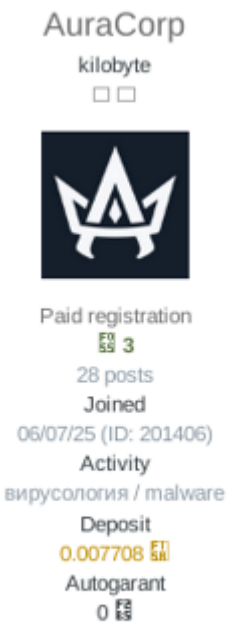
Aura Stealer #2 beatin the obfuscation

Published: 2025-10-29 · Archived: 2026-04-05 18:02:04 UTC

Hello party people

today we gonna deep dive into the topic of obfuscation in AuraStealer

just kiddin this aint chatgpt, this is my bad english.



Profile

So lets look into the obfuscation from Aura Stealer.

It uses a very similar technique like the one which is describe here https://cloud.google.com/blog/topics/threat-intelligence/lummac2-obfuscation-through-indirect-control-flow

It is called `indirect control flow` I'll use the `WinMain()` here as a good example.

we directly can see that there are no direct function calls.

```
// bad sp value at call has been detected, the output may be wrong!
int __stdcall WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nShowCmd)
{
  int v4; // eax
  int v5; // edx
  _DWORD v7[164]; // [esp+120h] [ebp-75Ch] BYREF
  int v8; // [esp+3B0h] [ebp-4CCh] BYREF
  int v9; // [esp+3B4h] [ebp-4C8h]

  ((void (*)(void))((char *)off_49AEE4 + 12476))();
  v7[0] = 1335530668;
  v4 = ((int (__cdecl *)(_DWORD *))((char *)off_49B1FC + 14100))(v7);
  ((void (__thiscall *)(char *, int *, _DWORD *, int))((char *)off_49A268 - 13741))(
    (char *)off_49B2A0 + 493,
    &v8,
    v7,
    v4);
  v5 = 85;
  if ( v9 == *((_DWORD *)off_49A208 + 1716) )
    v5 = 275;
  if ( !v9 )
    v5 = 275;
  return ((int (__stdcall *)(HINSTANCE, HINSTANCE, LPSTR, int))(*(char **)((char *)&off_4998B0 + v5 * 4)
                                                                + dword_49A400[v5]))(
           hInstance,
           hPrevInstance,
           lpCmdLine,
           nShowCmd);
}
```

```
text:0042F6C6 _WinMain@16     proc near               ; CODE XREF: __scrt_common_main_seh(void)+F3↓p
text:0042F6C6
text:0042F6C6 var_4C8         = dword ptr -4C8h
text:0042F6C6 hInstance       = dword ptr  8
text:0042F6C6 hPrevInstance   = dword ptr  0Ch
text:0042F6C6 lpCmdLine       = dword ptr  10h
text:0042F6C6 nShowCmd        = dword ptr  14h
text:0042F6C6
text:0042F6C6                 push    ebp
text:0042F6C7                 mov     ebp, esp
text:0042F6C9                 push    ebx
text:0042F6CA                 push    edi
text:0042F6CB                 push    esi
text:0042F6CC                 and     esp, 0FFFFFFF0h
text:0042F6CF                 sub     esp, 870h
text:0042F6D5                 mov     esi, esp
text:0042F6D7                 mov     eax, 30BCh
text:0042F6DC
text:0042F6DC loc_42F6DC:                             ; DATA XREF: .data:0049A168↓o
text:0042F6DC                 add     eax, off_49AEE4
text:0042F6E2                 call    eax
text:0042F6E4                 mov     dword ptr [esi+20h], 658943857
text:0042F6EB                 mov     eax, [esi+20h]
text:0042F6EE                 xor     eax, 68DC3BDDh
text:0042F6F3                 lea     edi, [esi+120h]
text:0042F6F9                 mov     [edi], eax
text:0042F6FB                 mov     eax, 3714h
text:0042F700                 add     eax, off_49B1FC
text:0042F706                 push    edi
text:0042F707                 call    eax
text:0042F709                 add     esp, 4
text:0042F70C                 mov     ecx, 1EDh
text:0042F711                 add     ecx, off_49B2A0
text:0042F717                 mov     edx, 0FFFFCA53h
text:0042F71C                 add     edx, off_49A268
text:0042F722                 lea     ebx, [esi+3B0h]
text:0042F728                 push    eax
text:0042F729                 push    edi
text:0042F72A                 push    ebx
text:0042F72B                 call    edx
text:0042F72D
text:0042F72D loc_42F72D:                             ; DATA XREF: .data:00499FF4↓o
text:0042F72D                 mov     eax, [ebx+4]
text:0042F730                 mov     ecx, off_49A208
text:0042F736                 cmp     eax, [ecx+1AD0h]
text:0042F73C                 mov     ecx, 44Ch
text:0042F741                 mov     edx, 154h
text:0042F746                 cmovz   edx, ecx
text:0042F749                 test    eax, eax
text:0042F74B                 cmovz   edx, ecx
text:0042F74E                 mov     ecx, off_4998B0[edx]
text:0042F754
text:0042F754 loc_42F754:                             ; DATA XREF: .data:00499C98↓o
text:0042F754                 add     ecx, dword_49A400[edx]
text:0042F75A                 jmp     ecx
text:0042F75A _WinMain@16     endp
```

lets take a look at the first call.

```
 .text:0042F6D7                mov     eax, 30BCh
.text:0042F6DC
.text:0042F6DC loc_42F6DC:
.text:0042F6DC                add     eax, off_49AEE4 ;  dword ptr [0x49aee4]
.text:0042F6E2                call    eax
```

so what is happening here? We can see `mov eax, 30BCh` that means the register holds the value `30BCh` now after that we `add eax, off_49AEE4` `off_49AEE4` lays in `.data` and points to `loc_450461+3`

at `loc_450461+3` the address would be `0x00450464` important we get the address not the data because of the `off_49AEE4` which is the same like `dword ptr [0x49aee4]`

what that means for the calculation of our functions address? we just have to calculate `0x30BC + 0x00450464 = 0x00453520` so `eax = 0x00453520`

`call 0x00453520` should be a function now lets see and boom we find a function here :3

```
.text:00453520 ; int *sub_453520()
.text:00453520 sub_453520      proc near
.text:00453520
.text:00453520 var_143C        = dword ptr -143Ch
.text:00453520 var_1438        = dword ptr -1438h
.text:00453520 var_1430        = dword ptr -1430h
.text:00453520 var_142C        = dword ptr -142Ch
.text:00453520 var_1428        = dword ptr -1428h
.text:00453520 var_1424        = dword ptr -1424h
.text:00453520 var_1420        = dword ptr -1420h
.text:00453520 String          = word ptr -141Ch
.text:00453520 var_1418        = dword ptr -1418h
.text:00453520 var_1414        = qword ptr -1414h
.text:00453520 var_140C        = xmmword ptr -140Ch
.text:00453520 var_13FC        = dword ptr -13FCh
.text:00453520 var_13F8        = dword ptr -13F8h
.text:00453520 var_13F4        = dword ptr -13F4h
.text:00453520 var_13F0        = dword ptr -13F0h
.text:00453520 var_13EC        = dword ptr -13ECh
.text:00453520 var_13E8        = dword ptr -13E8h
.text:00453520 var_13E4        = dword ptr -13E4h
.text:00453520 var_13E0        = dword ptr -13E0h
.text:00453520 var_13DC        = dword ptr -13DCh
.text:00453520 var_13D8        = dword ptr -13D8h
.text:00453520 var_13D4        = dword ptr -13D4h
.text:00453520 var_13D0        = qword ptr -13D0h
.text:00453520 var_13C8        = dword ptr -13C8h
.text:00453520 var_13C4        = dword ptr -13C4h
.text:00453520 var_13C0        = dword ptr -13C0h
.text:00453520 var_13BC        = dword ptr -13BCh
.text:00453520 var_13B8        = dword ptr -13B8h
.text:00453520 var_13B4        = dword ptr -13B4h
.text:00453520 var_13B0        = qword ptr -13B0h
.text:00453520 var_13A8        = dword ptr -13A8h
.text:00453520 var_13A4        = dword ptr -13A4h
.text:00453520 var_13A0        = dword ptr -13A0h
.text:00453520 Block           = dword ptr -139Ch
.text:00453520 var_1398        = dword ptr -1398h
.text:00453520 var_1394        = qword ptr -1394h
.text:00453520 var_138C        = xmmword ptr -138Ch
.text:00453520 var_18          = dword ptr -18h
.text:00453520
.text:00453520                 push    ebp
.text:00453521                 mov     ebp, esp
.text:00453523                 push    ebx
.text:00453524                 push    edi
.text:00453525                 push    esi
.text:00453526                 and     esp, 0FFFFFFF0h
.text:00453529                 mov     eax, 1430h
.text:0045352E                 call    __alloca_probe
.text:00453533                 mov     eax, 210B86E4h
.text:00453538                 mov     ecx, 3967A763h
.text:0045353D                 lea     edi, [esp+143Ch+String]
.text:00453541                 mov     [edi+4], ecx
.text:00453544                 mov     [edi], eax
```

As we can see we have many patterns like this with `jmp/call` variation

```
mov     eax, [esp+174h]
cmp     eax, 8
mov     ecx, 0E8h
mov     edx, 104h
cmovnb  edx, ecx
mov     ecx, off_495294
mov     ecx, [ecx+edx-4AFFh]
mov     esi, off_494B9C
add     ecx, [esi+edx-3757h]
jmp     ecx
```

```
lea     eax, ds:2[eax*2]
mov     ecx, off_495248
mov     edx, 0FFFFDB5Eh
add     edx, [ecx-11E8h]
push    eax
push    dword ptr [esp+74h]
call    edx
add     esp, 8
mov     eax, [esp+174h]
cmp     eax, 8
mov     ecx, 0E8h
mov     edx, 104h
cmovnb  edx, ecx
mov     ecx, off_495294
mov     ecx, [ecx+edx-4AFFh]
mov     esi, off_494B9C
add     ecx, [esi+edx-3757h]
jmp     ecx
```

```
lea     eax, ds:2[eax*2]
mov     ecx, off_494E00
mov     edx, 0FFFFCD35h
add     edx, [ecx+4935h]
push    eax
push    dword ptr [esp+14h]
call    edx
```

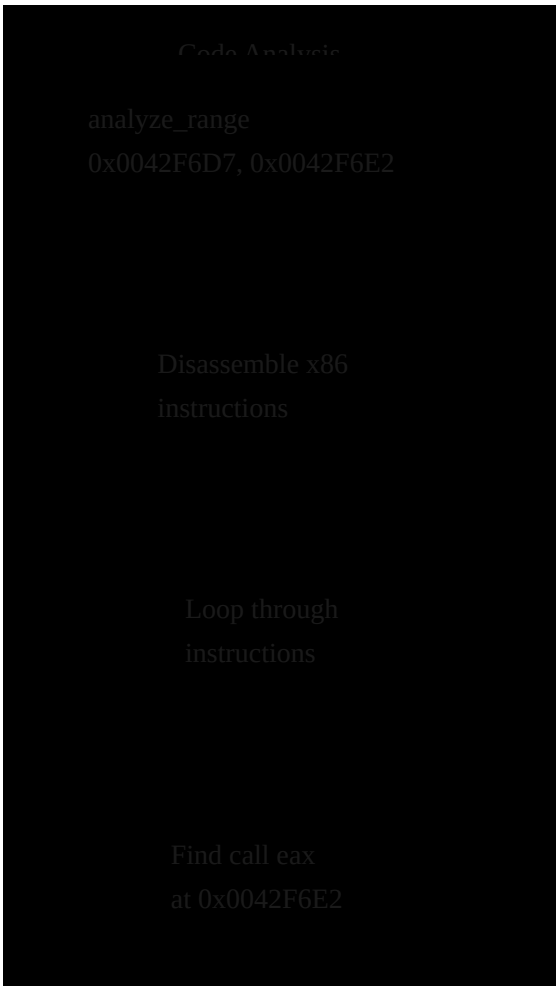
So what now?

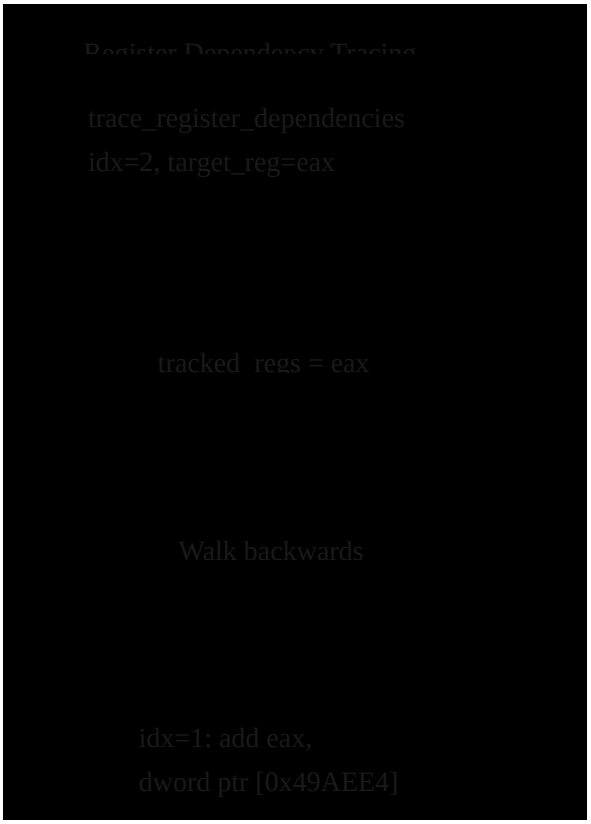
first i tried hardcoding every patttern…
but then I had the idea to just look for `call/jmp register` in the dissassembled code and trace back every register which is important for the calculation of our final call address.

Turns out I should have read the whole article “LummaC2: Obfuscation Through Indirect Control Flow” https://cloud.google.com/blog/topics/threat-intelligence/lummac2-obfuscation-through-indirect-control-flow

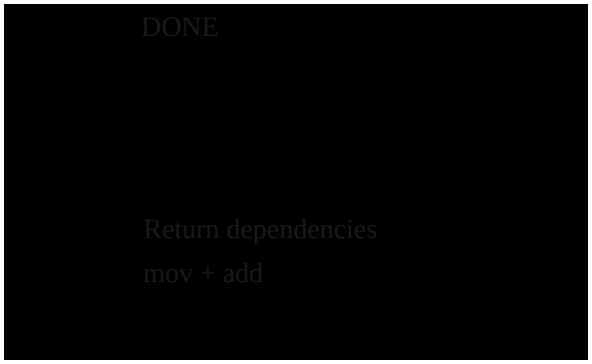
Here it is described as `symbolic backward slicing`, but hey atleast i had the idea too, just need to fail for 3 days straight wondering how many patterns are left.

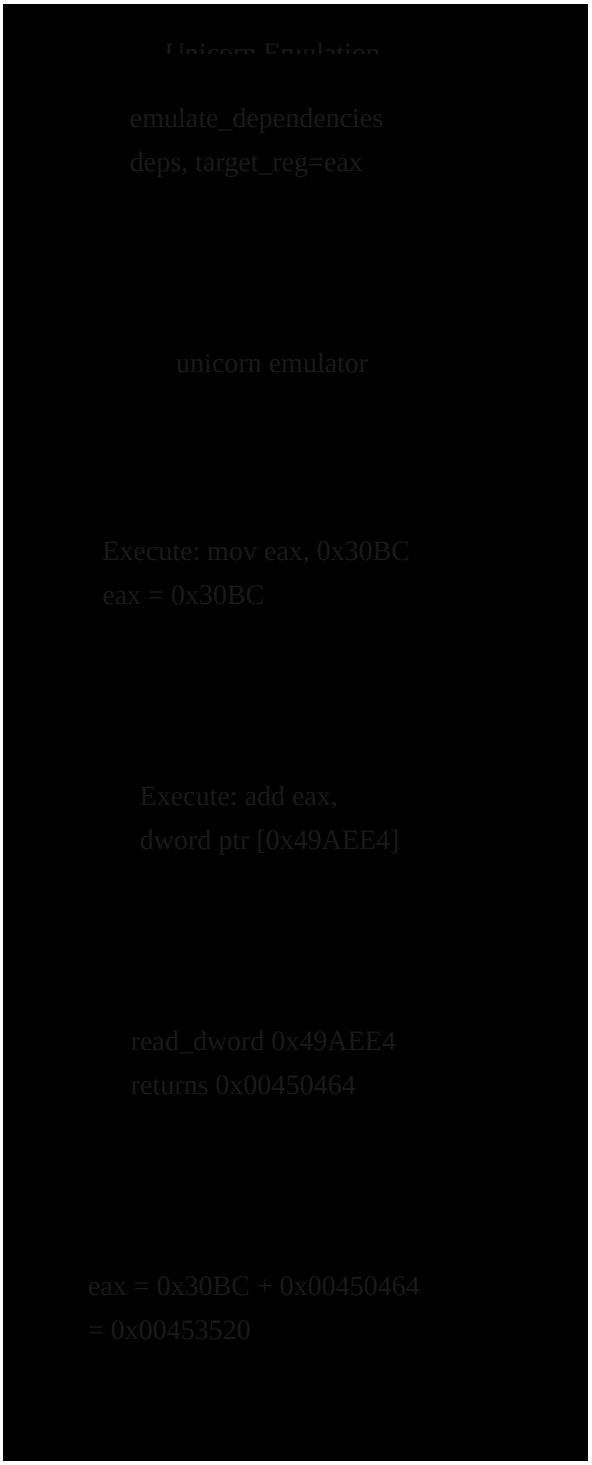
```
 ; earlier example from WinMain we calculated
0x0042F6D7: mov eax, 30BCh
0x0042F6DC: add eax, dword ptr [0x49AEE4]
0x0042F6E2: call eax
```


Code Analysis
analyze_range
0x0042F6D7, 0x0042F6E2
Disassemble x86
instructions
Loop through
instructions
Find call eax
at 0x0042F6E2


Register Dependency Tracing
trace_register_dependencies
idx=2, target_reg=eax
tracked regs = eax
Walk backwards
idx=1: add eax,
dword ptr [0x49AEE4]


get_register_written
returns eax
Add to dependencies
get_registers_read
returns eax, [0x49AEE4]
tracked_regs = eax
idx=0: mov eax, 30BCh
get_register_written
returns eax
Add to dependencies
get_registers_read
returns empty
tracked_regs = empty


DONE
Return dependencies
mov + add


Unicorn Emulation
emulate_dependencies
deps, target_reg=eax
unicorn emulator
Execute: mov eax, 0x30BC
eax = 0x30BC
Execute: add eax,
dword ptr [0x49AEE4]
read_dword 0x49AEE4
returns 0x00450464
eax = 0x30BC + 0x00450464
= 0x00453520

Reads a 4-byte integer from a virtual address by finding the correct section and unpacking the bytes.

```
def read_dword(self, va):
       for s in self.sections.values():
           if s['start'] <= va < s['end']:
               off = va - s['start']
               if off + 4 <= len(s['data']):
                   return struct.unpack('<I', s['data'][off:off+4])[0]
       return None
```

Extracts raw bytes from a specified virtual address range within the binary’s sections.

```
def get_code_range(self, start_va, end_va):
   for s in self.sections.values():
       if s['start'] <= start_va < s['end']:
           return bytes(s['data'][start_va - s['start']:end_va - s['start']])
   return None
```

Returns the set of registers that are read by an instruction (excluding destination registers for mov/lea).

```
def get_registers_read(self, inst):
   regs = set()
   start = 1 if inst.mnemonic in ['mov', 'lea'] else 0
   for i, op in enumerate(inst.operands):
       if i >= start or inst.mnemonic in ['cmp', 'test', 'sub', 'add', 'xor', 'or', 'and']:
           if op.type == X86_OP_REG:
               regs.add(op.reg)
           elif op.type == X86_OP_MEM:
               if op.mem.base: regs.add(op.mem.base)
               if op.mem.index: regs.add(op.mem.index)
   return regs
```

Walks backward through instructions to find all instructions that contribute to computing the value of a target register.

```
def trace_dependencies(self, instructions, target_idx, target_reg, max_lookback=100):
   deps, tracked, seen = [], {target_reg}, set()
   for idx in range(target_idx - 1, max(0, target_idx - max_lookback), -1):
       inst = instructions[idx]
       if inst.mnemonic in ['jmp', 'je', 'jne', 'jz', 'jnz', 'jg', 'jl', 'jge', 'jle', 'ja', 'jb', 'jae', 'jbe
```

```
            continue
        if inst.operands and inst.operands[0].type == X86_OP_REG and inst.operands[0].reg in tracked and \
            inst.mnemonic in ['mov', 'lea', 'add', 'sub', 'xor', 'or', 'and', 'shl', 'shr', 'sar', 'imul', 'mul
            deps.insert(0, inst)
            seen.add(inst.address)
            tracked.remove(inst.operands[0].reg)
            tracked.update(self.get_registers_read(inst))
        if not tracked: break
    return deps
```

Uses Unicorn Engine to execute the dependency chain and resolve the final value of the target register.

```
def emulate_dependencies(self, deps, target_reg):
    if not deps: return None
    mu = Uc(UC_ARCH_X86, UC_MODE_32)
    code_base, stack_base = 0x1000000, 0x2000000
    mu.mem_map(code_base, 0x10000)
    mu.mem_map(stack_base, 0x10000)
    mu.reg_write(UC_X86_REG_ESP, stack_base + 0x8000)

    for s in self.sections.values():
        try:
            mu.mem_map(s['start'], ((len(s['data']) + 0xFFF) // 0x1000) * 0x1000)
            mu.mem_write(s['start'], bytes(s['data']))
        except: pass
```

Disassembles a code range and finds all indirect calls/jumps, then traces and resolves their target addresses.

```
def analyze_range(self, start_va, end_va):
        code = self.get_code_range(start_va, end_va)
        if not code:
            print(f"Failed to read code range 0x{start_va:08x} - 0x{end_va:08x}")
            return
        instructions = list(self.cs.disasm(code, start_va))
        for idx, inst in enumerate(instructions):
            if inst.mnemonic in ["call", "jmp"] and inst.operands and inst.operands[0].type == X86_OP_REG:
                target_reg = self.cs.reg_name(inst.operands[0].reg)
                print(f"\n[+] Analyzing: 0x{inst.address:08x}: {inst.mnemonic} {target_reg}")
                deps = self.trace_dependencies(instructions, idx, inst.operands[0].reg)
                target_value = self.emulate_dependencies(deps, inst.operands[0].reg)
                if target_value is not None:
                    print(f"\n[*] Target resolved: 0x{target_value:08x}\n")
```

Output

```
------------------------------------------------------------------------------------------------------------------------------------
ADDRESS      INSTRUCTION                                EAX        ECX        EDX        EBX        ESP        EBP        ESI        EDI
------------------------------------------------------------------------------------------------------------------------------------
0x0042f6d7  mov eax, 0x30bc                             00000000   00000000   00000000   00000000   02008000   00000000   00000000   00000000
0x0042f6dc  add eax, dword ptr [0x49aee4]               000030bc   00000000   00000000   00000000   02008000   00000000   00000000   00000000
------------------------------------------------------------------------------------------------------------------------------------

[*] Target resolved: 0x00453520


[+] Analyzing: 0x0042f707: call eax
------------------------------------------------------------------------------------------------------------------------------------
ADDRESS      INSTRUCTION                                EAX        ECX        EDX        EBX        ESP        EBP        ESI        EDI
------------------------------------------------------------------------------------------------------------------------------------
0x0042f6fb  mov eax, 0x3714                             00000000   00000000   00000000   00000000   02008000   00000000   00000000   00000000
0x0042f700  add eax, dword ptr [0x49b1fc]               00003714   00000000   00000000   00000000   02008000   00000000   00000000   00000000
------------------------------------------------------------------------------------------------------------------------------------

[*] Target resolved: 0x00403937


[+] Analyzing: 0x0042f72b: call edx
------------------------------------------------------------------------------------------------------------------------------------
ADDRESS      INSTRUCTION                                EAX        ECX        EDX        EBX        ESP        EBP        ESI        EDI
------------------------------------------------------------------------------------------------------------------------------------
0x0042f717  mov edx, 0xffffca53                         00000000   00000000   00000000   00000000   02008000   00000000   00000000   00000000
0x0042f71c  add edx, dword ptr [0x49a268]               00000000   00000000   ffffca53   00000000   02008000   00000000   00000000   00000000
------------------------------------------------------------------------------------------------------------------------------------

[*] Target resolved: 0x004038ec


[+] Analyzing: 0x0042f75a: jmp ecx
------------------------------------------------------------------------------------------------------------------------------------
ADDRESS      INSTRUCTION                                EAX        ECX        EDX        EBX        ESP        EBP        ESI        EDI
------------------------------------------------------------------------------------------------------------------------------------
0x0042f741  mov edx, 0x154                              00000000   00000000   00000000   00000000   02008000   00000000   00000000   00000000
0x0042f74e  mov ecx, dword ptr [edx + 0x4998b0]         00000000   00000000   00000154   00000000   02008000   00000000   00000000   00000000
0x0042f754  add ecx, dword ptr [edx + 0x49a400]         00000000   0042edfe   00000154   00000000   02008000   00000000   00000000   00000000
------------------------------------------------------------------------------------------------------------------------------------

[*] Target resolved: 0x0042f760
```

full script can be found here

https://github.com/01Xyris/RE-Malware/blob/main/AuraStealer/find_cff.py

Now we just need to patch this in the binary. takes the last instruction before the call and patches it with # `mov reg, calculated_address` example

```
 0x0042F6D7: mov eax, 30BCh
0x0042F6DC: add eax, dword ptr [0x49AEE4]
0x0042F6E2: call eax
```

after

```
 0x0042F6D7: mov eax, 30BCh
0x0042F6DC: mov eax, 0x00453520 ; calculated_address
            nop                 ; nop padding so size is the same
0x0042F6E2: call eax
```

```
int __stdcall WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nShowCmd)
{
  int v4; // eax
  _DWORD v6[164]; // [esp+120h] [ebp-75Ch] BYREF
  int v7; // [esp+3B0h] [ebp-4CCh] BYREF

  sub_453520();
  v6[0] = 1335530668;
  v4 = sub_403937(v6);
  sub_4038EC((char *)off_49B2A0 + 493, &v7, v6, v4);
  JUMPOUT(0x42F760);
}
```

The full deobfuscation script can be found here

https://github.com/01Xyris/RE-Malware/blob/main/AuraStealer/aura_patch_obf.py

```
AURA STEALER DEOBFUSCATION COMPLETE
PATCHED 3451 INDIRECT CALLS/JUMPS ʕ/ˋ。ˎ。´\
Patched binary saved to: patched_target.exe
Analysis complete.
```

Source: https://blog.xyris.mov/posts/aura-stealer-%232-beatin-the-obfuscation/