2024 DUBLIN

2 - 4 October, 2024 / Dublin, Ireland

AUTOMATICALLY DETECT AND SUPPORT AGAINST ANTI-DEBUG WITH IDA/GHIDRA TO STREAMLINE DEBUGGING PROCESS

Takahiro Takeda

LAC Cyber Emergency Center, Japan

takahiro.takeda@lac.co.jp

ABSTRACT

Malware authors often employ anti-debugging techniques to obstruct analysis. When executed on a debugger, the malware detects the debugger and either stops its subsequent actions or behaves differently from usual, making analysis difficult. The number of anti-debugging implementations varies with each malware. Notably, malware spread through mass-mailing campaigns that affect many organizations, and popular ransomware, have been confirmed to possess multiple anti-debugging techniques. For example, anti-debugging techniques include VM detection, which checks for a debugging environment, detection of breakpoints (which temporarily pause program execution during debugging), and time difference detection, which utilizes the difference in execution time when analysing malware with a debugger.

'AntiDebugSeeker' is an open-source plugin for the binary analysis tools IDA and Ghidra, which are frequently utilized by analysts. It streamlines the malware analysis process by automatically identifying the anti-debugging techniques embedded within *Windows* malware. Code with anti-debug capabilities often overlaps with techniques used for anti-analysis, as well as with the preparatory steps for process injection, which are frequently employed by malware. Therefore, by flexibly customizing the detection rules, it is possible not only to identify anti-debugging features but also to understand the functionalities of the malware. Furthermore, the tool also provides functionalities to explain these anti-debugging measures and approaches to the corresponding functions. This enhances the analyst's ability to understand and counteract the malware's evasion techniques effectively, offering a more comprehensive understanding and response strategy against such threats. This paper provides a comprehensive explanation of AntiDebugSeeker, detailing its features and usage. It offers an in-depth understanding of the tool and illustrates how these features can effectively be applied in various threat scenarios.

INTRODUCTION OF ANTIDEBUGSEEKER

AntiDebugSeeker is a program for automatically identifying and extracting potential anti-debugging techniques used by malware and displaying them in IDA [1] or Ghidra [2].

The main functionalities of this plugin are as follows:

1. Extraction of APIs that are potentially being used by the malware for anti-debugging.
2. In addition to APIs, extraction of anti-debugging techniques based on key phrases that serve as triggers, as some anti-debugging methods cannot be comprehensively identified by API calls alone.

As a note, for packed samples, it is more effective to run this plugin after unpacking and fixing the Import Address Table.

ANTIDEBUGSEEKER FOR IDA

Files required to run the program

For the purpose of facilitating the detection of anti-debugging techniques using IDA, the following three files must be placed within the plugin directory of the IDA software:

- anti_debug.config – this file contains a set of rules designed to identify various anti-debugging techniques.
- anti_debug_techniques_descriptions.json – this file provides detailed descriptions of the rules detected.
- AntiDebugSeeker.py – this script serves as the core program for anti-debugging detection.

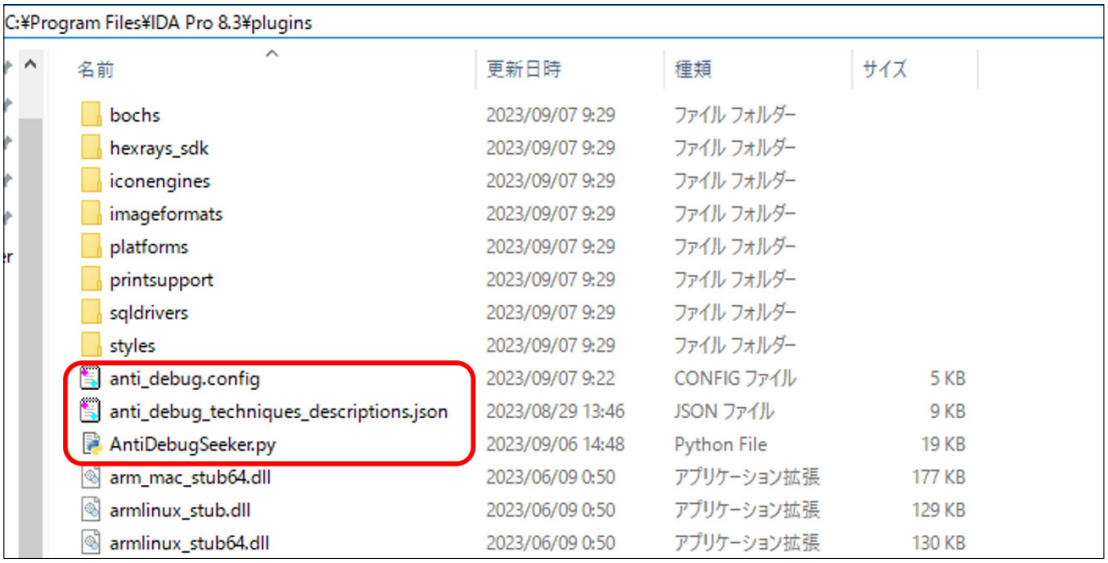


Figure 1: Three files to run in IDA.

Regarding anti_debug.config

The anti_debug.config file contains rules for detecting anti-debugging features. It is divided into two sections: Anti_Debug_API and Anti_Debug_Technique.

In the Anti_Debug_API section, you can freely create categories and add any number of APIs you want to detect (exact match).

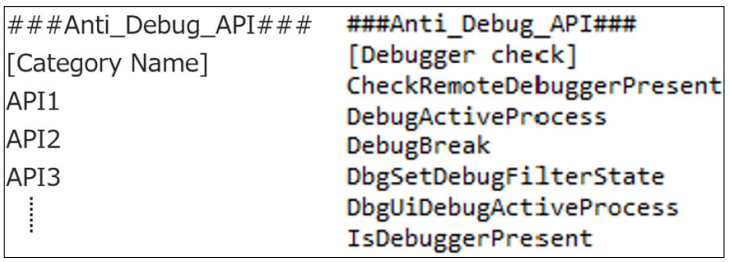


Figure 2: Anti_Debug_API section.

In the Anti_Debug_Technique section, you can set up to three keywords (partial match) under a single rule name.

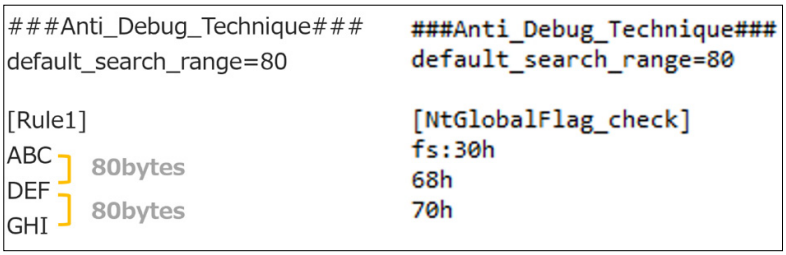


Figure 3: Anti_Debug_Technique section.

The basic flow of the search is as follows:

The search begins with the first keyword. If it is found, the second keyword is then searched for within a specified number of bytes (the default is 80 bytes). This same process is applied when searching for the third keyword.

Search Target:

Disassembly (Opcode, Operand)

Comments

API based on Import Table

Should you wish to alter the predefined default values and tailor the search parameters, append ‘search_range=value’ to the specified keyword. This adjustment facilitates the customization of the search scope for each established rule within your configuration.

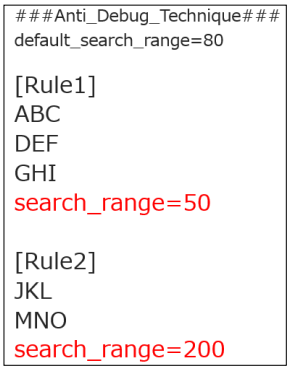


Figure 4: How to specify the search range.

Regarding anti_debug_techniques_descriptions.json

This JSON file provides explanations for the rules defined in the Anti_Debug_Technique section of the anti_debug.config file, detailing why they were detected and describing the rules themselves. This information can also be useful for reference in case of false detections. Comments are recorded in IDA for the addresses that were detected.

```
{
   "VMware_I/O_port" : "detect a VM environment based on the VMware I/O port",
   "VMware_magic_value" : "detect a VM environment based on the VMware magic value.",
   "HeapTailMarker": "Malware can detect if it's on a debug heap by checking the HeapTailMarker value. ¥nIf this value is ABABABAB, it indicates execution in a debug environment; otherwise, it's a normal execution.",
   "KernelDebuggerMarker": "Detect Kernelmode Debugger(KdDebuggerEnabled)",
   "DbgBreakPoint_RET": "This detection may be due to the first byte of the DbgBreakPoint function being replaced with 0xC3, which corresponds to the 'ret' instruction. ¥nThis prevents a debugger from successfully attaching to the process.",
   "DbgUiRemoteBreakin_Debugger_Terminate": "When a debugger tries to attach to a process, the process automatically terminates.",
                    ¦
                    ¦
                    ¦
"Commucate_function" : "Detected by characters related to '/' and '443', indicating communication capabilities. ¥nThis suggests potential connections to a C2 server and the ability to detect analysis environments."
}
```


Figure 5: The contents of anti_debug_techniques_descriptions.json.

The functionality of AntiDebugSeeker

Upon activation of the plugin using the keyboard shortcut Ctrl + Shift + D, the system initiates an analysis sequence. Subsequent to the completion of this analysis, a user interface screen entitled 'Anti Debug Detection Results' will be displayed. This interface presents the findings of the analysis, enabling detailed examination of the detected anti-debugging techniques.

Category Name	Possible Anti-Debug API	Address	Possible Anti-Debug Technique	Address
Analysis Environment Check	SetupDiGetClassDevsA	0x401022		
Analysis Environment Check	SetupDiEnumDeviceInfo	0x401043		
Analysis Environment Check	SetupDiGetDeviceRegistryPr…	0x401062		
Analysis Environment Check	SetupDiGetDeviceRegistryPr…	0x401068		
Analysis Environment Check	SetupDiGetDeviceRegistryPr…	0x401092		
Check Invalid Close->Exception	CloseHandle	0x401410		
Check Invalid Close->Exception	CloseHandle	0x401419		
Check Invalid Close->Exception	CloseHandle	0x40141E		
User Interaction Check	GetCursorInfo	0x40161B		
			Opened_Exclusively_Check	0x4016cf
Check Invalid Close->Exception	CloseHandle	0x401707		
Time Check	Sleep	0x40184F		
Check Invalid Close->Exception	CloseHandle	0x40185D		
Check Invalid Close->Exception	CloseHandle	0x40194D		
Time Check	Sleep	0x4019A8		
Memory Manipulation	VirtualProtectEx	0x4019C7		
			Memory_EXECUTE_READWRITE_2	0x4019d1
Memory Manipulation	VirtualProtectEx	0x4019DD		
Memory Manipulation	VirtualProtectEx	0x401A11		
Check Invalid Close->Exception	CloseHandle	0x401E35		
Thread Execute	ResumeThread	0x402170		
Time Check	WaitForSingleObject	0x40217E		
Thread Manipulation	SuspendThread	0x402191		
Thread Execute	ResumeThread	0x4022C3		
Time Check	WaitForSingleObject	0x4022D1		
Thread Manipulation	SuspendThread	0x4022E4		
Thread Execute	ResumeThread	0x40235B		
Check Invalid Close->Exception	CloseHandle	0x402409		
Check Invalid Close->Exception	CloseHandle	0x402423		
			Opened_Exclusively_Check	0x402b79
Check Invalid Close->Exception	CloseHandle	0x402BB3		
Check Invalid Close->Exception	CloseHandle	0x402E5D		
Time Check	GetTickCount	0x402F60		
Time Check	GetTickCount	0x402F66		
Time Check	GetTickCount	0x402FCF		
Check Invalid Close->Exception	CloseHandle	0x40364D		
			Opened_Exclusively_Check	0x4036cf

Figure 6: Screen of Anti Debug Detection Results tab.

The column structure of Anti Debug Detection Results is as follows:

- Category Name
  API category name defined in the Anti_Debug_API as listed in anti_debug.config.
- Possible Anti-Debug API
  List of detected APIs displayed.
- Address
  Address where the detected API is being used.
- Possible Anti-Debug Technique
  Detection name identified by the keyword defined in Anti_Debug_Technique as listed in anti_debug.config.

- Address
  Address of the first detected keyword.
- Address Transition
  By double-clicking on the detected line, you will jump to the address specified.

After running the plugin, detected APIs and keywords are highlighted in different colours.

Detection Category	Color
Anti_Debug_API	Green
Anti_Debug_Technique	Orange

Figure 7: Differences in colour by category.

Furthermore, should an API detailed within the Anti_Debug_API section be identified, the corresponding category name is annotated as a comment (Figure 8). Similarly, upon detection of a rule name within the Anti_Debug_Technique section, a commentary derived from the JSON file illustrated in Figure 5 is appended as a note to the initially detected keyword (Figure 9).

```
call    sub_401D30
mov     ds:WriteProcessMemory, eax ; Write Data OnTheMemory
mov     edx, 9D00A761h
mov     eax, [ebp+var_8]
call    sub_401D30
mov     ds:ReadProcessMemory, eax ; MemoryRead,ProcessInspection
mov     edx, 9ABFB8A6h
mov     eax, [ebp+var_8]
call    sub_401D30
mov     ds:VirtualAllocEx, eax ; Memory Manipulation
mov     edx, 6B416786h
mov     eax, [ebp+var_8]
call    sub_401D30
mov     ds:GetCurrentProcessId, eax
mov     edx, 774393E8h
mov     eax, [ebp+var_8]
call    sub_401D30
mov     ds:GetModuleFileNameA, eax
mov     edx, 2EE4F10Dh
mov     eax, [ebp+var_8]
call    sub_401D30
mov     ds:CopyFileA, eax
mov     edx, 19F78C90h
mov     eax, [ebp+var_8]
call    sub_401D30
mov     ds:Process32First, eax ; Process Check
mov     edx, 0D89AD05h
mov     eax, [ebp+var_8]
call    sub_401D30
mov     ds:GetCurrentProcess, eax
mov     edx, 0C930EA1Eh
mov     eax, [ebp+var_8]
call    sub_401D30
mov     ds:Process32Next, eax ; Process Check
mov     edx, 5BC1D14Fh
mov     eax, [ebp+var_8]
call    sub_401D30
mov     ds:CreateToolhelp32Snapshot, eax ; Process Check
mov     edx, 77CD9567h
```

Figure 8: Detected in the Anti_Debug_API section.

```
push    ebp
mov     ebp, esp
and     esp, 0FFFFFFF8h
mov     eax, large fs:30h ; NtGlobalFlag_check - The code is checking the NtGlobalFlag value at offset 0x68 from the Process Environment Block.
                          ; The value 70 is the sum of FLG_HEAP_ENABLE_TAIL_CHECK (0x10), FLG_HEAP_ENABLE_FREE_CHECK (0x20), and FLG_HEAP_VALIDATE_PARAMETERS (0x40).
sub     esp, 480h
test    byte ptr [eax+68h], 70h
push    esi
push    edi
jz      short loc_4BFFB2
```

```
mov     [ebp+Context.ContextFlags], 10010h ; Hardware_Breakpoints_Check - Check the debug registers DR0, DR1, DR2, and DR3 (CONTEXT_DEBUG_REGISTERS 0x10010)
                                           ; to determine if a hardware breakpoint has been set.
lea     eax, [ebp+Context]
push    eax             ; lpContext
call    ds:GetCurrentThread
push    eax             ; hThread
call    ds:GetThreadContext ; Thread Manipulation
mov     ecx, [ebp+var_4]
xor     eax, eax
xor     ecx, ebp        ; StackCookie
call    @__security_check_cookie@4 ; __security_check_cookie(x)
mov     esp, ebp
pop     ebp
retn
_main endp
```

```
push    ebp
mov     ebp, esp
push    ecx
mov     eax, large fs:30h ; BeingDebugged_check - The BeingDebugged field in the Process Environment Block (PEB) indicates whether the current process is being debugged or not.
movzx   eax, byte ptr [eax+2]
test    eax, eax
setnz   byte ptr [ebp+var_4]
cmp     [ebp+var_4], 0
jz      short loc_40102E
```

Figure 9: Detected in the Anti_Debug_Technique section.

Upon initiating the plugin using the Ctrl+Shift+D shortcut, the system not only presents the anti-debug detection results but also includes a newly added feature that displays a list of detected functions. From the perspectives of the anti-debug detection results and the detected function list, there is an enhanced understanding of both the detection outcomes and the overarching code structure. Concurrently, the debugging process is streamlined by systematically organizing information by function. Collectively, these windows support a dual-perspective approach to malware analysis.

Below is a list of the basic features of the Detected Function List:

- Display the anti-debug detection results, organized by function.
- For items detected by the rules in the Anti_Debug_Technique section, displayed in pink text, the rule descriptions can be viewed by hovering over them with the mouse.
- Clicking on the areas highlighted in yellow under the function names will navigate to those addresses.
- Entering text in the search bar initiates a search, and matching results are highlighted in blue.

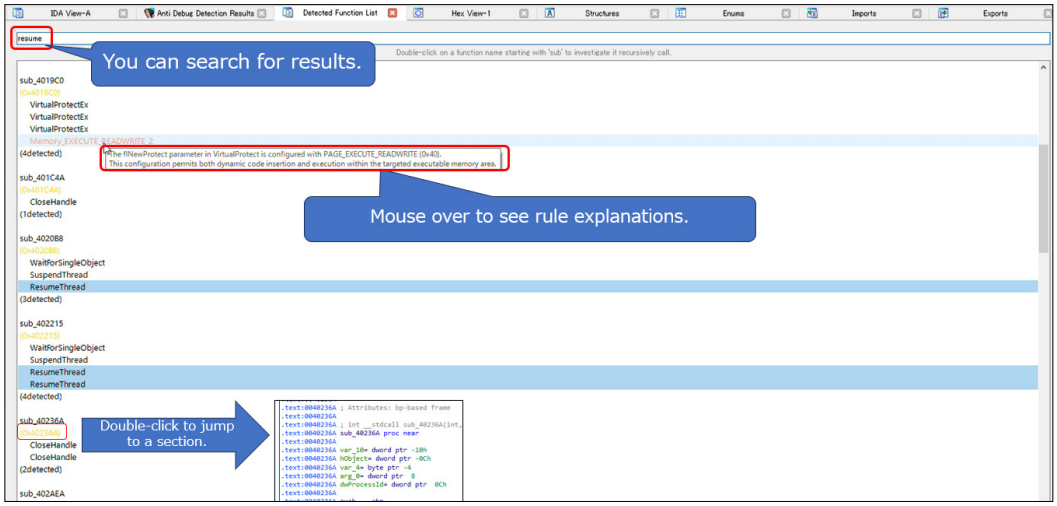


Figure 10: Screen of Detected Function List – basic functions.

The Detected Function List, in addition to its basic features, displays the hierarchical structure of where detected functions are being called from when function names starting with 'Sub' are double-clicked. In the hierarchical display, function names include 'depth:[number]' to indicate their depth from the Original Entry Point. If the text is in grey, it indicates that the function has been detected in the Anti Debug Detection Lists. Furthermore, double-clicking on the displayed function names navigates to the address where that function is called.

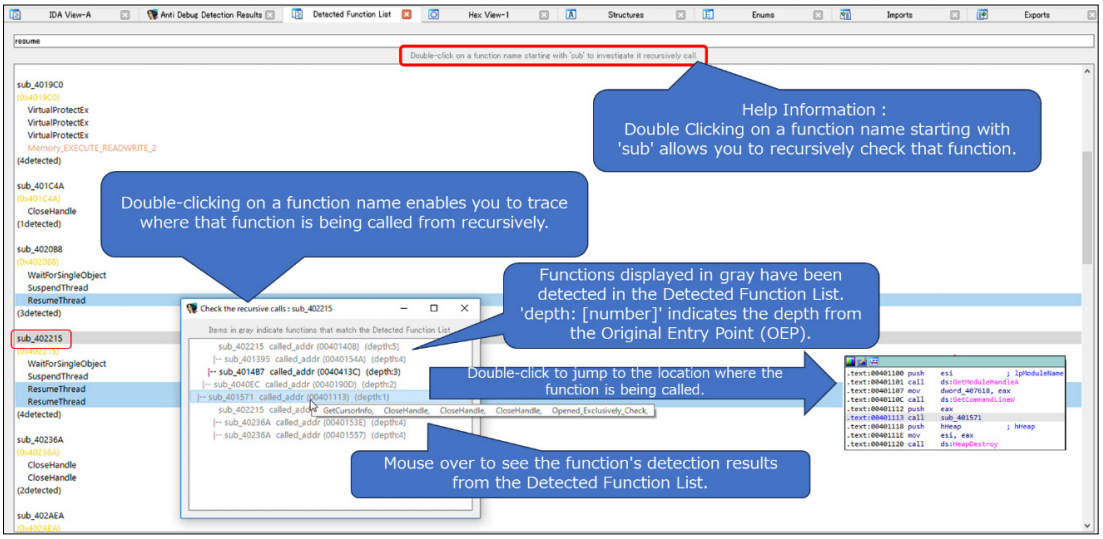


Figure 11: Screen of Detected Function List – recursive checking.

List of detectable anti-debugging techniques (all 47 rules)

VMware_I/O_port	NtGlobalFlag_check	Memory_Region_Tracking
VMware_magic_value	NtGlobalFlag_check_2	Check_BreakPoint_Memory_1

HeapTailMarker	HeapFlags	Check_BreakPoint_Memory_2
KernelDebuggerMarker	HeapForceFlags	Software_Breakpoints_Check
DbgBreakPoint_RET	Combination_of_HEAP_Flags	Hardware_Breakpoints_Check
DbgUiRemoteBreakin_Debugger_Terminate	Combination_of_HEAP_Flags_2	ChildProcess_Check
PMCCheck_RDPMC	ReadHeapFlags	Enumerate_Running_Processes
TimingCheck_RDTSC	ReadHeapFlags_2	ThreadHideFromDebugger
SkipPrefixes_INT1	DebugPrivileges_Check	NtQueryInformationProcess_PDPort
INT2D_interrupt_check	CreateMutex_AlreadyExist	NtQueryInformationProcess_PDFlags
INT3_interrupt_check	CreateEvent_AlreadyExist	NtQueryInformationProcess_PDObjectHandle
EXCEPTION_BREAKPOINT	Opened_Exclusively_Check	NtQuerySystemInformation_KD_Check
ICE_interrupt_check	EXCEPTION_INVALID_HANDLE_1	Extract_Resource_Section
DBG_PRINTEXCEPTION_C	EXCEPTION_INVALID_HANDLE_2	Commucate_function_String
TrapFlag_SingleStepException	Memory_EXECUTE_READWRITE_1	Commucate_function
BeingDebugged_check	Memory_EXECUTE_READWRITE_2	

ANTIDEBUGSEEKER FOR GHIDRA

The basic search logic is the same across the versions for IDA and Ghidra.

Files required to run the program

- AntiDebugSeeker.java (for Ghidra script version)
- Zip folder containing the compiled files: ghidra_11.0.1_AntiDebugSeeker.zip (for Ghidra module extension)
- anti_debug_Ghidra.config (converted for Ghidra: a file containing rules for detecting anti-debugging techniques)
- anti_debug_technique_descriptions_Ghidra.json (converted for Ghidra: a file containing descriptions of the detected rules)

How to run in the Ghidra scrip version

When utilizing Ghidra Script, one can select AntiDebugSeeker.java from the Script Manager and proceed to execute it.

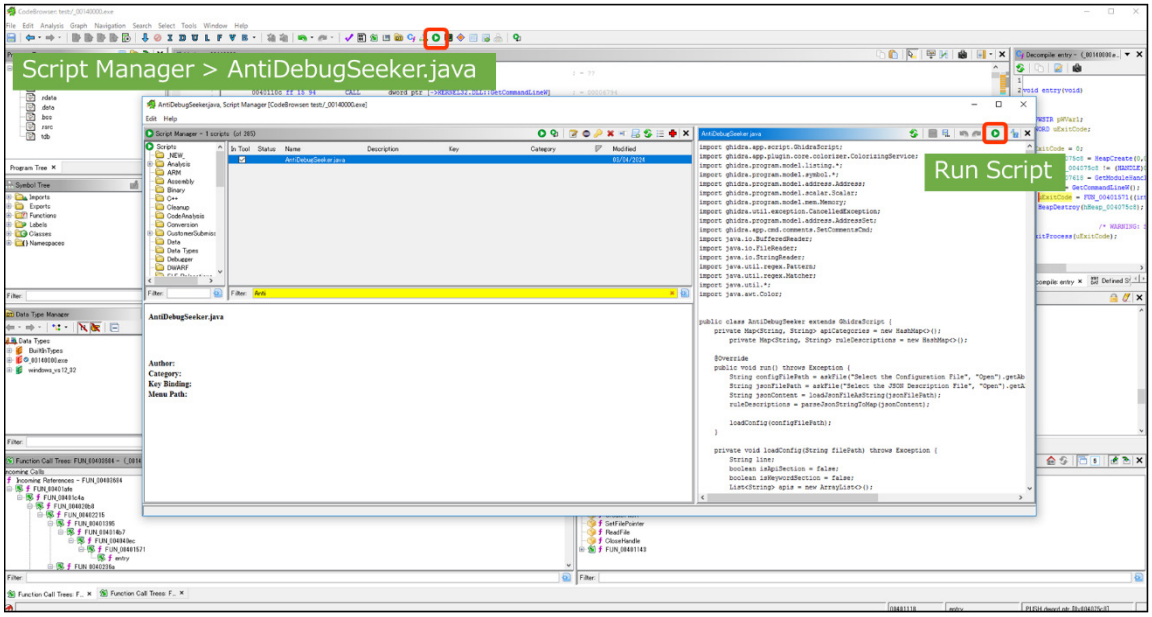


Figure 12: How to run in the Ghidra script version.

When the script is initiated, a prompt entitled ‘Select the Configuration File’ appears. Users are required to specify the anti_debug_Ghidra.config, which defines the detection rules, before selecting ‘Open’ to proceed.

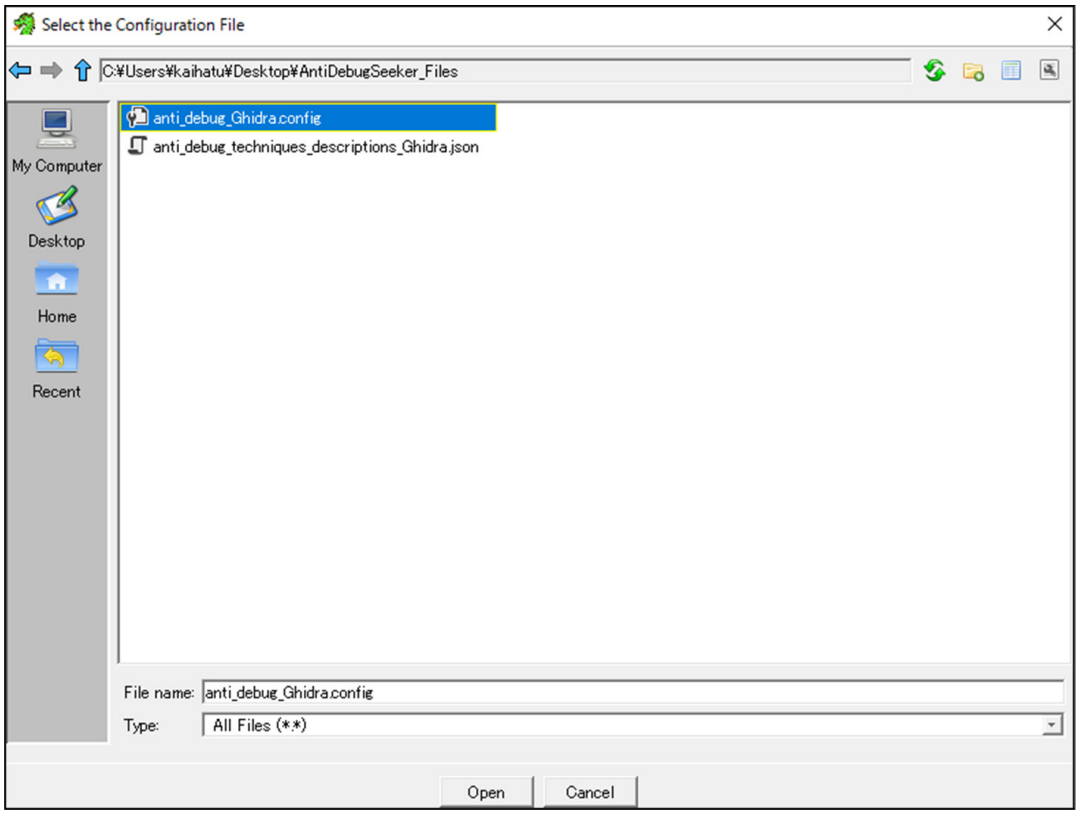


Figure 13: Select the configuration file, anti_debug_Ghidra.config.

Following the selection of the configuration file, a prompt titled 'Select the JSON Description File' appears. Users are required to specify the anti_debug_technique_descriptions_Ghidra.json file, which contains the descriptions of the detection rules, and then click 'Open' to continue.

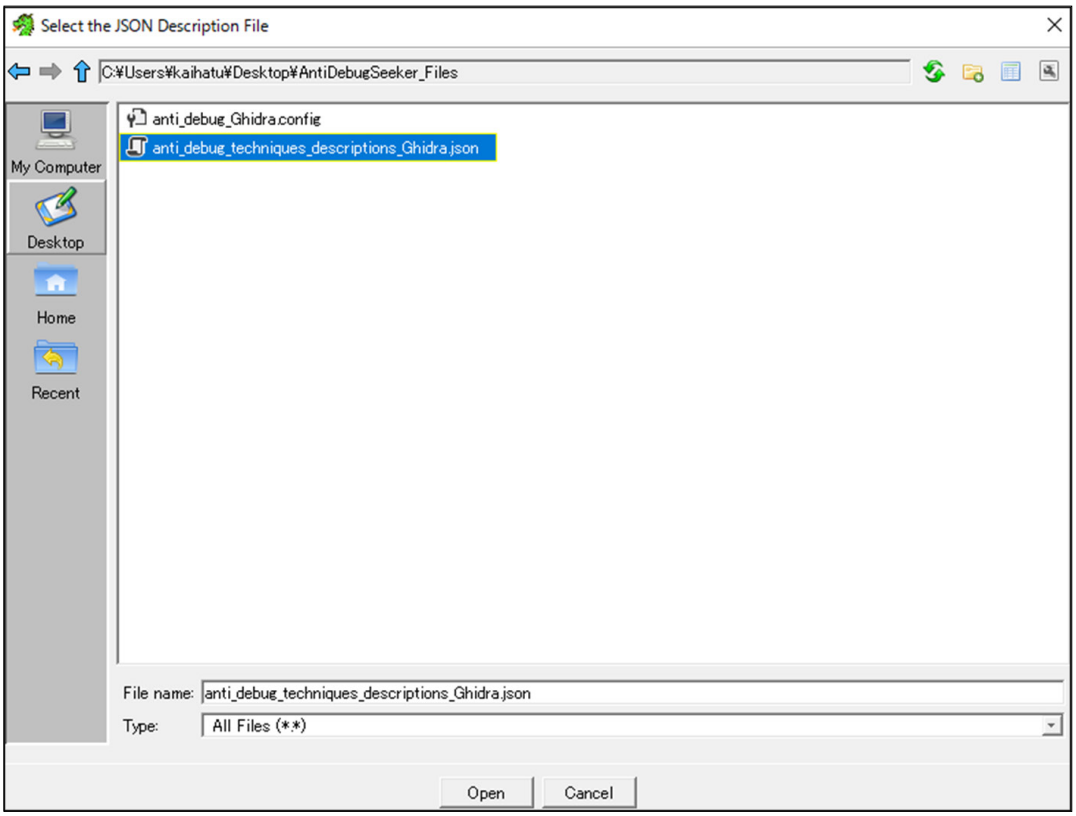


Figure 14: Select the description file, anti_debug_technique_descriptions_Ghidra.json.

How to set up and execute the Ghidra module extension

To integrate and operate the module version of AntiDebugSeeker within Ghidra, rather than utilizing a script, the following steps are necessary. The module version features a graphical user interface, which facilitates a visual comprehension of the analysis results, and is therefore recommended.

1. Select the file > Install Extensions
2. Click + button, then Select Install Extensions.
   (ghidra_11.0.1_PUBLIC_AntiDebugSeeker.zip)
   After Opening CodeBrowser
3. Select the file.
4. In the Configure menu, check ‘Examples.
5. Click ‘Configure,’ select ‘AntiDebugSeekerPlugin,’ and click ‘OK.

First, prior to launching Ghidra’s CodeBrowser, we will proceed with the steps necessary to install the AntiDebugSeeker plugin.

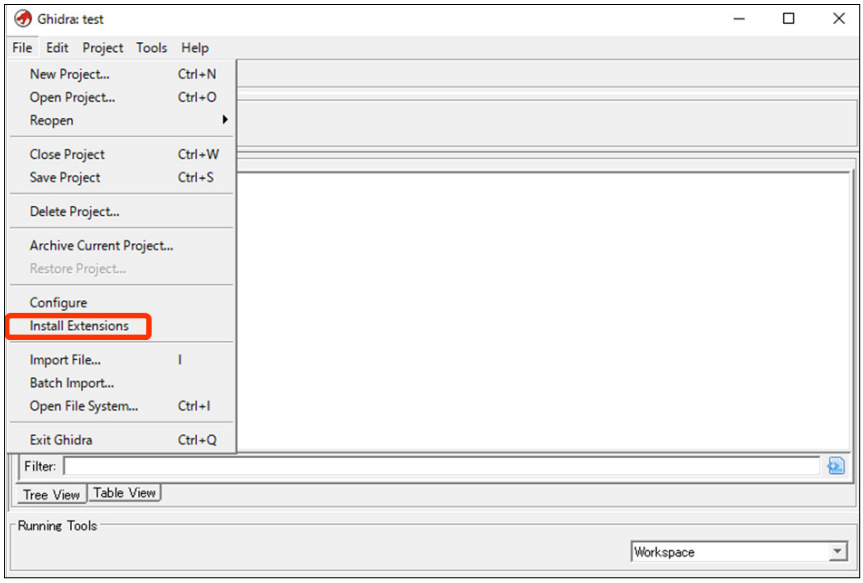


Figure 15: Select File > Install Extensions.

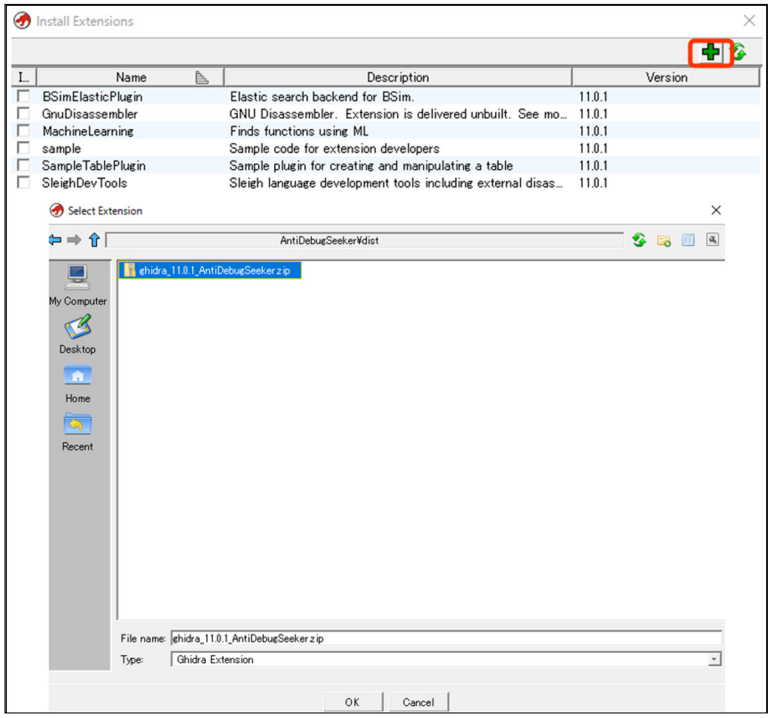


Figure 16: Install a ZIP file (compiled AntiDebugSeeker).

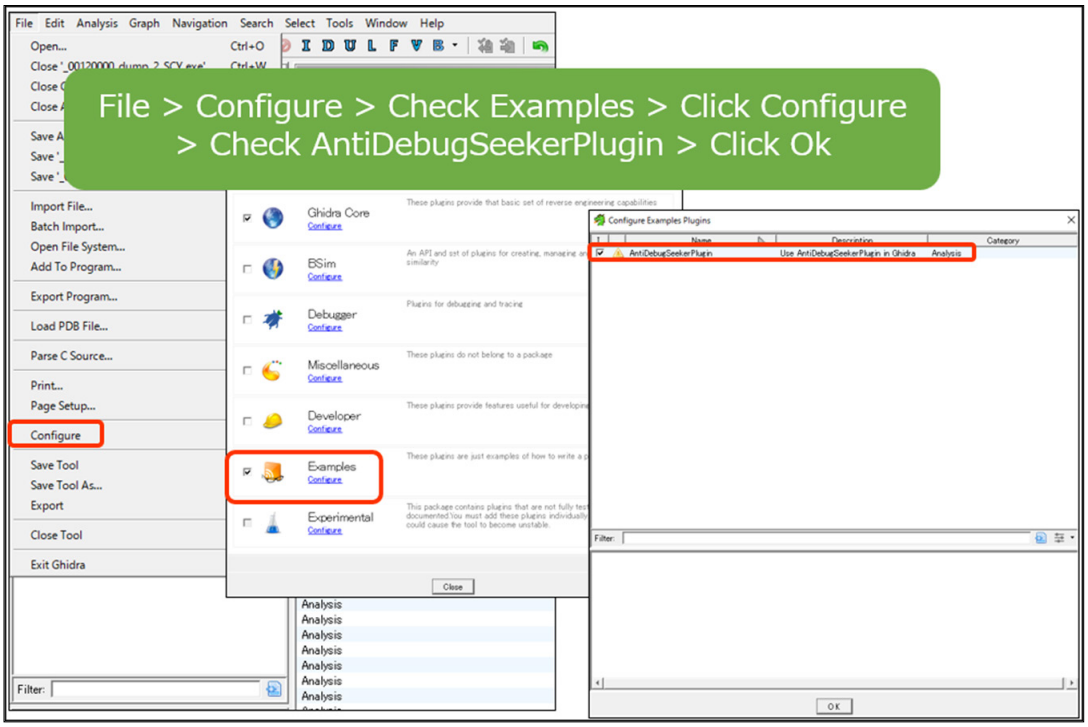


Figure 17: How to integrate the module version of AntiDebugSeeker.

After integration, 'AntiDebugSeekerPlugin' will be available in the Window menu; users are advised to click on it to continue.

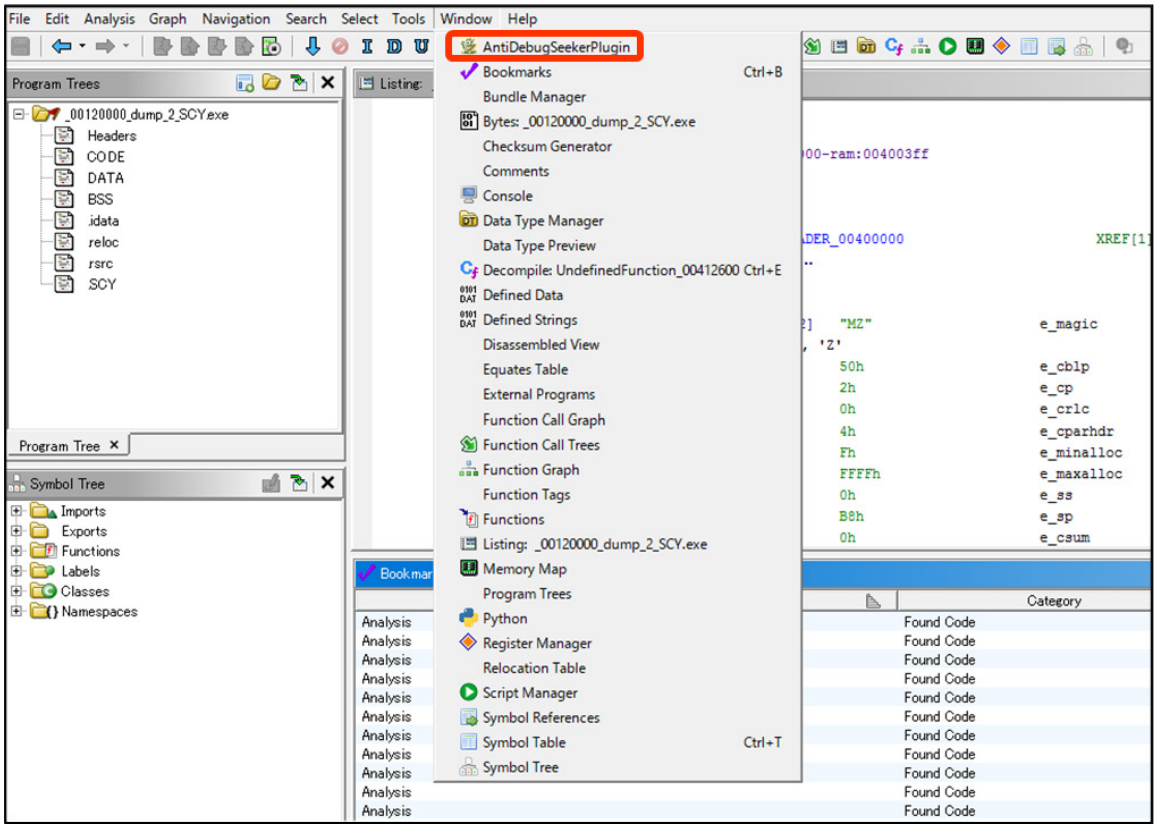


Figure 18: Select AntiDebugSeekerPlugin from the Window tab.

When the plugin is launched, as illustrated in Figure 19, three clickable buttons are available: 'Start Analyze', 'Display Only the Detection Results' and 'Detected Function List'. Clicking the 'Start Analyze' button loads the configuration and JSON files and initiates the analysis.

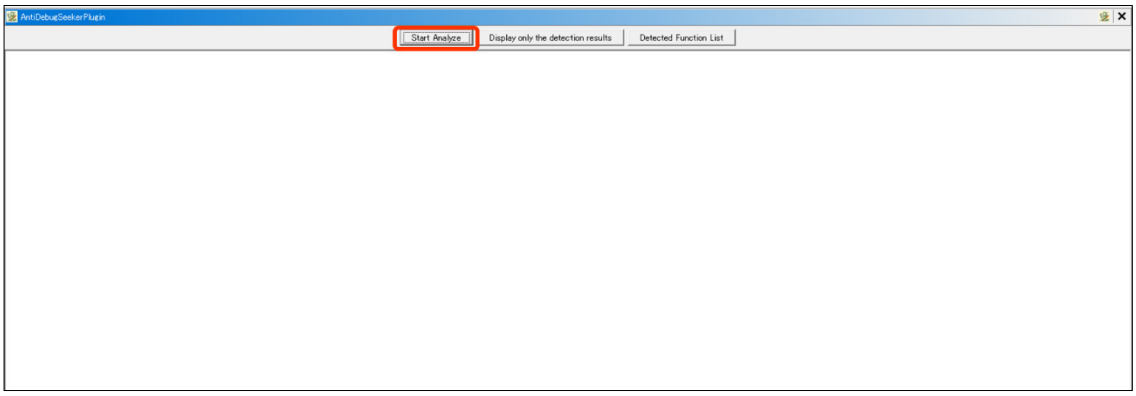


Figure 19: Startup screen.

A screen with a small dragon and file selection options will appear. 'Select Config File' is displayed. Specify the anti_debug_Ghidra.config file that defines the detection rules, and then click 'Open'.

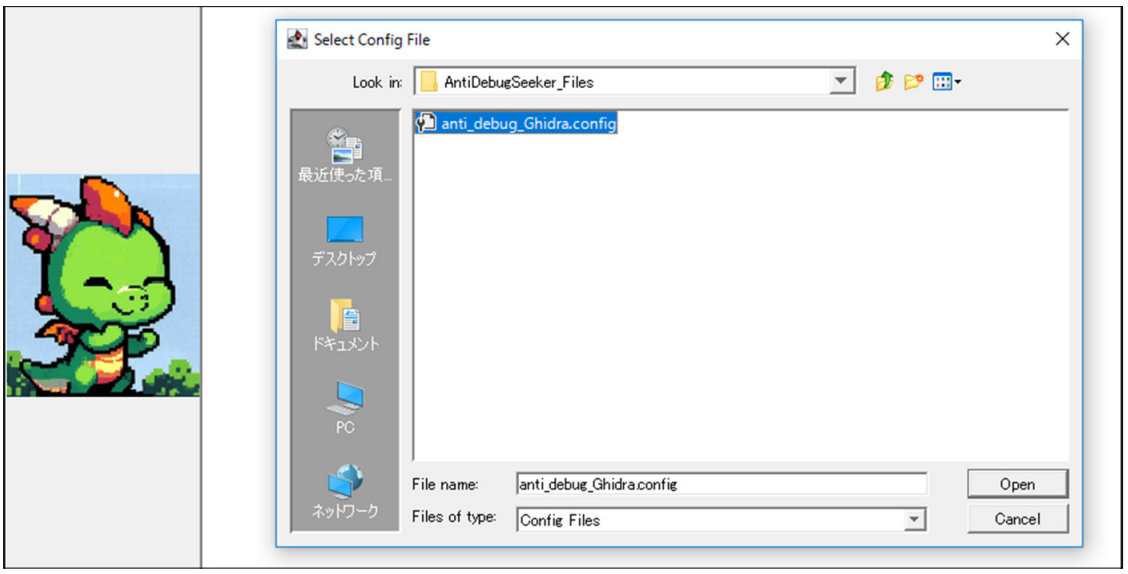


Figure 20: Select the config file.

Similar to the Ghidra script version, a prompt to 'Select JSON File' will be displayed. Specify the anti_debug_technique_descriptions_Ghidra.json file, which contains the descriptions of the detection rules, and then click 'Open'.

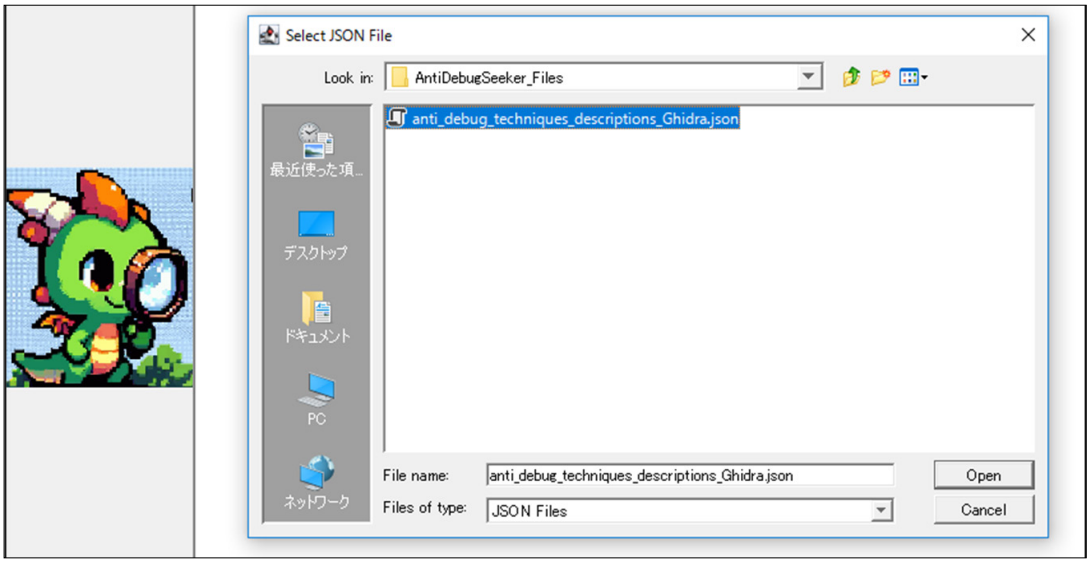


Figure 21: Select the JSON file.

VERIFYING THE RESULTS, GHIDRA SCRIPT + MODULE EXTENSION

Ghidra script: check console-scripting

The results of the detection can be reviewed from the Console - Scripting screen. The message 'AntiDebugSeeker Process Finished' indicates that the process has successfully completed.

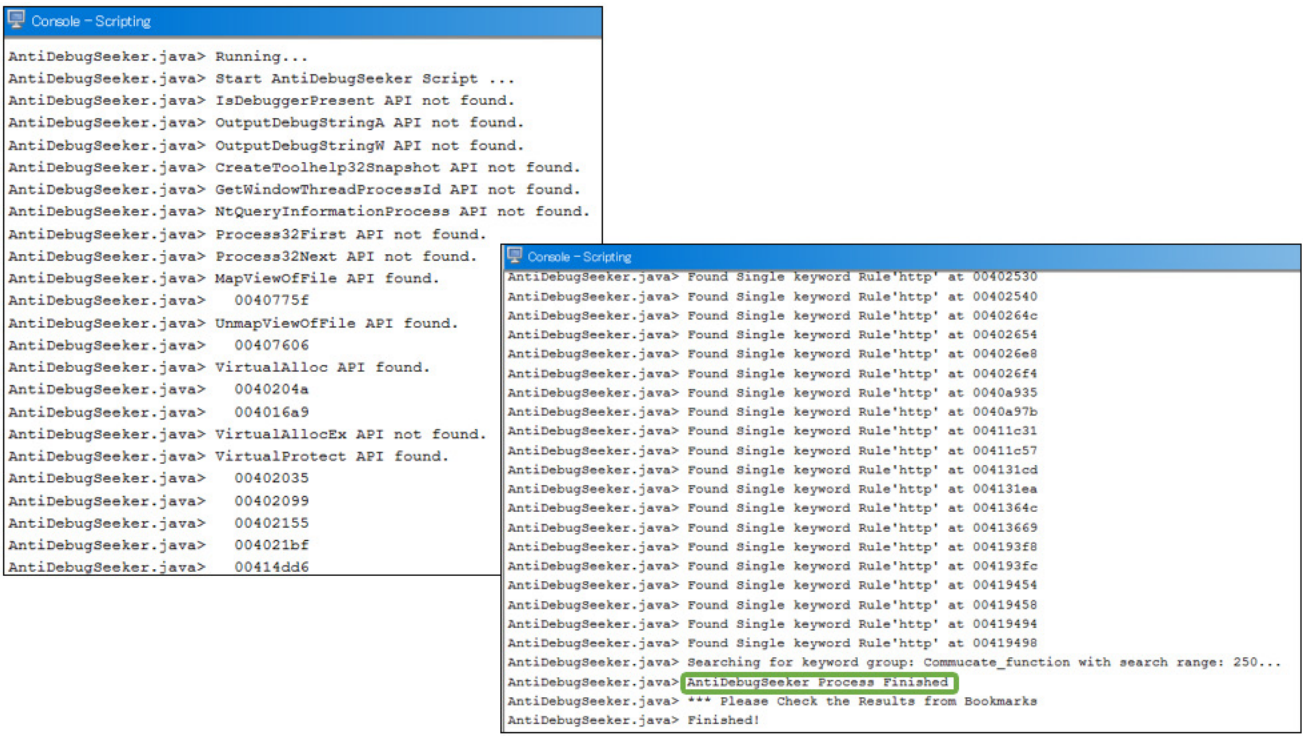


Figure 22: Script analysis completion screen.

Ghidra module extension: check text area

When the analysis is complete, 'Analysis Complete' will be displayed. Monitor the progress on the right side of the screen using the progress bar. Once the bar reaches 100%, the analysis is complete. On the displayed screen, you can review both the detected and undetected items. At this point, the results are already registered in Ghidra's Bookmark feature.

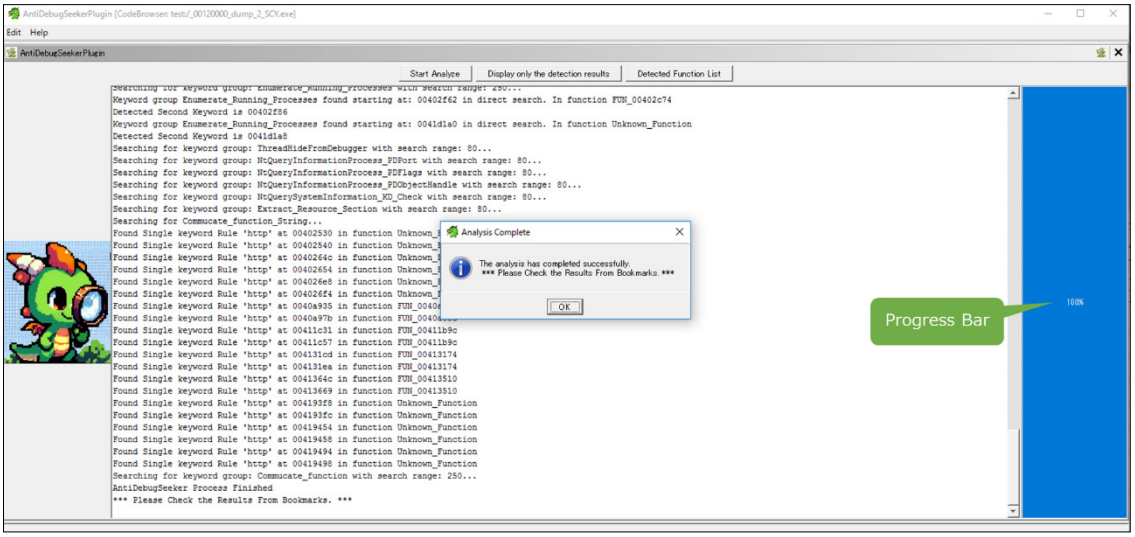


Figure 23: Plugin analysis completion screen.

Upon detection by Anti_Debug_Technique, the system will display the name of the detection rule, the position of the detected keyword within the rule, and the address where the detection occurred.

```
Keyword group Memory_EXECUTE_READWRITE_1 found starting at: 0040203f in direct search. In function FUN_00402004
Detected Second Keyword is 00402041
Detected Third Keyword is 0040204a
Searching for keyword group: Memory_EXECUTE_READWRITE_2 with search range: 20...
Keyword group Memory_EXECUTE_READWRITE_2 found starting at: 0040202d in direct search. In function FUN_00402004
Detected Second Keyword is 00402035
Keyword group Memory_EXECUTE_READWRITE_2 found starting at: 0040214d in direct search. In function FUN_00402114
Detected Second Keyword is 00402155
Keyword group Memory_EXECUTE_READWRITE_2 found starting at: 00414dc2 in direct search. In function entry
Detected Second Keyword is 00414dd6
Searching for keyword group: Memory_Region_Tracking with search range: 250...
Searching for keyword group: Check_BreakPoint_Memory_1 with search range: 80...
Searching for keyword group: Check_BreakPoint_Memory_2 with search range: 80...
Searching for keyword group: Software_Breakpoints_Check with search range: 300...
Searching for keyword group: Hardware_Breakpoints_Check with search range: 80...
Searching for keyword group: Enumerate_Running_Processes with search range: 250...
Keyword group Enumerate_Running_Processes found starting at: 00402f62 in direct search. In function FUN_00402c74
Detected Second Keyword is 00402f86
```

Figure 24: Display when detected by the rules specified in Anti_Debug_Technique.

Clicking the 'Display only the detection results' button, as shown in Figures 25 and 26, displays only the detected items, making it easy to review the results.

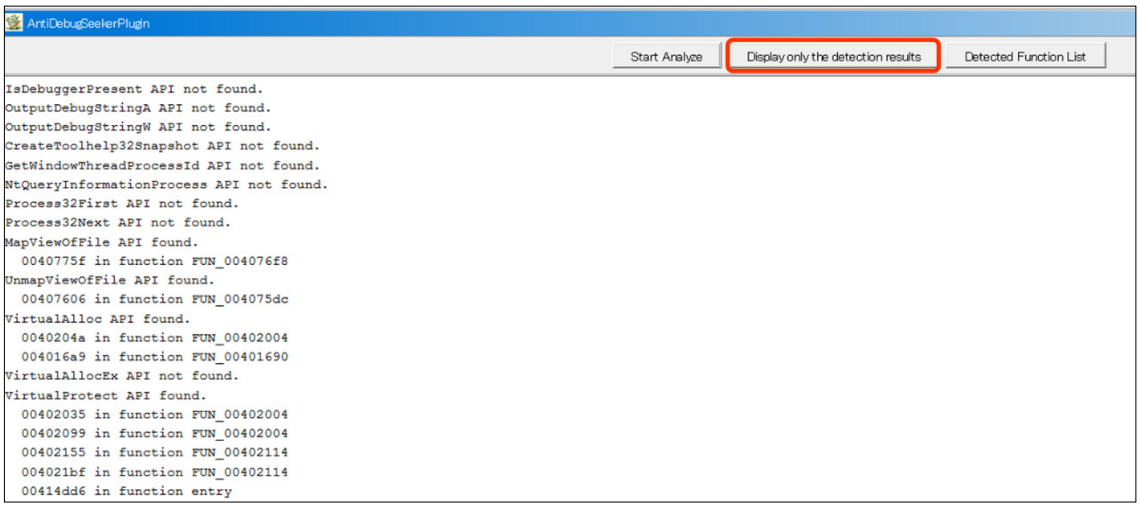


Figure 25: Before processing 'Display only the detection results'.

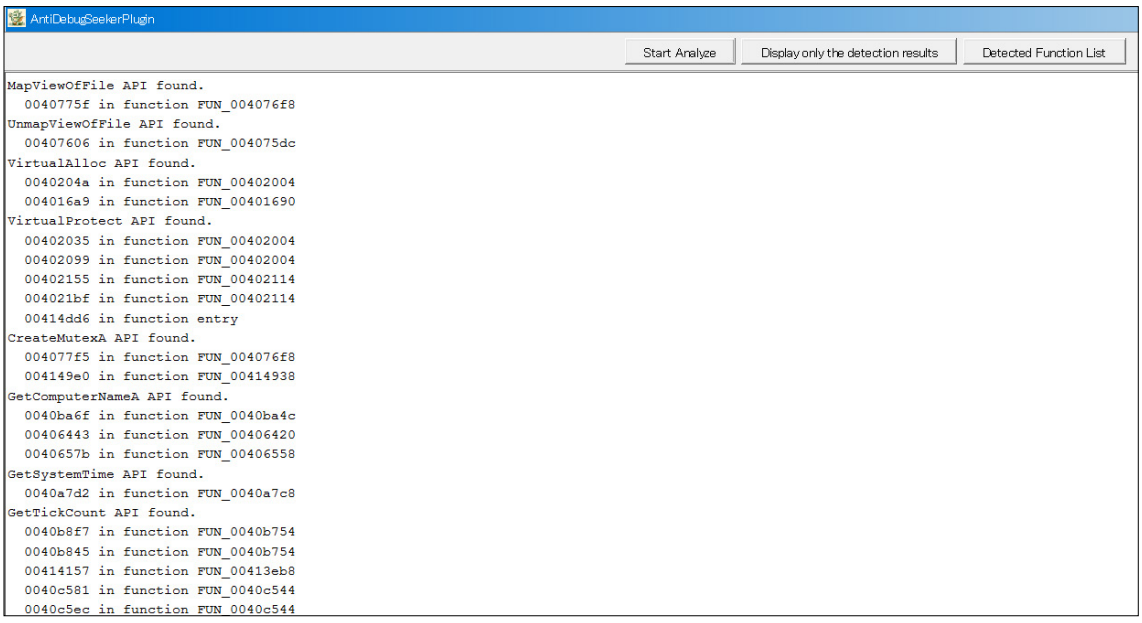


Figure 26: After processing 'Display only the detection results'.

In the states depicted in Figures 25 and 26, before and after processing 'Display only the detection results', clicking the 'Detected Function List' button groups the detection results by function. This organization facilitates a clearer understanding of the anti-debugging features at the function level.

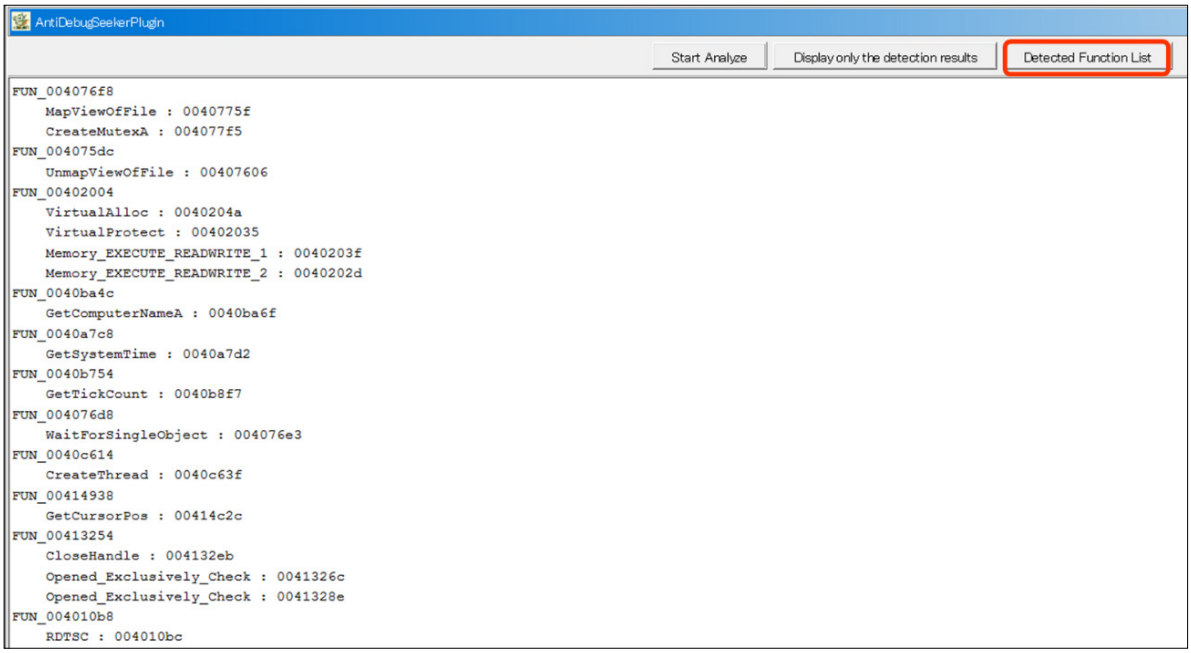


Figure 27: After processing 'Detected Function List'.

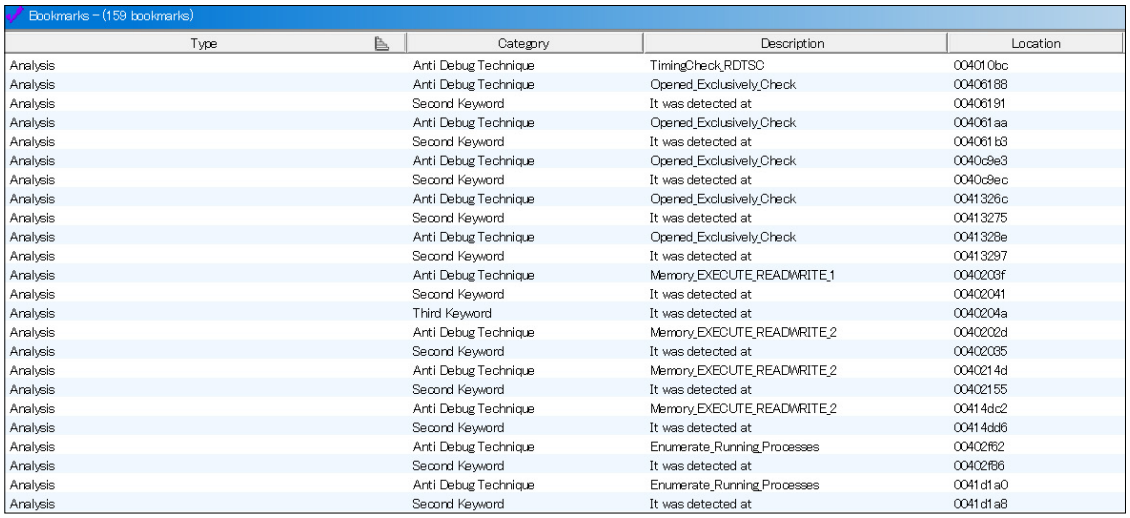
Ghidra script / module extension: check Bookmarks

In Figures 28 and 29, the detection results are registered in Ghidra's Bookmark, allowing for easy verification. The category labelled 'Potential of Anti Debug API' indicates detections based on the rules specified in the Anti_Debug_API section of the anti_debug_Ghidra.config file. Similarly, the category labelled 'Anti Debug Technique' signifies detections based on the rules in the Anti_Debug_Technique section. Additionally, entries such as 'Second Keyword' or 'Third Keyword' under 'Anti Debug Technique' denote the specific locations where the defined keywords were detected.

Bookmarks – (159 bookmarks)

Type	Category	Description	Location
Analysis	Potential of Anti Debug API	Memory Manipulation : MapViewOfFile	0040775f
Analysis	Potential of Anti Debug API	Memory Manipulation : UnmapViewOfFile	00407606
Analysis	Potential of Anti Debug API	Memory Manipulation : VirtualAlloc	0040204a
Analysis	Potential of Anti Debug API	Memory Manipulation : VirtualAlloc	004016a9
Analysis	Potential of Anti Debug API	Memory Manipulation : VirtualProtect	00402035
Analysis	Potential of Anti Debug API	Memory Manipulation : VirtualProtect	00402089
Analysis	Potential of Anti Debug API	Memory Manipulation : VirtualProtect	00402155
Analysis	Potential of Anti Debug API	Memory Manipulation : VirtualProtect	004021bf
Analysis	Potential of Anti Debug API	Memory Manipulation : VirtualProtect	0041 4dd6
Analysis	Potential of Anti Debug API	Mutual Exclusion : CreateMutexA	004077f5
Analysis	Potential of Anti Debug API	Mutual Exclusion : CreateMutexA	004149e0
Analysis	Potential of Anti Debug API	Analysis Environment Check : GetComputerNameA	0040ba6f
Analysis	Potential of Anti Debug API	Analysis Environment Check : GetComputerNameA	00406443
Analysis	Potential of Anti Debug API	Analysis Environment Check : GetComputerNameA	0040657b
Analysis	Potential of Anti Debug API	Time Check : GetSystemTime	0040a7d2
Analysis	Potential of Anti Debug API	Time Check : GetTickCount	0040b8f7
Analysis	Potential of Anti Debug API	Time Check : GetTickCount	0040b845
Analysis	Potential of Anti Debug API	Time Check : GetTickCount	00414157
Analysis	Potential of Anti Debug API	Time Check : GetTickCount	0040c581
Analysis	Potential of Anti Debug API	Time Check : GetTickCount	0040c5ec
Analysis	Potential of Anti Debug API	Time Check : WaitForSingleObject	004076e3
Analysis	Potential of Anti Debug API	Time Check : WaitForSingleObject	004070ea
Analysis	Potential of Anti Debug API	Time Check : WaitForSingleObject	00413942
Analysis	Potential of Anti Debug API	Time Check : WaitForSingleObject	00406f20
Analysis	Potential of Anti Debug API	Time Check : WaitForSingleObject	00410e16

Figure 28: After processing 'Detected Function List'.

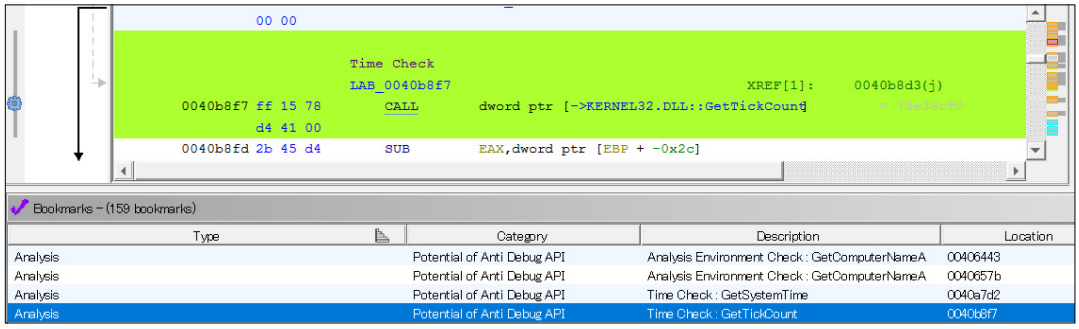


Bookmarks – (159 bookmarks)

Type	Category	Description	Location
Analysis	Anti Debug Technique	TimingCheck_RDTSC	004010bc
Analysis	Anti Debug Technique	Opened_Exclusively_Check	00406188
Analysis	Second Keyword	It was detected at	00406191
Analysis	Anti Debug Technique	Opened_Exclusively_Check	004061aa
Analysis	Second Keyword	It was detected at	004061b3
Analysis	Anti Debug Technique	Opened_Exclusively_Check	0040c9e3
Analysis	Second Keyword	It was detected at	0040c9ec
Analysis	Anti Debug Technique	Opened_Exclusively_Check	0041326c
Analysis	Second Keyword	It was detected at	00413275
Analysis	Anti Debug Technique	Opened_Exclusively_Check	0041328e
Analysis	Second Keyword	It was detected at	00413297
Analysis	Anti Debug Technique	Memory_EXECUTE_READWRITE_1	0040203f
Analysis	Second Keyword	It was detected at	00402041
Analysis	Third Keyword	It was detected at	0040204a
Analysis	Anti Debug Technique	Memory_EXECUTE_READWRITE_2	0040202d
Analysis	Second Keyword	It was detected at	00402035
Analysis	Anti Debug Technique	Memory_EXECUTE_READWRITE_2	0040214d
Analysis	Second Keyword	It was detected at	00402155
Analysis	Anti Debug Technique	Memory_EXECUTE_READWRITE_2	00414dc2
Analysis	Second Keyword	It was detected at	00414dd6
Analysis	Anti Debug Technique	Enumerate_Running_Processes	00402f62
Analysis	Second Keyword	It was detected at	00402f86
Analysis	Anti Debug Technique	Enumerate_Running_Processes	0041d1a0
Analysis	Second Keyword	It was detected at	0041d1a8

Figure 29: After processing 'Detected Function List'.

Items detected by the Anti Debug API are highlighted with a green background, and the rule name is annotated as a PRE comment.



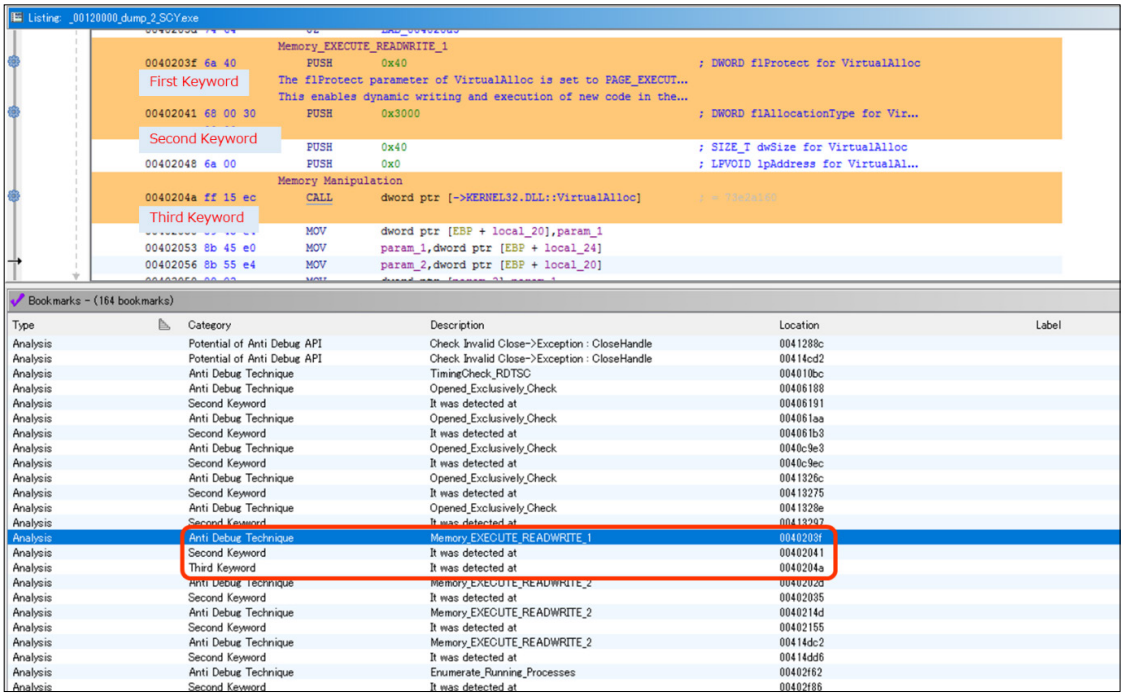
```
                    00 00
                    Time Check
                    LAB_0040b8f7                          XREF[1]:   0040b8d3(j)
0040b8f7 ff 15 78   CALL        dword ptr [->KERNEL32.DLL::GetTickCount]
         d4 41 00
0040b8fd 2b 45 d4   SUB         EAX,dword ptr [EBP + -0x2c]
```

Bookmarks – (159 bookmarks)

Type	Category	Description	Location
Analysis	Potential of Anti Debug API	Analysis Environment Check : GetComputerNameA	00406443
Analysis	Potential of Anti Debug API	Analysis Environment Check : GetComputerNameA	0040657b
Analysis	Potential of Anti Debug API	Time Check : GetSystemTime	0040a7d2
Analysis	Potential of Anti Debug API	Time Check : GetTickCount	0040b8f7

Figure 30: Disassembly screen showing detections from the Anti_Debug_API section of anti_debug_Ghidra.config.

Items detected by the Anti Debug Technique are highlighted with an orange background, and the rule name is annotated as a PRE comment. The details of the rule are displayed as a POST comment, derived from the data in the loaded JSON file.



```
Memory_EXECUTE_READWRITE_1
0040203f 6a 40      PUSH        0x40                    ; DWORD flProtect for VirtualAlloc
                    The flProtect parameter of VirtualAlloc is set to PAGE_EXECUT...
                    This enables dynamic writing and execution of new code in the...
00402041 68 00 30   PUSH        0x3000                  ; DWORD flAllocationType for Vir...
00402046 6a 00      PUSH        0x40                    ; SIZE_T dwSize for VirtualAlloc
00402048 6a 00      PUSH        0x0                     ; LPVOID lpAddress for VirtualAl...
                    Memory Manipulation
0040204a ff 15 ec   CALL        dword ptr [->KERNEL32.DLL::VirtualAlloc]
00402050 89 45 e0   MOV         dword ptr [EBP + local_20],param_1
00402053 8b 45 e0   MOV         param_1,dword ptr [EBP + local_24]
00402056 8b 55 e4   MOV         param_2,dword ptr [EBP + local_20]
```

First Keyword

Second Keyword

Third Keyword

Bookmarks – (164 bookmarks)

Type	Category	Description	Location	Label
Analysis	Potential of Anti Debug API	Check Invalid Close->Exception : CloseHandle	0041288c	
Analysis	Potential of Anti Debug API	Check Invalid Close->Exception : CloseHandle	00414cd2	
Analysis	Anti Debug Technique	TimingCheck_RDTSC	004010bc	
Analysis	Anti Debug Technique	Opened_Exclusively_Check	00406188	
Analysis	Second Keyword	It was detected at	00406191	
Analysis	Anti Debug Technique	Opened_Exclusively_Check	004061aa	
Analysis	Second Keyword	It was detected at	004061b3	
Analysis	Anti Debug Technique	Opened_Exclusively_Check	0040c9e3	
Analysis	Second Keyword	It was detected at	0040c9ec	
Analysis	Anti Debug Technique	Opened_Exclusively_Check	0041326c	
Analysis	Second Keyword	It was detected at	00413275	
Analysis	Anti Debug Technique	Opened_Exclusively_Check	0041328e	
Analysis	Second Keyword	It was detected at	00413297	
Analysis	Anti Debug Technique	Memory_EXECUTE_READWRITE_1	0040203f	
Analysis	Second Keyword	It was detected at	00402041	
Analysis	Third Keyword	It was detected at	0040204a	
Analysis	Anti Debug Technique	Memory_EXECUTE_READWRITE_2	0040202d	
Analysis	Second Keyword	It was detected at	00402035	
Analysis	Anti Debug Technique	Memory_EXECUTE_READWRITE_2	0040214d	
Analysis	Second Keyword	It was detected at	00402155	
Analysis	Anti Debug Technique	Memory_EXECUTE_READWRITE_2	00414dc2	
Analysis	Second Keyword	It was detected at	00414dd6	
Analysis	Anti Debug Technique	Enumerate_Running_Processes	00402f62	
Analysis	Second Keyword	It was detected at	00402f86	

Figure 31: Disassembly screen showing detections from the Anti_Debug_Technique section of anti_debug_Ghidra.config.

CONCLUSION

AntiDebugSeeker is a tool designed to automatically detect and analyse anti-debugging features commonly found in malware. For those who have little experience in analysing anti-debugging malware and find it challenging, the detailed descriptions of the rules in the JSON file can serve as a valuable reference. While this tool specializes in detecting anti-debugging techniques, it also highlights common malware techniques through its detection rules. To make this tool accessible to a broader audience, versions have been developed for both IDA and Ghidra. The functional differences between the IDA and Ghidra versions are minimal, allowing users to choose the version that best suits their preferences.

I will update this work as promptly as possible upon the discovery of new anti-debugging techniques. Alternatively, if you report a new technique, I will ensure it is incorporated in the updates. Your collaboration in the development of this tool would be greatly appreciated.

REFERENCES

[1] IDA_Plugin_AntiDebugSeeker. https://github.com/LAC-Japan/IDA_Plugin_AntiDebugSeeker.

[2] Ghidra_AntiDebugSeeker. https://github.com/LAC-Japan/Ghidra_AntiDebugSeeker.