

# Attacking Emotet's Control Flow Flattening

[news.sophos.com/en-us/2022/05/04/attacking-emotets-control-flow-flattening/](https://news.sophos.com/en-us/2022/05/04/attacking-emotets-control-flow-flattening/)

Andreas Klopsch

May 4, 2022



Emotet has been one of the most professional and long-lasting cybercrime services and malware infections in the threat landscape. Notorious since shortly after its debut in 2014, the botnet was disrupted in January 2021 by a multinational law enforcement effort that sidelined its activity for almost a year. Unfortunately, in November 2021 the botnet re-emerged and once again began to appear on Sophos' radar.

To protect our customers, SophosLabs is always looking for the most significant techniques, tactics, and procedures used to distribute and deliver Emotet. In this post, we'll look at Control Flow Flattening (CFF), one of several obfuscation tactics Emotet's developers use to make detection and reverse engineering of the malware's payload more difficult. We'll provide a brief example of CFF applied to a simple hello-world program, and then discuss how Sophos researchers address CFF in Emotet's code. We'll finish by summarizing the challenges and problems we encountered during research.

Emotet's internals have been covered by many researchers, but we have not seen discussions on de-obfuscating its use of Control Flow Flattening so far.

## Emotet: Resurgence and Tenacity

Figure 1 shows the volume of Emotet payloads detected in our sandbox systems in the first quarter of 2022. As the chart shows, we receive multiple Emotet submissions daily; we believe that the recurring larger spikes result from large-scale campaigns being kicked off by the malware's distributors. This is a sensible assumption; Emotet is mainly distributed via email spam, and more malicious emails naturally lead to more sandbox submissions.

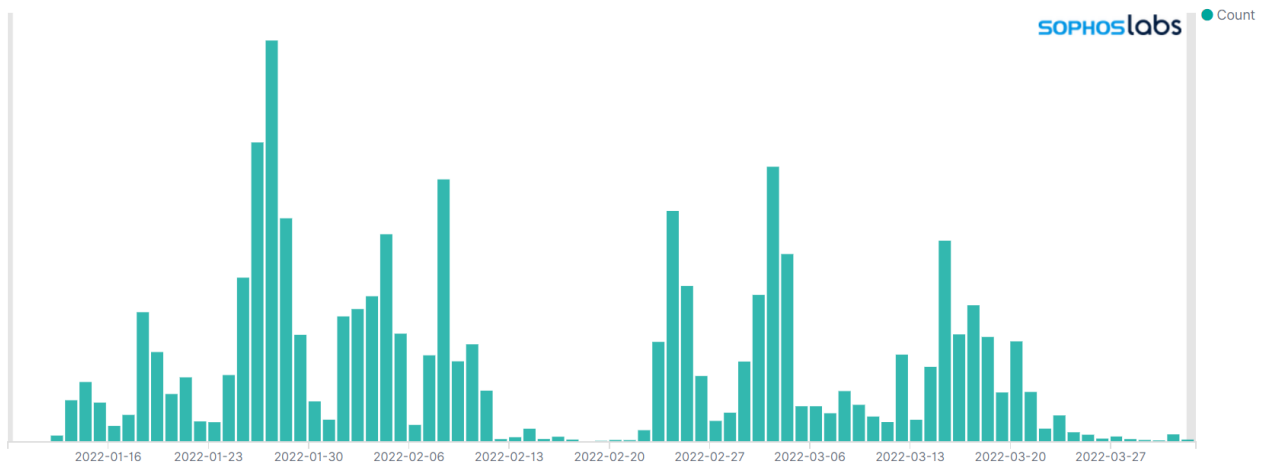


Figure 1: Timeline of 2022 Emotet detections in SophosLabs' sandbox systems

In addition to Emotet's delivery mechanism and prevalence, we also analyze the final payload in depth. Thus, we noticed Control Flow Flattening in an unpacked Emotet sample. Control Flow Flattening hides program flow by putting all function blocks next to each other. It is a well-known obfuscation technique used to conceal the purpose of software. While extracting the original code from a flattened binary is inherently challenging, we have successfully adapted some existing toolsets to deobfuscate the majority of Emotet payload functionality.

### What is Control Flow Flattening?

Control Flow Flattening is a technique that aims to obfuscate program flow by taking away tidy program structures in favor of putting the program blocks inside a loop with a single switch statement controlling program flow.

First, the body of the function is broken into basic blocks, and then the blocks are put next to each other on the same level. A visualization of this transformation can be seen in Figure 2. Control Flow Flattening can be combined with other obfuscation techniques, such as API Hashing or String Encryption. Some of the most prominent obfuscators for flattening functions are [OLLVM](#) and [Tigress](#).

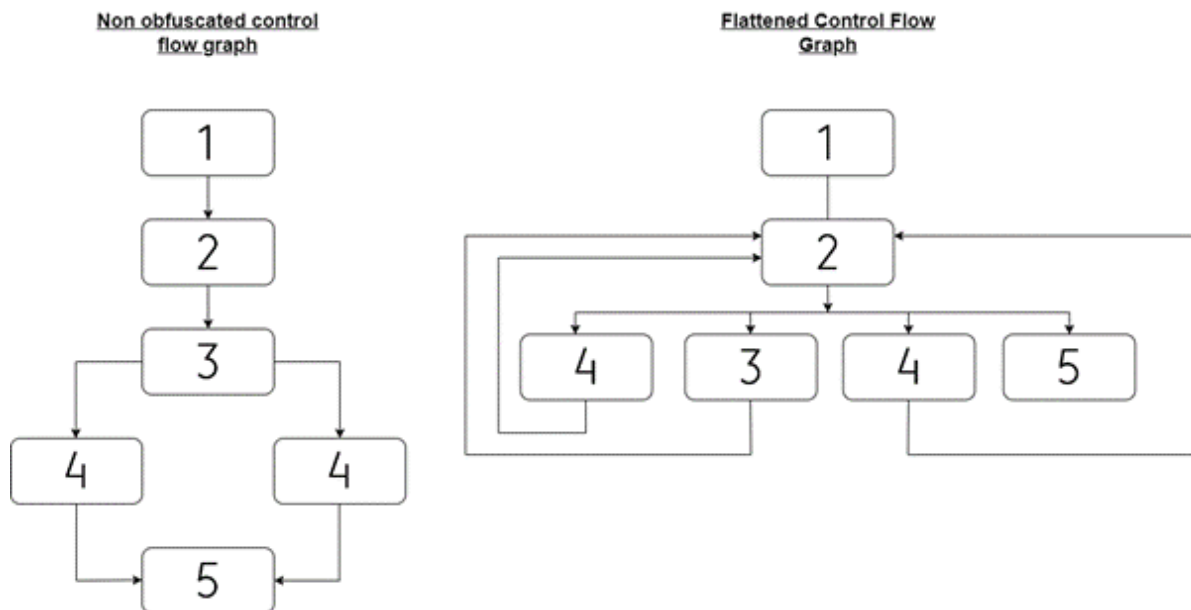


Figure 2: Comparing a flattened and non-flattened control flow graph (CFG)

Let's take a quick look at a simplified example of CFF in action.

### Flattening Hello World

For demonstration purposes, we've compiled a simple program written in C. On the left side of Figure 3, an annotated control flow graph (CFG) of the binary is shown. On the right side you can see the decompiled output generated by the Hex-Rays Decompiler.

In this figure, no obfuscation techniques have been applied. The Hex-Rays Decompiler has no trouble generating an easy-to-read high-level language representation of the disassembly. Even without a decompiler, an experienced reverse engineer can simply follow the control flow graph to understand its purpose.

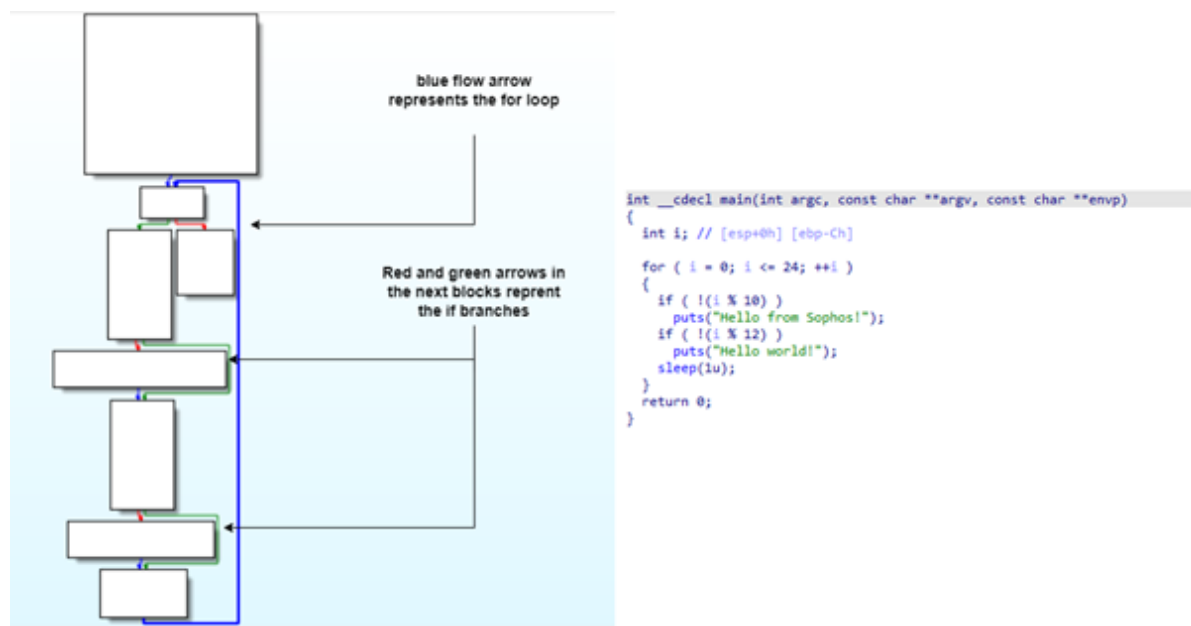


Figure 3: Control Flow Graph and decompiled output of sample program

Now we'll flatten the function and compare the results. Figure 4 displays the CFG and decompiled output after Control Flow Flattening was applied. On the left side, we see that the number of basic blocks has more than doubled, and reading the decompiled output is not possible any more without spending a significant amount of time analyzing it.



Figure 4: Annotated example of a flattened function

Overall, CFF introduces the following problems to hamper our analysis:

- The control flow is concealed. Instead of being able to follow the blocks, a control flow dispatcher block is implemented. This block determines which blocks are executed next.
- A state variable annotated as `stateVar` in the decompiled output is updated with high entropy variables throughout the function. The state variable is used by the control flow dispatcher to decide which block gets executed next.
- The two problems above lead to a highly complex decompiled output. While it is still possible to follow the execution flow, the time and effort needed to understand the function is significantly larger than it would be compared to the decompiled output in Figure 3.

## Unflattening Emotet

To deobfuscate Emotet's use of Control Flow Flattening, we started with a review of existing tools and research on CFG deobfuscation. Some of those include:

- [Hex-Rays Microcode API vs. Obfuscating Compiler](#) by Rolf Rolles
- [Defeating Compiler-Level Obfuscations used in APT10 Malware](#) by VMWare's Threat Analysis Unit
- [Deobfuscation: recovering an OLLVM-protected program](#) by Francis Gabriel of Quarkslab
- [D810: A journey into control flow unflattening](#) by Boris Batteaux of eShard

For deeper dives into the algorithm behind CFG Unflattening, the articles referenced above provide a wealth of information.

In Figure 5 you can see the decompiled output and CFG of a function in an unpacked Emotet sample. Excluding the Control Flow Flattening applied here, the output might seem confusing, because Emotet applies more than just one obfuscation technique. (If you are not familiar with those other techniques, an appendix at the end of this article briefly explains the other obfuscation techniques .)

First, the function calls `OpenSCManagerA` to retrieve a handle to the Service Control Manager. Next, it calls `OpenServiceW` to open an existing service. If opening the service succeeds, the opened service will be deleted via `DeleteService`. Finally, the opened handles will be closed. If the service was deleted successfully, the function returns 1, otherwise 0.

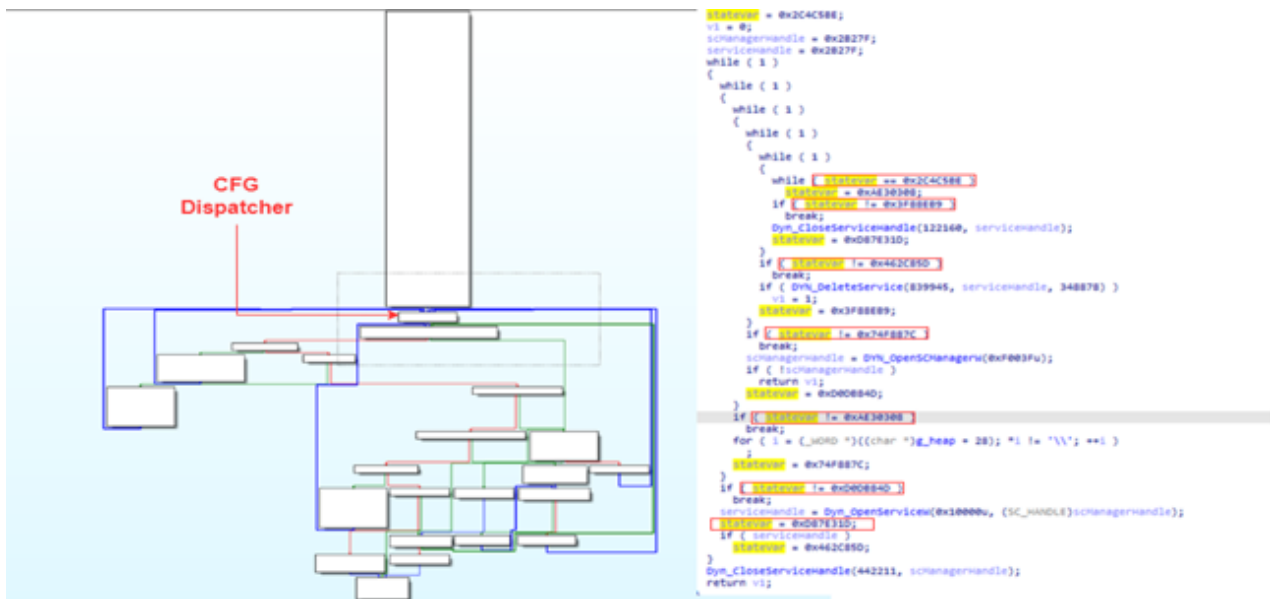


Figure 5: Annotated example of a flattened function

If we compare the decompiled output of Figure 3 and Figure 4, we can see multiple similarities, and we can identify the CFG dispatcher again. In the decompiled output, we see a variable we annotated as `stateVar`. Like the output in Figure 3, this is our state variable that is constantly updated and used by the dispatcher to determine which block is executed next.

On a high level, if we want to restore the control flow, we need to:

1. Identify the dispatcher block and states
2. For each block, identify the corresponding constant and find the address of the next block to execute based on dispatcher and state variable value
3. Patch the outbound dispatcher blocks to jump to the address of the original next block



Instead of patching and operating on the disassembly directly, we make use of the Hex-Rays Microcode API. Microcode is an intermediate language used by the Hex-Rays decompiler. During decompilation, the decompiler steps through different maturity phases. The different phases are displayed in Figure 6 below. The API allows us to hook the decompilation progress and operate on the microcode instead of patching the disassembly directly.



Figure 6: IDA Microcode maturity levels

## Adjusting the Tool

We used an IDAPython fork of the Rolf Rolles' HexRaysDeob tool as our foundation. Like the fork, we are operating solely on the maturity level MMAT\_LOCOPT, the third level in the figure above. As seen in Figure 6, that maturity level includes information about inbound and outbound blocks, which are necessary to correctly identify dispatcher blocks. Furthermore, the original code was based around the MMAT\_LOCOPT layer. Changing the layer would have required plenty more investigation, verification, and adjustments of the existing code than keeping the layer. Below is a summarization of changes we applied on the existing code base.

## Handling Multiple/Related Dispatchers

In multiple functions, running the deobfuscation algorithm on a single dispatcher did not generate an output we were satisfied with. Analysis showed that more complex functions might contain multiple nested dispatchers instead of one. We added additional logic to identify and run the algorithm on multiple dispatchers. This option can be turned on or off by setting the RUN\_MLTPLE\_DISPATCHERS flag to True or False. In Figure 7 below, you can see an example of a function with two potential dispatchers.

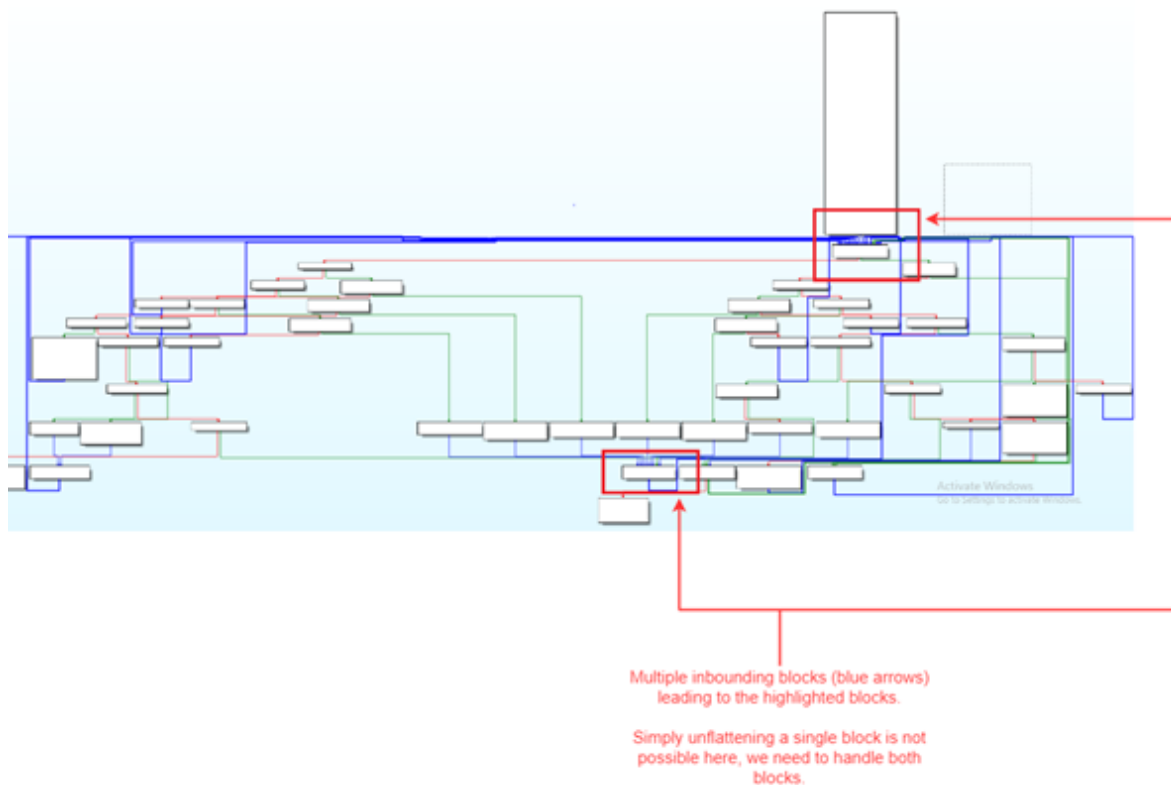


Figure 7: Example of a function with two potential dispatchers

### Risky Search for the Cluster Head

A flattened block might be implemented by multiple microcode blocks. To find the end of the region, the original algorithm by Rolf Rolles generates a dominator tree and uses the generated information to determine the end of a region, or the beginning of a cluster. In some cases, the algorithm failed to find the cluster head. We added an additional function to determine the cluster head as a fallback. We believe that the original algorithm by Rolf Rolles is more reliable; however, evaluation showed that the fallback algorithm still led to good results and improved the decompiled output.

### Adding Additional Patterns and Small Code Updates

In some cases, the existing logic failed in patching all flattened blocks. After analyzing multiple functions, we identified various patterns that reoccurred throughout the whole binary. We added additional logic to identify and unflatten blocks following these patterns to the existing code base. Finally, we adjusted the overall code a little. Some of the changes include:

- The IDAPython fork of Rolf Rolles' HexRaysDeob tool was based on Python2.7. We updated several parts of the code to match Python3 standards.

- In the original version of the tool, if the “run” function was invoked once, the plugin got activated and if the tool determined the function as flattened via an algorithm, it attempted to unflatten it. During implementation and testing, we experienced crashes of IDA Pro when using the IDAPython microcode API. This might lead to a corrupted IDB database. As an additional safety mechanism, the address of the target function must be added to the array “white\_list” to allow unflattening. Overall, we recommend saving often and keeping a separate IDB copy when using the tool.

Out of 254 functions, we categorized 68 functions as flattened. From these 68 functions, we were able to unflatten 38 successfully. Nineteen functions were partially flattened, and 11 functions failed. By “unflattened successfully,” we mean cases in which our script failed to unflatten a maximum of 3 states. “Partially unflattened” means that most of the function remains flattened, but our tool was able to unflatten some blocks. Finally, “failed” means that we were not able to deobfuscate a single block in the function.

Figure 8 below shows the function from Figure 5 after our script was applied.

```

v0 = 0;
v1 = 0xB78D;
scManagerHandle = 0x2B27F;
serviceHandle = 0x2B27F;
stateVar = 0xAE30308;
while ( stateVar != v1 )
{
    if ( stateVar == 0xAE30308 )
    {
        for ( i = (_WORD *)((char *)g_heap + 28); *i != '\\'; ++i )
        ;
        // First open a handle to the ServiceControlManager
        scManagerHandle = DYN_OpenSCManagerW(0xF003Fu);
        if ( !scManagerHandle )
            return v0;
        // Try to open a service, service name is stored in g_heap+28++
        serviceHandle = Dyn_OpenServiceW(0x10000u, (SC_HANDLE)scManagerHandle);
        stateVar = 0xD87E31D;
        v1 = 0x462C85D;
        // If failed, quit instantly
        if ( serviceHandle )
            stateVar = 0x462C85D;
    }
    else if ( stateVar == 0xD87E31D )
    {
        goto INSTANT_QUIT;
    }
}
if ( DYN_DeleteService(839945, serviceHandle) )
    v0 = 1;
Dyn_CloseServiceHandle(122160, serviceHandle);
INSTANT_QUIT:
Dyn_CloseServiceHandle(442211, scManagerHandle);
return v0;
}

```

Figure 8: Analyzed function after the CFG unflattening tool was applied

## IoCs

**Description** SHA256

---



---

Packed Emotet	9a0286ec0a3e7ea346759c9497c8b5c7c212fa2c780a1cabb094134bf492a51b
---------------	--

---

Unpacked Emotet	1bbce395c839c737fdc983534b963a1521ab9693a5b585f15b8a4950adea5973
-----------------	--

Our unflattening tool is [now available](#) on the SophosLabs Github. (For those interested in such things, we also recommend to your attention a [CFF-unpacking tool](#) released by ESET several years back to address control-flow flattening in the Stantinko botnet – another example of why, since attackers freely share tactics, techniques, and procedures among themselves, defenders are wise to do the same.)

## Conclusion and Limitations

Control Flow Flattening is a complex topic, and the purpose of this article is to share our experience and results attacking Emotet's Control Flow Flattening. While we made multiple adjustments and met with some success, our solution is not able to deobfuscate all functions completely. Among the outstanding issues:

- The algorithm to detect nested dispatchers is simple. Therefore, we have added an option to turn it on and off. In rare cases, a faulty output is generated if the nested dispatcher is enabled.
- In many functions, we had to deal with conditional states. Depending on the outcome of, for example, a WINAPI function, the state variable changes to a different value at runtime. Additional patching and insertion of microcode instructions would be needed to unflatten these conditional blocks.
- Our main approach was to add logic for reoccurring patterns in the binary. As our work progressed, we realized that a microcode emulator might have been a better choice, or would have been an adjustment that led to more unflattened blocks.
- During development and evaluation, we experienced multiple crashes. We are all humans and we make mistakes, so some crashes will result from bugs in our code. However, judging from the error messages, we believe that there is a deeper-rooted problem in the Python port of the Microcode API. Therefore, we recommend saving often and keeping a copy of the IDB file.

Overall, we recommend that researchers always cross-check their results and not trust the output blindly. Control Flow Flattening used in conjunction with other obfuscation techniques certainly complicates the process of reverse engineering Emotet, but the technique we've described helps to even the odds against researchers examining this high-profile malware.

## Appendix: Emotet and Code Obfuscation

---

When sharing the decompiled output of functions in an Emotet sample, it is impossible not to encounter other Emotet obfuscation techniques beyond CFF. This appendix covers the most prevalent obfuscation techniques we have identified in an unpacked Emotet sample. Keep in mind that Emotet is usually delivered in a packed form and needs to be unpacked first.

## String Encryption

Emotet contains encrypted strings in its unpacked form. Before usage, strings will be decrypted and freed again right after each serves its purpose.









	Up	p	sub_10002C79+124C	call	DecryptString; Microsoft Primitive Provider
	Do...	p	sub_10004A13+A29	call	DecryptString; %s%s.exe
	Do...	p	PersistViaService+652	call	DecryptString; %s\rundll32.exe "%s\%", %s
	Do...	p	sub_100065BD+BFC	call	DecryptString; ObjectLength
	Do...	p	sub_100065BD+C9E	call	DecryptString; SHA256
	Do...	p	sub_100065BD+CC6	call	DecryptString; SHA256
	Do...	p	sub_100093A7+612	call	DecryptString; %s\regsvr32.exe -s "%s"
	Do...	p	sub_100093A7+707	call	DecryptString; %s%s.dll

Figure 9: Cross references of DecryptString function with corresponding decrypted string

## API Hashing

Emotet uses API Hashing to conceal the usage of API functions. The malware calculates the hash of exported function names for a given DLL. If the calculated hash matches the constant pushed onto the stack at method invocation, the pointer to the exported function will be retrieved.

```

030 59                pop     ecx
02C F7 F1          div     ecx
02C 83 C4 04       add     esp, 4
028 BA 0E 02 00 00 mov     edx, 20Eh
028 89 45 FC        mov     [ebp+var_4], eax
028 B9 34 80 84 94 mov     ecx, 94848034h
028 81 75 FC E4 A9 08 00 xor     [ebp+var_4], 8A9E4h
028 8B 45 FC        mov     eax, [ebp+var_4]
028 8B 45 F8        mov     eax, [ebp+var_8]
028 8B 45 F4        mov     eax, [ebp+var_C]
028 68 B0 65 87 E5 push    0E58765B0h
02C E8 40 FE FF FF call    ApiHash
02C 83 C4 10       add     esp, 10h
01C 56             push    esi
020 FF D0         call    eax
01C 5E             pop     esi
018 8B E5        mov     esp, ebp
004 5D             pop     ebp
000 C3             retn
DYN_LoadLibraryW endp

```

Figure 10: Disassembly of ApiHash function invocation

In most cases, API Hashing calls and their corresponding dynamic call are wrapped into separate functions. We have automated this analysis, and functions with prefix DYN\_ are functions determined at runtime via API Hashing.

## Junk Instructions

Emotet embeds junk instructions to confuse reverse engineers. Junk instructions are instructions that do not serve any purpose except to complicate and slow down analysis. Figure 11 below shows an example of a junk instruction block.

```
000 55          push  ebp
004 8B EC      mov   esp, esp
004 81 EC 28 07 00 00  sub  esp, 720h
72C 53          push  ebx
730 56          push  esi
734 57          push  edi
738 FF 75 18   push  [ebp+arg_10]
73C 0B D9      mov   ebx, ecx
73C FF 75 14   push  [ebp+arg_C]
740 FF 75 10   push  [ebp+arg_8]
744 FF 75 0C   push  [ebp+arg_4]
748 FF 75 08   push  [ebp+arg_0]
74C 52          push  edx
750 53          push  ebx
754 E8 ED 76 FF FF  call nullsub_1
754 C7 45 E8 55 3F 00 00  mov  [ebp+var_18], 3F55h
754 83 C4 1C   add  esp, 1Ch
738 01 75 E8 34 8A FB D4  xor  [ebp+var_18], 004F88A34h
738 33 D2      xor  edx, edx
738 01 75 E8 22 52 10 40  xor  [ebp+var_18], 40105222h
738 09 31 30 CF 01   mov  ecx, 1CF3031h
738 01 45 E8 43 A2 00 00  add  [ebp+var_18], 0A2A3h
738 01 75 E8 96 89 EC 94  xor  [ebp+var_18], 94EC8996h
738 C7 45 E8 2F DF 00 00  mov  [ebp+var_20], 000DF2Fh
738 C1 65 E8 0F          shl  [ebp+var_20], 0Fh
738 C1 65 E8 00          shl  [ebp+var_20], 00h
738 8B 45 E8          mov  eax, [ebp+var_20]
738 6A 7C          push 7Ch ; '|'
73C 5E          pop  esi
738 F7 F6          div  esi
738 6A 62          push 62h ; 'b'
73C 09 45 E8          mov  [ebp+var_20], eax
73C 33 D2      xor  edx, edx
73C 01 75 E8 07 F2 E6 01  xor  [ebp+var_20], 1E6F207h
73C C7 45 AB CD A0 90 00  mov  [ebp+var_58], 90A0CDh
73C C1 6D AB 03          shr  [ebp+var_58], 3
73C 6B 45 AB 00          imul eax, [ebp+var_58], 00h
-- --
```

Figure 11: Example of junk instructions in unpacked Emotet sample

## Stack Obfuscation

Another interesting technique that confuses the IDA decompiler is the way in which Emotet passes parameters to functions. In Figure 12 below, we show how DYN\_BCryptEncrypt is invoked.

DYN\_BCryptEncrypt first resolves the API function BCryptEncrypt and stores the pointer to this function in register EAX. The function is then called via call EAX. Instead of just pushing the necessary parameters, this method pushes values onto the stack not being used by the actual EAX call. This leads to generation of a function signature that is much harder to read than normal.

```

.text:100223BB 0A8 FF 74 24 54      push    [esp+0A8h+var_54]
.text:100223BF 0AC FF B4 24 88 00 00 00  push    [esp+0ACh+var_24]
.text:100223C6 0B0 51              push    ecx
.text:100223C7 0B4 57              push    edi
.text:100223C8 0B8 FF 74 24 74      push    [esp+0B8h+var_44]
.text:100223CC 0BC 50              push    eax
.text:100223CD 0C0 A1 18 52 02 10    mov     eax, dword_10025218
.text:100223D2 0C0 FF 30           push    dword ptr [eax]
.text:100223D4 0C4 51              push    ecx
.text:100223D5 0C8 FF 74 24 78      push    [esp+0C8h+var_50]
.text:100223D9 0CC FF B4 24 90 00 00 00  push    [esp+0CCh+var_3C]
.text:100223E0 0D0 FF 74 24 50       push    [esp+0D0h+var_80]
.text:100223E4 0D4 FF 74 24 4C       push    [esp+0D4h+var_88]
.text:100223E8 0D8 FF 74 24 60       push    [esp+0D8h+var_78]
.text:100223EC 0DC 8B 4C 24 74       mov     ecx, [esp+0DCh+var_68]
.text:100223F0 0DC FF 76 04         push    dword ptr [esi+4]
.text:100223F3 0E0 8B 16           mov     edx, [esi]
.text:100223F5 0E0 E8 3D C6 FF FF     call   DYN_BCryptEncrypt
.text:100223FA 0E0 83 C4 3C         add     esp, 3Ch

```

Figure 12: Multiple values being pushed onto the stack before DYN\_BCryptEncrypt is invoked

```

int __usercall DYN_BCryptEncrypt@eax(
    PCHAR pbInput@cedx,
    int cbInput,
    int a3,
    int a4,
    int cbOutput,
    int cbIV,
    int a7,
    int a8,
    BCRYPT_KEY_HANDLE hkey,
    int pcbResult,
    int a11,
    int pbOutput,
    int a13,
    int a14,
    int a15,
    int dwFlags)
{
    int (__stdcall *BCryptEncrypt)(BCRYPT_KEY_HANDLE, PCHAR, ULONG, _DWORD, PCHAR, ULONG, ULONG, ULONG); // eax
    nullsub_1();
    BCRYPTEncrypt = (int (__stdcall *))(BCRYPT_KEY_HANDLE, PCHAR, ULONG, _DWORD, PCHAR, ULONG, ULONG, ULONG)ApiNash(419, 2101713412, 1494469899);
    return BCRYPTEncrypt(hkey, pbInput, cbInput, 0, 0, cbIV, (PCHAR)pbOutput, cbOutput, pcbResult, dwFlags);
}

```

Figure 13: Corresponding generated function signature