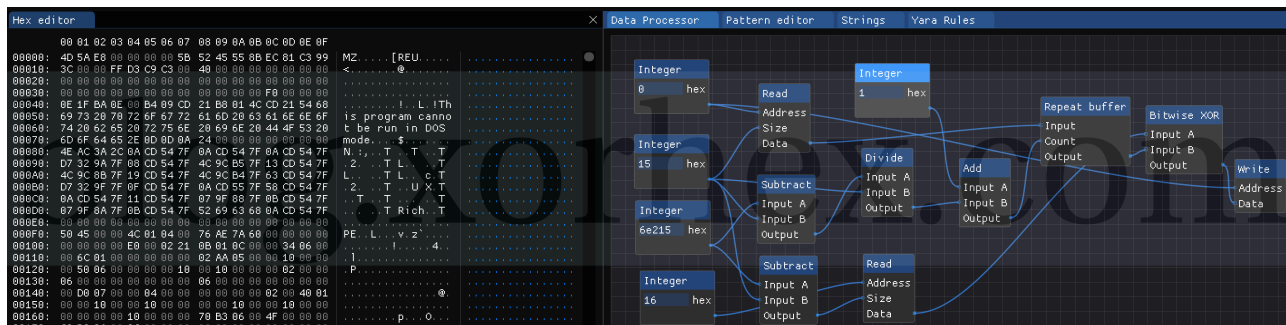


RedDelta PlugX Undergoing Changes and Overlapping Again with Mustang Panda PlugX Infrastructure

Published: 2021-06-02 · Archived: 2026-04-05 22:20:21 UTC



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 infrastructure 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 .

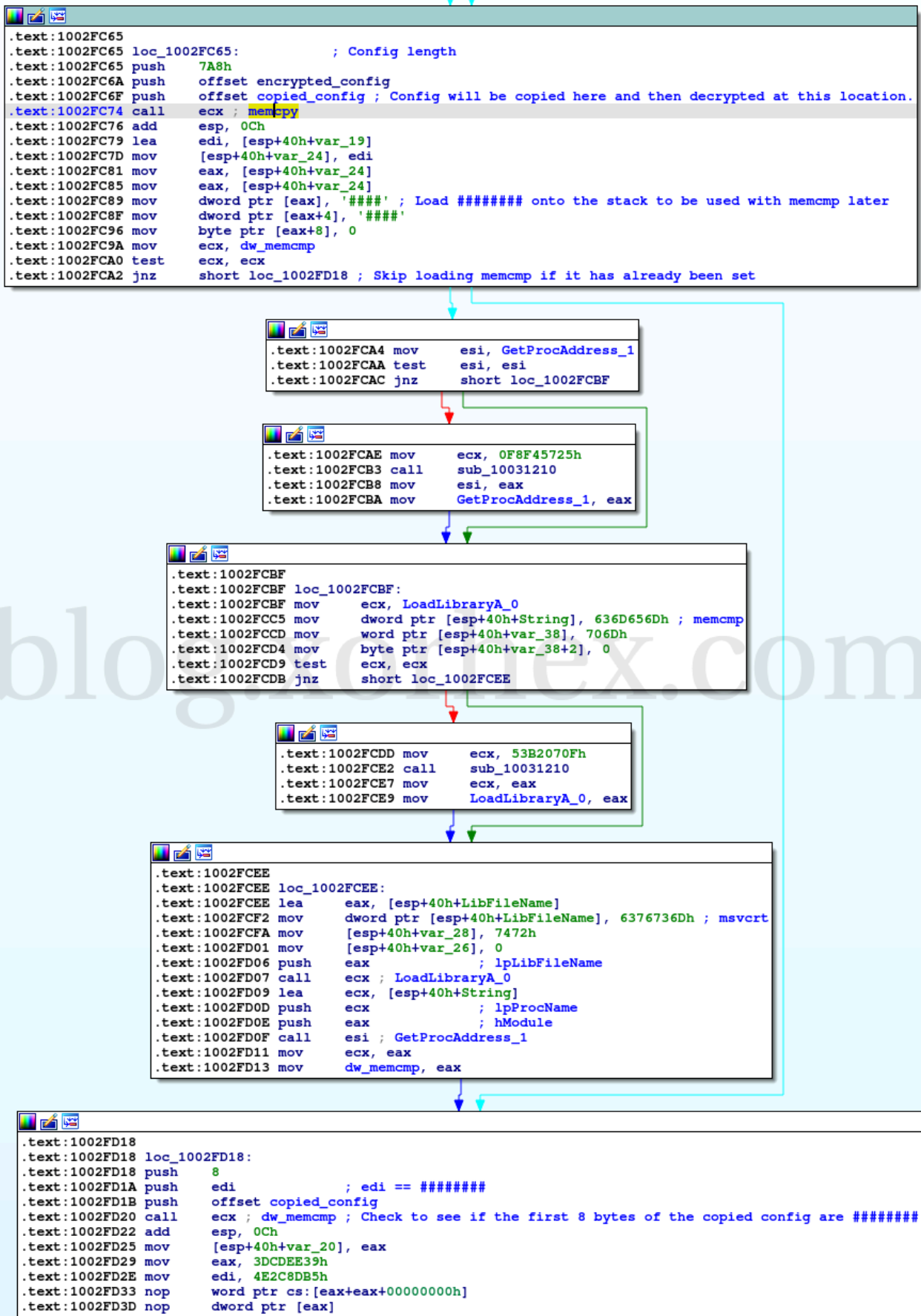


Figure 1: RedDelta Variant Config Check (ec1c29cb6674ffce989576c51413a6f9cbb4a8a41cbd30ec628182485a937160)

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 encrypted with a 21 byte XOR key prepended to the binary.

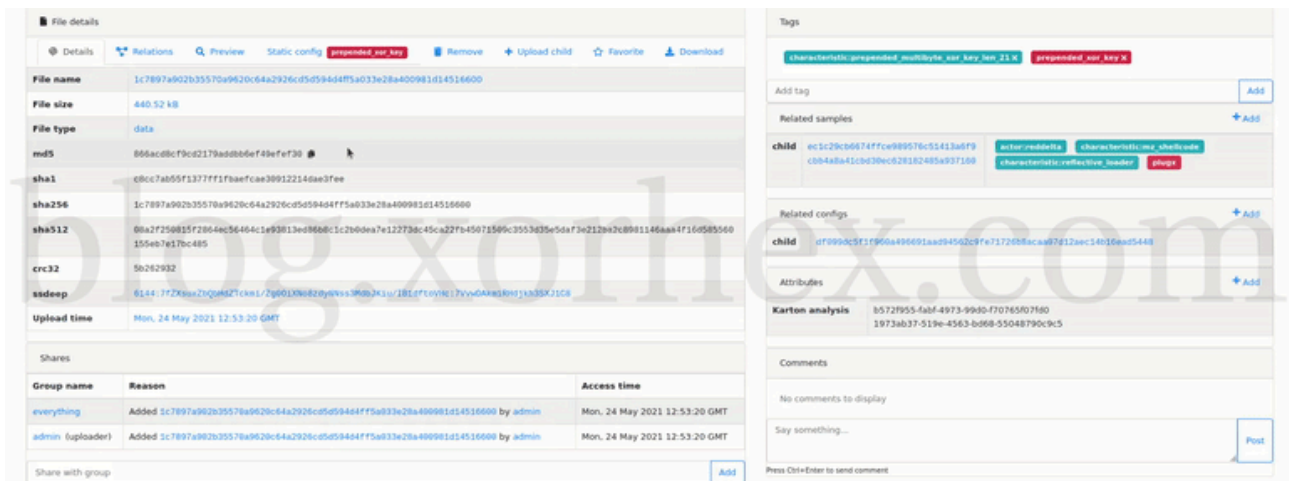


Figure 2: MWDB

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	a1eb4ce6aa0c35ca4e8285c32b59cd0dfb34018b3f454d4fa4cbe9906534d8
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.

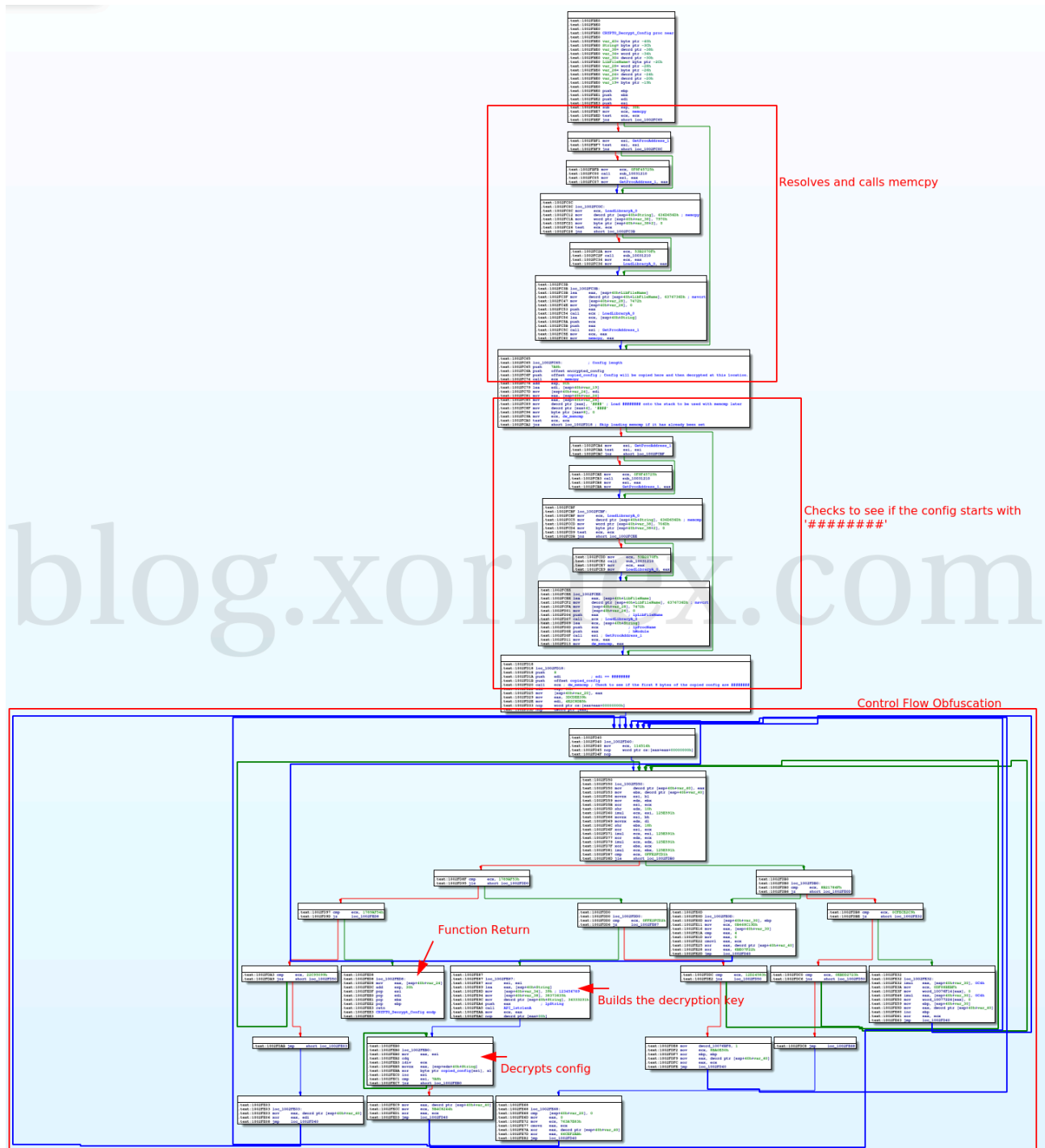


Figure 3:1c7897a902b35570a9620c64a2926cd5d594d4ff5a033e28a400981d14516600

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_1000FD10
add     esp, 10h
```

Builds the decryption key

Decrypts config

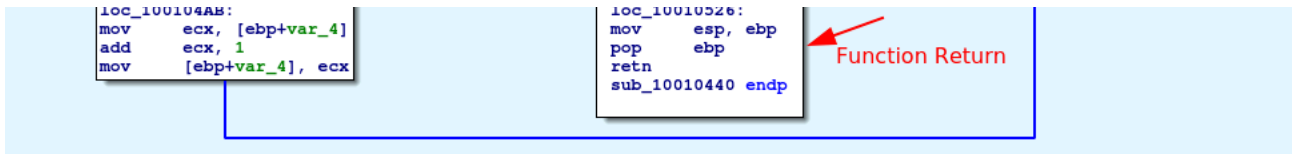


Figure 4: dba437c9030b5f857ce9820a0c9e2c252fd8aeda71c2101024d3576c446972a0

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).

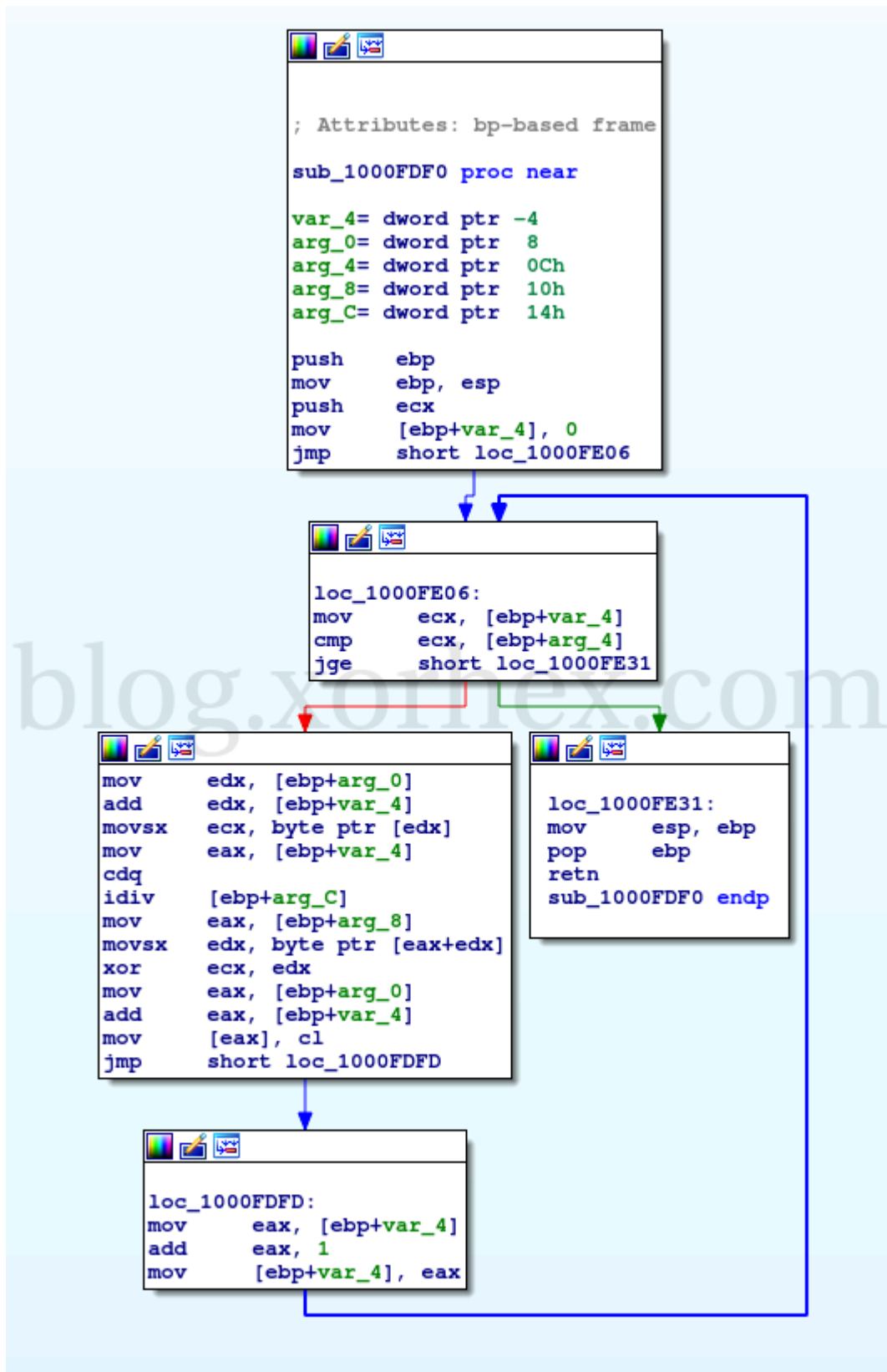


Figure 5: XOR Routine - dba437c9030b5f857ce9820a0c9e2c252fd8aeda71c2101024d3576c446972a0

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

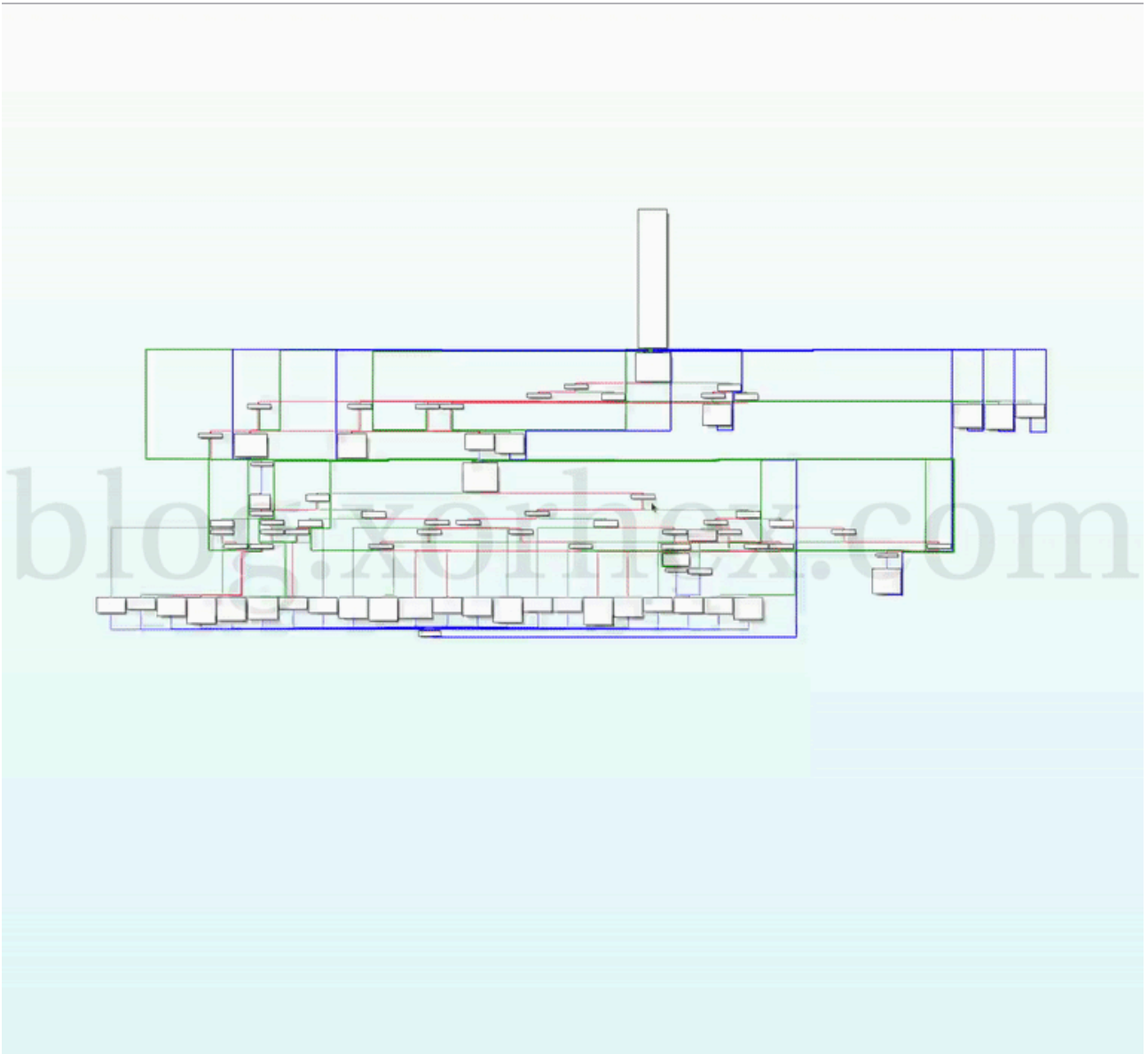


Figure 6: Control Flow Obfuscation - API Hashing Function

Notice the multiple comparison statements controlling which branches are taken (Figure 6), making it harder to follow the execution 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 modifying 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 consistently 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 Technique - 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.

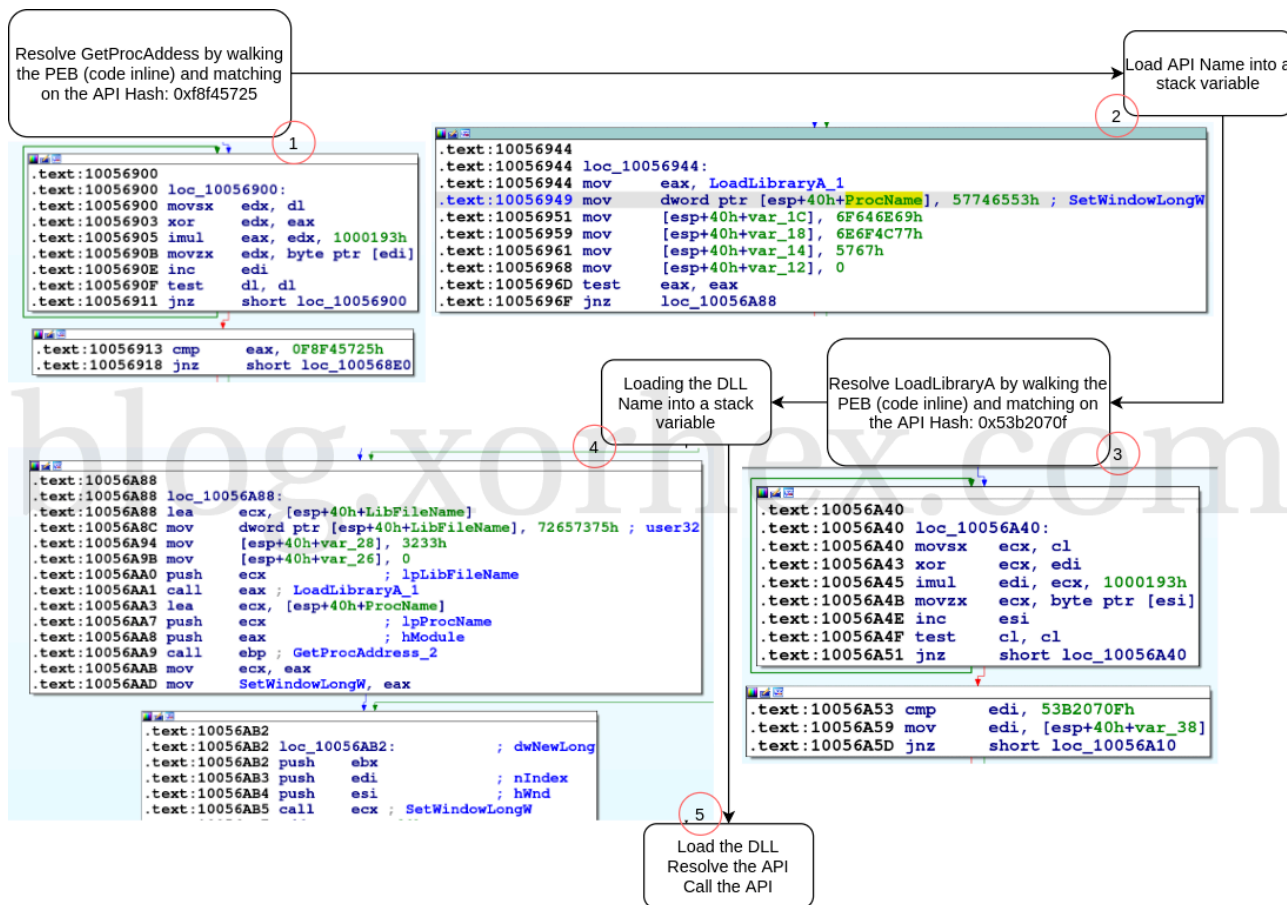


Figure 8: Dynamic API Calling (Method 1)

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.

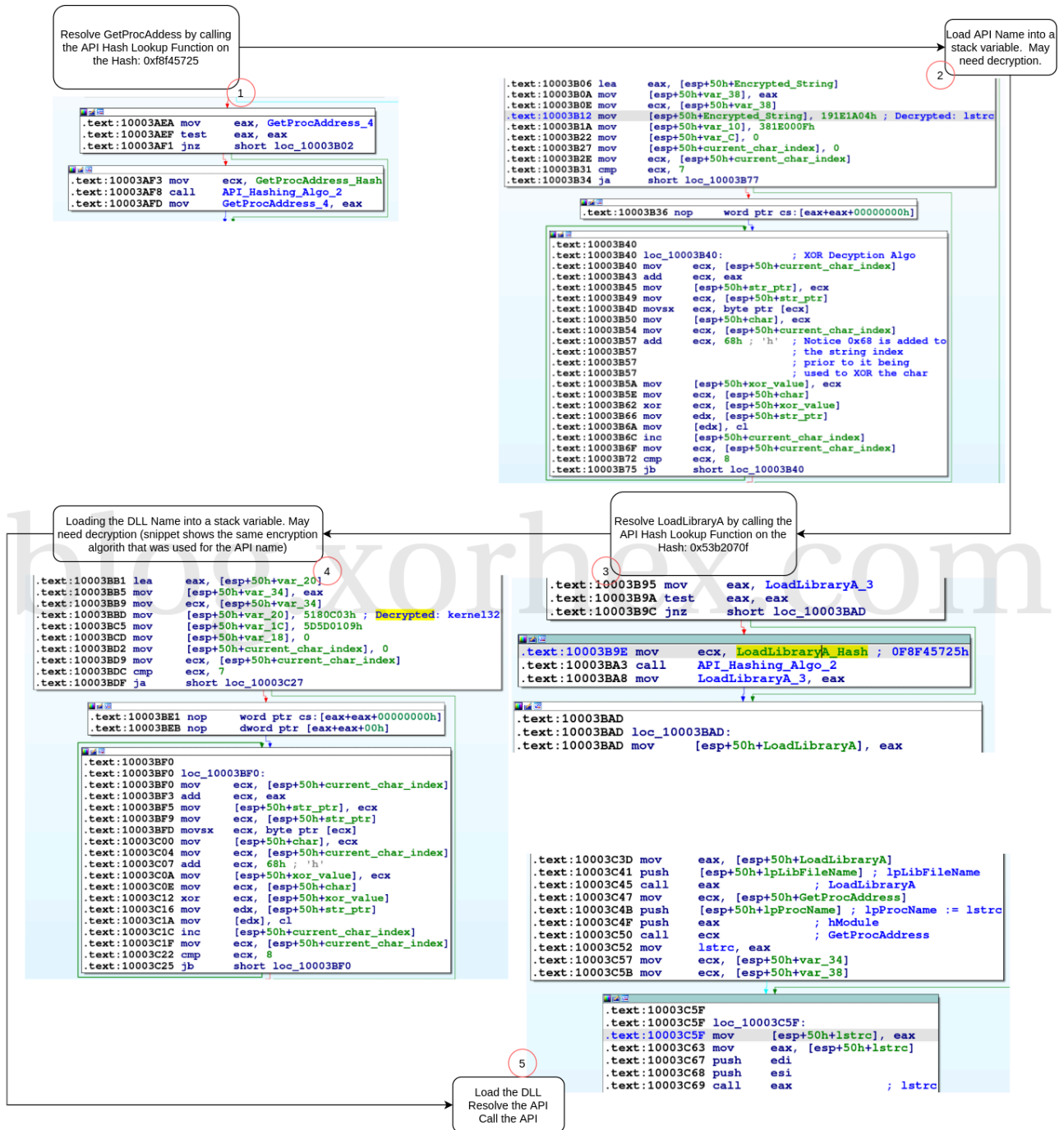


Figure 9: Dynamic API Calling (Method 2)

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

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 Obfuscation

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.

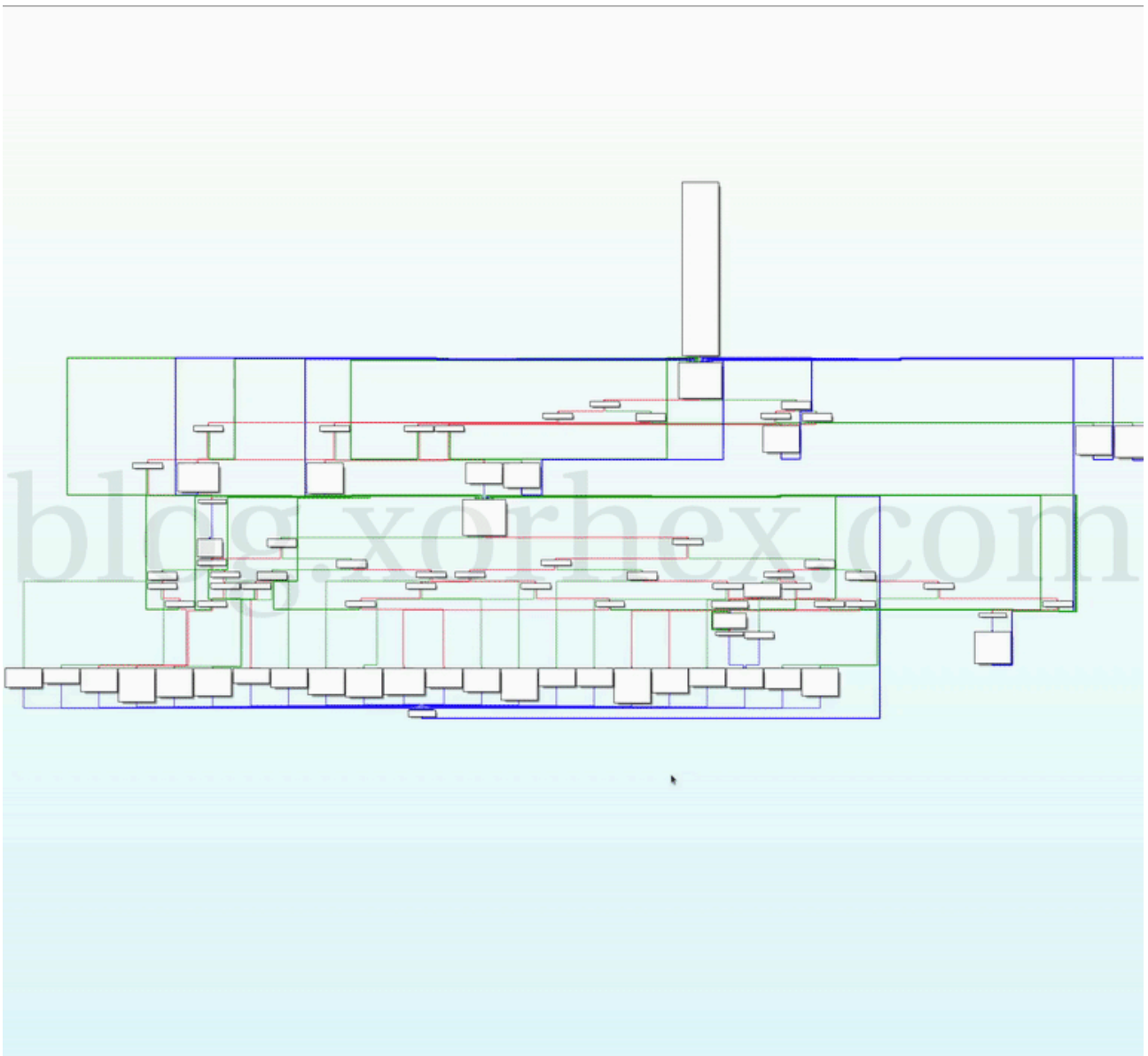


Figure 10: Control Flow Obfuscation - API Hashing Loop

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
- f631e8f0c723ccbc5b26387f4100351de2e158b6770e962733734be6ca119d5
- 76f44175f88984367ad62c81d1dcc947b1a26d6832fd33569d2c21113c1ddee2

Source: <https://blog.xorhex.com/blog/reddeltaplugxchangeup/>