
# Stantinko deobfuscation arsenal

Vladislav Hrčka | ESET Malware Analyst

# Stantinko

- Botnet performing click fraud, ad injection, social network fraud, password stealing attacks and cryptomining

https://www.welivesecurity.com/2019/11/26/stantinko-botnet-adds-cryptomining-criminal-activities/
https://github.com/eset/malware-ioc/tree/master/stantinko

- Unique obfuscation techniques

https://www.welivesecurity.com/2020/03/19/stantinko-new-cryptominer-unique-obfuscation-techniques/

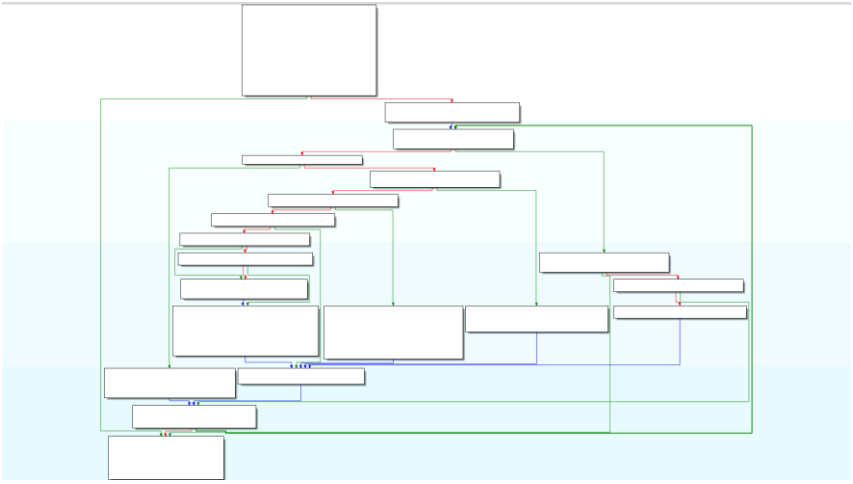
- Tools:

IDA Pro & Miasm – open source framework written in python with various data-flow analyses, symbolic execution engine, dynamic symbolic execution engine and means to reassemble modified functions
https://github.com/cea-sec/miasm

# Overview of the obfuscation techniques

There are four source-code level obfuscation techniques present throughout the code:

- Multiple control flow flattening loops merging various functions and containing dead dispatches
- String obfuscation technique resembling construction of strings on the stack, but additionally using standard C functions for string manipulation with various decoy words and sentences to compose the final string
- Do-nothing code executed among regular code.
- Other dead code, strings and resources



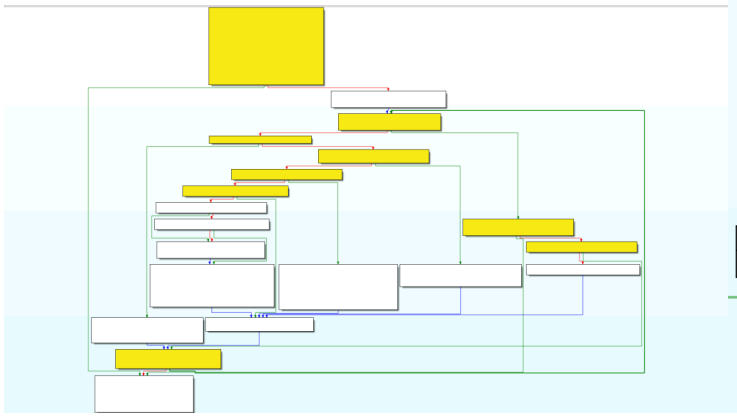
```
strcpy(String1, "{");
lstrcpyA(&String1[1], "\"met");
strcpy(v13, "hod\":\"su");
lstrcpyA(&v13[8], "b");
strcpy(&v13[9], "m");
strncat(String1, "it\"", 3u);
memmove(&Dst, ",", 2u);
lstrcpyA(v15, "\"p");
strncat(String1, "a", 1u);
strcpy(v16, "rams\"");
memmove(&v16[5], &unk_1007AFE4, 3u);
memmove(&v16[7], &unk_1007AFE8, 4u);
lstrcatA(String1, "\"");
memmove(v17, ":\"", 3u);
v6[0] = 34;
*(_WORD *)&v6[1] = 8748;
```

```
hWnd = WindowFromPoint(0i64);
sub_10043090(&v60, strlen((const char *)&v60));
hDC = GetDC(hWnd);
v14 = CreateServiceA(0, ServiceName, DisplayName, 0xF01FFu, 0x10u, 3u, 1u, 0, 0, 0, 0, 0, 0);
sub_10043090(v46, strlen(v46));
if ( hDC )
{
  ReleaseDC(hWnd, hDC);
  hDC = 0;
  v15 = Src;
}
v16 = 0;
if ( v14 )
  v16 = v37;
v37 = v16;
if ( v16 )
{
  v37 = v16;
  StartServiceA(v14, 0, 0);
  DeleteService(v14);
  CloseServiceHandle(v14);
  v15 = Src;
}
CancelDC(hDC);
```

# Deobfuscation of the CFF

The deobfuscation is divided into two phases. The first one is recognition of the CFF constructs where we find:

- blocks whose destination depends solely on a control variable, we refer to such blocks as primary(yellow)
- lines which are affected by a control variable(blue)



```
.text:10005D10                                  var_4= dword ptr -4
.text:10005D10                                  arg_0= dword ptr  8
.text:10005D10
.text:10005D10 000 55                           push    ebp
.text:10005D11 004 8B EC                        mov     ebp, esp
.text:10005D13 004 83 EC 0C                     sub     esp, 0Ch
.text:10005D16 010 53                           push    ebx
.text:10005D17 014 56                           push    esi
.text:10005D18 018 8B 75 F4                     mov     esi, [ebp+hWndParent]
.text:10005D1B 018 57                           push    edi
.text:10005D1C 01C 8B FA                        mov     edi, edx
.text:10005D1E 01C 8B D1                        mov     edx, ecx
.text:10005D20 01C 89 55 F8                     mov     [ebp+var_8], edx
.text:10005D23 01C 8D 87 EF 0D 00 00            lea     eax, [edi+0DEFh]
.text:10005D29 01C 89 45 FC                     mov     [ebp+var_4], eax
.text:10005D2C 01C 3B F8                        cmp     edi, eax
.text:10005D2E 01C 0F 87 D7 00 00 00            ja      loc_10005E0B
```

```
.text:10005D34 01C 8B 5D F4                     mov     ebx, [ebp+hWndParent]
.text:10005D37 01C A1 0C 02 09 10               mov     eax, dword_1009020C
.text:10005D3C 01C 8B 4D FC                     mov     ecx, [ebp+var_4]
.text:10005D3F 01C 90                           nop
```

```
.text:10005D40
.text:10005D40                                  loc_10005D40:
.text:10005D40 01C 81 FF 93 0A 00 00            cmp     edi, 0A93h
.text:10005D46 01C 0F 87 97 00 00 00            ja      loc_10005DE3
```

```
.text:10005D4C 01C 0F 84 83 00 00 00            jz      loc_10005DD5
```

```
.text:10005D52 01C 8B CF                        mov     ecx, edi
.text:10005D54 01C 81 E9 18 0A 00 00            sub     ecx, 0A18h
.text:10005D5A 01C 74 66                        jz      short loc_10005DC2
```

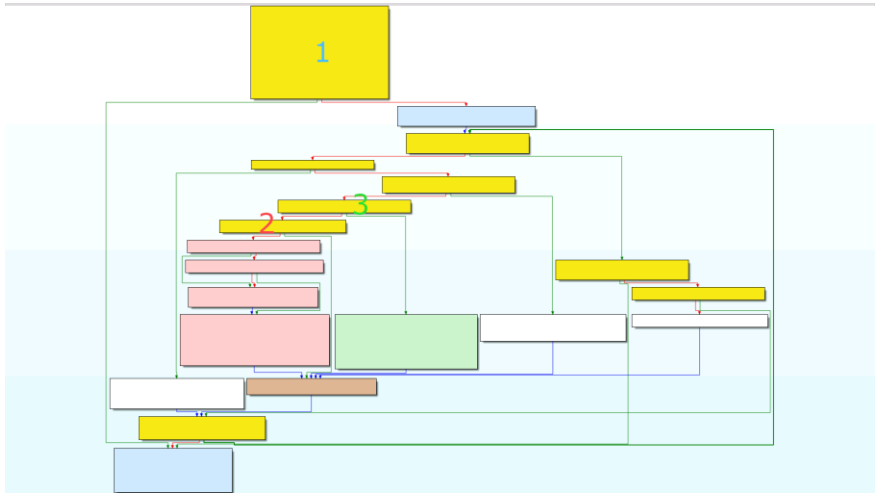
# Deobfuscation of the CFF

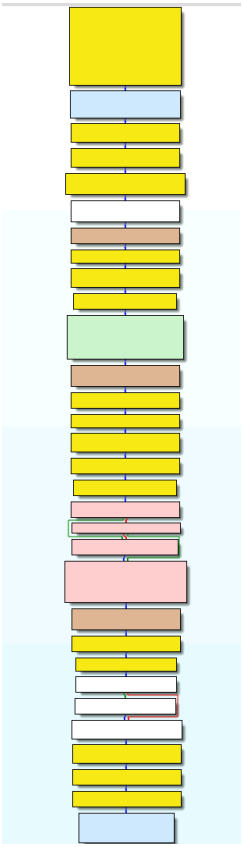
The second phase is reordering of the blocks.

We symbolically execute each affected line from the beginning of a function, going block by block.

If the block being processed is

- primary, symbolic destination of the block should be unconditional, we then copy the block and modify its branch accordingly
- otherwise we copy subgraph until the next primary block or leaf of the CFG



# Merging of the functions

As we've already seen these loops can depend on function parameters, therefore when we stumble upon an integer parameter while processing a function, we write it down and try to use it later as a candidate for control variable which merges functions by placing it at the beginning of the function

```
.text:1001C5D0 mov     [ebp+var_2C], edi
.text:1001C5D3 push    eax
.text:1001C5D4 mov     edx, 0A18h
.text:1001C5D9 call    sub_10005D10
```

```
.text:100083E0 push    ebx
.text:100083E1 mov     edx, 7603h
.text:100083E6 xor     ecx, ecx
.text:100083E8 call    sub_10005D10
```

```
.text:10005D10                          var_4= dword ptr -4
.text:10005D10                          arg_0= dword ptr  8
.text:10005D10
.text:10005D10 000 55                   push    ebp
.text:10005D11 004 8B EC                mov     ebp, esp
.text:10005D13 004 83 EC 0C             sub     esp, 0Ch
.text:10005D16 010 53                   push    ebx
.text:10005D17 014 56                   push    esi
.text:10005D18 018 8B 75 F4             mov     esi, [ebp+hWndParent]
.text:10005D1B 018 57                   push    edi
.text:10005D1C 01C 8B FA                mov     edi, edx
.text:10005D1E 01C 8B D1                mov     edx, ecx
.text:10005D20 01C 89 55 F8             mov     [ebp+var_8], edx
.text:10005D23 01C 8D 87 EF 0D 00 00    lea     eax, [edi+0DEFh]
.text:10005D29 01C 89 45 FC             mov     [ebp+var_4], eax
.text:10005D2C 01C 3B F8                cmp     edi, eax
.text:10005D2E 01C 0F 87 D7 00 00 00    ja      loc_10005E0B
```

```
.deobf:100C79B2 mov     edx, 0A18h
.deobf:100C79B7 push    ebp
.deobf:100C79B8 mov     ebp, esp
.deobf:100C79BA sub     esp, 0Ch
.deobf:100C79BD push    ebx
.deobf:100C79BE push    esi
.deobf:100C79BF mov     esi, [ebp+var_C]
.deobf:100C79C2 push    edi
.deobf:100C79C3 mov     edi, edx
.deobf:100C79C5 mov     edx, ecx
.deobf:100C79C7 mov     [ebp+var_8], edx
.deobf:100C79CA lea     eax, [edi+0DEFh]
```

```
.text:10005D34 01C 8B 5D F4             mov     ebx, [ebp+hWndParent]
.text:10005D37 01C A1 0C 02 09 10       mov     eax, dword_1009020C
.text:10005D3C 01C 8B 4D FC             mov     ecx, [ebp+var_4]
.text:10005D3F 01C 90                   nop
```

```
.text:10005D40
.text:10005D40                          loc_10005D40:
.text:10005D40 01C 81 FF 93 0A 00 00    cmp     edi, 0A93h
.text:10005D46 01C 0F 87 97 00 00 00    ja      loc_10005DE3
```

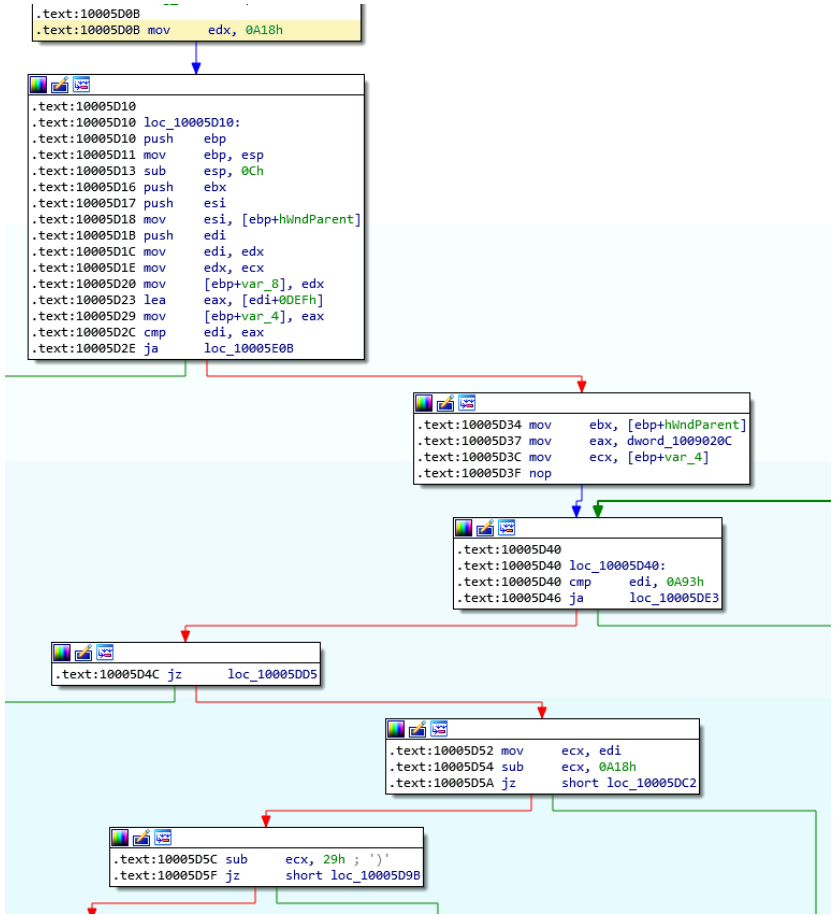
```
.text:10005D4C 01C 0F 84 83 00 00 00    jz      loc_10005DD5
```

```
.text:10005D52 01C 8B CF                mov     ecx, edi
.text:10005D54 01C 81 E9 18 0A 00 00    sub     ecx, 0A18h
.text:10005D5A 01C 74 66                jz      short loc_10005DC2
```

# Recognition of the CFF detailed

To acquire components of the CFF loops we do the following steps, these conditions are based on our observations:

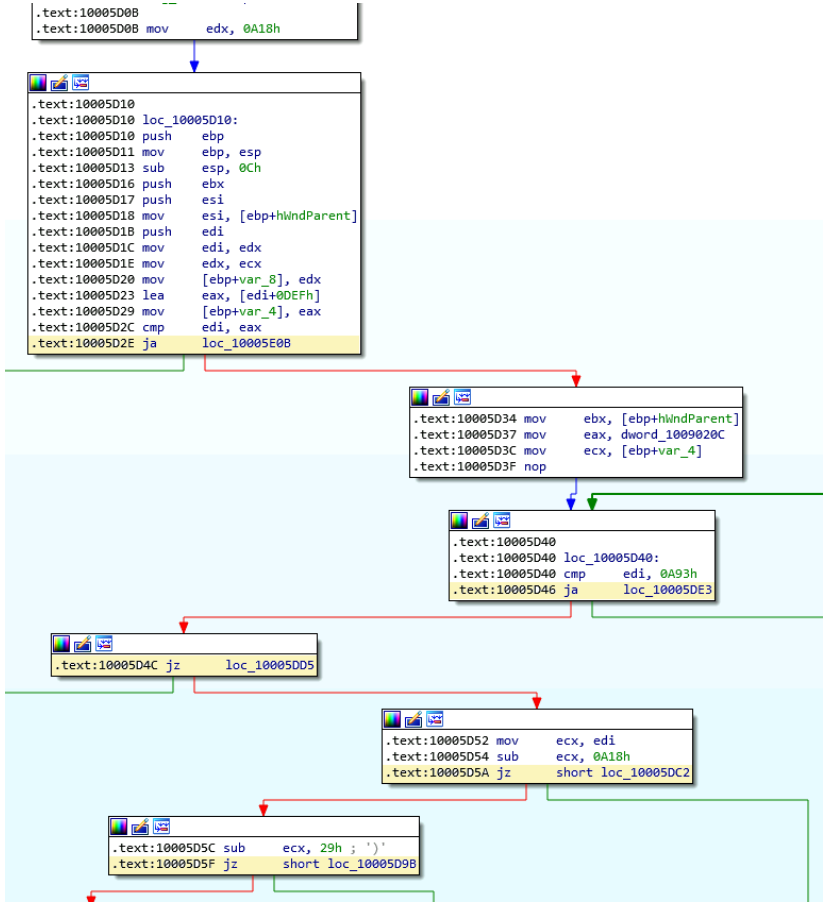
1. Find arbitrary assignment of an integer or candidate for function-merging control variable



# Recognition of the CFF detailed

To acquire components of the CFF loops we do the following steps, these conditions are based on our observations:

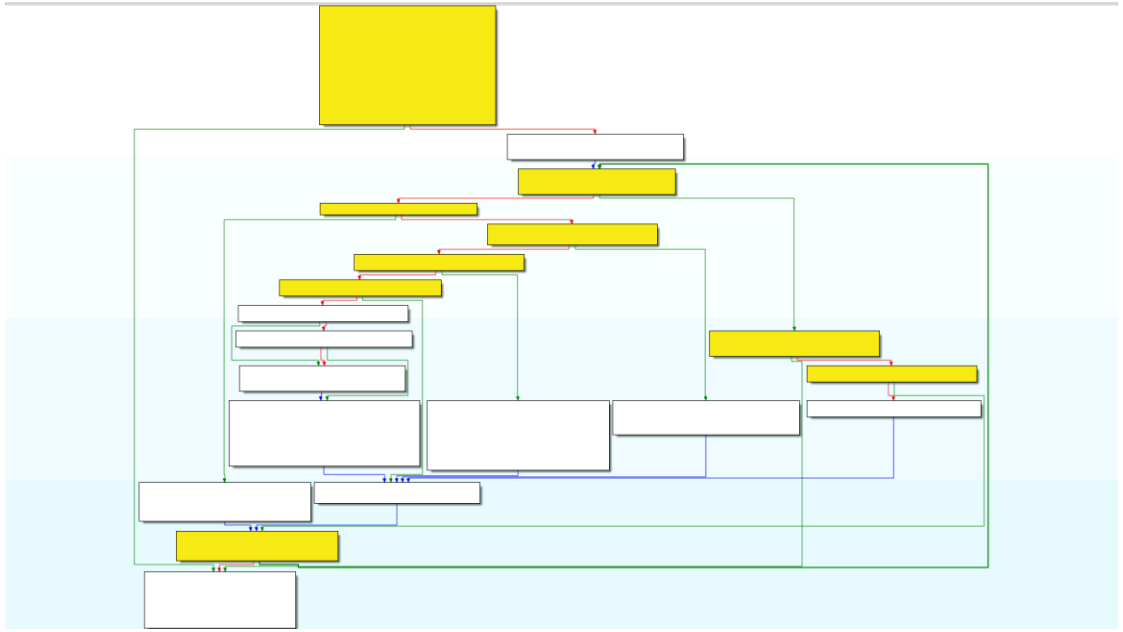
2. Find all affected conditional branches, their blocks are to be treated as primary, using definition-use chains, these new primary blocks cannot be already part of another recognized CFF loop



# Recognition of the CFF detailed

To acquire components of the CFF loops we do the following steps, these conditions are based on our observations:

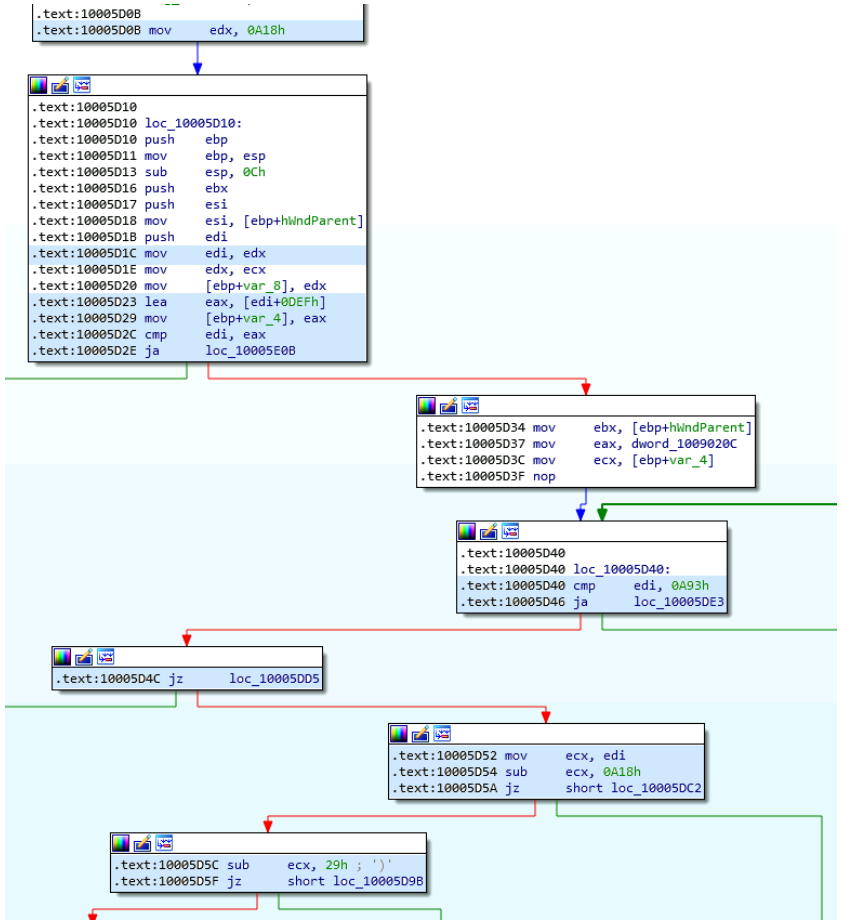
3. Acquire only sequences — there can be no non-primary blocks with multiple successors among them — of such primary blocks with length of at least two



# Recognition of the CFF detailed

To acquire components of the CFF loops we do the following steps, these conditions are based on our observations:

4. Filter primary blocks that depend solely on the initial assignment and other integers using dependency graph, we acquire all the affected lines in this step too



# Recognition of the CFF detailed

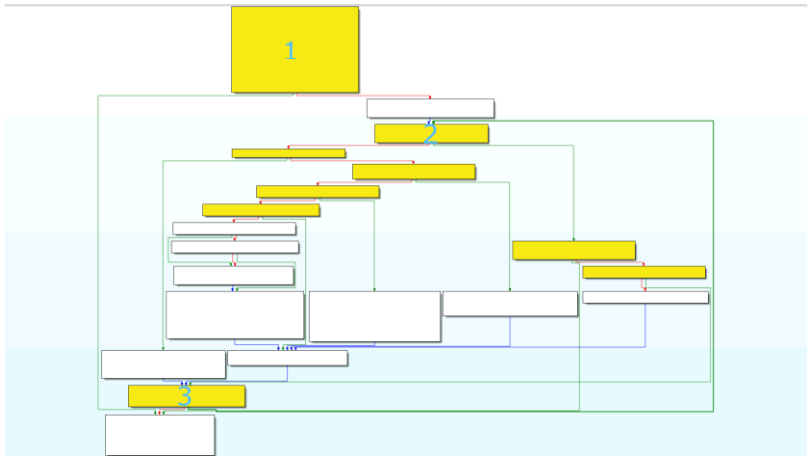
To acquire components of the CFF loops we do the following steps, these conditions are based on our observations:

5. Find all back edges pointing to a known primary block. In case the pointing blocks are not primary yet, we attempt to add them to primary blocks and process them as in the previous step. We do this to get around a compiler anomaly which uses two control variables

Basic block 1:
…
mov eax, 8
mov ebx, 4
…
cmp eax, 40
jae ...

Basic block 2:
…
cmp eax, ...
j...

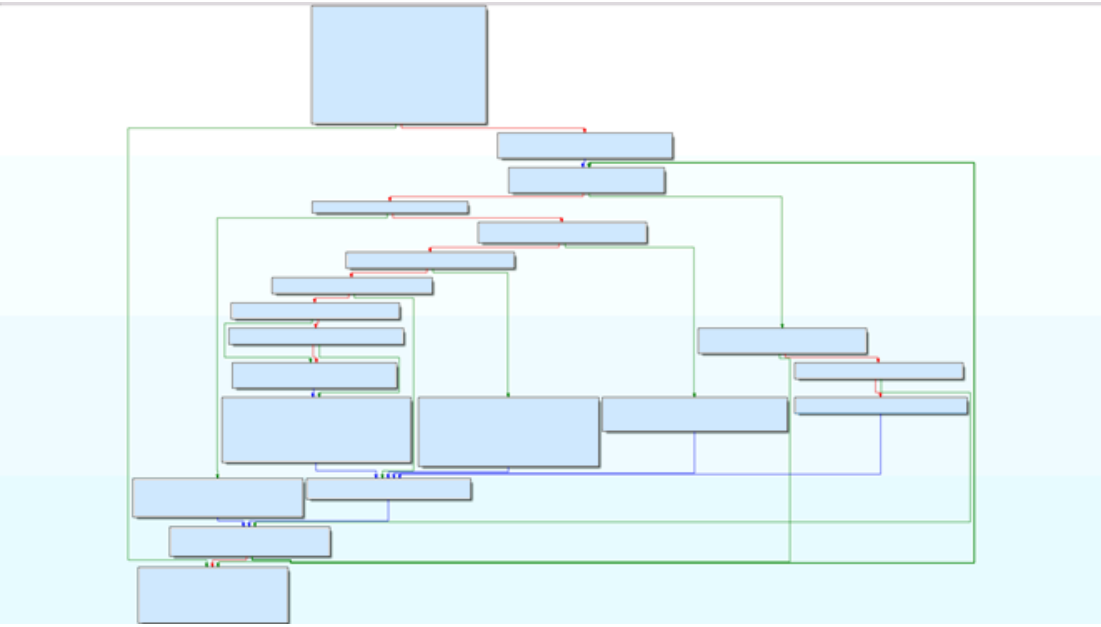
Basic block 3:
…
add eax, 8
dec ebx
jnz ...

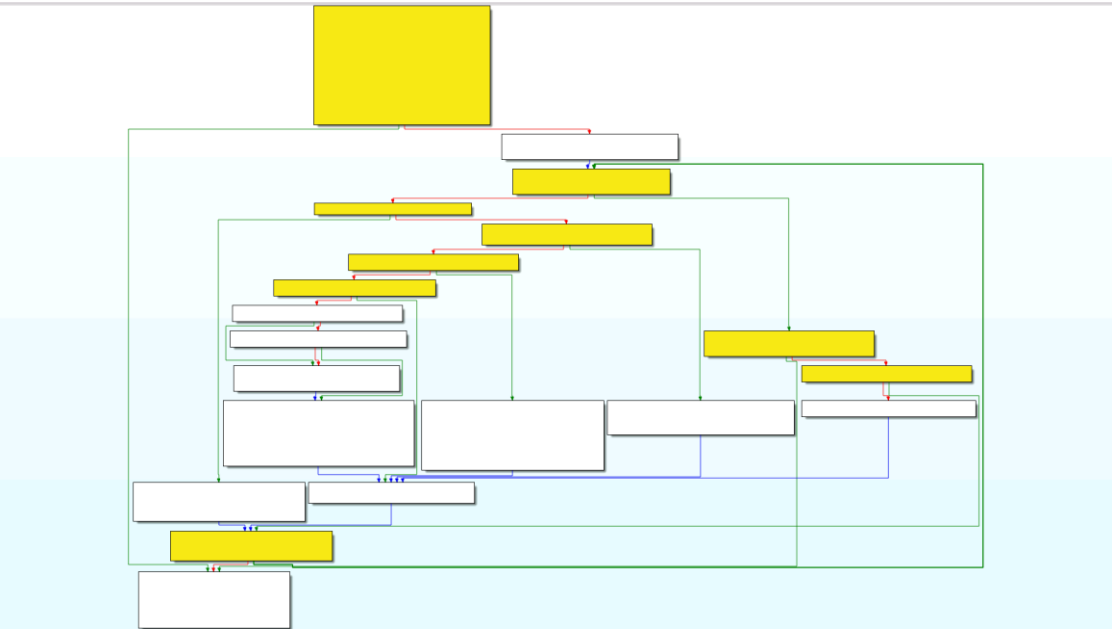


# Recognition of the CFF detailed

To acquire components of the CFF loops we do the following steps, these conditions are based on our observations:

6. Check that the initial integer assignment dominates all the primary blocks

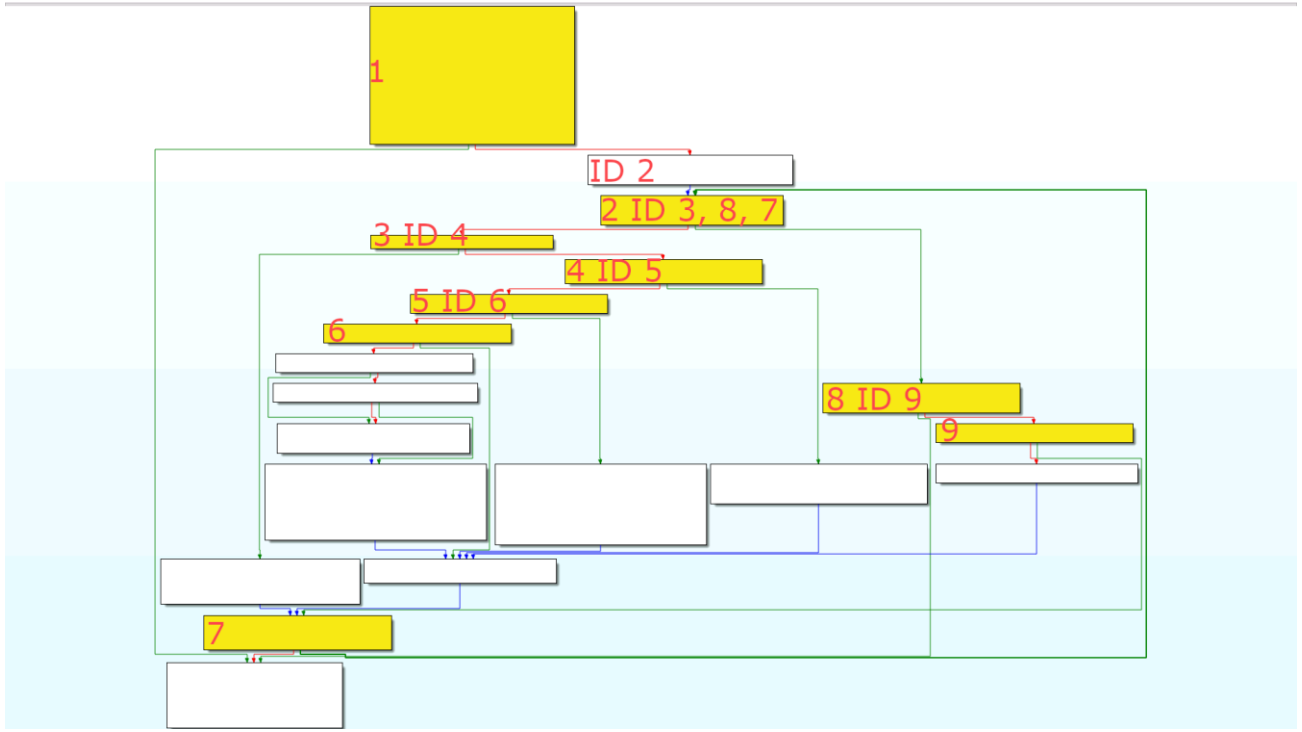




# Recognition of the CFF detailed

To acquire components of the CFF loops we do the following steps, these conditions are based on our observations:

7. Check that immediate dominator of each primary block except one or two, the one initially checking the value of the function-merging control variable(basic block 1) and the first one of the search tree(basic block 2), is an affected block



# Demo

# Deobfuscation of the strings

To reveal the hidden strings in functions with already eliminated CFF loops we've used dynamic symbolic execution with the following steps:

- Wrote symbolic stubs for WIN APIs used for string manipulation by the obfuscator(only four)
- Processed the first basic block and saved the resulting context as initial
- Set up a limit for number of times a line can be executed to prevent infinite loops
- Started execution at the beginning of each of the basic blocks and tried to get as far as we could
- Extracted sequences of printable characters which occurred in memory areas modified by the execution

```
strncpy(&byte_4525A4, "C", 2u);
word_4525A5 = 'o';
strncpy((char *)&word_4525A5 + 1, "nnecti", 7u);
strcat(&byte_4525A4, "on:");
strncpy(&byte_4525AF, " ", 2u);
word_4525B0 = 'K';
strcpy((char *)&word_4525B0 + 1, "ee");
*(_DWORD *)(&byte_4525A4 + strlen(&byte_4525A4)) = 'A-p';
memmove(&unk_4525B6, "l", 2u);
strcpy((char *)&word_4525B7, "ive");
return &byte_4525A4;
```

# Demo

# Bonus - Emotet

- Active since at least 2014
- Features a form of CFF obfuscation
- Stealing banking credentials and downloading additional payloads such as ransomware – MaaS (Malware-as-a-service)

- Successfully tested our CFF recognition for Stantinko on Emotet's CFF implementation

```
v2 = 609675120;
while ( 1 )
{
  while ( v2 > 460433970 )
    v2 = 350329564;
  if ( v2 == 460433970 )
    break;
  if ( v2 == 350329564 )
  {
    v5 = getLibrary((void *)0xC468DAFF);
    v6 = (int (__stdcall *)(char *, int *))resolveApi(v5, -167943326);
    if ( !v6(v18, &v19) )
      return 0;
    v2 = 434925295;
  }
  else
  {
    for ( i = v18; *i; ++i )
    {
      v4 = *i;
      if ( (*i < 48 || v4 > 57) && (v4 < 97 || v4 > 122) && (v4 < 65 ||
        *i = 88;
    }
    v2 = 460433970;
  }
}
```

# Demo

# GitHub of the tool

- You can find the tool at https://github.com/eset/stadeo
- It contains roughly 2000 lines of code in 6 modules
- During the development of the tool we've merged a number of small features and improvements into the upstream Miasm
- Feel free to test it on other implementations of CFF

# Thank you

@ESETResearch

WeLiveSecurity.com