Nymaim Malware: Deep Technical Dive – Adventures in Evasive Malware

arielkoren.com/blog/2016/11/02/nymaim-deep-technical-dive-adventures-in-evasive-malware/

AK

November 2, 2016

Nymaim is mostly known worldwide as a downloader, although it seems they evolved from former versions, now having new functionalities to obtain data on the machine with no need to download a new payload. Some of the exported functionalities allow harvesting passwords and browsers data from the machine, hidden on the file system until communication occurs. Payloads downloaded from the C&C are not saved locally on the machine but instead are loaded dynamically to memory with a unique internal calling convention.

One of the signature features I noticed when I began analyzing the Nymaim payload were the novel anti-reverse engineering and obfuscation techniques. Frustrating the analyzer many different code pieces for the same function requires piecing them together in order to fully understand the code. Most of the code is heavily obfuscated using ‘spaghetti code’ methods but we’ll dive into that in a 1 (bit).

In addition to the already obfuscated code, the DGA (Domain generation algorithm) use quite an interesting technique to make sure it won’t be sink-holed easily, as well as further challenging analyzation.

In this blog, I will review the anti-reverse engineering techniques the malware authors implemented in the code, explain the unique DGA they made, and show different automation concepts to conquer the code and make the analyzer's life a lot easier.

And so it begins…

In general, when I dive into a new malware, I begin with a set of goals or objectives I need to discover and understand such as the DGA mechanism of a malware, or analyzing the protocol and functionality. When I focus on the DGA for instance, while debugging, I expect the malware to hit (at some point) a DNS resolving function such as `getaddrinfo`, `gethostbyname` or any similar API. Unfortunately, Nymaim hit none of the expected DNS resolving APIs exported. Confused for a moment, I decided to try a breakpoint on the `sendto` function and indeed the breakpoint is hit. It is a crafted DNS request with a messed up Call Stack and a hardcoded dns server. I can't conclude anything definitive, I have to find the caller to the `sendto` function manually. Following the RETs and JMPs I finally get to the function called the `sendto` function. But wait, it looks so weird! (Dramatic drumming…)

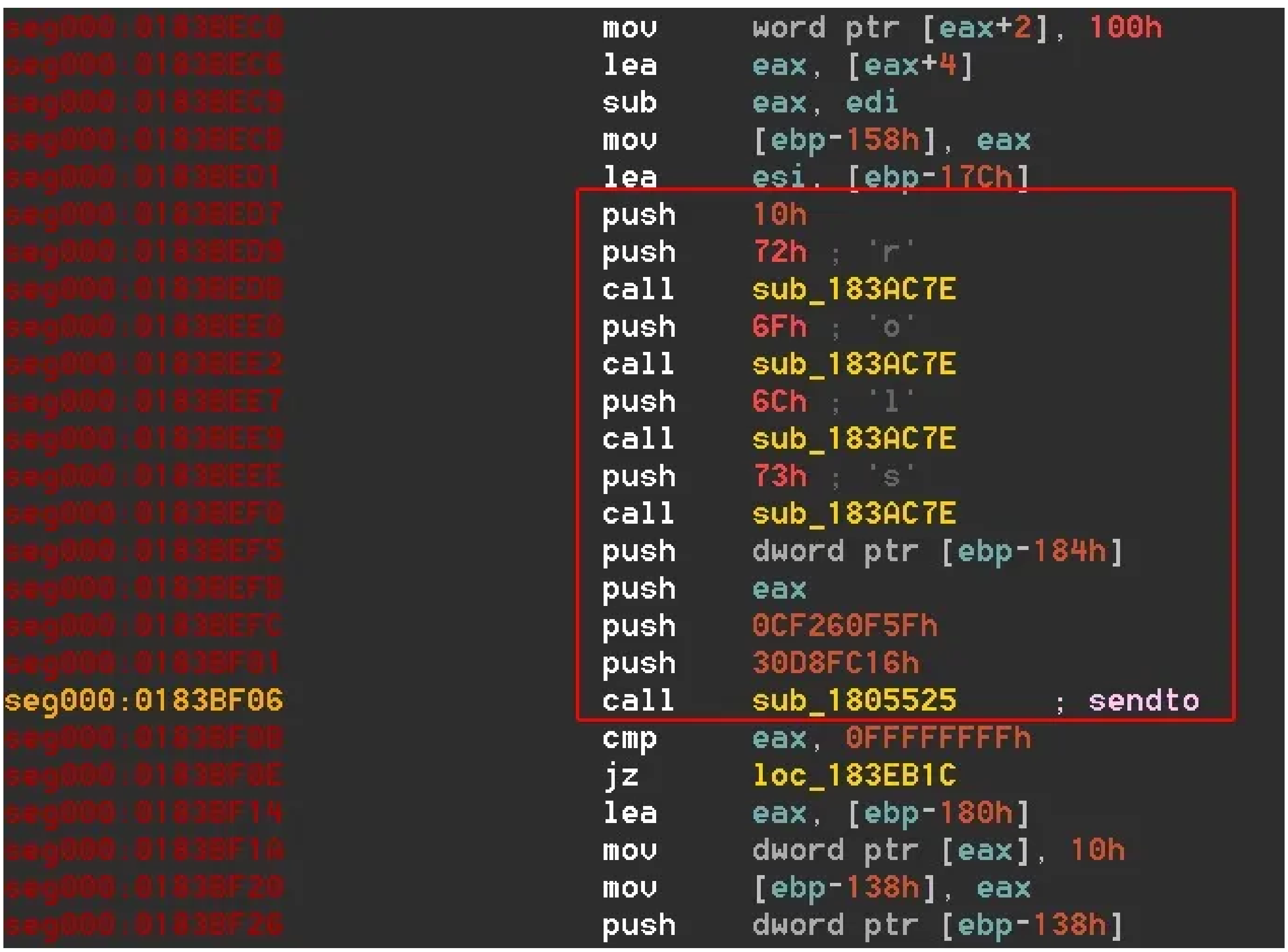


Fig. 1, The calling convention to the sendto function

no way this is the `sendto` function!

So, it continues! Obfuscation is legit code protection

Let us examine the IDA snippet above (Fig. 1), while keeping in mind what the `sendto` function looks like:

```
WS2_32!sendto(SOCKET s,
              const char *buf,
              int len,
              int flags,
              const struct sockaddr *to,
              int tolen)
```

There are 6 arguments in total. After static analysis of the code, the arguments passed on the stack don't make much sense in terms of what `sendto` is expecting (value wise). Also there are 9 push opcodes in total. Something fishy is going on in there. Let's examine the last call function `call sub_1805525` which is the OPCODE I returned to manually from the `sendto` function.
<SpoilerAlert>

This function is one of many spaghetti functions found in the code

</SpoilerAlert>

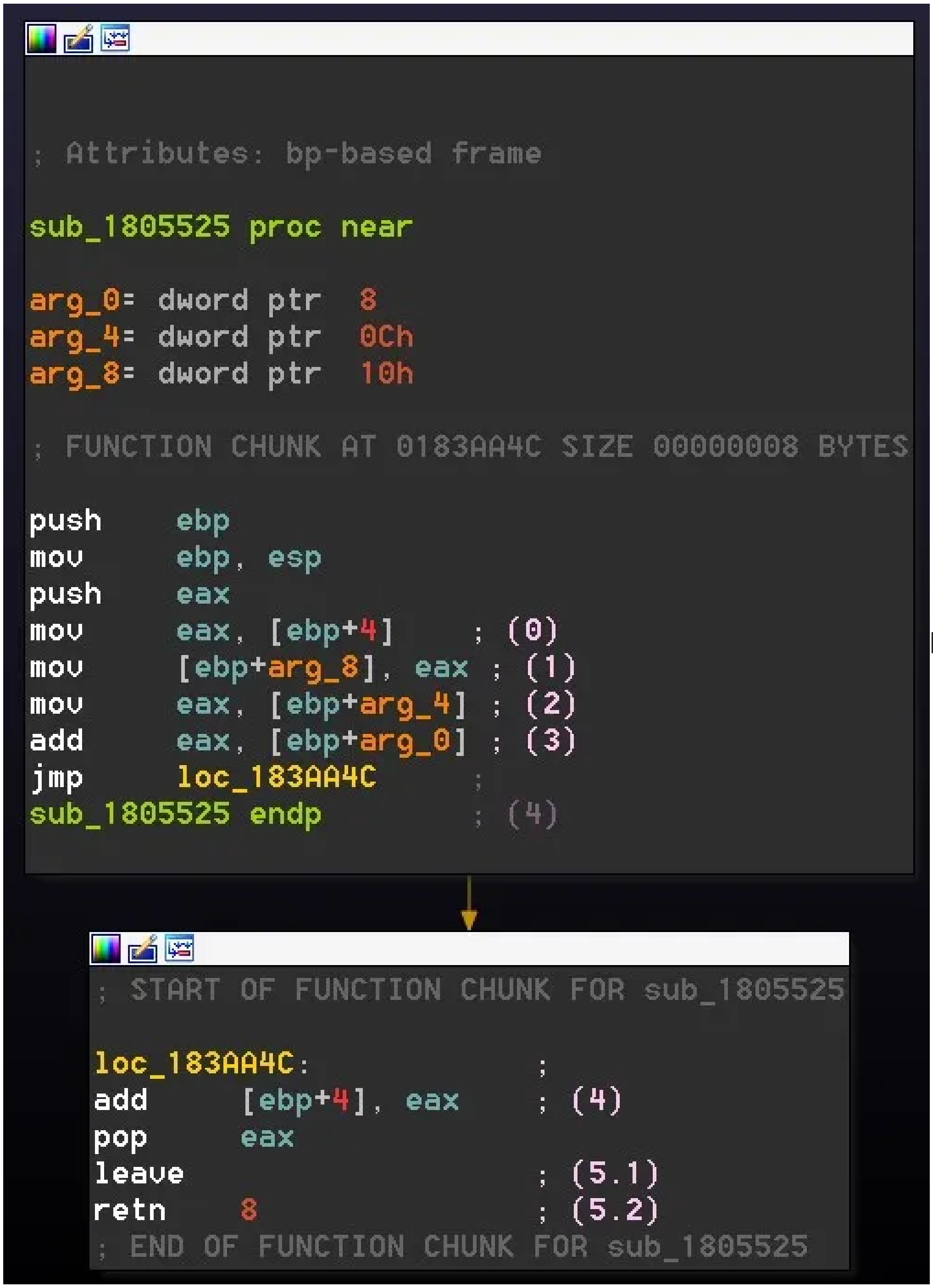


Fig. 2, How the called function looks like

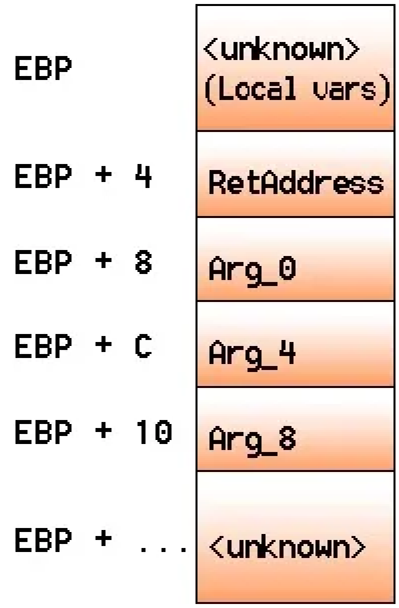
To fully comprehend what is going on, we will first have to understand how the stack would look after calling this function in terms of EBP offsets:

First of all pushing EAX (arg_8) and then two more DWORDS, arg_4 (0xCF260F5F) and arg_0 (0x30D8FC16).

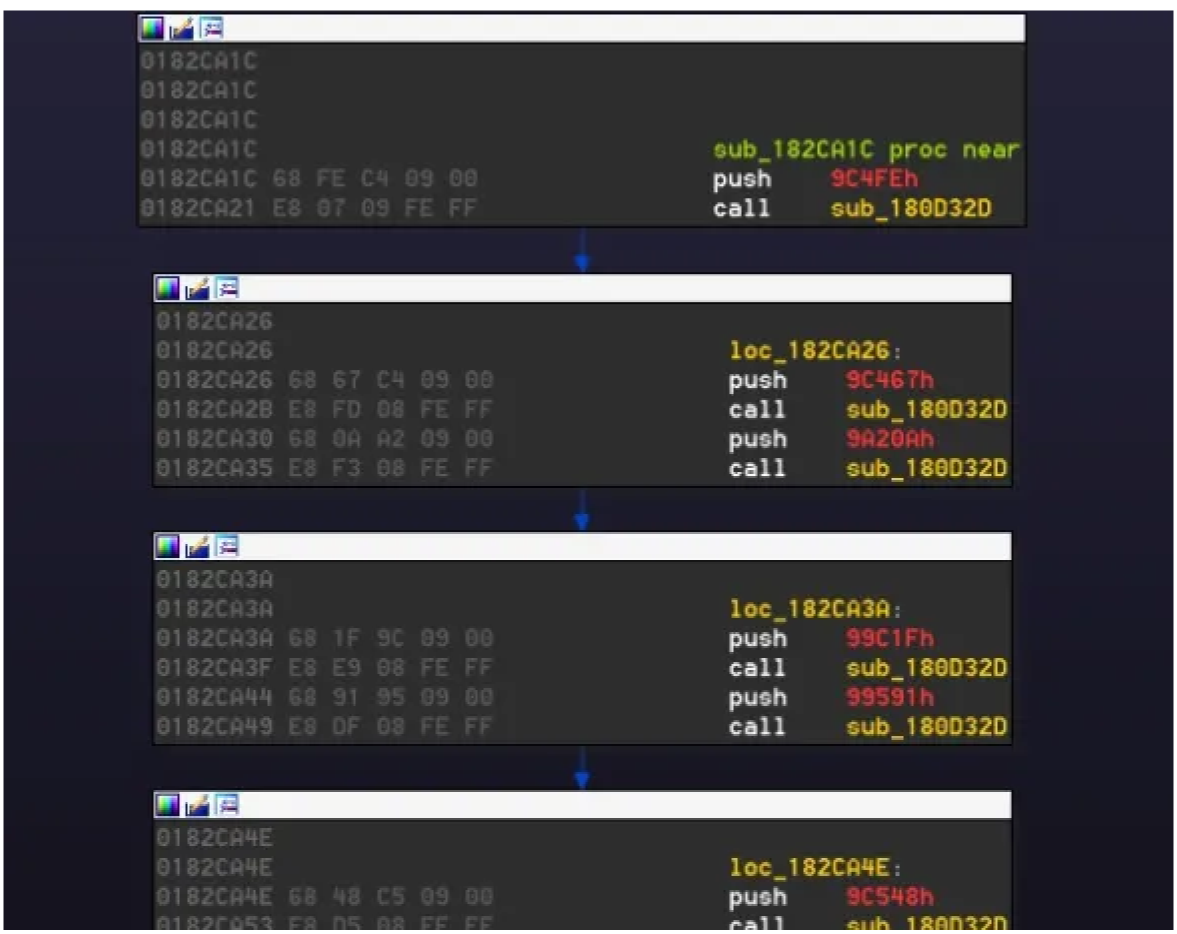
Then calling the function (call sub_1805525) which will put the appropriate ret address as the last value on the stack and that's all we need to know stack-wise for now when calling this function.

Then, inside the called function, the function's prologue happens

```
push ebp
mov ebp, esp
```



This puts into the base register (EBP) the current stack address to relatively point to stack variables using EBP and not ESP. Let's see what this function does exactly (As seen on the IDA snippet above):
(0) + (1) Overwrite `arg_8` with the `RetAddress`, (2) + (3) sum the values of the two DWORDS pushed on the stack (`arg_0 + arg_4`), (4) the result from the last operation will be added to the `arg_8` which was already overwritten with the `RetAddress`.
Basically it receives two numbers and a dummy stack value, 3 arguments in total. Resulting in a new return address with the value of `[ReturnAddress + arg_0 + arg_4]`.
Xreferencing this whole mathematical function shows me it is being called from 36 m
ore places. There are dozens (!) more variants of this function and about 2600 different places in which all of the variants being called inside the code.
Back to analyzing, the new address should be:
`[0x0183BF0B + 0x30D8FC16 + 0XCF260F5F]`, cutting the 32 bit part will result in `[0x0182CA80]`



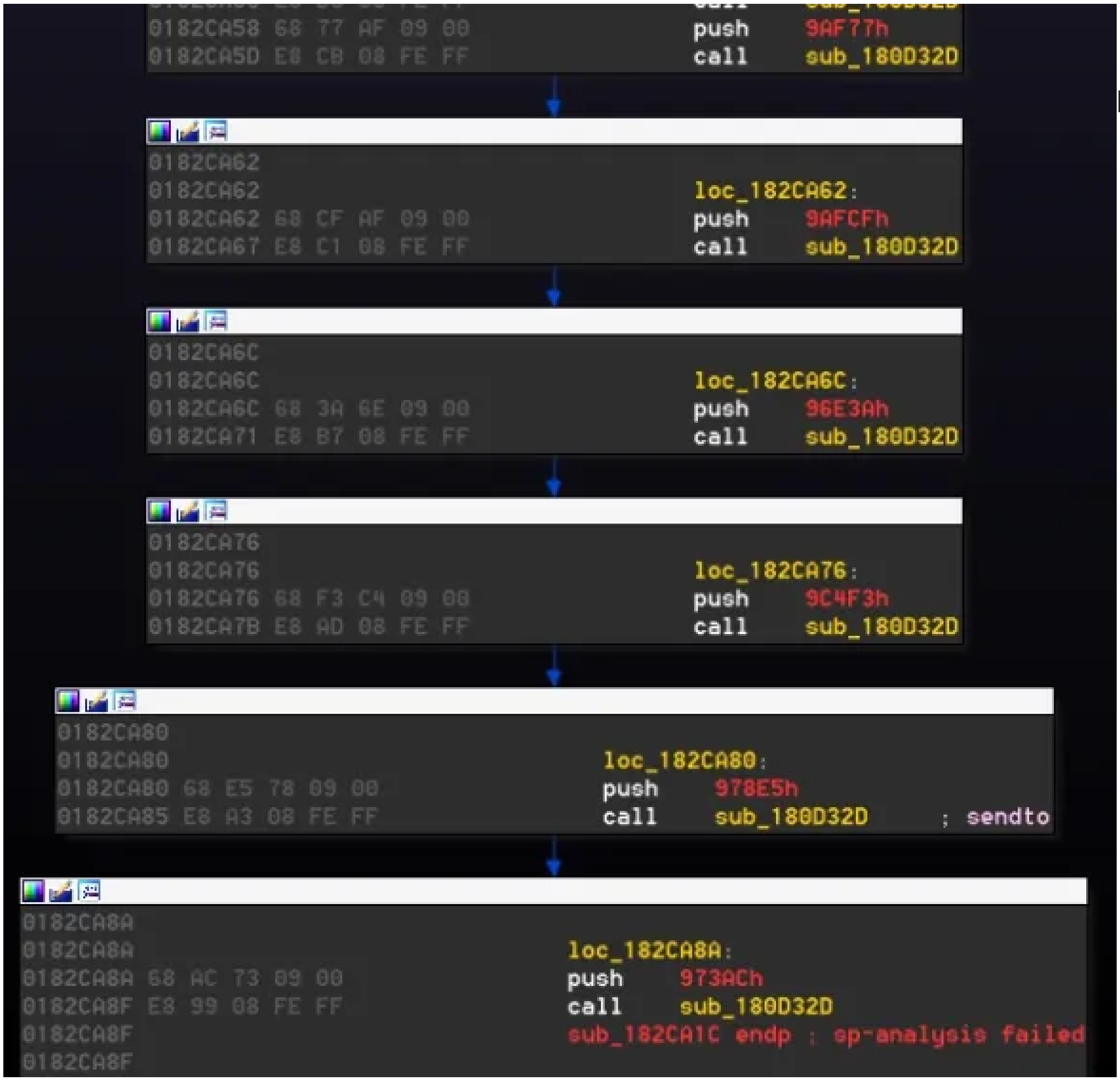


Fig. 3, API obfuscation for some api calls, sendto commented

Great success! The above snippet (Fig. 3) is another part of the obfuscation. The function that would be called next (`sub_180D32D`) is some API-Wrapper. Actually there are no standard API calls anywhere in the code, everything is calculated dynamically… everything. It's terrible I know.

Diving into that API-Wrapper function is possible (and actually required for the most part). However I won't do that in the scope of this blog post.

So this spaghetti calling convention messes up the code and I will have to fix it if I want to do any effective static analysis of it. Before I present the solution for this problem, however, Let's examine the rest of the unresolved issues in the calling function to `sendto` .

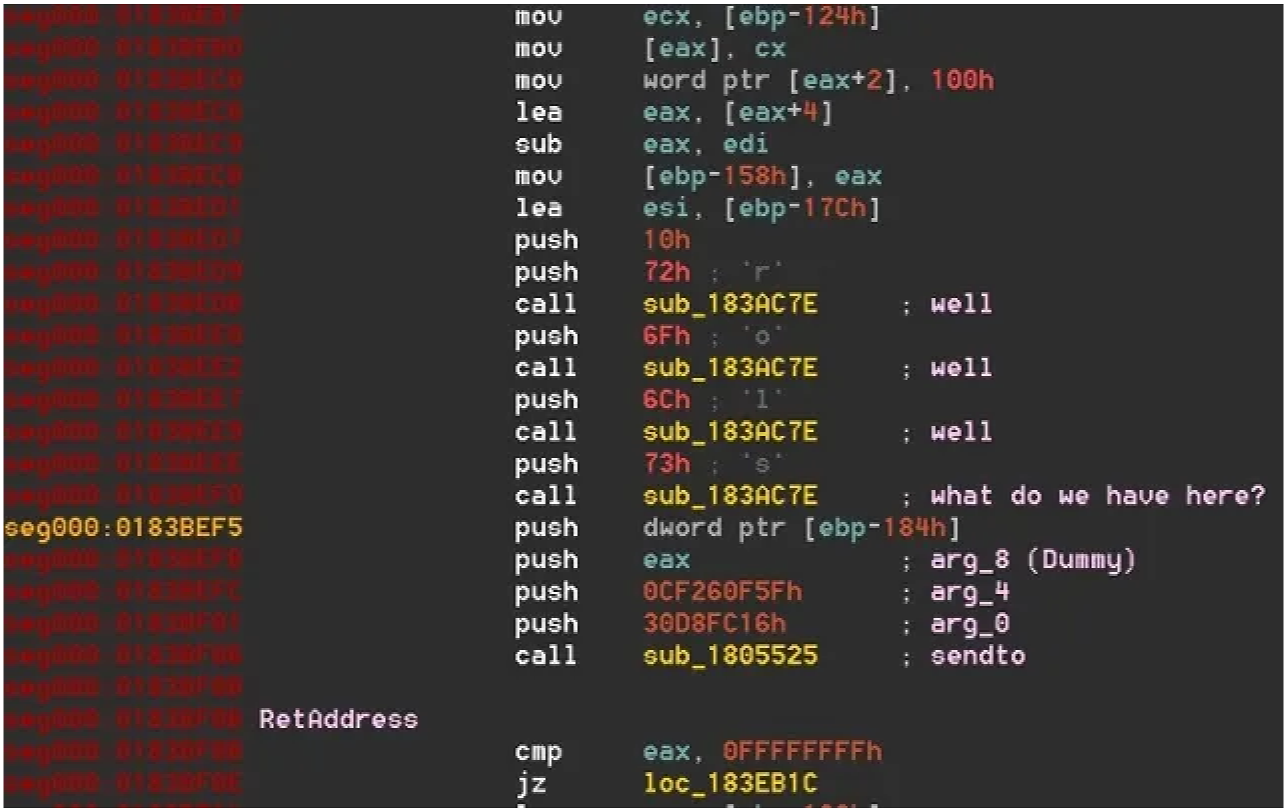


Fig. 4, Caller to the sendto function, extra unresolved code

The next thing we need to investigate, is the repeated function `sub_183AC7E`

Fig. 5, Push reg obfuscation

I will make it easy, This is a huge switch-case of putting a register value on the stack dependent of the given value. For example, the following code (our `sendto` scenario):

```
seg000:0183BED7              push 10h
seg000:0183BED9              push 72h ; 'r'
seg000:0183BEDB              call sub_183AC7E
seg000:0183BEE0             push 6Fh ; 'o'
seg000:0183BEE2              call sub_183AC7E
seg000:0183BEE7              push 6Ch ; 'l'
seg000:0183BEE9              call sub_183AC7E
seg000:0183BEEE              push 73h ; 's'
seg000:0183BEF0             call   sub_183AC7E
seg000:0183BEF5              push dword ptr [ebp-184h]
```

Can be translated to

```
push 10h
push esi
push ebx
push eax
push edi
push dword ptr [ebp-184h]
```

Now i can peacfully say i know everything i need to de-obfuscate this `sendto` call (Well not everything, i did skip the API-Wrapper function, but everything besides that) With all this new information at hand, we can move on to the next part

Tomāto-Tomăto, Potāto-Potăto It’s all the same

The two problems i aim to solve, fixing that spaghetti code calling convention, and to fix the `push_reg` function. These two functions rule most of the code, so fixing these two should be a huge step forward in understanding the code and statically analyzing it.

So how is it done? Easy, Magic!

or in its unofficial name, IDA-Python, scripting an automation process to go over all of the code, wherever one of these functions occur, fix it and change it to a simpler and more readable code format while retaining the same functionality.

So let’s get practical shall we? Starting with the `push_reg` function

I need to change every call to that function, which is made up of two opcodes:

```
6A XX               push <BYTE>
E8 XX XX XX XX      call <DWORD>
```

Push and Call, which are both in total 7 bytes in memory. If I could replace these 7 bytes with the appropriate values of the Push <Register> and do it over all of the code, it will be a big step in de-obfuscating the code.

So now that I know exactly what I want to replace, I wrote a script which does exactly that:

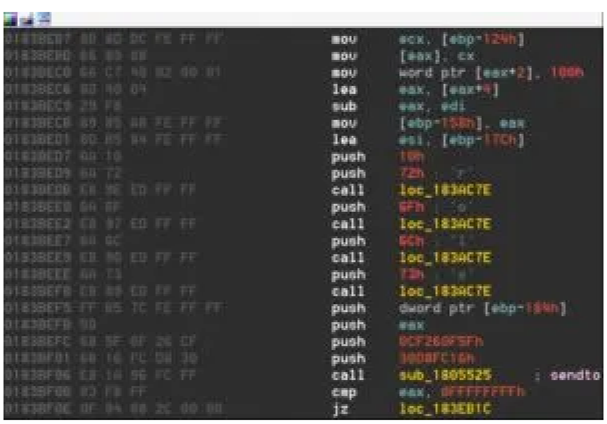
```
PUSH_REGISTER_ADDR = 0x0183AC7E
PUSH_REG_VALUE = 0x6C
SIZEOF_PUSH_BYTE = 2

def fix_reg_push(function_address):
               patched_counter = 0
               unpatched_counter = 0
               values_to_patch = {PUSH_REG_VALUE : 0x50,        # push eax
                            PUSH_REG_VALUE + 1 : 0x51,      # push ecx
                            PUSH_REG_VALUE + 2 : 0x52,      # push edx
                            PUSH_REG_VALUE + 3 : 0x53,      # push ebx
                            PUSH_REG_VALUE + 5 : 0x55,      # push ebp
                            PUSH_REG_VALUE + 6 : 0x56,      # push esi
                            PUSH_REG_VALUE + 7 : 0x57,      # push edi}

            # Go through all xrefs
            for xcall in XrefsTo(function_address):

                            # Make code if is not already
                            opcode_length = idc.MakeCode(xcall.frm -
SIZEOF_PUSH_BYTE)
                            if SIZEOF_PUSH_BYTE != opcode_length:
                                            print " [*] fix_reg_push not code
[0x%08X]" % push_addr
                                            not_code_counter += 1
                                            continue

                            # Obtain previous opcode address
                            push_addr = idc.PrevHead(xcall.frm)

                            # Sanity check 2
                            if "push" != GetMnem(push_addr):
                                            print " [*] fix_reg_push not push
instruction [0x%08X]" % push_addr
                                            print GetMnem(push_addr)
                                            not_push_counter += 1
                                            continue

                            # Get new value
                            push_value = GetOperandValue(push_addr, 0)
                            byte_val = values_to_patch.get(push_value, None)
                            if None == byte_val:
                                            print " [*] fix_reg_push unexpected push
value [0x%08X]" % push_addr
                                            bad_push_counter += 1
                                            continue

                            # Patch code
                            idaapi.patch_word(push_addr, 0x04EB) # EB 04 -> Jmp
$+4...
                            idaapi.patch_long(push_addr + 2, 0x90909090) #
                        idaapi.patch_byte(push_addr + 6, byte_val)

                            patched_counter += 1
            print " [*] fix_reg_push - Total: [%d]\npatched functions:
```

```
[%d]\nunpatched functions: [%d]" % (patched_counter + unpatched_counter,
patched_counter, unpatched_counter)


def main():
            fix_reg_push(PUSH_REGISTER_ADDR)

if "__main__" == __name__:
            main()
```

The code above is separated into a couple of sections:

Calling my `fix_reg_push` function with the appropriate function address which handles the push register by value
Running through all the Xrefs of the function and making IDA identify the bytes as code if it hasn’t already. Otherwise there would be issues identifying opcodes later in the script
Making sure the xref is valid and working as expected. I don’t want to create any weird code patches so I make some necessary sanity checks
Patching the code, changing the 7 original bytes to `PUSH <reg>` and `JMP <byte>` for better code clarity.

Lets examine the before and after results:

Before	**After**
	

As you can see, I translated the reg_push functions (all of them) to a readable simple de-obfuscated push opcodes which have a length of one byte. I could have just done a NOP-slide for the rest of the bytes left, but I decided to implement a jmp opcode instead with the memory I had left to overwrite. It’s a matter of taste. The code became much more readable and now I can finally read which register represents which value on the stack. This function was fixed at over 3,900 places in the code. So it was definitely worth it.

And that’s it for the first part.

Patching the code on IDA made everything a lot more readable in terms of static analysis. Next, there is still that spaghetti calling convention I will have to fix, but as I investigated more of the code, I noticed there are dozens of variations with different calculations being made, and for each one of those, there are a dozen more duplications which look identical to each other. The only logical thing left for me to do, was to make a regex to find every matching function.

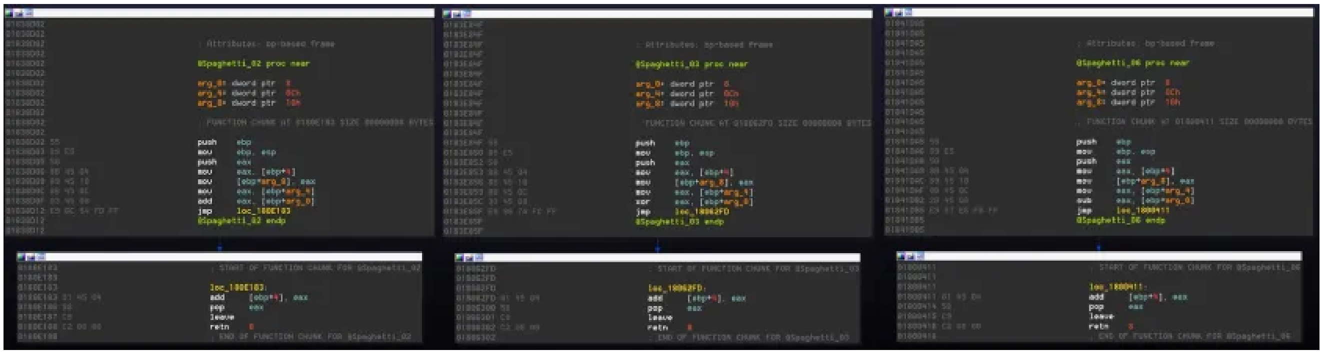


Fig. 6, three spaghetti functions found on the code, using add, xor and sub for calculations
Fortunately finding the common base between all functions wasn’t so hard. All of them have more or less the same prologue, and pretty much the same epilogue. So creating some kind of byte regex to find all of them (and fix them!) isn’t very hard. So I’ve done just that.

After automatically finding all of these spaghetti functions, I will patch the code just as I have done with the ‘*push_reg*‘ functions. Only this time I have a lot more “space” in terms of bytes to do so

Fig. 7, focusing on the sendto call
In total, there are 16 bytes, that I would like to change to just CALL (5 bytes), so I have enough space to override as I want. This method is practically the same as the method I used before. So there is no reason to put another code block to show how its done. Looking for all variants of these functions gave me a result of almost 100 different variations, with a total of approximately 3,000 different Xrefs in the code (for all variants).

The final result after patching both the spaghetti calling convention and the push registry by value:

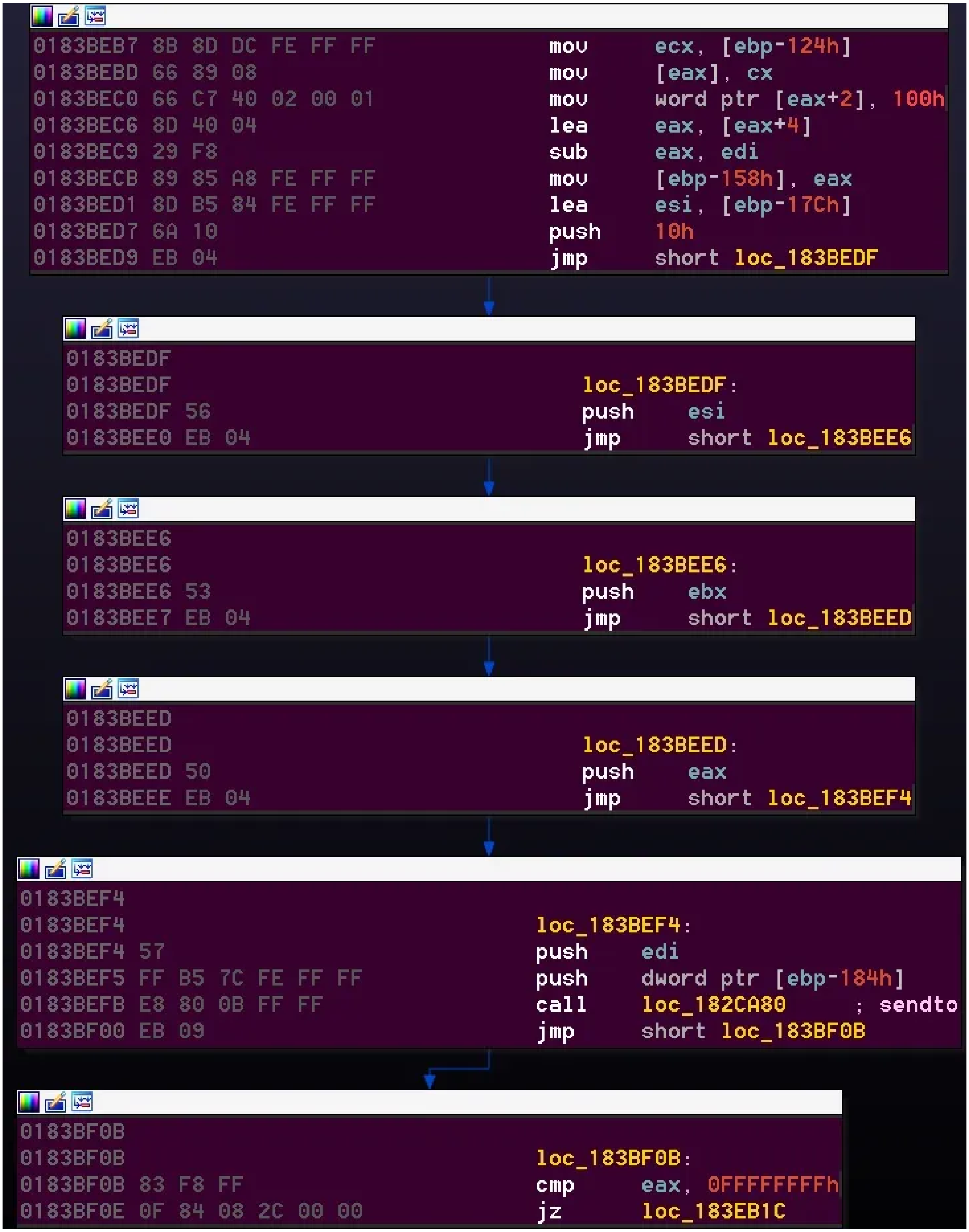


Fig. 8, Final patch

You Can Run, But You Can’t Hide…

Finally, having the important parts de-obfuscated, I could continue on to the DGA. Let me pre announce, the authors intent to avoid being sinkholed payed off, good job! It has been a while since I've seen someone trying to protect their code and their DGA as much as they did. So let's get to it

Most malwares who have a DGA use some value which changes periodically. This one is no different and is based on the current date to calculate it's DGA (Day, Month, Year). Though it's not as simple as it sounds: Instead of using some sort of builtin linear random function (such as *msvcsrt!rand* and *msvcrt!srand*), they implemented their own functions for making random numbers and setting the initial seed. Their MagicSeed (I'm going to use that term a lot), means the number calculated every day, generated by the current date for example is made out of 128 bits. Every time anything needs to obtain the MagicSeed's value, the MagicSeed changes as well. So I had to follow all of the code very carefully, not to miss anything regarding the MagicSeed's usage.

How It All Works

I will now explain how the malware reaches the C&C server and the obfuscation made behind the DGA.

As you would expect from any malware, they make a simple domain list using a MagicSeed, then try to resolve each of the domains created, using google's dns servers to prevent being dns-sinkholed, until one is being resolved and that would usually be the C&C server. However, this is not our case because it would be too boring to talk about just that wouldn't it?

So as it gets more complicated, as when trying to resolve all of the generated domains, only the first domain which will be resolved into exactly two different IP addresses. For example, these domains (which are generated at 30/09/2016):

Generated Domain	Resolved IP addresses
jfwwqi.com	
avljz.net	4.2.0.1 4.2.0.2 4.2.0.3
hlrhtvl.com	
mcodqfban.com	192.168.0.1 192.168.0.2
xdvhfogmw.pw	13.37.80.80
obsvi.com	
igcvdloatwf.in	

Generated Domain	Resolved IP addresses
zcekjgrmmx.in	

The only domain that will be used from this list would be

mcodqfban.com	192.168.0.1 192.168.0.2

Because it is being resolved into two different IP addresses.

Yet, these two IP addresses have no direct connection to the C&C server. They are just going to be another stepping stone in Nymaim's logic in order to create a new MagicSeed number.

And with that new MagicSeed, create a new domain list. with exactly the same algorithm as the first domain list was generated, But hold on, there is more:

Before trying to use this newly created domain list, a checksum algorithm is used over the newly created domain list, and the result is compared with a builtin checksums list.

This probably means that the domains themselves are finite and have probably been pre-bought, or they are just waiting for the right time to buy a new domain that matches their checksum list.

After the list passes the checksum check, the first domain in the list is taken and its TLD is changed to ".COM".

After all this effort, I would guess that domain is all that is left and the IP addresses matching the resolving of this domain are what would be the C&C server. However my guess was wrong. The IPs resolved from that newly created domain are not yet the correct IP addresses of the C&C servers. For every IP address we get from the DNS request, a loop of xoring and rotation calculations are being made over each of the IP Addresses in order to obtain the real C&C server IP addresses (**Finally!**). Let's summarize everything with a pseudo code:

```
tlds = [".net", ".com", ".in", ".pw"]

GenerateDomains(magic_num)
{
               domains = []

            seed = CreateUniqueSeed(TODAYS_DATE)
            rand = GetRandomNumber(seed)

            for(int i=0; i<16; i++)
            {
                            domain_str = GenString(rand, seed, magic_num)
                            domain_str += tlds[GetRandomNumber(seed)
                            domains += [domain_str]
            }
            return domains
}

ResolveDomains(domain_list)
{
               for(i =0, i<16; i++)
               {
                            ip_addresses = DnsResolve(domain_list[i])
                            if (2 == ip_addresses.length())
                                            return ip_addresses
            }
}

Main()
{
               domains = GenerateDomains(0)
            ips = ResolveDomains(domains)

            new_domains = GenerateDomains(ips)
            domain = new_domains[0].replace(".com")

            real_ips = ResolveDomains(domain)
            real_ips = XorIPS(real_ips)

            CommunicateWithRealServer(real_ips)
}
```

- GenerateDomains
  - Creating a unique seed based on current date
  - Generate random number from seed
  - Create a domain string from generated random number and the seed
  - Create a new random, use it to append TLD
  - Returns a list of 16 domains created
- ResolveDomains
  - Trying to resolve domain list ip addresses
  - Check if exactly 2 IP addresses were obtained in the dns request
  - Return list of resolved addresses

- Main
  - Generating first list of domains
  - Get good matching ip addresses (Only 2 ip addresses)
  - Generate new list of domains from the ips we got
  - Change TLD of the first domain from the list generated
  - Resolve domain
  - Obtain real C&C ip addresses through calculations
  - Communicate with C&C

I have also added a graph form for convenience

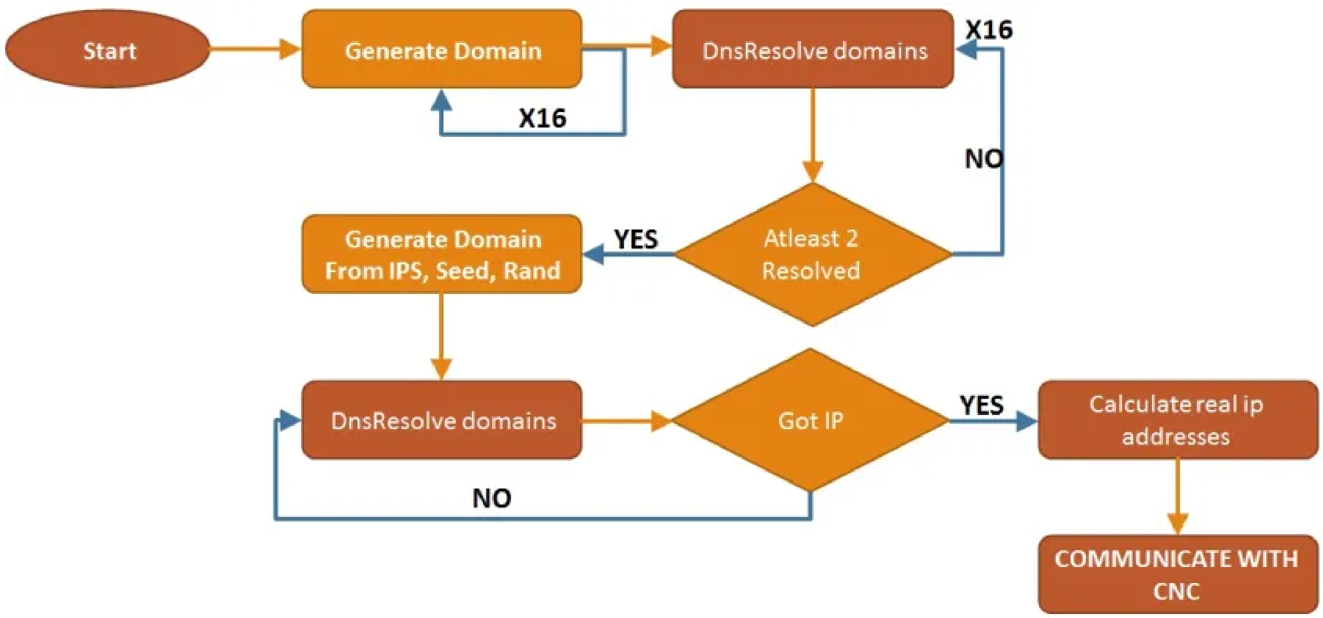


Fig. 9, Graph format of the pseudo code

This is a lot of stuff to do in order just to get a C&C server IP address. Those little tricks they used made it harder to reverse and understand the Nymaim code, and harder to sink-hole the malware as well.

So here we see prime example of how malware authors try to avoid being sink-holed by using obfuscation methods as protection for their code.
But then again, everything can be conquered and beaten if you wear on your malware thinking-cap and put your mind into it.

Ref analyzed sample:

c41ffc1fd6e3f5157181b6e45f45f4fe