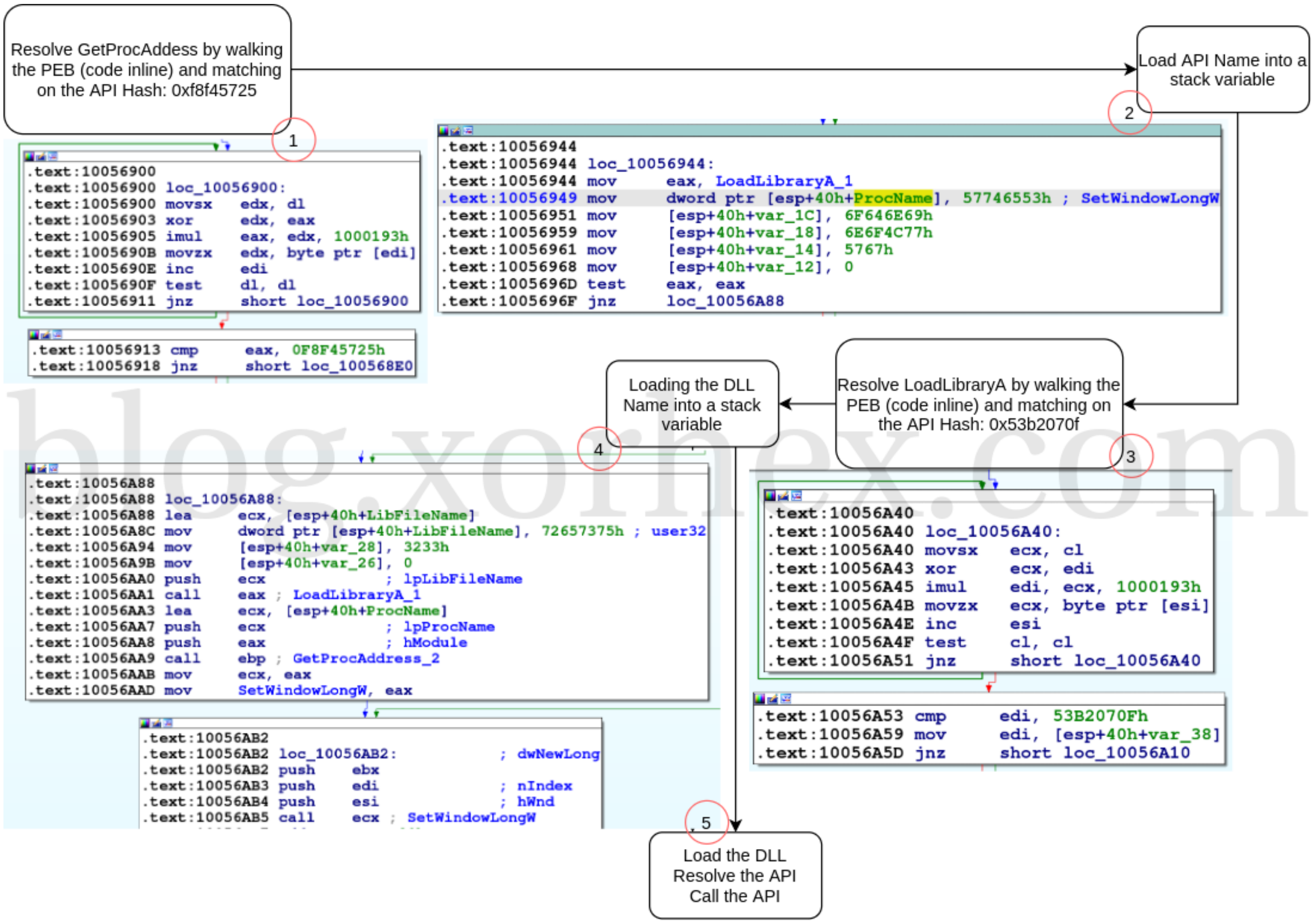
Figure 7: Prior Dynamic API Calling (dba437c9030b5f857ce9820a0c9e2c252fd8aeda71c2101024d3576c446972a0)

Notice the API name is built on the stack and then resolved using GetProcAddress. They do this consistantly throughout the binary when dynamically resolving API calls.

The new sample, ec1c29cb6674ffce989576c51413a6f9cbb4a8a41cbd30ec628182485a937160, changes things up a bit by using two slightly different variations on the same pattern to resolve Windows API calls.

Added Techinque - Method 1

The first method involves using API hashing to get `LoadLibraryA` and `GetProcAddress` function pointers. The API hashing code is placed inline with the rest of the function. It’s not separated into its own function. This is an important distinction as the next method does separate it out. The API name is built on the stack between resolving `GetProcAddress` and `LoadLibraryA`. The final step is to execute the Windows function hidden in the stack string. The diagram below outlines the process in more detail.



Resolve GetProcAddess by walking the PEB (code inline) and matching on the API Hash: 0xf8f45725

```
.text:10056900
.text:10056900 loc_10056900:
.text:10056900 movsx   edx, dl
.text:10056903 xor     edx, eax
.text:10056905 imul    eax, edx, 1000193h
.text:1005690B movzx   edx, byte ptr [edi]
.text:1005690E inc     edi
.text:1005690F test    dl, dl
.text:10056911 jnz     short loc_10056900
```

```
.text:10056913 cmp     eax, 0F8F45725h
.text:10056918 jnz     short loc_100568E0
```

Load API Name into a stack variable

```
.text:10056944
.text:10056944 loc_10056944:
.text:10056944 mov     eax, LoadLibraryA_1
.text:10056949 mov     dword ptr [esp+40h+ProcName], 57746553h ; SetWindowLongW
.text:10056951 mov     [esp+40h+var_1C], 6F646E69h
.text:10056959 mov     [esp+40h+var_18], 6E6F4C77h
.text:10056961 mov     [esp+40h+var_14], 5767h
.text:10056968 mov     [esp+40h+var_12], 0
.text:1005696D test    eax, eax
.text:1005696F jnz     loc_10056A88
```

Resolve LoadLibraryA by walking the PEB (code inline) and matching on the API Hash: 0x53b2070f

```
.text:10056A40
.text:10056A40 loc_10056A40:
.text:10056A40 movsx   ecx, cl
.text:10056A43 xor     ecx, edi
.text:10056A45 imul    edi, ecx, 1000193h
.text:10056A4B movzx   ecx, byte ptr [esi]
.text:10056A4E inc     esi
.text:10056A4F test    cl, cl
.text:10056A51 jnz     short loc_10056A40
```

```
.text:10056A53 cmp     edi, 53B2070Fh
.text:10056A59 mov     edi, [esp+40h+var_38]
.text:10056A5D jnz     short loc_10056A10
```

Loading the DLL Name into a stack variable

```
.text:10056A88
.text:10056A88 loc_10056A88:
.text:10056A88 lea     ecx, [esp+40h+LibFileName]
.text:10056A8C mov     dword ptr [esp+40h+LibFileName], 72657375h ; user32
.text:10056A94 mov     [esp+40h+var_28], 3233h
.text:10056A9B mov     [esp+40h+var_26], 0
.text:10056AA0 push    ecx                     ; lpLibFileName
.text:10056AA1 call    eax ; LoadLibraryA_1
.text:10056AA3 lea     ecx, [esp+40h+ProcName]
.text:10056AA7 push    ecx                     ; lpProcName
.text:10056AA8 push    eax                     ; hModule
.text:10056AA9 call    ebp ; GetProcAddress_2
.text:10056AAB mov     ecx, eax
.text:10056AAD mov     SetWindowLongW, eax
```

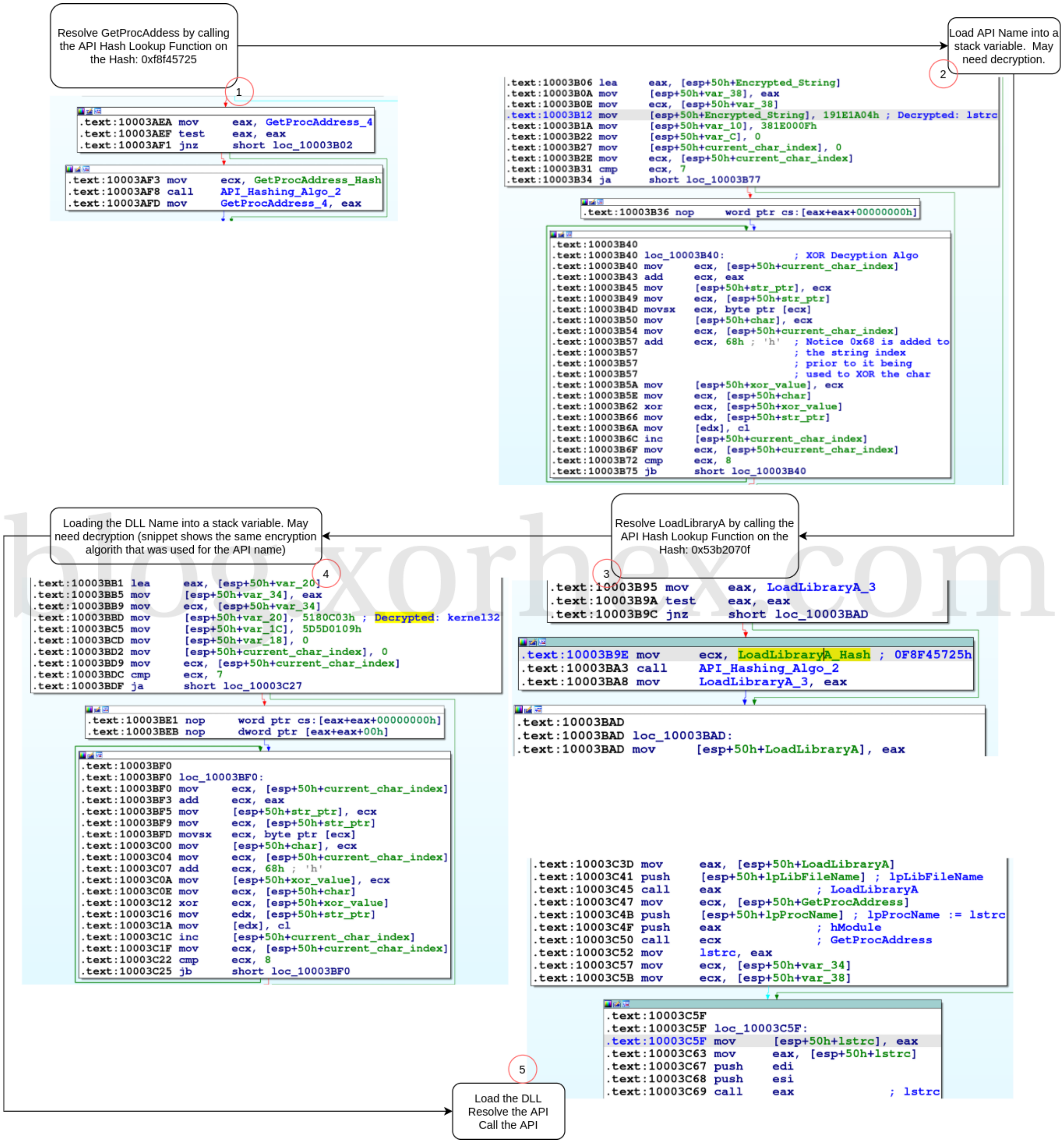
```
.text:10056AB2
.text:10056AB2 loc_10056AB2:                   ; dwNewLong
.text:10056AB2 push    ebx
.text:10056AB3 push    edi                     ; nIndex
.text:10056AB4 push    esi                     ; hWnd
.text:10056AB5 call    ecx ; SetWindowLongW
```

Load the DLL
Resolve the API
Call the API

Figure 8: Dynamic API Calling (Method 1)

Added Technique - Method 2

The second method is similar to the first but the Windows API hashing algorithm is placed into a separate function. The Windows API names may or may not be encrypted. Figure 9 shows the function's flow with the stack strings being encrypted. The unencrypted strings follow this same pattern minus the decryption loop.



① Resolve GetProcAddess by calling the API Hash Lookup Function on the Hash: 0xf8f45725

```
.text:10003AEA mov     eax, GetProcAddress_4
.text:10003AEF test    eax, eax
.text:10003AF1 jnz     short loc_10003B02
```

```
.text:10003AF3 mov     ecx, GetProcAddress_Hash
.text:10003AF8 call    API_Hashing_Algo_2
.text:10003AFD mov     GetProcAddress_4, eax
```

② Load API Name into a stack variable. May need decryption.

```
.text:10003B06 lea     eax, [esp+50h+Encrypted_String]
.text:10003B0A mov     [esp+50h+var_38], eax
.text:10003B0E mov     ecx, [esp+50h+var_38]
.text:10003B12 mov     [esp+50h+Encrypted_String], 191E1A04h ; Decrypted: lstrc
.text:10003B1A mov     [esp+50h+var_10], 381E000Fh
.text:10003B22 mov     [esp+50h+var_C], 0
.text:10003B27 mov     [esp+50h+current_char_index], 0
.text:10003B2E mov     ecx, [esp+50h+current_char_index]
.text:10003B31 cmp     ecx, 7
.text:10003B34 ja      short loc_10003B77
```

```
.text:10003B36 nop     word ptr cs:[eax+eax+00000000h]
```

```
.text:10003B40
.text:10003B40 loc_10003B40:                   ; XOR Decyption Algo
.text:10003B40 mov     ecx, [esp+50h+current_char_index]
.text:10003B43 add     ecx, eax
.text:10003B45 mov     [esp+50h+str_ptr], ecx
.text:10003B49 mov     ecx, [esp+50h+str_ptr]
.text:10003B4D movsx   ecx, byte ptr [ecx]
.text:10003B50 mov     [esp+50h+char], ecx
.text:10003B54 mov     ecx, [esp+50h+current_char_index]
.text:10003B57 add     ecx, 68h ; 'h'  ; Notice 0x68 is added to
.text:10003B57                         ; the string index
.text:10003B57                         ; prior to it being
.text:10003B57                         ; used to XOR the char
.text:10003B5A mov     [esp+50h+xor_value], ecx
.text:10003B5E mov     ecx, [esp+50h+char]
.text:10003B62 xor     ecx, [esp+50h+xor_value]
.text:10003B66 mov     edx, [esp+50h+str_ptr]
.text:10003B6A mov     [edx], cl
.text:10003B6C inc     [esp+50h+current_char_index]
.text:10003B6F mov     ecx, [esp+50h+current_char_index]
.text:10003B72 cmp     ecx, 8
.text:10003B75 jb      short loc_10003B40
```

③ Resolve LoadLibraryA by calling the API Hash Lookup Function on the Hash: 0x53b2070f

```
.text:10003B95 mov     eax, LoadLibraryA_3
.text:10003B9A test    eax, eax
.text:10003B9C jnz     short loc_10003BAD
```

```
.text:10003B9E mov     ecx, LoadLibraryA_Hash ; 0F8F45725h
.text:10003BA3 call    API_Hashing_Algo_2
.text:10003BA8 mov     LoadLibraryA_3, eax
```

```
.text:10003BAD
.text:10003BAD loc_10003BAD:
.text:10003BAD mov     [esp+50h+LoadLibraryA], eax
```

④ Loading the DLL Name into a stack variable. May need decryption (snippet shows the same encryption algorith that was used for the API name)

```
.text:10003BB1 lea     eax, [esp+50h+var_20]
.text:10003BB5 mov     [esp+50h+var_34], eax
.text:10003BB9 mov     ecx, [esp+50h+var_34]
.text:10003BBD mov     [esp+50h+var_20], 5180C03h ; Decrypted: kernel32
.text:10003BC5 mov     [esp+50h+var_1C], 5D5D0109h
.text:10003BCD mov     [esp+50h+var_18], 0
.text:10003BD2 mov     [esp+50h+current_char_index], 0
.text:10003BD9 mov     ecx, [esp+50h+current_char_index]
.text:10003BDC cmp     ecx, 7
.text:10003BDF ja      short loc_10003C27
```

```
.text:10003BE1 nop     word ptr cs:[eax+eax+00000000h]
.text:10003BEB nop     dword ptr [eax+eax+00h]
```

```
.text:10003BF0
.text:10003BF0 loc_10003BF0:
.text:10003BF0 mov     ecx, [esp+50h+current_char_index]
.text:10003BF3 add     ecx, eax
.text:10003BF5 mov     [esp+50h+str_ptr], ecx
.text:10003BF9 mov     ecx, [esp+50h+str_ptr]
.text:10003BFD movsx   ecx, byte ptr [ecx]
.text:10003C00 mov     [esp+50h+char], ecx
.text:10003C04 mov     ecx, [esp+50h+current_char_index]
.text:10003C07 add     ecx, 68h ; 'h'
.text:10003C0A mov     [esp+50h+xor_value], ecx
.text:10003C0E mov     ecx, [esp+50h+char]
.text:10003C12 xor     ecx, [esp+50h+xor_value]
.text:10003C16 mov     edx, [esp+50h+str_ptr]
.text:10003C1A mov     [edx], cl
.text:10003C1C inc     [esp+50h+current_char_index]
.text:10003C1F mov     ecx, [esp+50h+current_char_index]
.text:10003C22 cmp     ecx, 8
.text:10003C25 jb      short loc_10003BF0
```

```
.text:10003C3D mov     eax, [esp+50h+LoadLibraryA]
.text:10003C41 push    [esp+50h+lpLibFileName] ; lpLibFileName
.text:10003C45 call    eax                ; LoadLibraryA
.text:10003C47 mov     ecx, [esp+50h+GetProcAddress]
.text:10003C4B push    [esp+50h+lpProcName] ; lpProcName := lstrc
.text:10003C4F push    eax                ; hModule
.text:10003C50 call    ecx                ; GetProcAddress
.text:10003C52 mov     lstrc, eax
.text:10003C57 mov     ecx, [esp+50h+var_34]
.text:10003C5B mov     ecx, [esp+50h+var_38]
```

```
.text:10003C5F
.text:10003C5F loc_10003C5F:
.text:10003C5F mov     [esp+50h+lstrc], eax
.text:10003C63 mov     eax, [esp+50h+lstrc]
.text:10003C67 push    edi
.text:10003C68 push    esi
.text:10003C69 call    eax                ; lstrc
```

⑤ Load the DLL
Resolve the API
Call the API

Figure 9: Dynamic API Calling (Method 2)

The next difference noted is around string obfuscation.

String Obfuscation

The older samples primarily make use of stack strings to hide from tools like strings.exe. This new sample uses a mixture of stack strings with and without XOR encryption. The index of the character being decrypted makes up one part of the XOR key for that letter. The second part is a constant they add to it to get the final XOR key.

The string decryption function (for example, Figure 9) can be represented in Python as:

```
def str_decrypt(value: [bytes], xor_key_modified_by: int) -> str:
  plain_text = []
  for idx, val in enumerate(value):
    plain_text.append(chr(val^(idx+xor_key_modified_by)))
  return ''.join(plain_text)
```

To call it, pass an array of bytes and the constant to add to the XOR key. For example:

```
>>> str_decrypt([0x03, 0x0c, 0x18, 0x05, 0x09, 0x01, 0x5d, 0x5d], 0x68)
'kernel32'
```

They don't always modify the XOR key by `0x68`; sometimes they use other values like `0x2c`.

Mustang Panda and RedDelta Infrastructure Overlap

This PlugX's config contains two previously seen Mustang Panda command and control servers;

- 101.36.125.203
- vitedannews.com

Infrastructure Pivot

Content Loading..

Click a Node to Load Details Below

There are 7 samples in our repository that share the IP, 101.36.125.203, and one other sample that shares the domain, vitedannews.com. All of these samples contain the `XXXXXXXX` config value check making them the Mustang Panda variant. This RedDelta variant (ec1c29cb6674ffce989576c51413a6f9cbb4a8a41cbd30ec628182485a937160) makes the second instance where the IP/Domains overlap with the "original" Mustang Panda PlugX variant. More about the first instance can be found on ThreatConnect's blog. This second infrastructure overlap further strenghtens our theory of them being the same group or at least sharing personnel/infrastructure.

Conclusion

Over all we believe Mustang Panda will continue evolving the RedDelta variant to help further thwart detection as time goes on. Historically the .dat file (the encrypted PlugX file) is loaded using a sideloaded dll which does the loading, decrypting, and passing execution on to this PlugX binary. These three files are sometimes packaged using a self extracting SFX file. We can't be certain that the updated variant was delivered in the same fashion, but that would be something to look for.

Feedback welcomed via Twitter.

Appendix

API Hashing

Spotting the Hashing Routine Inside Control Flow Obfucation

Taking our knowledge of API hashing algorithms (most, if not all, API hashing routines loop through each character of the API name and apply the hashing algorithm to it) we can find the only part of the algorithm we really care about, the hashing routine. The GIF starts from a zoomed out position to identify a loop for inspection before zooming in on the hashing algorithm loop.

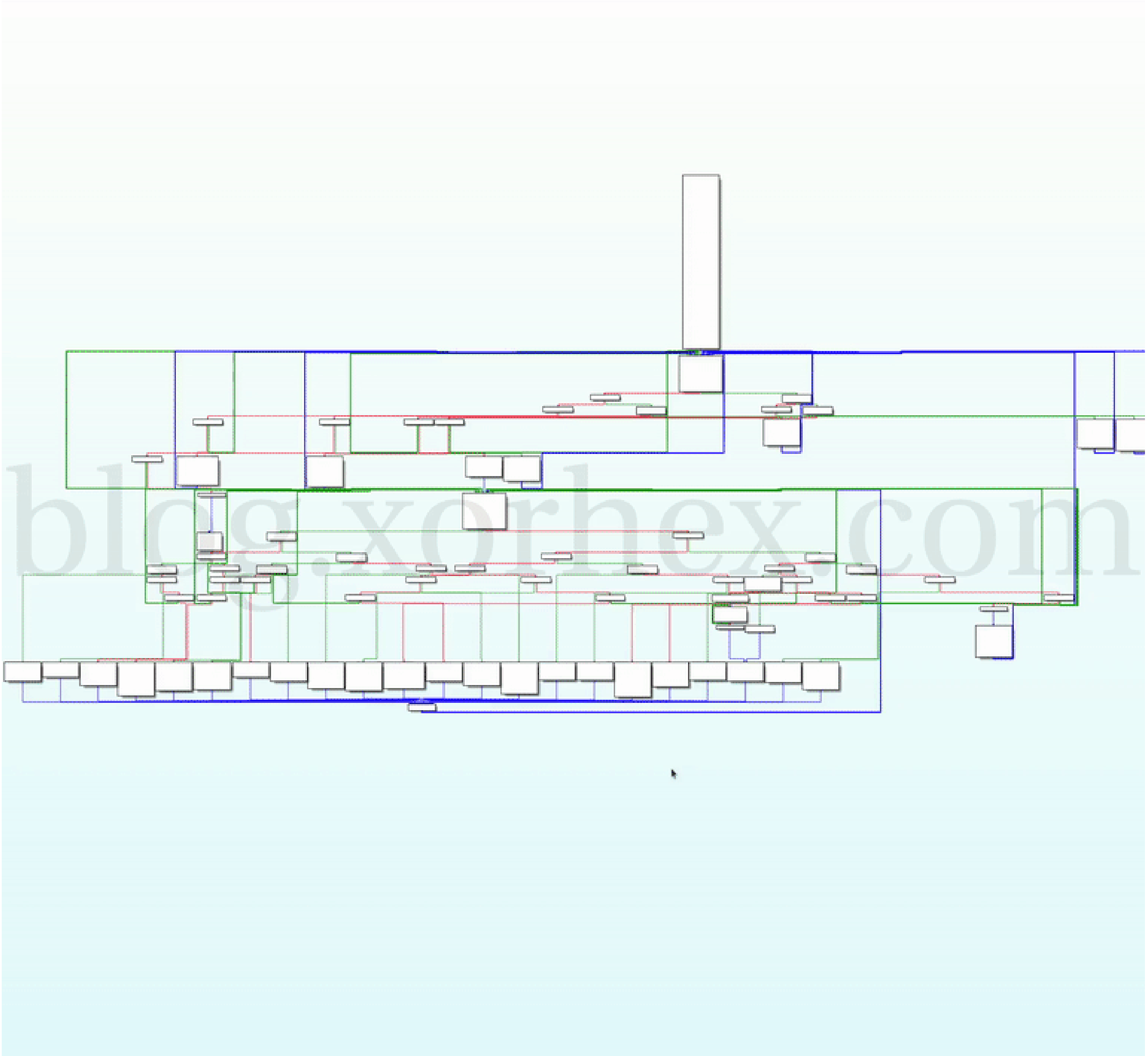


Figure 10: Control Flow Obfuscation - API Hashing Loop

Rewriting the Hashing Algorithm in Python

The API hashing algorithm can be re-written in Python as:

```
def hasher(name: str):
  ebp = 0x811c9dc5
  for dl in name:
    edx = ord(dl) ^ ebp
    ebp = (edx * 0x1000193) & 0xffffffff
  print(hex(ebp))
```

- LoadLibraryA == 0x53b2070f
- GetProcAddress == 0xf8f45725

IOCs

- 1c7897a902b35570a9620c64a2926cd5d594d4ff5a033e28a400981d14516600
- ec1c29cb6674ffce989576c51413a6f9cbb4a8a41cbd30ec628182485a937160
- 101.36.125.203
- vitedannews.com
- dba437c9030b5f857ce9820a0c9e2c252fd8aeda71c2101024d3576c446972a0
- a1eb4ce6eaa0c35ca4e8285c32b59cd0dfb34018b3f454d4fa4cebe9906534d8
- 2304891f176a92c62f43d9fd30cae943f1521394dce792c6de0e097d10103d45
- 2f58a869711d2b28e6ecaac25cc2166daa46f7adfb719b7dd334e01c1474ca9b
- 2bfd100498f70938dedef42116af09af2db77ef1315edcea0ffd62c93015ddf5
- b87d1c01daee804c7330d5ac6273e5dcba886e1663c929709c158fd45b11a7ba
- 4e30cfa4f3d3bd6192818c5619eb7f6a26a408ae9fd62a7629059f47466f757b
- 2531af12360e29b73b545210e1cbdfc2459c95e2827d3246e9d6933820a808dd
- 4b1dbb3fc4adba3a83a563e5e86afb56136a1f9ba0293ad21a00e031b88b2ad9
- f631e8f0c723cccbc5b26387f4100351de2e158b6770e962733734be6ca119d5
- 76f44175f88984367ad62c81d1dcc947b1a26d6832fd33569d2c21113c1ddee2