



Scripting IDA Debugger to Deobfuscate Nymaim

Author:	GovCERT.ch
Version:	v1.00
Document Typ:	White Paper
Document Title:	Scripting IDA Debugger to Deobfuscate Nymaim
Source:	www.govcert.ch

Table of Contents

Summary	2
Recap of Main Nymaim Obfuscation for CALLs	3
Obfuscation of JMPs	5
The Hidden Second Argument	6
Non-constant Arguments...?	7
An Obfuscator Obfuscates an Obfuscator, which Obfuscates an Obfuscator,	8
Deobfuscator: Finding the Obfuscation Functions	9
The Action Starts... Debugger Scripting	12
API Calls	15
Remaining Obfuscation Techniques	19

Summary

Nymaim is active worldwide since at least 2013 and is also responsible for many infections in Switzerland. Sinkhole Data shows that Nymaim is responsible for about 2% of infected devices¹ in Switzerland that hit sinkholes the last few days. Nymaim uses powerful code obfuscation techniques. These techniques have already been discussed several times. Many approaches use code emulation. We'd like to present an approach in this paper to do so by directly using IDA's debugger feature and IDAPython to do the same, as it might be the more generic approach in certain cases. Also, we follow a slightly different approach to actually find all the obfuscation functions, and make the deobfuscation a bit more generic. No additional Python modules are required.

¹ <https://www.govcert.admin.ch/statistics/drone/>

Recap of Main Nymaim Obfuscation for CALLs

We're looking at the sample with MD5 22cb5ad102b418a7f7078e679ef68b66. To get rid of the first packer layer, the easiest procedure for most samples seems to be to just set a breakpoint to `GlobalFree` and wait until the region to be freed points to a PE file, which can then be dumped to disk. For some samples, an initial endless loop causing exceptions need to be bypassed manually by toggling the Z flag.

The resulting unpacked binary is then loaded into IDA. IDA's navigation bar shows that a lot of code is unreferenced (brown parts), which is a direct effect of the main code obfuscation technique of Nymaim:



The obfuscation technique has already been described in several publications, here are some good papers on it (just a selection, there are out more):

- <http://www.botconf.eu/wp-content/uploads/2016/11/PR18-Nymaim-ORTEGA.pdf>
- <http://www.seculert.com/blogs/nymaim-deep-technical-dive-adventures-in-evasive-malware>
- <http://www.welivesecurity.com/2013/08/26/nymaim-obfuscation-chronicles/>
- https://bitbucket.org/daniel_plohmann/idapatchwork is a project to deobfuscate Nymaim using PyEmu

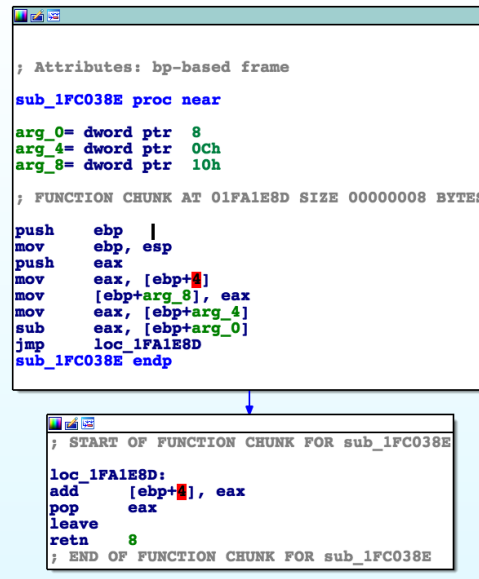
Why using a debugger approach instead of emulation? Of course both work, and the approaches are similar. Emulation does not require a full PE file, which is a big advantage. The debugger approach does not require any additional Python modules, and might get along better with very special code or instructions; it also works for resolving Windows API calls, which is more difficult to do in an emulator (without emulating a full OS), but naturally it only works in a Windows environment. Also, debugger scripts should only be executed in a VM setup where snapshots can be replayed, because there is an infection risk. In the case of Nymaim, there is no real additional advantage for the debugger approach (except for resolving API calls), but it is a good case study to try it out.

We'll start with a short recap how the main code obfuscation technique of Nymaim works. Most of this is already described elsewhere (except maybe the part about non-constant arguments below), but we'll repeat it here for consistency reasons.

The following fragment shows a code fragment, where a CALL target is obfuscated:

<code>:01FA16B5 6A FF</code>	<code>push</code>	<code>0FFFFFFFh</code>
<code>:01FA16B7 50</code>	<code>push</code>	<code>eax</code>
<code>:01FA16B8 68 04 1F 29 12</code>	<code>push</code>	<code>12291F04h</code>
<code>:01FA16BD 68 A9 5A 27 12</code>	<code>push</code>	<code>12275AA9h</code>
<code>:01FA16C2 E8 C7 EC 01 00</code>	<code>call</code>	<code>sub_1FC038E</code>

The first PUSH at address 01FA16B5h is just an argument for the final function. Then we see a "PUSH EAX", which is just a placeholder on the stack – as we'll see in the code below, the obfuscation code only takes 2 arguments (it ends in a "RETN 8"), so that PUSH seems redundant; however, it will be overwritten with the return address by the obfuscation code. The next 2 PUSHs of constants are the actual parameters for the obfuscator; they are used to calculate the final address of the function to be called. The obfuscator at address 1FC038Eh looks like this:



```

; Attributes: bp-based frame
sub_1FC038E proc near
arg_0= dword ptr 8
arg_4= dword ptr 0Ch
arg_8= dword ptr 10h
; FUNCTION CHUNK AT 01FA1E8D SIZE 00000008 BYTES
push    ebp
mov     ebp, esp
push    eax
mov     eax, [ebp+4]
mov     [ebp+arg_8], eax
mov     eax, [ebp+arg_4]
sub     eax, [ebp+arg_0]
jmp     loc_1FA1E8D
sub_1FC038E endp

; START OF FUNCTION CHUNK FOR sub_1FC038E
loc_1FA1E8D:
add     [ebp+4], eax
pop     eax
leave
retn    8
; END OF FUNCTION CHUNK FOR sub_1FC038E

```

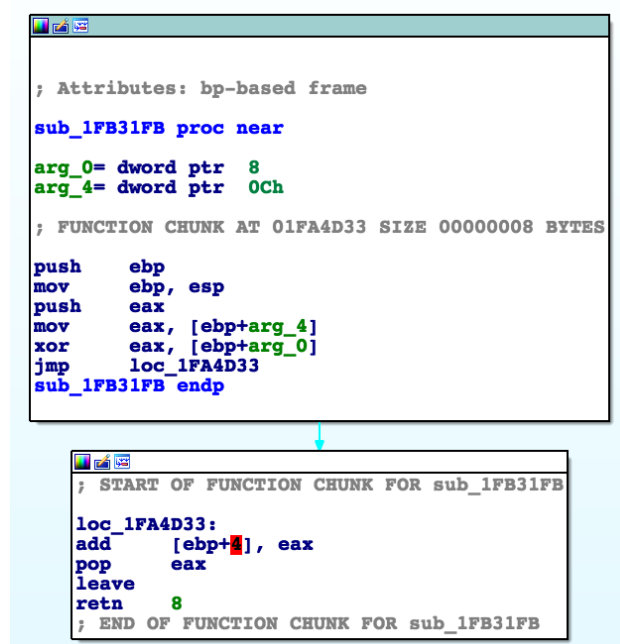
Please note the final “RETN 8”, which suggests the code is consuming 2 arguments from the stack (the 2 most recent PUSHs). After the standard function prolog, the code first reads [EBP+4] into EAX, which is the return address to the calling code before. This value is then stored into [EBP+10h] (or “arg_8” – it looks like a third argument, but of course that’s a misinterpretation of IDA), which is the placeholder mentioned before. The 2 constants pushed by the caller are now used for some arithmetic operation, in this case a subtraction, and then added to the return address – an instruction, which **overwrites [EBP+4]** – this instruction is actually the trigger instruction we will search for in our deobfuscator. The result of this – the **first element on the stack before the RETN instruction is executed** – is the calculated address for the final function, and “RETN 8” actually jumps to it. “RETN 8” also removes the 2 constants from the stack, so the final function will now find the actual return address on the stack – the placeholder mentioned above – followed by the arguments (in this case FFFFFFFFh).

Obfuscation of JMPs

There is also another variation of the obfuscator, as the following calling code shows:

```
.code:01FB7440 89 18          mov     [eax], ebx
.code:01FB7442 68 94 07 C7 60        push    60C70794h
.code:01FB7447 68 E3 1F C6 60        push    60C61FE3h
.code:01FB744C E8 AA BD FF FF    call    sub_1FB31FB
.code:01FB7451 55                push    ebp
.code:01FB7452 89 E5          mov     ebp, esp
```

The main difference is the lack of the placeholder on the stack, also we don't see any arguments. A more subtle difference is that the code immediately behind the CALL ("PUSH EBP", ...) looks like a function prolog of another function, which is rather surprising after an actual CALL. Now let's look at the obfuscator:



This code is very similar to the previous obfuscator. One obvious difference is that an XOR instruction is used instead of a SUB. But there's another difference: *the placeholder is not written back* – which naturally would be hard to do, as the caller did not push any placeholder. The explanation is quite easy: in this case, not a CALL, but a JMP is obfuscated, so the obfuscator CALL is not returning. This also explains why the code behind the initial CALL looks like a function prolog – because that's what it is. Our deobfuscator must actually detect this situation and take care of this. We do this by scanning the final RETN instruction (here "RETN 8"), which suggests 2 arguments, and check if the code tries to write anything into an argument "above" it (i.e., with an offset of 8 bytes more than the RETN offset, taking return address and stored EBP into calculation); this is also a difference to the emulation approaches, and a generalization, as it would also work with more than 2 arguments.

A simple approach to deobfuscate such calls would be to find the obfuscation functions, analyze them (it's mostly XOR, ADD or SUB), get all XREFS, and patch back the calling code, NOP-ing out what's no longer required. Unfortunately, that's a lot of work: Our sample has more than 60 variations of the obfuscation code, and they must all be found and analyzed. Our goal is to make this more efficient and generic by not analyzing these calls at all, but just call them in an automated way via IDA debugger. Our first idea was to use IDA's **Appcall** feature, which allows to call functions from Python and is a great and underestimated tool.

Unfortunately, there is a variation of the obfuscation functions that makes this hard to achieve.

The Hidden Second Argument

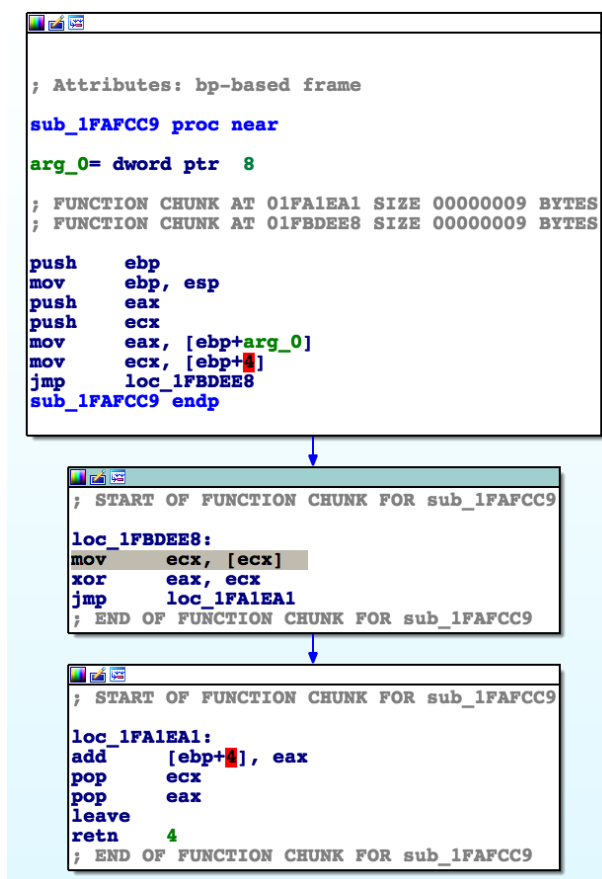
Some locations where an obfuscator is called look strange – here is an example:

```

:01FAF56B E8 C1 45 01 00      call     sub_1FC3B31
:01FAF570 FF B6 F4 00 00+          push     dword ptr [esi+0F4h]
:01FAF576 FF B6 F0 00 00+          push     dword ptr [esi+0F0h]
:01FAF57C FF B6 E8 00 00+          push     dword ptr [esi+0E8h]
:01FAF582 68 E0 EC 1A 23          push     231AEC0h
:01FAF587 E8 3D 07 00 00      call     sub_1FAFCC9
:01FAF58C 5C                          pop      esp
:01FAF58C                          ; END OF FUNCTION CHUNK FOR sub_1FA85D0
:01FAF58C                          ; -----
:01FAF58D FE 1A 23              db 0FEh, 1Ah, 23h
:01FAF590                          ; -----

```

Obviously this is again a JMP obfuscation (no placeholder pushed, and junk code behind the CALL), but we see only one constant pushed to the stack. Where is the second argument? Or is there none? Let's look at the obfuscator code:



This looks once more very similar. The relevant difference is the “MOV ECX, [ECX]” instruction; ECX contains the return address and points to the seeming chunk code behind the CALL, actually to the byte sequence “5C FE 1A 23” or 0x231afe5c. This is an alternative way to feed the obfuscator with the second argument for the calculation (here an XOR).

Unfortunately, this makes the use of IDA's Appcall hard, as this second argument is not passed to the obfuscator by a standard calling convention. We would have to write wrapper code to facilitate Appcall. So we decided not to use Appcall at all, but standard breakpoints – more to this below. Before, there are 2 additional tricky cases of the obfuscator code...

Non-constant Arguments...?

This is a rather rare case, and we're not sure if it is actually dealt with by existing deobfuscators, because the way these functions are called does not match the usual pattern:

```

:01FA58C1 E8 D9 CD 00 00      call     sub_1FB269F
:01FA58C6 FF 75 08              push    dword ptr [ebp+8]
:01FA58C9 56                      push    esi
:01FA58CA 53                      push    ebx
:01FA58CB FF 75 F0              push    dword ptr [ebp-10h]
:01FA58CE E8 30 EC 00 00      call     sub_1FB4503
:01FA58D3 3D 4E 00 00 C0          cmp     eax, 0C000004Eh

```

No constant values are pushed to the stack at all. As the obfuscator code below suggests, the “PUSH ESI” is the placeholder (so it is a CALL obfuscation), “PUSH EBX” a (dummy) second argument, and “PUSH DWORD PTR [EBP-10h]” the first and actually only non-constant argument for the obfuscator. The “PUSH DWORD PTR [EBP-8]” at the very start is the argument for the final function. Now let's look at the obfuscator code:

```

; Attributes: bp-based frame
sub_1FB4503 proc near
arg_0= dword ptr 8
arg_8= dword ptr 10h
; FUNCTION CHUNK AT 01FA28DA SIZE 00000008 BYTES

push    ebp
mov     ebp, esp
push    eax
mov     eax, [ebp+8]
mov     [ebp+arg_8], eax
mov     eax, [ebp+arg_0]
jmp     loc_1FA28DA
sub_1FB4503 endp

; START OF FUNCTION CHUNK FOR sub_1FB4503
loc_1FA28DA:
mov     [ebp+8], eax
pop     eax
leave
retn    8
; END OF FUNCTION CHUNK FOR sub_1FB4503

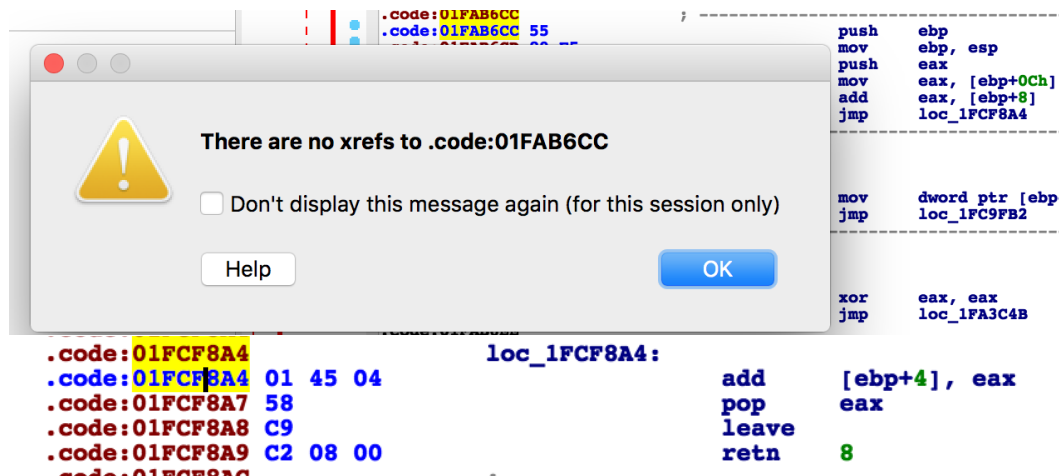
```

As you can see, this code completely ignores the second argument (“PUSH EBX” above) and just copies the first argument into EAX, without any arithmetic operation. So it finally obfuscates a “CALL DWORD PTR [EBP-10h]” – this is also the code that needs to be patched in the caller.

The bad news for this case is that it can't be treated correctly via debugger (or emulation) instrumentation, because the arguments are not constant. The good news is that there are only very few of these cases - maybe a dozen or so – and they all use the same obfuscator code (though via several copies). We decided to assume this to be the rule for these cases, and to just output warnings, so they can be verified manually. This actually is the only case where the deobfuscator possibly requires manual intervention.

An Obfuscator Obfuscates an Obfuscator, which Obfuscates an Obfuscator, ...

There is another strange situation – look at this obfuscation code. It obviously is one, adding its arguments; the code jumped to is shown just below. But.... there are no cross references to it:



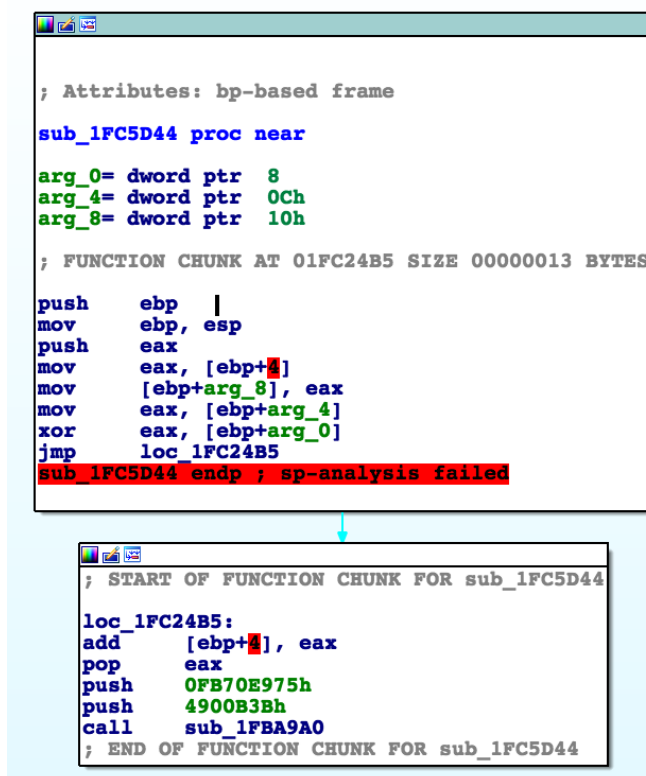
Is this just some orphaned code? But wait, here is another funny situation:

```

:01FA138A E8 10 13 01 00      call    sub_1FB269F
:01FA138F 53                      push    ebx
:01FA1390 68 08 8B E3 5C          push    5CE38B08h
:01FA1395 68 26 96 E2 5C          push    5CE29626h
:01FA139A E8 A5 49 02 00      call    sub_1FC5D44
:01FA139F 83 F8 00              cmp     eax, 0

```

This calling code looks completely normal... but look at its obfuscator function:



There is no function epilog (“LEAVE” and “RETN 8” instructions)... Instead, we see a nested call to another obfuscator. In other words, the obfuscation can be recursive. This nesting also explains the orphaned code seen just before. Most probably, the CALL to that obfuscation code is obfuscated itself, though we can’t know that for sure yet.

Is this nesting, which looks really annoying, a huge problem? Well, actually it is not. It just means that the deobfuscation procedure needs be applied twice, or maybe more times – until no additional obfuscation calls are found anymore (but in all samples we checked, two iteration steps were sufficient). We assume that the obfuscator itself is also applied more than one time.

Deobfuscator: Finding the Obfuscation Functions

Now, how can an automatic deobfuscator deal with this? Our approach is to first find all obfuscation functions present in the code, using IDAPython. We decided not to search for the code fragments that call the obfuscators, but for the obfuscation functions themselves. Then we follow their cross references. We also prefer to not scan for certain structures or instruction sequences in the obfuscation calls – we can’t know if this might at some point include permutations or PRNGs functions instead of simple add, sub or xor. So we tried to find the smallest common denominator. To us, a very good candidate seems to be the *instruction towards the end of these functions that writes to [EBP+4]*. [EBP+4] is the return address on the stack in any function that uses a standard stack frame (one assumption we do make). Normal code should never ever overwrite its return address; any instruction doing this smells like obfuscation code.

Here is the Python code we use to find all these instructions (it’s embedded into a Nymaim class):

```
def scanManglerInstructions(self):
    for sgm in Segments():
        for head in Heads(sgm, SegEnd(sgm)):
            if isCode(GetFlags(head)):

                inst = DecodeInstruction(head) # object-oriented disassembly engine

                # Make sure we actually have 2 operands
                if inst.Op2.type == 0:
                    continue

                # Make sure we write something into [ebp+4]
                if inst.Op1.type != idaapi.o_displ or \
                   inst.Op1.specflag1 != 0 or \
                   inst.Op1.reg != procregs.ebp.reg or \
                   inst.Op1.addr != 4:
                    continue

                self.manglerInstructions.append(head)
```

Note that we decided to use the *object oriented disassembly interface* (`DecodeInstruction`) instead of the text oriented one (`GetDisasm`), as this allows to allows us to replace regex and other string operations by direct property accesses. In our opinion, this makes the code cleaner and safer. An even better approach would be to use the *Python Sark module* to write more readable code (instead of accessing `specflag1` literally, which basically is the

SIB byte), but we decided against using third party modules in this case study. Using Sark, the code looks cleaner:

```
def scanManglerInstructions(self):
    for sgm in Segments():
        for head in Heads(sgm, SegEnd(sgm)):
            if isCode(GetFlags(head)):
                inst = sark.instruction.Instruction(head) # Sark disassembly engine

                # Make sure we actually have 2 operands
                if len(inst.operands)<2:
                    continue

                # Make sure we write something into [ebp+4]
                opl = inst.operands[0]
                if opl.base != "ebp" or opl.index is not None or opl.displacement!=4:
                    continue

                self.manglerInstructions.append(head)
```

Of course we could also use other disassembly engines, like Capstone or Distorm3. Anyway, the code checks for instructions with 2 operands, where the first is [EBP+4], which is what we want.

After we have all these instructions, we must make sure they are actually followed by the expected function epilog:

```
for head in self.manglerInstructions:
    # Make sure only pop and leave instructions follow until retn (and maybe nop + jmp):
    nextHead = head
    epilogOK = None

    while epilogOK is None:
        nextHead = Rfirst(nextHead) # next instruction in flow (this follows jmp's)
        inst = DecodeInstruction(nextHead)
        if inst.itype in (idaapi.NN_jump, idaapi.NN_leave, idaapi.NN_pop, idaapi.NN_nop):
            continue
        epilogOK = (inst.itype == idaapi.NN_retn)

    if not epilogOK:
        continue

    retnAddr = inst.ip # address of retn address
    retnCorr = 0 # stack correction (number of bytes for args)

    if inst.Opl.type == idaapi.o_imm:
        retnCorr = inst.Opl.value
        if retnCorr % 4 != 0:
            print "WARNING: %08x %s - strange stack correction 4"%(head, GetDisasm(head))
            nbrArgs = retnCorr >> 2

    print "%08x: %s, returns at %08x, %d args" % (head, GetDisasm(head), retnAddr, nbrArgs)
```

Note we follow the code using “Rfirst”, as this follows JMP instructions (Nymaim’s code is often chunked). We also allow NOP instructions; they are inserted by our deobfuscator, so this allows additional deobfuscator iterations to take place. We also see how useful the object-oriented interface for the disassembling engine is in this case. Also note that we do ex-

tract the first operand of the RETN instruction; as the obfuscator calls seem to use stdcall convention (another assumption we make), this value tells us how many arguments the function expects – in Nymaim, it usually is 4 or 8 (1 or 2 arguments). We try to be a bit more generic in allowing more than 2 arguments by evaluating and memorizing this value. So we know how many PUSHs we need to see in the calls (and NOP them out), and we can find out if it is a CALL or JMP obfuscation.

We must also find the function start. We could rely on IDA's own mechanism to find it, but we decided to scan all instructions back in an iterative manner. This way we can also see if the function ever writes anything back to the stack "above" the expected arguments (i.e. the possible placeholder), which suggests the code is a CALL obfuscator and not a JMP obfuscator; note that we know the number of expected arguments from the RETN instruction ("nbrArgs" above):

```
fctAddress = BADADDR # function start, as soon as we find it
# Parse instructions back (to see if this hides a call or a jmp):
prevs = [[head]] # instructions still to scan (a todo queue)
# Note: this is actually backtrack search until we see a CALL.
#       We store instruction chains in the list.
seen = [head] # to avoid cycles
stopSearch = False
isCall = False # set, if we see a write "above" the actual arguments

while (not stopSearch) and prevs:
    chain = prevs.pop(0)
    thisHead = chain[0]
    inst = DecodeInstruction(thisHead)

    # If we see a store to the stack element below the args, this is the placeholder
    if inst.Op2.type!=0 and inst.Op1.type==idaapi.o_displ and inst.Op1.specflag1 == 0 and \
        inst.Op1.reg == procregs.ebp.reg and inst.Op1.addr == retnCorr + 8:
        isCall = True

    if inst.itype == idaapi.NN_call: # We assume that obfuscation calls do not contain calls
        continue # of course, in a next deobfuscation iteration, this might disappear

    # A push ebp means we found the function start:
    if inst.itype==idaapi.NN_push and inst.Op1.type==idaapi.o_reg and \
        inst.Op1.reg==procregs.ebp.reg:

        fctAddress = inst.ip
        stopSearch = True

    prevHead = RfirstB(thisHead) # This again follows back Jumps - there might be several ones:
    while prevHead != BADADDR:
        if prevHead not in seen: # avoid cycles
            prevs.append([prevHead] + chain)
            seen.append(prevHead)
        prevHead = RnextB(thisHead, prevHead)

    if not prevs: # no more instructions to search, and no function start found
        break

if fctAddress != BADADDR:
    self.manglerFunctions[fctAddress] = (retnAddr, nbrArgs, isCall, chain)
```

Now we have collected all obfuscators and their relevant information:

- Address of the function, which allows to follow cross references
- Address of RETN instruction – here we'll set a breakpoint in order to read the final target from the stack
- The number of arguments to the obfuscator, and so the number of preceding PUSHs in the cross references we need to NOP out
- Whether we need to patch in a CALL or a JMP
- And (for debug purpose) the chain of instructions in the obfuscator

Note that this generic search algorithm also allowed us to find the unusual variant with the non-constant arguments.

The Action Starts... Debugger Scripting

Now we need to start the debugger to resolve all calls to these obfuscators:

```
LoadDebugger('win32',0)
SetDebuggerOptions(DOPT_ENTRY_BPT)
if StartDebugger("", "", "") == -1:
    return
```

We just make sure the debugger stops at the entry point, and start it. Now we iterate through the functions found above (and write a log file):

```
resolves = [] # contains information we will patch at the very end

with open("nymiam.txt", 'wb') as f:
    for (fct,v) in nymaim.manglerFunctions.iteritems():
        (retn,args,isCall,chain) = v
        f.write("\n-----\n%08x, ret at %08x, %d args, %s\n" % (fct,retn,args,
                                                                "CALL" if isCall else "JMP"))

        for head in chain:
            f.write(" %08x: %s\n" % (head, GetDisasm(head)))
        caller = RfirstB(fct)
        while caller != 0xffffffff:
            ...
```

And iterate all calls to it – we follow the preceding instructions back for the number of PUSHs to know what all we need to patch (one more PUSH in the case of CALLs for the placeholder):

```
f.write("XREF: %08x %s\n" % (caller, GetDisasm(caller)))
head = caller
nbrInst = args # number of PUSH instructions we need to NOP out
if isCall:
    nbrInst += 1
argsAreImmediate = True # To detect the nasty cases with non-constant PUSHs
idxStorePush = 1 if isCall else 0 # this push is allowed to be non-immediate
itype = "CALL" if isCall else "JMP"
firstPushInstr = None

while nbrInst:
    head = PrevHead(head) # Hopefully this prefix code is not chunked...
    inst = DecodeInstruction(head)
    if inst.itype == idaapi.NN_push and firstPushInstr is None:
        # Memorize first push (in some cases, this must be replaced by a call)
        firstPushInstr = GetDisasm(head)
```

```

    if nbrInst>idxStorePush and inst.itype==idaapi.NN_push and inst.Opl.type!=idaapi.o_imm:
        argsAreImmediate = False

    f.write("        %08x %s\n" % (head, GetDisasm(head)))
    startAddr = head
    if inst.itype != idaapi.NN_nop: # count PUSHs
        nbrInst -= 1

if argsAreImmediate:
    resolved = nymaim.resolve(startAddr, retn)
    f.write("        ==> %08x resolves to %s %08x\n" % (startAddr, itype, resolved))
    resolves.append([startAddr, caller, resolved, isCall, firstPushInstr])
else:
    f.write("WARNING: no immediate arguments at %08x - will assume pushing arg1"%startAddr)
    resolves.append([startAddr, caller, None, isCall, firstPushInstr])

caller = RnextB(fct, caller)

```

Note that we store for every patch required the first and last address of the code fragment to be patched later on (as assume this fragment is not itself chunked, otherwise the code has to be changed a bit), and the final address determined by the debugger (or “None” in the case the first argument is just the CALL target), whether we need to patch a JMP or CALL, and the actual first argument for our special case. What remains is the actual resolver – this is where the action happens, and it is rather simple:

```

def resolve(self, startAddr, endAddr):
    print "Resolving from %08x to %08x"%(startAddr,endAddr)
    AddBptEx(endAddr, 1, BPT_EXEC)
    SetRegValue(startAddr, "EIP")
    GetDebuggerEvent(WFNE_SUSP|WFNE_CONT,-1)
    val = DbgDword(GetRegValue("ESP"))
    DelBpt(endAddr)
    return val

```

At the very end, the patches are actually applied – unfortunately we do need to do some string fixes when re-assembling the CALL code for the non-constant argument case mentioned above. This is because IDA skips “DWORD PTR” in indirect memory operands in its disassembly engine, but expects it in the assembly engine – that’s the ugliness of the text-oriented interface (combined with incompatible assembly and disassembly engines in IDA), but here there’s no easy alternative to this, **sigh**:

```

StopDebugger()
pcnt1, pcnt2 = 0,0 # counters

for addr,call,target,isCall,firstPushInstr in resolves:

    opcode = "call" if isCall else "jmp"

    if target is None:
        # Special case, call to first argument which is not a constant, requires re-assembly
        cinstr = "" # new call instruction, will be assembled later on
        print "first-arg-patch at %08x-%08x with opcode %s"%(addr,call+5,opcode)

        # Ugly dword ptr fix
        if firstPushInstr.find("[")>0 and firstPushInstr.find("dword ptr")<0:
            cinstr = firstPushInstr.replace("push","%s dword ptr"%opcode)

```

```

else:
    cinstr = firstPushInstr.replace("push",opcode)

    ok, code = Assemble(addr, cinstr)

    if ok: # NOP out rest
        code += "\x90" * (call+5-addr-len(code))
    else:
        print "ERROR: Cannot assemble '%s' on %08x" % (firstPushInstr, addr)

    for c in code:
        PatchByte(addr, ord(c))
        addr += 1

    pcnt1 += 1

else:
    # Normal code patch
    print "normal    patch at %08x-%08x for %80x with opcode %s" % \
        (addr,call+5,target,opcode)

    while addr < call: # NOP out all before actual call
        PatchByte(addr,0x90)
        addr += 1

    # And just change the call target address
    PatchDword(call+1, (target - (call+5)) & 0xffffffff)

    if not isCall: # in this case, also change the opcode to JMP opcode:
        PatchByte(call, 0xe9)

    pcnt2 += 1
print "%d normal patches, %d first-arg-patches applied" % (pcnt2,pcnt1)
print "Apply patches to binary, reload, and re-apply script until no more patches are made"

```

That's it. In our sample 2508 places were patched (plus 23 first-arg ones). Surprisingly, it only takes about a minute to execute everything. The blog shows a funny to watch animated GIF how it actually looks – while it seems to be quite slow, only about every 20th call is actually animated.

The approach can of course also be done using an emulator. We modified the script a bit and tried it using the Unicorn engine, as a change to PyEmu. This requires an additional initialization step, where the whole IDA database is copied into the emulator and a stack is created, which we put in the constructor of our class:

```

if useUnicorn:
    maxMem = 0
    self.mu = Uc(UC_ARCH_X86, UC_MODE_32) # instantiates the emulator
    for s in Segments():
        s,e = SegStart(s), SegEnd(s)
        size = e-s
        if size%4096 > 0: # a little precaution (probably not necessary)
            size = (size/4096 + 1)*4096
        self.mu.mem_map(s, size)
        self.mu.mem_write(s, GetManyBytes(s, e-s))
        maxMem = s+size # Unused memory for the stack

    # Allocate some stack memory:
    self.mu.mem_map(maxMem, 81920)
    self.stackPtr = maxMem+80000

```

```
self.mu.reg_write(UC_X86_REG_ESP, self.stackPtr)
```

Of course we don't need to start the debugger at all, and the resolve function changes to:

```
def resolve(self, startAddr, endAddr):
    if self.useUnicorn:
        self.mu.emu_start(startAddr, endAddr)
        val, = struct.unpack("<I", self.mu.mem_read(self.mu.reg_read(UC_X86_REG_ESP), 4))
        return val
    else:
        ... # Code shown above
```

Whatever we choose, debugger or emulator, we're not done yet, this was just the first iteration. Unfortunately, there lacks a clean way in IDA to re-analyze everything, and this would not really help for the second debugger run anyway, because the debugger needs a patched executable. But we can use IDA's "Apply patches to input file" feature in the "Edit" / "Patch program" submenu to actually apply the patches to the binary (Output: "Applied 35341/35341 patch(es)"). After this, IDA must be restarted using this binary, and the process must be repeated. In our case, the second run patches about 218 more places (plus 13 first-arg ones), making a total of 2762.

The final navigation bar now also shows much more of blue:



API Calls

Nymaim contains similar obfuscation for **API calls**. It is not trivial resolving them automatically using an emulator, because the actual resolving of the calls often includes scanning through all loaded libraries (often using the PEB of the process), which is information not really available in an emulator, except if a full operating system is emulated. In these cases, the actual debugger interface becomes vital.

Most of the API calls in Nymaim are calls to a proxy function, that just pushes one constant onto the stack, then jumps to another function, which pushes another constant to the stack and finally calls the main resolver-and-execution function. We use the following code (slightly simplified) to find these calls; the main loop fills the `found` array with the calling code, and `mainFcts` array with the resolver/execution function (there are 2 entry points into it, hence a list is required – in our sample, 20114B7h, which jumps to the second entry point 1FAF221h):

```
for head in Heads(sgm, SegEnd(sgm)):
    if isCode(GetFlags(head)):
        inst = DecodeInstruction(head)
        if inst is None or inst.itype != idaapi.NN_push or inst.Op1.type != idaapi.o_imm:
            continue
        scanHead = NextHead(head)
        if scanHead == BADADDR:
            continue
        inst = DecodeInstruction(scanHead)
        if inst is None or inst.itype != idaapi.NN_call or inst.Op1.type != idaapi.o_near:
            continue
        scanHead = inst.Op1.addr
        inst = DecodeInstruction(scanHead)
```

```

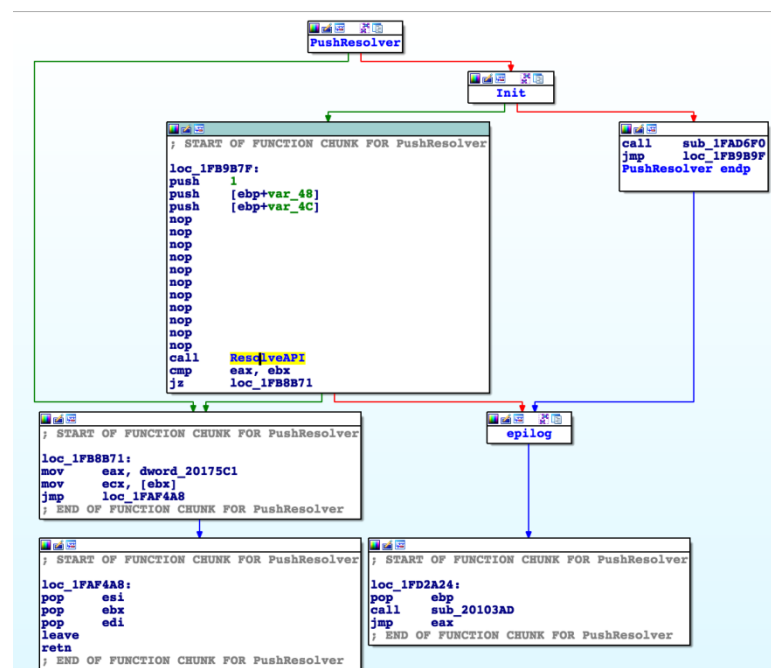
if inst is None or inst.itype != idaapi.NN_push or inst.Op1.type != idaapi.o_imm:
    continue
scanHead = NextHead(scanHead)
if scanHead == BADADDR:
    continue
inst = DecodeInstruction(scanHead)
if inst is None or inst.itype != idaapi.NN_call or inst.Op1.type != idaapi.o_near:
    continue
scanHead = inst.Op1.addr
inst = DecodeInstruction(scanHead)

if inst is None or (inst.itype == idaapi.NN_push and inst.Op1.type == idaapi.o_imm):
    continue

if scanHead not in mainFcts:
    mainFcts.append(scanHead)
    print "Main function: %08x"%scanHead
found.append(head)

```

Now we try to find the resolver function within this “push-resolver” code, which is calling the API function after having resolved it. We’d like to set the breakpoint in between the resolution and execution of the API call. The following graph shows the code (with some collapsed blocks to save space), where the relevant “call ResolveAPI” is highlighted. “ResolveAPI” is also called directly several times (i.e., not via the PUSH resolver), so we’ll also scan its cross references, and set a breakpoint on its return instruction.



Note the nop's are the result of the previous deobfuscation steps. We can use the “push 1” instruction as a fingerprint to actually find the right call behind, and then search for its retn instruction, using a state machine; at the end, `resolver` contains the retn instruction of the resolver; `resolverHead` is used to also find the direct calls (not going via push resolver):

```
queue = [mainFct]
state = 0
while queue:
    head = queue.pop()
    if head in seen:
```



```

        continue

    seen.append(head)
    inst = DecodeInstruction(head)
    if inst is None:
        continue
    for f in getFollowers(head):
        queue.append(f)

    # Condition 1: we must see a "push 1"
    if state==0 and inst.itype==idaapi.NN_push and inst.Op1.type==idaapi.o_imm and inst.Op1.value==1:
        state = 1

    # After this, we scan for the next call:
    elif state==1 and inst.itype==idaapi.NN_call:
        # In this function, we must search RETN, this is where we'll find the resolved address in EAX
        resolverHead = inst.Op1.addr
        state = 2
        queue = [resolverHead]
    elif state==2 and inst.itype == idaapi.NN_retn:
        resolver = inst.ip
        break

```

All calling locations are stored in a `callers` list, containing 4 entries (`startAddress`, `lengthToNopOutInBytes`, `DLLname`, `fcnName`); a length of 0 indicates actual proxy functions. The dynamic resolving code in debugger mode now looks like:

```

def resolveAPI(self):
    AddBptEx(self.resolver, 1, BPT_EXEC)
    SetDebuggerOptions(0)

    if StartDebugger("", "", "") == -1:
        return False

    memState = idatools.MemoryState() # separate package to resolve API addresses
    memState.updateSections()

    for entry in self.callers:
        startAddr = entry[0]
        SetRegValue(startAddr, "EIP")
        if GetDebuggerEvent(WFNE_SUSP|WFNE_CONT,-1) == 16:
            apiAddr = GetRegValue("EAX")
            symbol, dll = memState.lookup(apiAddr)
            if symbol is None: # In case new libraries were loaded meanwhile
                memState.updateSections()
                symbol, dll = memState.lookup(apiAddr)
            entry[2] = dll
            entry[3] = symbol
            print "%08x => %08x (%s, %s)"%(startAddr,apiAddr,dll,symbol)

    DelBpt(self.resolver)
    StopDebugger()

```

The code uses a private Python package (`idatools`) that allows to take memory snapshots of all PE files in the debugger memory, and then tries to resolve addresses to symbols in their export tables. One issue of the resolver is its dependency on certain tables calculated in an initialization step. So we just let the code run one time until the resolver breakpoint hits (which is usually a `NtReadVirtualMemory`) and the initialization happened. Without this, the resolver would not work properly. We just store the resolutions in a dictionary; in the actual

code, it is written out as a text file and then read back in, as it often needs some small manual edits due to caching issues (we don't show this code in the report, because it's lengthy and does not contain new information).

At the very end, the final patching can be done; all proxy functions are renamed accordingly. Direct calls are replaced by a call to an additional dummy function (just containing a return instruction), which is renamed accordingly:

```
def applyAPIPatches(self):
    if self.section is None:
        self.section = idatools.makeSegment(1024) # Here we create dummy API functions
    ptr = self.section

    translations = {} # to avoid double renames
    done = []

    for addr,size,dll,symbol in self callers:
        if size==0:
            if GetFunctionAttr(addr,FUNCATTR_START) == addr: # standard proxy functions
                MakeName(addr, symbol)
                translations[symbol] = addr
                done.append(addr)
                continue

    for addr,size,dll,symbol in self callers: # Embedded and direct API calls
        if addr in done:
            continue
        if size==0:
            # an embedded push # call within function:
            size = DecodeInstruction(addr).size
            scan = NextHead(addr)
            searching = True

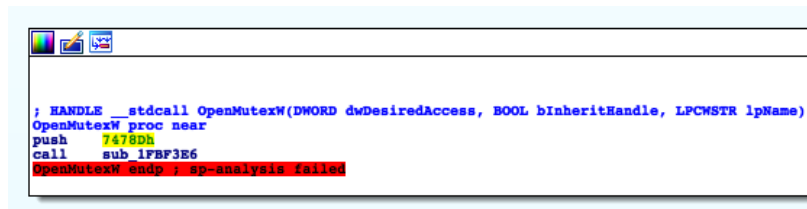
            while scan != BADADDR:
                inst = DecodeInstruction(scan)
                if inst is None:
                    break
                scan = NextHead(scan)
                size += inst.size
                if inst.itype != idaapi.NN_nop:
                    break
            if inst and inst.itype != idaapi.NN_call or size<5:
                continue

        if symbol not in translations: # Additional dummy function must be created
            PatchByte(ptr, 0xC9) # dummy function, a simple retn
            MakeFunction(ptr)
            MakeName(ptr, symbol)
            translations[symbol] = ptr

        PatchByte(addr, 0xE8) # Patch in a call instruction
        PatchDword(addr+1, (translations[symbol] - (addr+5)) & 0xffffffff)
        for i in range(5,size): # Nop out rest, if required
            PatchByte(addr+i, 0x90)
        AddCodeXref(addr, translations[symbol], fl_CN | XREF_USER)
        print "%08x: %s via %08x"%(addr,symbol,translations[symbol])
```

```
ptr += 1 # ready for next dummy function
```

Of course this does not produce a standalone PE file including correct IAT – this would be additional work, but could be done as well. But by renaming the functions correctly, IDA will apply the function prototypes and structure types. Ignoring the actual code. You won't see it in the import table, but have everything else, as this example shows:



Remaining Obfuscation Techniques

One remaining and still open issue is the fact the obfuscator also obfuscated the C library, and IDA is not able to apply it's FLIRT signatures successfully; even after the deobfuscation, the code is still different from the original, and the FLIRT signatures don't match. A BinDiff run might help here, but we did not actually try this out.

The registry-proxy functions (where a value is translated into pushing one of the registers) were easy to manually patch in our sample, as the values were just the required opcodes minus 0x20, and there's only such function. It could also be solved in a more generic debugger way: one individual value would initially be written into each of the registers (1 to 7, skipping ESP), and then checked which of these values is returned. Again, this is nothing new to what was discussed above.

Finally, there are a lot of short inline code pieces (like XOR-ing EAX with a constant), which are just a bit annoying. They can comparably easy be statically patched, if required. No debugger required in these cases.