TA505 Get2 Analysis

goggleheadedhacker.com/blog/post/13

Jacob Pimental November 24, 2019

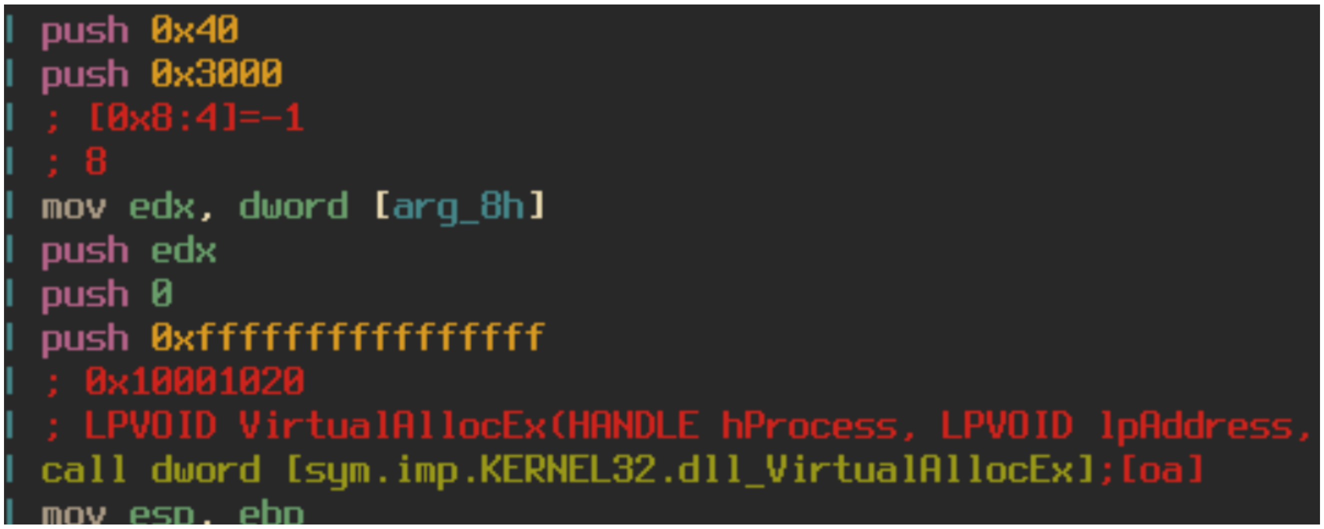
24 November 2019

The TA505 group debuted Get2 and SDBot last month in a new phishing campaign. While there have been some great analyses on the SDBot RAT that is dropped, there have not been many on the Get2 downloader. I wanted to take this opportunity to do my own analysis on it. I will not be going over the macro-enabled word document itself, just the DLL that is dropped. There are also two versions of the dll, x86 and x64. This analysis will focus on the x86 version. If you want to follow along you can get the sample from Hybrid Analysis here.

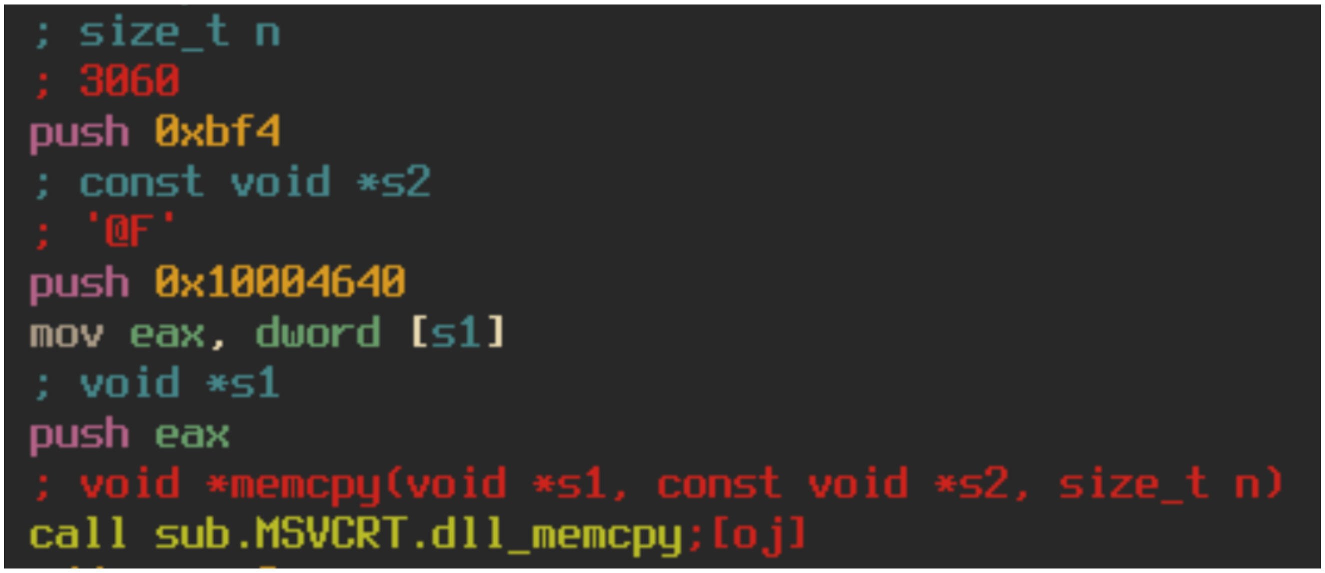
Obfuscation

The Get2 DLL that comes from the malicious word document is pretty heavily obfuscated and packed using a custom packing mechanism. The code contains multiple loops in unnecessary places in order to distract reverse engineers from the actual functionality. It also contains calls to multiple bogus functions that don't return anything of importance. These bogus functions tend to contain multiple loops and calls to more bogus functions. It is very easy to go down a rabbit hole while analyzing this binary.

On top of having fake instructions, the DLL also contains self-modifying code. The DLL calls VirtualAlloc to allocate 3060 bytes of memory. It then dumps data into the newly created space using memcpy.

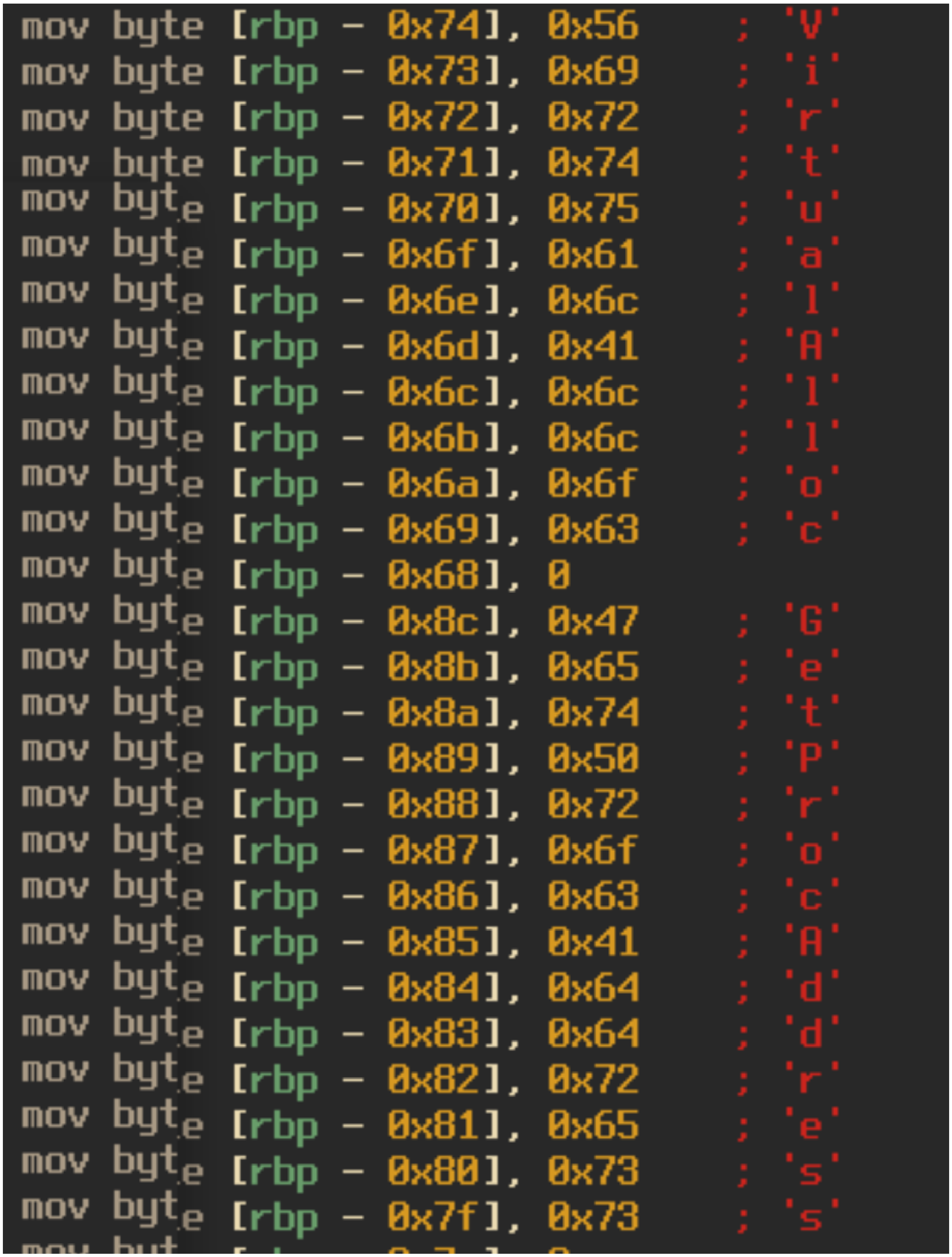


```
| push 0x40
| push 0x3000
| ; [0x8:4]=-1
| ; 8
| mov edx, dword [arg_8h]
| push edx
| push 0
| push 0xffffffffffffffff
| ; 0x10001020
| ; LPVOID VirtualAllocEx(HANDLE hProcess, LPVOID lpAddress,
| call dword [sym.imp.KERNEL32.dll_VirtualAllocEx];[oa]
| mov esp, ebp
```



```
; size_t n
; 3060
push 0xbf4
; const void *s2
; '@F'
push 0x10004640
mov eax, dword [s1]
; void *s1
push eax
; void *memcpy(void *s1, const void *s2, size_t n)
call sub.MSVCRT.dll_memcpy;[oj]
```

It will then decrypt this data by taking every dword, xoring it by 0x6949, rotating the bits left by 4 and adding 0x77777778. I have created a small python script to emulate this functionality using r2pipe, radare2's API. You can find that here. After the decryption occurs, we can see new code formed in memory that is executed. This code appears to be importing functions such as VirtualAlloc, GetProcAddress, VirtualProtect, LoadLibraryA, and VirtualFree. Which leads us to believe that more unpacking is necessary.

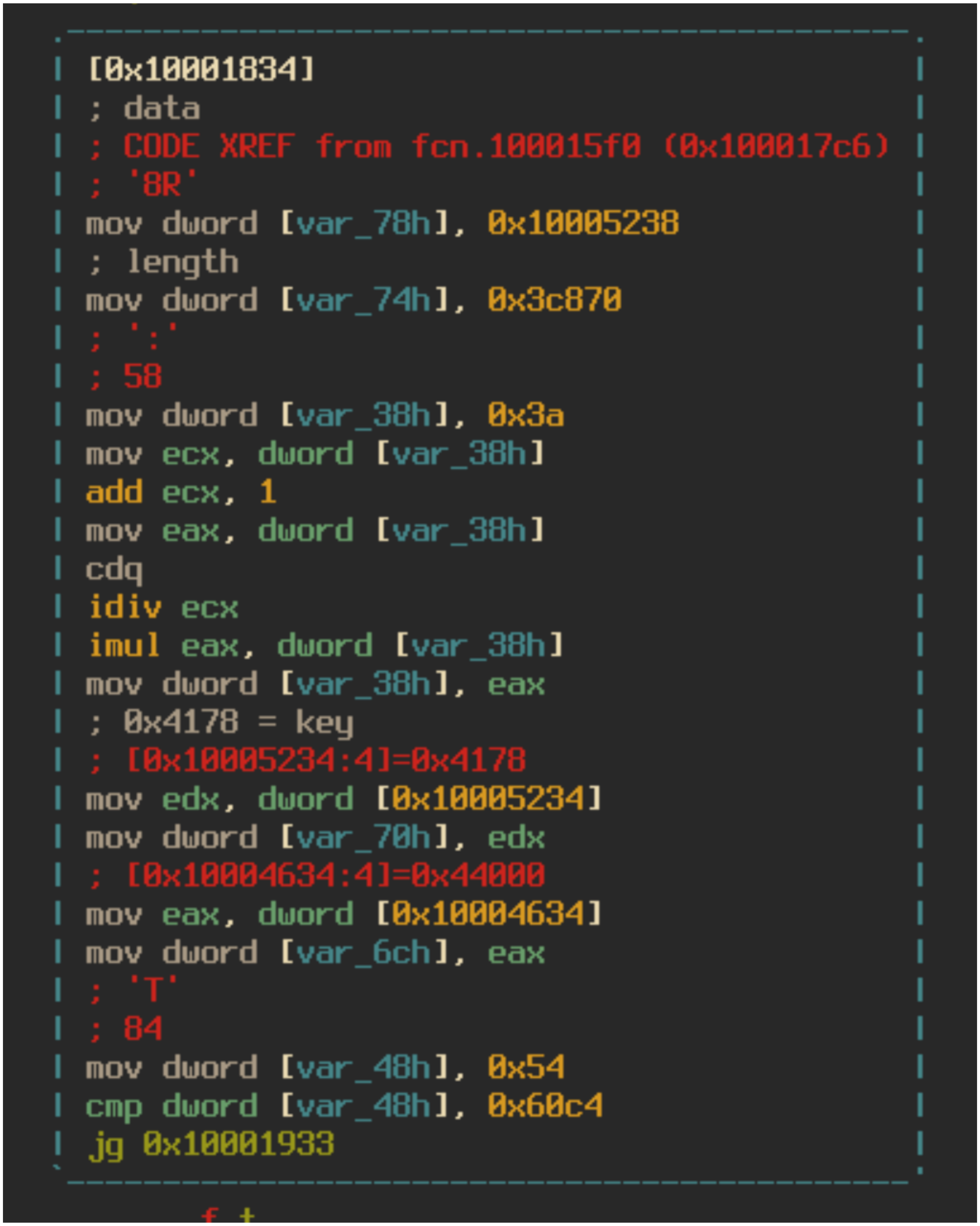


```
mov byte [rbp - 0x74], 0x56      ; 'V'
mov byte [rbp - 0x73], 0x69      ; 'i'
mov byte [rbp - 0x72], 0x72      ; 'r'
mov byte [rbp - 0x71], 0x74      ; 't'
mov byte [rbp - 0x70], 0x75      ; 'u'
mov byte [rbp - 0x6f], 0x61      ; 'a'
mov byte [rbp - 0x6e], 0x6c      ; 'l'
mov byte [rbp - 0x6d], 0x41      ; 'A'
mov byte [rbp - 0x6c], 0x6c      ; 'l'
mov byte [rbp - 0x6b], 0x6c      ; 'l'
mov byte [rbp - 0x6a], 0x6f      ; 'o'
mov byte [rbp - 0x69], 0x63      ; 'c'
mov byte [rbp - 0x68], 0
mov byte [rbp - 0x8c], 0x47      ; 'G'
mov byte [rbp - 0x8b], 0x65      ; 'e'
mov byte [rbp - 0x8a], 0x74      ; 't'
mov byte [rbp - 0x89], 0x50      ; 'P'
mov byte [rbp - 0x88], 0x72      ; 'r'
mov byte [rbp - 0x87], 0x6f      ; 'o'
mov byte [rbp - 0x86], 0x63      ; 'c'
mov byte [rbp - 0x85], 0x41      ; 'A'
mov byte [rbp - 0x84], 0x64      ; 'd'
mov byte [rbp - 0x83], 0x64      ; 'd'
mov byte [rbp - 0x82], 0x72      ; 'r'
mov byte [rbp - 0x81], 0x65      ; 'e'
mov byte [rbp - 0x80], 0x73      ; 's'
mov byte [rbp - 0x7f], 0x73      ; 's'
```

Unpacking

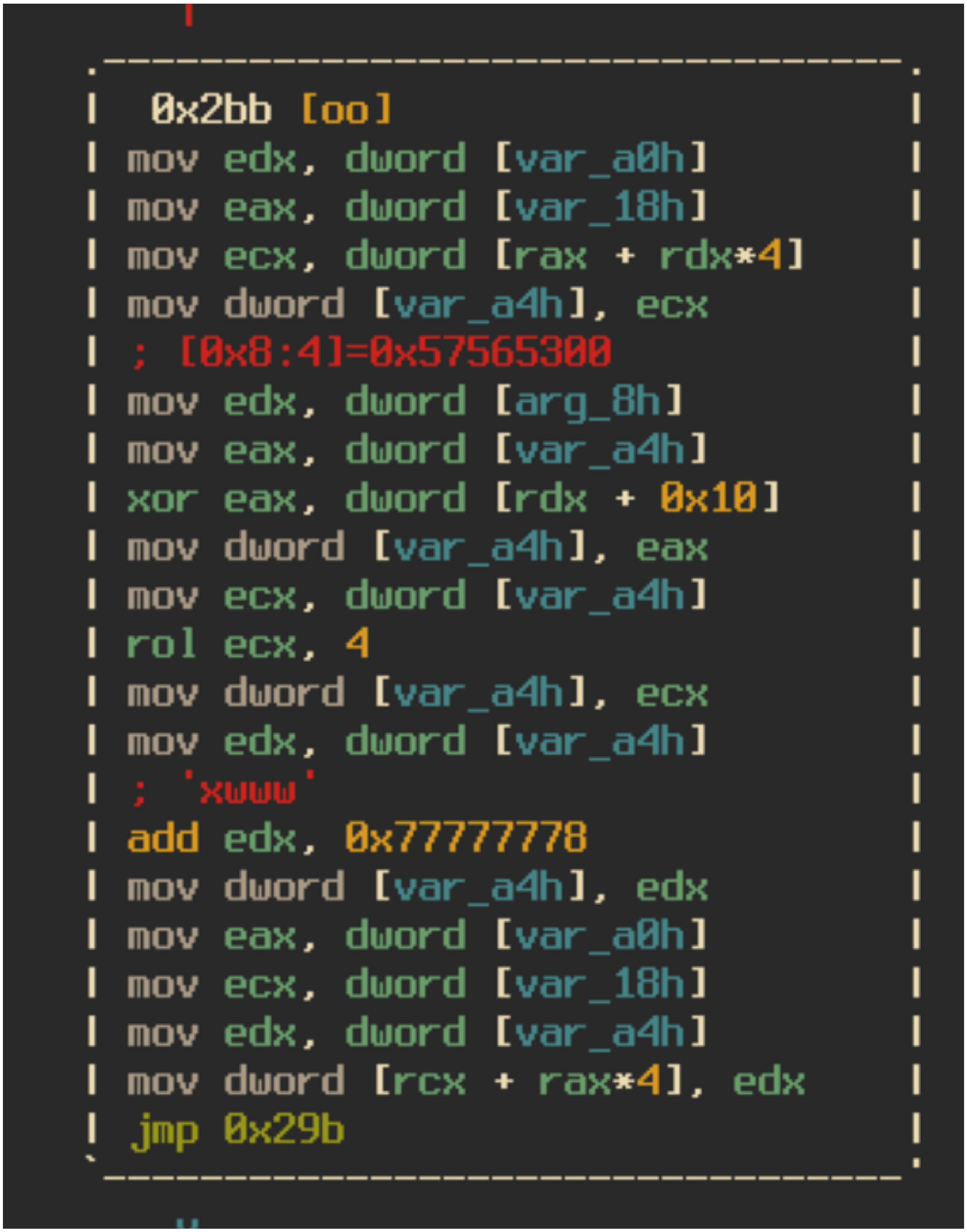
We can see that earlier there was data moved into the memory space at `[ebp – 78]`, `[ebp – 74]`, and `[ebp – 70]` which is referenced in this new set of code based on the offset of the argument, `[ebp + 8]`. The data moved into `[ebp – 78]` contains a list of bytes that will be decrypted. `[ebp – 74]` contains 0x3c870 which is the length of the data to be decrypted, and `[ebp – 70]` contains 0x4178, the decryption key. This round of decryption is slightly different from the first. The first thing that the malware does is loop through each index of the encrypted data and if the index is divisible by two then it will skip two bytes and move the data at that index into a buffer. For clarity, this follows the pattern [2, 3, 6, 7, 10, 11, 14…] and can be represented as the following python code:

```
compressed_data = b''
x = 0
while x < len(data):
    if x % 2 == 0:
        x += 2
    compressed_data += bytes([data[x]])
    x += 1
```



```
[0x10001834]
; data
; CODE XREF from fcn.100015f0 (0x100017c6)
; '8R'
mov dword [var_78h], 0x10005238
; length
mov dword [var_74h], 0x3c870
; ':'
; 58
mov dword [var_38h], 0x3a
mov ecx, dword [var_38h]
add ecx, 1
mov eax, dword [var_38h]
cdq
idiv ecx
imul eax, dword [var_38h]
mov dword [var_38h], eax
; 0x4178 = key
; [0x10005234:4]=0x4178
mov edx, dword [0x10005234]
mov dword [var_70h], edx
; [0x10004634:4]=0x44000
mov eax, dword [0x10004634]
mov dword [var_6ch], eax
; 'T'
; 84
mov dword [var_48h], 0x54
cmp dword [var_48h], 0x60c4
jg 0x10001933
```

Once the data is moved into the new buffer the malware will follow the normal decryption process we saw earlier by xoring each dword and rotating the bits left by 4, this time using the key 0x4178.



```
0x2bb [oo]
mov edx, dword [var_a0h]
mov eax, dword [var_18h]
mov ecx, dword [rax + rdx*4]
mov dword [var_a4h], ecx
; [0x8:4]=0x57565300
mov edx, dword [arg_8h]
mov eax, dword [var_a4h]
xor eax, dword [rdx + 0x10]
mov dword [var_a4h], eax
mov ecx, dword [var_a4h]
rol ecx, 4
mov dword [var_a4h], ecx
mov edx, dword [var_a4h]
; 'xwww'
add edx, 0x77777778
mov dword [var_a4h], edx
mov eax, dword [var_a0h]
mov ecx, dword [var_18h]
mov edx, dword [var_a4h]
mov dword [rcx + rax*4], edx
jmp 0x29b
```

The resulting data looks a like a mangled PE header, which leads us to believe that there is more deobfuscation necessary. The final stage of this process is quite complicated. It involves multiple ways of moving bytes from our newly decrypted data to an empty buffer, but no actual decryption occurs. The paths that the deobfuscation algorithm could take are:

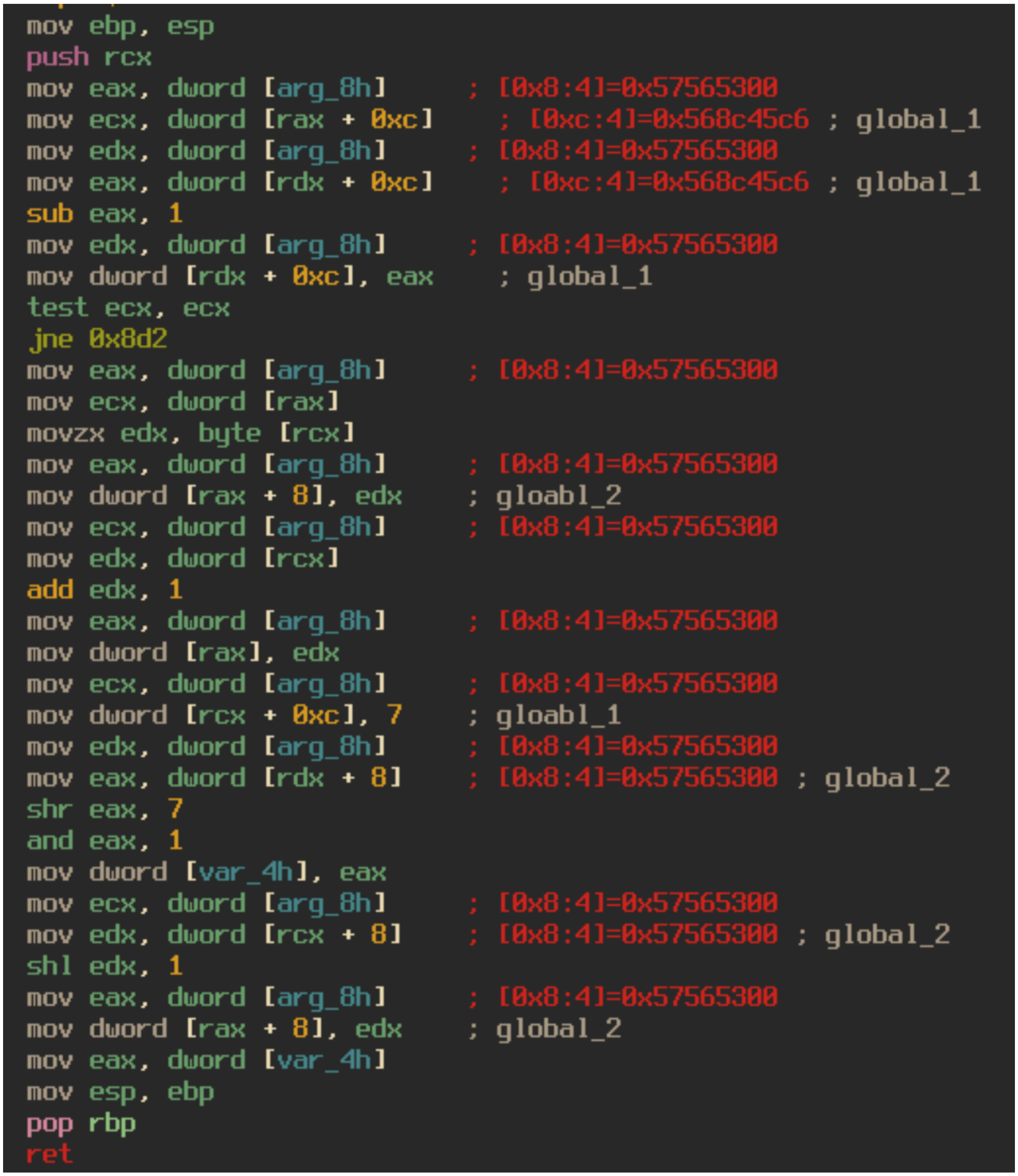
- To simply copy one byte from the current index of the decrypted data into the empty buffer
- To copy previous data that was inserted into the empty buffer into the current index of the buffer
- To move the byte 00 into the current index of the buffer x number of times

These paths were all dependent on a "check" function that would take into account the current index of the encrypted data, and two global variables. The check function works as follows:

- Check if global_1 is 0
- If So:
    - Move the current byte of the encrypted data into global_2
    - Increment the current index of the encrypted data by 1
    - Move 7 into global_1
- Else:

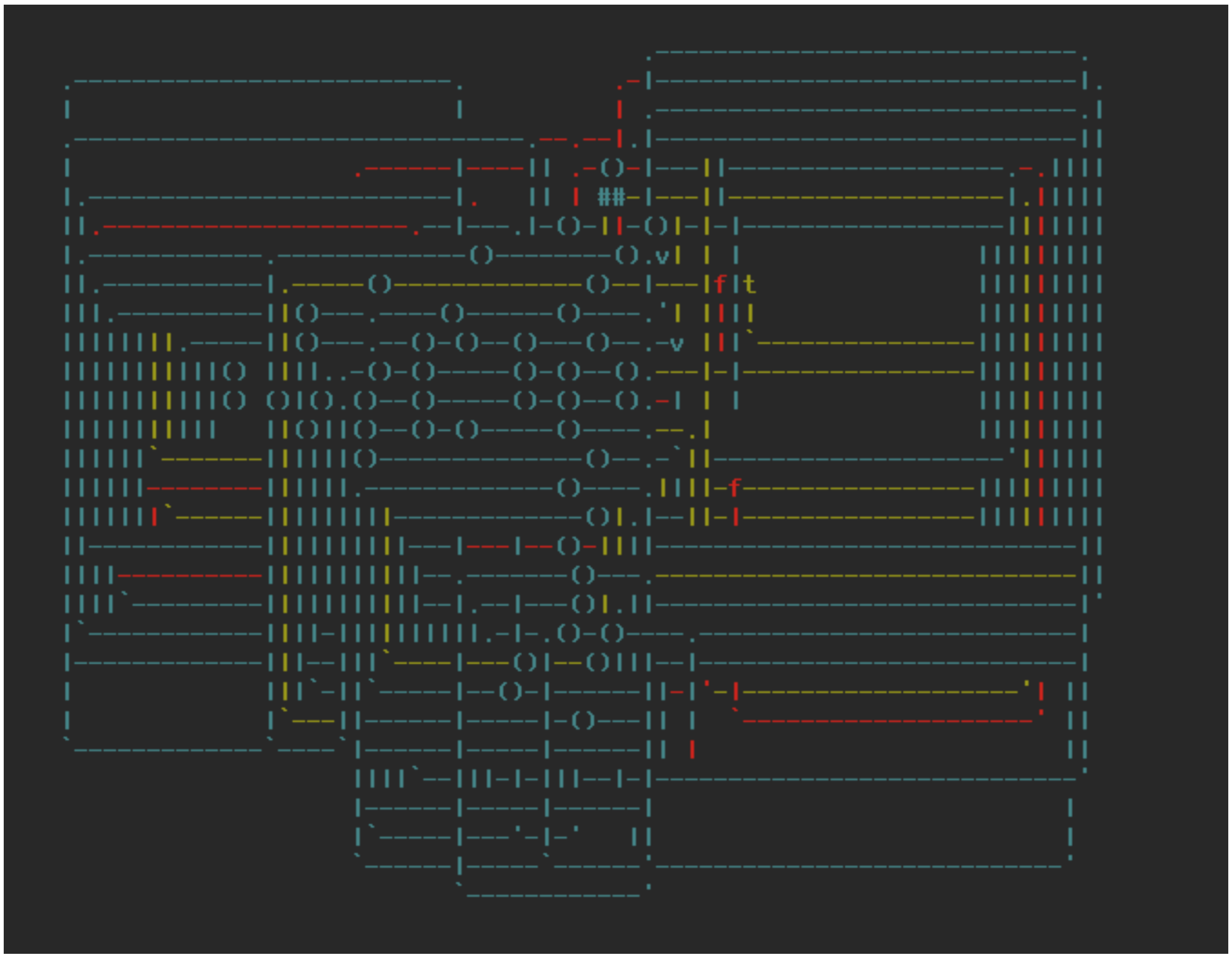
    global_1 = global_1 – 1
- Move the product of a bitwise shift right of global_2 by 7, anded by 1 ((gloabl_2 » 7) & 1) into the return register
- global_2 = global_2 shifted left by 1
- return

I have made a python script that mimics the unpacking functionality and writes out the final payload to a file. You can find that here.



```
mov ebp, esp
push rcx
mov eax, dword [arg_8h]      ; [0x8:4]=0x57565300
mov ecx, dword [rax + 0xc]    ; [0xc:4]=0x568c45c6 ; global_1
mov edx, dword [arg_8h]      ; [0x8:4]=0x57565300
mov eax, dword [rdx + 0xc]    ; [0xc:4]=0x568c45c6 ; global_1
sub eax, 1
mov edx, dword [arg_8h]      ; [0x8:4]=0x57565300
mov dword [rdx + 0xc], eax    ; global_1
test ecx, ecx
jne 0x8d2
mov eax, dword [arg_8h]      ; [0x8:4]=0x57565300
mov ecx, dword [rax]
movzx edx, byte [rcx]
mov eax, dword [arg_8h]      ; [0x8:4]=0x57565300
mov dword [rax + 8], edx     ; gloabl_2
mov ecx, dword [arg_8h]      ; [0x8:4]=0x57565300
mov edx, dword [rcx]
add edx, 1
mov eax, dword [arg_8h]      ; [0x8:4]=0x57565300
mov dword [rax], edx
mov ecx, dword [arg_8h]      ; [0x8:4]=0x57565300
mov dword [rcx + 0xc], 7     ; gloabl_1
mov edx, dword [arg_8h]      ; [0x8:4]=0x57565300
mov eax, dword [rdx + 8]     ; [0x8:4]=0x57565300 ; global_2
shr eax, 7
and eax, 1
mov dword [var_4h], eax
mov ecx, dword [arg_8h]      ; [0x8:4]=0x57565300
mov edx, dword [rcx + 8]     ; [0x8:4]=0x57565300 ; global_2
shl edx, 1
mov eax, dword [arg_8h]      ; [0x8:4]=0x57565300
mov dword [rax + 8], edx     ; global_2
mov eax, dword [var_4h]
mov esp, ebp
pop rbp
ret
```

Here is a tinygraph view of the final deobfuscation function which shows just how complicated the algorithm is:



Extracted Payload

The extracted binary is UPX packed. We can simply unpack it using the command `upx -d <packed_binary>`. Looking at the exports of the newly unpacked binary we can see the function getandgodll_Win32.dll_IKAJSL. This is most likely where execution will continue.

This exported function seems to call one function then exit. It is safe to assume that the called function will be the main function for this binary. This main function will grab the UserName of the user the malware is running as, the name of the PC, the version of Windows the malware is running on, and a list of the currently running processes. It then concatenates this data into the string:

`“&D=<ComputerName>&U=<UserName>&OS=<WindowsVersion>&PR=<Process list>”`

It will then send a POST request with these parameters to the C2 using the WinHttp library, with the useragent:

`Mozilla/5.0 (Windows NT 6.1) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/41.0.2228.0 Safari/537.36`

The response will send back a list of URLs that contain the final payload (normally SDBot). The Get2 downloader will download these payloads and run them on the victim’s machine.

Conclusions

Overall this was a very interesting and fun sample to analyze. The code was complicated and seemed sophisticated, which is what you would expect from this threat actor. I am hoping this article helps others with analyzing this particular downloader. I am always open to feedback, so feel free to send me messages on my Twitter or LinkedIn letting me know what I can improve on in these articles.

Thanks for reading and happy reversing!

Radare2, Malware Analysis, Malware Windows, Scripting, Automation, r2pipe, unpacking

More Content Like This: