Detection and Blocking Method against DLL Injection Attack Using PEB-LDR of ICS EWS in Smart IoT Environments

Junwon Kim[1], Jiho Shin[2], Jung Taek Seo[3*]

[1] Department of Information Security Engineering, Gachon University, South Korea
[2] Police Science Institute, Korean National Police University, South Korea
[3] Department of Computer Engineering, Gachon University, South Korea
junone@gachon.ac.kr, suchme@police.go.kr, seojt@gachon.ac.kr

Abstract

Modern Industrial Control System (ICS) can provide vast functions as the introduction of IT technology is established along with the introduction of the IoT environment. Engineering Workstation (EWS) used by ICS is widely used to efficiently manage and control industrial devices including smart IoT devices. However, the DLL injection attack in ICS is not high in difficulty compared to the risk, but it can cause fatal malfunction. If an attack is carried out targeting the EWS, it may cause erroneous operation in many control devices, including IoT devices, cause fatal accidents throughout the Supervisory Control and Data Acquisition (SCADA) system. In this paper, we present a method to detect DLL injection attacks by specializing in EWS used in ICS in IoT environment and purpose a method to detect data changes due to DLL injection attacks by analyzing and utilizing *PEB-LDR* data. Also, we purpose a method to detect and block execution when a malicious DLL is suspected to be loaded by DLL injection.

Keywords: Industrial Control System (ICS), Internet of Things (IoT), Engineering Workstation (EWS), Process Environment Block (PEB), Dynamic Link Library (DLL), Injection

1 Introduction

In recent years, the succession of the IT technology of the Industrial Control System (ICS) has changed it to a more flexible and effective operation method, and the modern ICS is utilized in infrastructures and improved productivity and operational efficiency such as power plant, transportation, smart city. Engineering Workstation (EWS) used in ICS is an integrated tool that can manage Programmable Logic Controller (PLC) logic and monitor industrial devices. However, the introduction of IT technology not only improved the level of ICS operation, but also led to the appearance of new attacks that did not appear before, such as an increase in the attack surface and connection to an external network. Unlike the information system, the existing ICS operates based on a closed network, so it is recognized that it is safe from external attacks. Conversely, this awareness is an important issue that can lead to negligence of internal workers' security and cause potential risk that can cause physical destruction, property damage. In recent years, as smart IoT and industrial IoT devices start to converge with ICS and operate together, it is difficult to rely on the physical environment security of ICS any longer [1]. Now, as IoT technology is introduced into ICS, security-related functions such as key management, intrusion detection, and additional access control are required [1]. A lot of time has been devoted to research on intelligent methods to detect and identify risk factors of installed industrial IoT devices [3-4]. This operating environment includes the smart IoT mobile environment in which ICS supports IoT-based mobility [5], and ICS security is also unavoidable for proper computing operation. Among ICS threats based on IoT environment; Dynamic Link Library (DLL) injection attack is not high in execution difficulty compared to attack severity, and an attacker can easily control the system, which is a serious attack that can cause fatal accidents to the infrastructure. For example, the Stuxnet malware caused malicious behavior by damaging the EWS installed on a network PC that was closed in a nuclear facility [6]. As a result, it is impossible to identify if the actual attack was caused by a DLL injection attack used by a malicious intruder. Thus, the logic data of Siemens PLC could not be observed normally. It resulted in physical destruction of the device [7]. As such, there is an attack tendency to change core functions targeting EWS, and DLL injection attack detection and defense technology targeting EWS is essential.

In this paper, we analyzed the behavioral tendency of attackers to perform DLL injection to detect DLL injection attacks specialized for EWS. In addition, we proposed a whitelist chain-based detection method using *PEB-LDR* data existing in the EWS process, and able to detect the intrusion of malicious DLLs through DLL injection. Also, we proposed the methods to detect the DLL injection and applied the whitelist chain concept that considers the operational characteristics of the IoT-based ICS environment, using the Microsoft Detour [8] tool together with a technology to block DLL injection, an attempt by an attacker to inject a DLL in the EWS process. This paper consists of 7 detailed sessions, which are as follows:

Session 2 describes the analysis of existing research cases to prevent DLL injection in the ICT environment and describes the direction to be supplemented by specializing in the ICS based-on IoT environment. Session 3 describes the results of observing the data of the Process Environment Block (PEB) structure containing process information, the results of examining the *Ldr* field to obtain the DLL information

*Corresponding Author: Jung Taek Seo; E-mail: seojt@gachon.ac.kr
DOI: 10.53106/160792642022072304022

referenced in the process, the analysis of the list data contained in the *Ldr* field. In Session 4, through the field analysis result of LDR_DATA_TABLE_ENTRY (*LDTE*) type analyzed in Session 3, it describes a method to create a whitelist chain that can detect DLL injection considering environmental factors that can be used for ICS based on IoT environment. Session 5 presents a method of acquiring information from a target EWS process to create the whitelist chain proposed in Session 4 and a monitoring method for detecting the inserted DLL. Session 6 describes the details and results of the detectability experiment by applying the whitelist chain-based collection method proposed in Sessions 4 and 5, to block the injected malicious DLL, the behavioral tendency of DLL injection from previous studies. By suggesting and applying a technique that can block it, proved that it is possible to ultimately block the DLL injection attack. Finally, Session 7 describes the experimental results and analysis contents were comprehensively summarized, and the future scope we intend to proceed.

1.1 Contributions

We presented the method to detect and block the DLL injection that aimed at EWS running on IoT Environment-based ICS and analyzed the PEB information that can identify or acquire DLL-related events and actions, identified the components of detailed fields. The above findings are expected to contribute as follows:

Information gathering. We investigated the PEB data structure to discover elements that can detect DLL injection in the Windows OS environment, and collected information from the *Ldr* detail field, a key field of PEB. These analysis results can be effectively utilized when acquiring DLL information used by the process.

Detection and blocking. By targeting the EWS used in the ICS environment, it is possible to minimize the attack damage by proposing a method to detect and block DLL injection attacks.

Versatility. The proposed technique can be applied not only to the EWS process, but also to important programs in other fields, which can protect against DLL injection, and can also be applied to target multiple processes whose availability may be compromised by an attacker.

2 Related Works

Although the DLL injection attack is a well-known and threatening attack in the IT field, ICS combined with IT technology is now forming a target that can threaten it. Relevant research includes the following categories: Klein et al. (2019) presented various trends and attack penetration processes of APIs that attackers call to perform intra-process injection [9]. In addition, Park et al. (2015) analyzed a malicious code execution routine that allows malware to execute code for Command and Control (C&C) in a process through Windows API hooking [10]. Through these research results, they studied the major APIs and function calling methods used 2 to succeed DLL injection and warned of the high fatality rate of DLL injection attacks. Originally, DLL injection was originally a method widely used for code patching and maintenance, but it proved that there are not a few cases of using it for malicious purposes. Due to this recognition, several techniques have been proposed to protect against high-risk DLL injection attacks. For example, Lee et al. (2020) to propose an anti-injection technique for DLL injection targeting Supervisory Control and Data Acquisition (SCADA), In order to block DLL injection, an attacker's tendency has been analyzed and a method of monitoring and blocking through Import Address Table (IAT) hooking has been proposed [7]. In addition, Berdajs et al. (2010) hooked the CreateRemoteThread function, which is mainly used for DLL injection, to block separate inspection codes through inline-based code insertion [11]. Sun et al. (2006) proposed an exploit blocking technique through API monitoring using Process Environment Block (PEB) data [12]. These studies succeeded in blocking the x86 based operating system, but it needs to conduct and verify an experiment on the x64 (AMD64) based operating system. Conclusively, there is a limit that the x86 call stack-based blocking technique is incompatible with the 64-bit method difference. In addition, APIs with suffixes such as *-Ex, -A, -W, -ExW,* etc. exist depending on the use of Unicode, multi-byte character set, and extended type of functions as the main API used for DLL injection. Hooking and verifying all these APIs has a disadvantage in that it is less effective in terms of performance. Crucially, there are cases where the DLL injection technique is used for good faith purposes such as patching and maintenance of vulnerable code, and when an injection event occurs, an additional method of determining whether it is a normal event is needed. As the proportion of programs expanded to 64-bit increases, additional defense techniques are also required. A study to block malicious DLL execution was also presented, Syed et al. (2015) proposed a technique to block malicious DLLs loaded in memory by detecting Windows API hooking [13], Mira (2019) proposed a method to detect abnormal activity by analyzing DLL data and monitoring the sequence of API calls to block the inflow of malicious DLLs [14]. However, as proposed by Shankarapani et al. (2011) [15], Yusirwan et al. (2015) [16], Bae et al. (2019) [17], presented the technical method to conceal and obfuscate the malware execution routine, these case of deliberately packed advanced malware, these malicious codes have limitations in lowering the success rate of not only API trend-based detection techniques but also code data-based detection techniques. Specially, since the DLL injection attack operates with a routine very similar to the existing injection-based good faith program, determining whether to determine the attack through code data that induces loading a malicious DLL has a problem that may cause a false diagnosis. For instance, the Stuxnet case also caused substantial damage by the injected malicious DLL, not the result of executing the code executing the DLL injection and has the disadvantage of having to go through 2 detection processes (injection, code execution) to block [18].

Recently, with the development of computing resources, various research cases using machine learning (AI) have appeared. Fan et al. (2015) proposed a method to detect malware by mining API log data [19], Hwang et al. (2017) conducted a study to block DLLs by learning the features of the section data area and the features of DLLs to analyze malware [20]. Ha et al. (2018) conducted a study to block the execution of malicious DLLs by learning API statistics for malicious DLLs [21]. In addition, Matsuda et al. (2020) investigated various methods based on DLL data and proved a recall result of 97.45% for the malware detection rate using

a deep learning algorithm [22]. This technique is highly accurate because it detects based on linear algebra values such as statistical estimation and regression analysis based on machine learning. However, in order to derive meaningful results, there is a disadvantage in that the learning step requires a lot of time and a lot of learning data, but the data must be composed of quantitative data. In other words, a detection technique based on machine learning requires many pre-processing and learning processes to derive results. In addition, in the industrial control system environment where the EWS process operates, there are restrictions on the application of detection techniques due to the use of limited resources for securing availability and the use of unpublished industrial protocols [7, 23]. Notably, since the industrial control system has a communication system based on a closed network and unidirectional data transmission, it is difficult to apply a detection technique for external inflow such as network monitoring [23-24].

When considering the ICS environment, verification is possible in both x86/x64 based environments as presented above, and a plan is needed to increase the efficiency of detection based on existing API hooking. In addition, it does not depend on an external connection such as a network and must be able to meet resource constraints that can minimize preprocessing for detection and blocking. To this end, we analyzed the existing API calling method to detect and block malicious DLL injection along with a whitelist chain design technique using *PEB-LDR* data and proposed a method to increase detection efficiency. This not only complements the existing limitations and disadvantages but can also be additionally applied to the IT field and contributed to the deduction of DLL injection detection and blocking method technologies.

3 Overview and Analysis of *PEB-LDR DATA (PLD)*

3.1 Observing PEB Data of Process

PEB, one of the basic data structures used in Windows OS, loads process information, and PEB information exists in all processes running in the OS [25]. PEB is defined as a single structure and is a data structure containing process load information [26]. Microsoft has disclosed the purpose of use of the seven fields constituting the PEB based on the date of analysis [26]. If "*nt!_PEB*" structure is mapped using WinDbg for the PEB address of an arbitrary process, Figure 1 displays not only the result that it can be verified that *Ldr* is included in the PEB structure but also the same as the result of inputting "*dt nt!_PEB @$PEB*" command in WinDbg. Among the PEB fields, we propose a detection method by using the *Ldr* field data that loads the information of the loaded module in Section 3.2.

```
0:007> dt nt!_PEB @$PEB
ntdll!_PEB
   +0x000 InheritedAddressSpace : 0 ''
   +0x001 ReadImageFileExecOptions : 0 ''
   +0x002 BeingDebugged    : 0x1 ''
   +0x003 BitField         : 0x84 ''
   +0x003 ImageUsesLargePages : 0y0
   +0x003 IsProtectedProcess : 0y0
   +0x003 IsImageDynamicallyRelocated : 0y1
   +0x003 SkipPatchingUser32Forwarders : 0y0
   +0x003 IsPackagedProcess : 0y0
   +0x003 IsAppContainer   : 0y0
   +0x003 IsProtectedProcessLight : 0y0
   +0x003 IsLongPathAwareProcess : 0y1
   +0x004 Padding0         : [4]  ""
   +0x008 Mutant           : 0xffffffff`ffffffff Void
   +0x010 ImageBaseAddress : 0x00007ff7`77220000 Void
   +0x018 Ldr              : 0x00007ff9`8d29a4c0 _PEB_LDR_DATA
   +0x020 ProcessParameters : 0x00000252`46223430 _RTL_USER_PR
   +0x028 SubSystemData    : 0x00007ff9`854c91d0 Void
   +0x030 ProcessHeap      : 0x00000252`46220000 Void
   +0x038 FastPebLock      : 0x00007ff9`8d29a0e0 _RTL_CRITICAL_
   +0x040 AtlThunkSListPtr : (null)
   +0x048 IFEOKey          : (null)
   +0x050 CrossProcessFlags : 0
   +0x050 ProcessInJob     : 0y0
   +0x050 ProcessInitializing : 0y0
   +0x050 ProcessUsingVEH  : 0y0
   +0x050 ProcessUsingVCH  : 0y0
```

Figure 1. *Ldr* field which located at PEB

3.2 Detailed Analysis for *PLD* Structure

The *Ldr* field is a data structure variable with *PLD* structure format, and the *PLD* structure consists of three fields [27]. Since *Ldr* contains the information of modules loaded in the process, we present a method for constructing information that can detect DLL injection by using this information in Section 3.3. However, before using *Ldr* data, we checked that the purpose of use of some data fields constituting the PEB structure and *PLD* is still not documented. The process of obtaining detailed information for each field is necessary, and research cases that analyze the purpose of use of each field constituting the *PLD* structure data including PEB data were referred to [9, 12]. Debugging tools such as WinDbg were used to learn detailed field information, and PEB and *PLD* symbols can be found by connecting to Microsoft Symbol Server [28]. Through this, the fields of each data structure can be briefly analyzed. Based on the operating system version Windows 10 Pro 64-bit, build 18362, it was verified that the details of the PEB and *PLD* field data differ from the previous research results, and this is estimated due to internal reasons of Microsoft. Particularly, it was verified that there are 9 fields in the *PLD* that we want to use for detection, Table 1 and Table 2 describes a detail of each field.

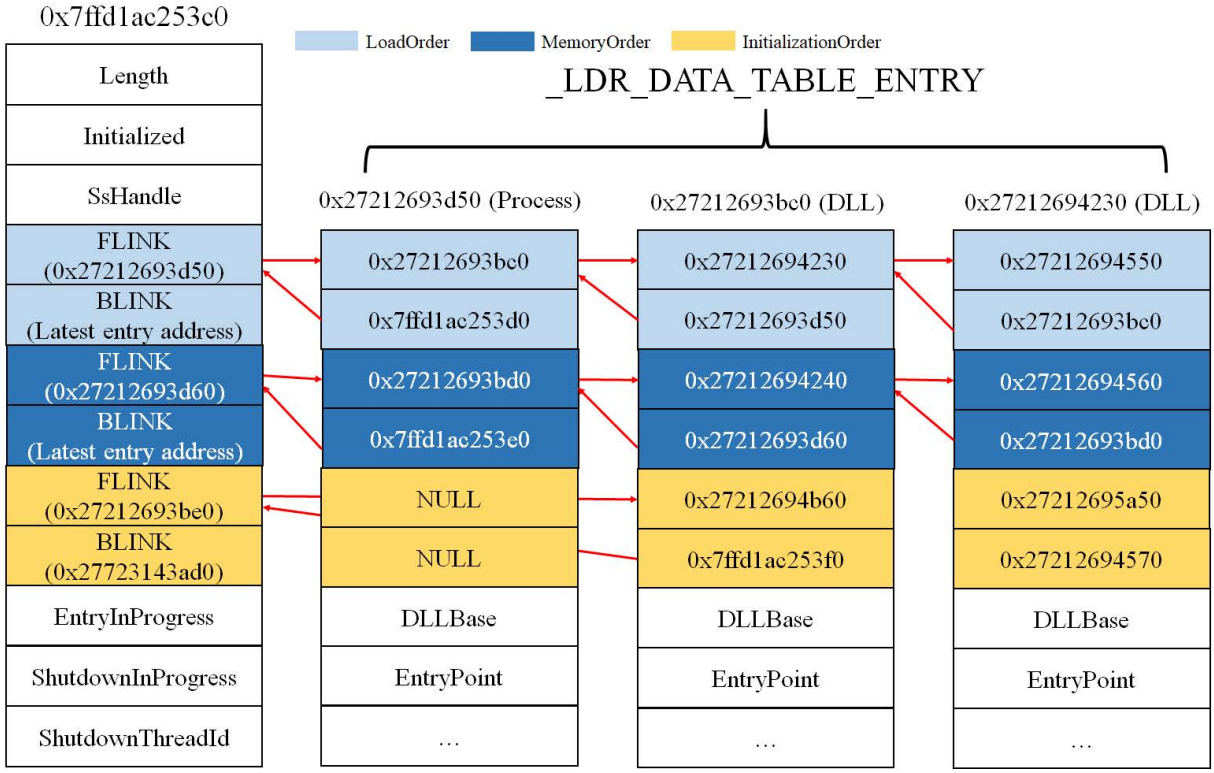


Figure 2. Example of LDTE linkages, depicted in a double linked list (Base on Windows 10 Pro 64-bit, build 18362)

Table 1. *PLD* structure data fields & data type (Based on Windows 10 Pro 64-bit, build 18362)

Field Name	Type
Length	ULONG
Initialized	UCHAR
SsHandle	HANDLE
InLoadOrderModuleList	LIST_ENTRY
InMemoryOrderModuleList	LIST_ENTRY
InInitializationOrderModuleList	LIST_ENTRY
EntryInProgress	PVOID
ShutdownInProgress	UCHAR
ShutdownThreadId	PVOID

Table 2. *PLD* structure data fields description (Based on Windows 10 Pro 64-bit, build 18362)

Field Name	Description of usage purpose
Length	The size of the structure used by ntdll.dll as Structure Version ID
Initialized	If *TRUE*, the Loader Data Session for the current process is initialized
SsHandle	Unidentified
InLoadOrder ModuleList	Contains the order in which modules are loaded, the address pointer is linked to a circular double-linked list
InMemory OrderModule List	Same as above, but contains memory arrangement order
InInitialization OrderModule List	Same as *InLoadOrder ModuleList*, but contains initialization order
EntryIn Progress	Not used in Windows 10
ShutdownIn Progress	Unidentified
Shutdown ThreadId	The thread ID suggested by the name, picked up from the UniqueThread member of *CLIENT_ID* in the Thread Environment Block (TEB) of the thread requesting process termination

3.3 Analysis of Changing the Specific Field Data & *LDR_DATA_TABLE_ENTRY (LDTE)* Structure

The fields constituting the *PLD* are described in Table 2. Among them, we continuously observed and analyzed changes in the field values of the *PLD* data each process operating in the OS. Among them, it is found that the data in a module-list (*–ModuleList*) field has a different value for each process, and it contains the DLL information that is being loaded into the program. The module-list includes 3 specific field as follows: *InLoadOrderModuleList (LOML)*, *InitializationOrderModuleList (IOML)*, and *InMemoryOrderModuleList (MOML)*, and the data stored in each field is different depending on the purpose of use. It can

be describing that the *LOML* field stores static DLL module information and process information that the process initially loads during the bootstrapping process. *IOML* is a structure that is almost like the data of the *LOML* field and has information about the DLL module that was initially loaded during bootstrap and performed until initialization but does not include process information [7]. *MOML* can verify that module information for all DLLs loaded in a dynamic manner is stored while the program is running. In the way that the module-list field loads module data, the variable with the LIST_ENTRY structure is loaded in a circular linked list as shown in Figure 2. It is connected to another LIST_ENTRY variable address using 2 fields (*Flink, Blink*) of the LIST_ENTRY structure, and the memory address loading the actual module information can be read by referring to the address values stored in the *Flink* and *Blink* fields. The data loading the actual module information is *LDTE* structure unit, and Microsoft documented and disclosed only 11 *LDTE* structure fields [27], but as a result of mapping code symbols, it can verify that there are 30 actual fields. However, we have selected 6 fields and 2 module-list *(OrderLink)* fields that can be used for DLL injection detection and blocking among 30 fields to increase efficiency and addresses for each field are based on x86 and x64. The results are shown in Figure 3 and Figure 4. The specific application plan for each field is described in Section 4.2.

```
0:007> dt nt!_LDR_DATA_TABLE_ENTRY 252`46223ce0-0x10
ntdll!_LDR_DATA_TABLE_ENTRY
   +0x000 InLoadOrderLinks : _LIST_ENTRY [ 0x00000252`462243f0 - 0x00000252`46
   +0x010 InMemoryOrderLinks : _LIST_ENTRY [ 0x00000252`46224400 - 0x00000252
   +0x020 InInitializationOrderLinks : _LIST_ENTRY [ 0x00000252`46224a20 - 0x
   +0x030 DllBase          : 0x00007ff9`8d130000 Void
   +0x038 EntryPoint       : (null)
   +0x040 SizeOfImage      : 0x1f5000
   +0x048 FullDllName      : _UNICODE_STRING "C:\WINDOWS\SYSTEM32\ntdll.dll"
   +0x058 BaseDllName      : _UNICODE_STRING "ntdll.dll"
   +0x068 FlagGroup        : [4]  "???"
   +0x068 Flags            : 0xa2c4
   +0x068 PackagedBinary   : 0y0
   +0x068 MarkedForRemoval : 0y0
   +0x068 ImageDll         : 0y1
   +0x068 LoadNotificationsSent : 0y0
   +0x068 TelemetryEntryProcessed : 0y0
   +0x068 ProcessStaticImport : 0y0
   +0x068 InLegacyLists    : 0y1
   +0x068 InIndexes        : 0y1
   +0x068 ShimDll          : 0y0
   +0x068 InExceptionTable : 0y1
   +0x068 ReservedFlags1   : 0y00
   +0x068 LoadInProgress   : 0y0
   +0x068 LoadConfigProcessed : 0y1
   +0x068 EntryProcessed   : 0y0
   +0x068 ProtectDelayLoad : 0y1
   +0x068 ReservedFlags3   : 0y00
   +0x068 DontCallForThreads : 0y0
   +0x068 ProcessAttachCalled : 0y0
   +0x068 ProcessAttachFailed : 0y0
   +0x068 CorDeferredValidate : 0y0
   +0x068 CorImage         : 0y0
   +0x068 DontRelocate     : 0y0
   +0x068 CorILOnly        : 0y0
   +0x068 ChpeImage        : 0y0
   +0x068 ReservedFlags5   : 0y00
   +0x068 Redirected       : 0y0
   +0x068 ReservedFlags6   : 0y00
```

Figure 3. LDTE symbol analysis result (Based on Windows 10 Pro 64-bit, build 18362)

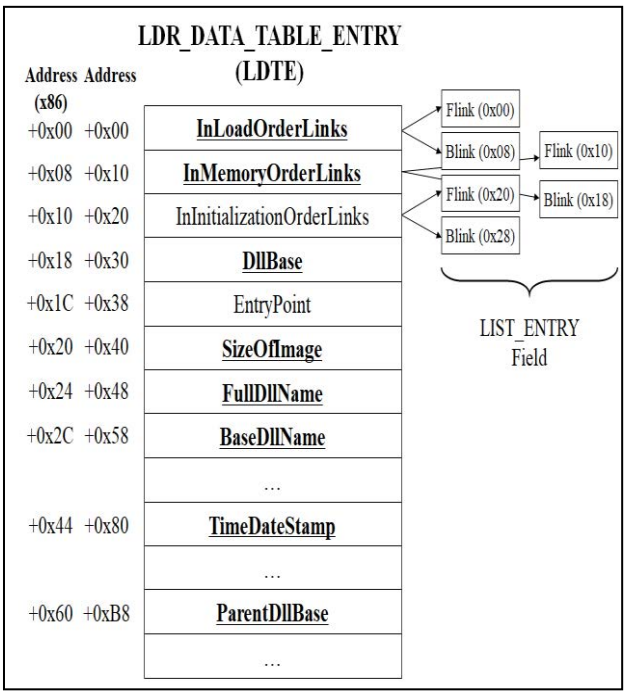


Figure 4. LDTE strcture data fields and selected (Based on Windows 10 Pro 64-bit, Build 18362)

4 Proposal of Generating Whitelist Chain Using *LDTE*

We propose a whitelist chain design to detect and block DLL injections and analyzed *LDTE* data changes to design the chain. In addition, the nodes constituting the chain were created by selecting several fields constituting the *LDTE*. A detailed method for this is proposed below in Section 4.1.

4.1 Analysis of Changing the Specific Field Data & *LDR_DATA_TABLE_ENTRY (LDTE)* Structure

We analyzed that the module-list fields of the *PLD* structure have LIST_ENTRY type, and that the addresses loaded in *LDTE* data units are connected. Since the purpose of use of each field of *LDTE* is not documented at all, it is difficult to accurately analyze all fields, but as a result, it can be checked that some field data is being changed with certain rules. First, the *OrderLinks* field existing in the *LDTE* field has the same link structure as the module-list of the *PLD* structure, which has the same data. In other words, the module-list field in the *PLD* structure is the same as *LDTE*'s *OrderLinks*, which is the same as connecting "*LDTE* nodes" in linked list format. Second, each time a DLL is loaded in each process, it has a different timestamp data. It is allocated to *LDTE*'s *TimeDateStamp* field, which is an Unsigned Long type, it can be expressed in a date format as shown in Figure 4 (Located at +0x080). However, not used for the purpose of identifying the date, it can be verified that it is a unique value by the deterministic compilation method in the process. This is a field that can identify the uniqueness of each DLL even if it has the same DLL name. Third, in the *ParentDllBase* field, when a child DLL is loaded due to various reasons such as API call from the parent DLL, dynamic loading, etc., the *ParentDllBase* field value, the *DLLBase* value of the parent

DLL are stored in the child DLL. That determines the loading order when the DLL is called. Based on the above, the 6 fields of *LDTE* were selected to design the whitelist chain and describes in Table 3 and Table 4.

Table 3. *PLD* structure data fields & data type (Based on Windows 10 Pro 64-bit, build 18362)

Field Name	Type
BaseDllName	UNICODE_STRING
FullDllName	UNICODE_STRING
SizeOfImage	ULONG
TimeStampDate	ULONG
DLLBase	PVOID (Address Pointer)
ParentDLLBase	PVOID (Address Pointer)

Table 4. *PLD* structure field description (based on Windows 10 Pro 64-bit, build 18362)

Field Name	Description
BaseDllName	DLL File Name
FullDllName	Contains the path value including the DLL file name
SizeOfImage	Size of image (DLL)
TimeStampDate	Hash value of the build file reproducible by the deterministic compile, not used for timestamp
DLLBase	Base Address of DLL
ParentDLLBase	Base Address of parent DLL when current DLL is loaded by DLL

4.2 Proposal of Whitelist Chain Design Method for DLL Injection Defense

In order to solve the DLL injection problem, we finally focused on the mechanism by which the malicious DLL works. DLL injection writes the DLL data to be infected in the process by using various approaches such as registry entry, the technique of creating a separate thread in the process using the CreateRemoteThread function, and the window hooking function. Ultimately, a function for loading DLL (LoadLibrary) and a function for loading data memory (VirtualAllocEx, WriteMemoryProcess) in the process is finally called in the injected process. Since this method uses the same technique in normal programs such as antivirus and patch programs, it is very difficult to detect them based on behavior. In addition, some DLLs operate on a Fileless basis or utilize an intelligent mechanism to finally deliver malicious DLL data to the victim by using a removable transmission medium [9], which can trigger DLL injection through file data monitoring. It is difficult to completely detect malicious malware. Therefore, we design a DLL whitelist chain to create a control group that can determine whether it is a normal DLL by using the resource constraints of the industrial control system environment, continuous work environment, and unique information of the DLL module loaded in *LDTE* data. The technique is proposed as follows. In first time, the *LDTE* data is read by referring to the PEB data address for the EWS process, and all connected *LDTE* data is read by referring to the *OrderLinks* field. For *LDTE* data sets $S = \{ a_0 , a_1 , \dots , a_n \}$, access the address a_i to get the structure field value for each *LDTE*. Here, there are 3 *OrderLinks* fields identical to *Ldr*, and the same *LOML*, *IOML*, and *MOML* fields as each field name can be mapped. At this time, there are dynamic and static methods for loading DLLs, so if it refers to *LOML* and *MOML*, all DLL module information used by the process can be retrieved. Among *LDTE* fields, *BaseDllName* and *FullDllName* fields contain DLL name data and are the most basic fields that can identify a DLL. However, malicious malware can attempt a DLL replacement attack to execute injection, and if this attack is successful, the values of both fields can be maliciously manipulated. Therefore, in order to check the actual data of the DLL, *TimeStampDate* and *ParentDLLBase* fields were additionally selected so that the authenticity of data manipulation can be determined. The *SizeOfImage* and *DLLBase* fields are fields that store the base address for the DLL and the size for the DLL, and the range and area of the address that the DLL uses in the process can be recognized. If it configures a total of 6 fields, it cans create one node, and if it connects all of the created nodes, you can configure it as a chain. At this time, the node data sheet can increase the efficiency of data access by building formatted data such as JavaScript Object Notation (JSON) and Extensible Markup Language (XML). An example of a data sheet designed based on JSON is shown in Figure 5.

Figure 5. whitelist chain example with JSON datasheet (based on x64)

Also, it is necessary to continuously update the whitelist chain, Industrial software such as EWS has a structure that continuously operates the same task without interruption due to the environmental characteristics of ICS. This is because the module I/O operations in which the DLLs used by the EWS are loaded and unloaded are constantly occurring, so we need a way to discover all the DLLs used by the EWS and update them in the chain. Considering the operating environment in IoT-based and stability of service provision, ICS goes through a process of continuously performing periodic tasks [5, 7]. From an information security perspective, availability is always a top priority even in the CIA triad, If the continuity of

the task is not guaranteed, or if it is interrupted, it is a serious incident that can cause great damage. That is why it is very important for ICS to keep the task constant at all times. In order to propose and use this method, it is assumed that EWS in a pure environment that is not yet subjected to DLL injection attacks is executed. Next, the memory address including the OrderLinks field data is read through the EWS process to be detected, and reads are performed on all structure data connected by a cyclic double linked list. At this time, while the EWS is executing the task, the node is created while reading the OrderLinks field data at a periodic time. If this process continues to run periodically, it can see the tendency of the EWS to load or unload a specific DLL and collect all the DLL information it uses. This reflects the environmental characteristics that the industrial control system continuously performs tasks, and the DLL module information used can be whitelisted. In summary, when there is a set $T = \{t_0, t_1, \ldots, t_n\}$ of points at which the chain creation operation is performed the information of the whitelist node chain is updated as each time passes from t_0 to t_n. Repeatedly, performing this increases the accuracy of DLL information used in EWS, it can be used as a control for detection and blocking. Figure 6 shows this process as the flow chart.

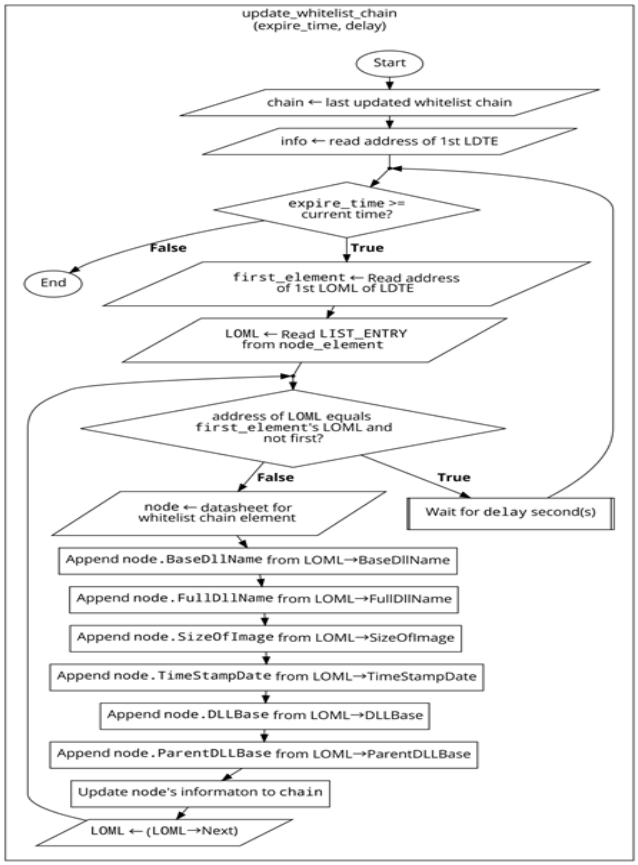


Figure 6. Flow chart for periodic whitelist chain updates (*LOML* based, *MOML* also can be applied in the same)

5 Whitelist Chain Based DLL Injection Detection & Proposal for Blocking Method

5.1 Proposal of DLL Injection Detection Method using Whitelist Chain

We propose a method that can detect DLL injection by using a whitelist chain based on the design technique proposed above. We proposed a method to create a whitelist chain updated with the latest information through Section 4.2, and a whitelist chain created based on the proposed method has a structure in which several nodes consisting of 6 fields is connected. In addition, we briefly presented how each field is used for detection through Section 4.1. In summary, it has the purpose of use as shown in Table 5.

Table 5. Purpose of use by selected field for detection

Field Name	Description
BaseDllName	Detect the DLL's name that was registered to chain
FullDllName	Detect the DLL's name that was registered to chain
SizeOfImage	Determine the use area address on process of DLL
TimeStampDate	Identifies the uniqueness of the DLL
DLLBase	Determine the use area address on process of DLL
ParentDLLBase	Determine if there is any abnormality in the calling relationship of DLL

We propose a method to detect this when it is assumed that a malicious DLL is loaded by an injection attack in the EWS process. Through the method described in Section 4.2, the EWS process reads all structures connected to *LOML* and *MOML* at periodic intervals to create a whitelist chain for normal DLLs used by EWS. As described in Section 2, this is an operational characteristic that reflects that EWS processes continuously perform tasks without interruption in most ICS environments [7, 23]. At this time, all DLLs loaded after the program bootstrapping process are loaded into *LOML*, and if it uses the ones connected in a circular linked list method as shown in Figure 2, it is determined whether the DLL is loaded by changing the load count through looping. If a DLL load is detected, information of nodes connected to the whitelist chain is read one by one to detect whether the DLL loaded in the process is normal. After that, it is determined whether the loaded DLL is normal by referring to the field value of the node. If there is no matching DLL information even after referring to all nodes, it is an abnormal DLL, and it is judged that a malicious DLL was loaded by DLL injection. The above methodology is summarized in Figure 7, Figure 8 shows how to detect a loaded DLL with a whitelist chain, and details of verifying the above methodology are covered in Section 6.1.

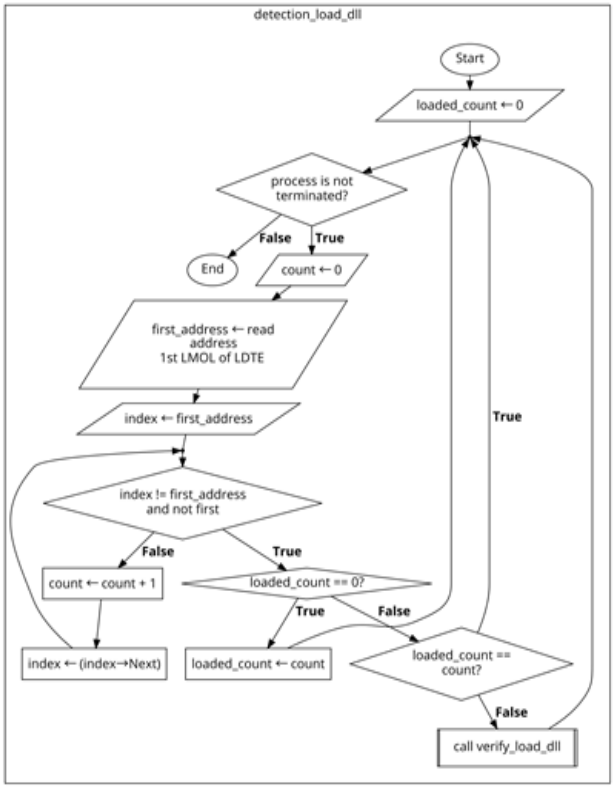


Figure 7. Flow chart for detecting load some DLL in the Process

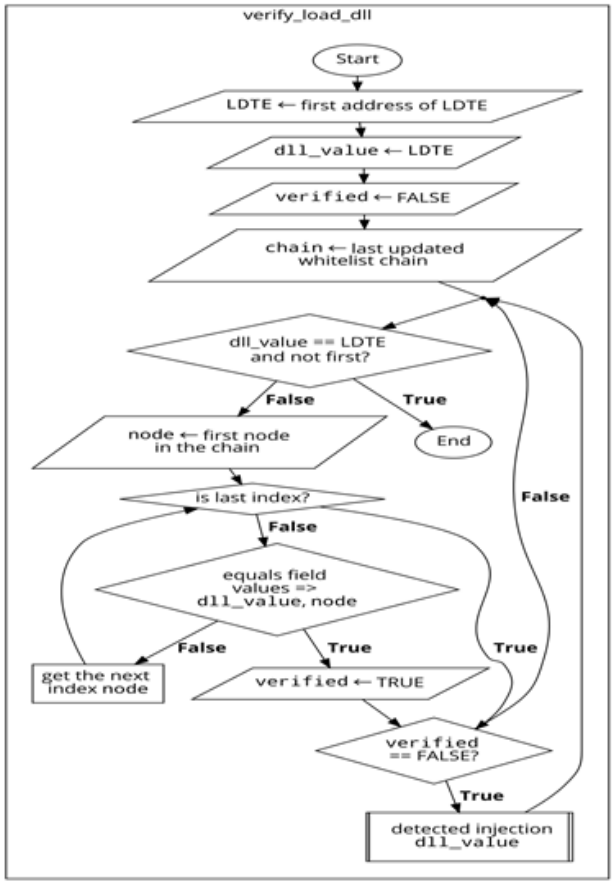


Figure 8. Flow chart for detecting injection when DLL is loaded in the process

5.2 Proposal of Blocking Method Based on Whitelist Chain to Prevent DLL Injection Execution

In addition to the detection method, a separate method is needed to block the execution of the injected DLL so that no actual damage occurs. When there is an arbitrary process, if the DLL is loaded into the process in a static or dynamic manner, the operating system guarantees thread-safe for the entire process until the loading is completed to improve safety.

In this process, it has been verified that if unloading is forcibly attempted, it can cause a serious access violation in the process. This is because the OS does not always guarantee the sequential execution of threads, and the CPU context switching is the cause [29]. In order to prevent the execution of the injected DLL detected in the step 5.1, we additionally conducted an experiment to disable the DLL through API calls for unloading. As a result, the process was forcibly terminated with a certain probability or the program was rebooted. In other words, it can cause a large loss in the availability of the operation from the point of view of the industrial control system. In addition, to ensure thread safety, unloading experiments were conducted through API calls based on Thread-Safe, but the result of unloading the DLL was shown after all the code in the injected DLL was executed. This means that it doesn't mean much to prevent DLL injection. Therefore, it is necessary to design a method that can block abnormal DLL injection techniques before the injected DLL is loaded into the process. We focused on the whitelist chain created using the technique proposed in Section 4.2 and the API used for DLL injection. For malicious DLL injection, calling of preceding codes for injecting malicious DLLs must be guaranteed, and several bypass techniques have emerged to execute the preceding codes [9].

To verify API calls to prevent DLL injection such as LoadLibrary, CreateRemoteThread, etc., it is necessary to hook the verification code for virtually all resident processes, which is a very inefficient operation. In other words, in order to guarantee the execution of malicious DLL, a separate thread is created in the victim process, and a method to detect this is proposed as follows. When the EWS process is loaded, it implements a routine that checks based on the whitelist chain by pre-hooking the main API that DLL injection calls to create a thread. At this time, the hooked function is used for verification, and it can check the presence or absence of DLL injection using the whitelist chain. The details are covered in Section 5.3, and the API hooking process for the verification code when loading the proposed process is described in Figure 9 and the hooking method in Figure 10. Additionally, we used the Microsoft's Detours library for API hooking for thread verification [8].

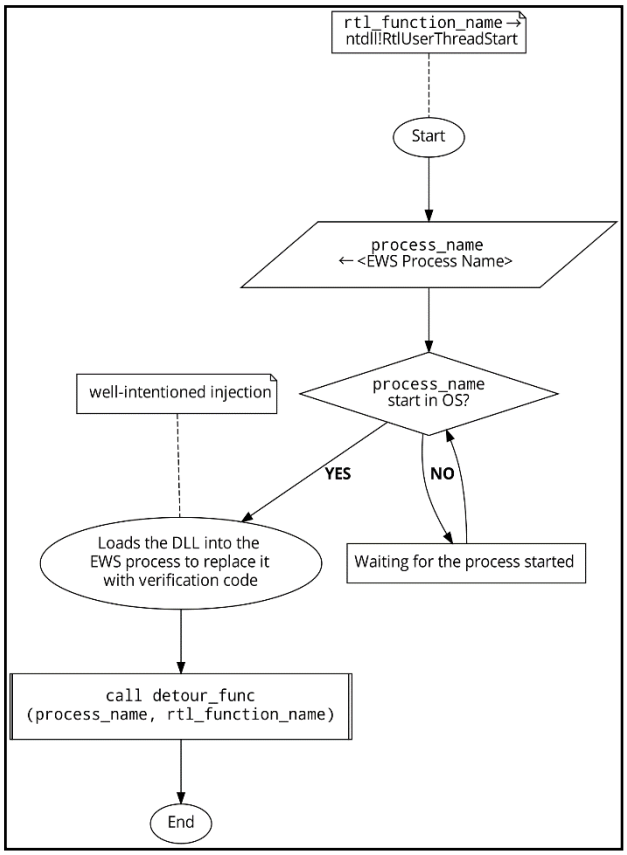


Figure 9. Flow chart for detecting load the DLL in the Process

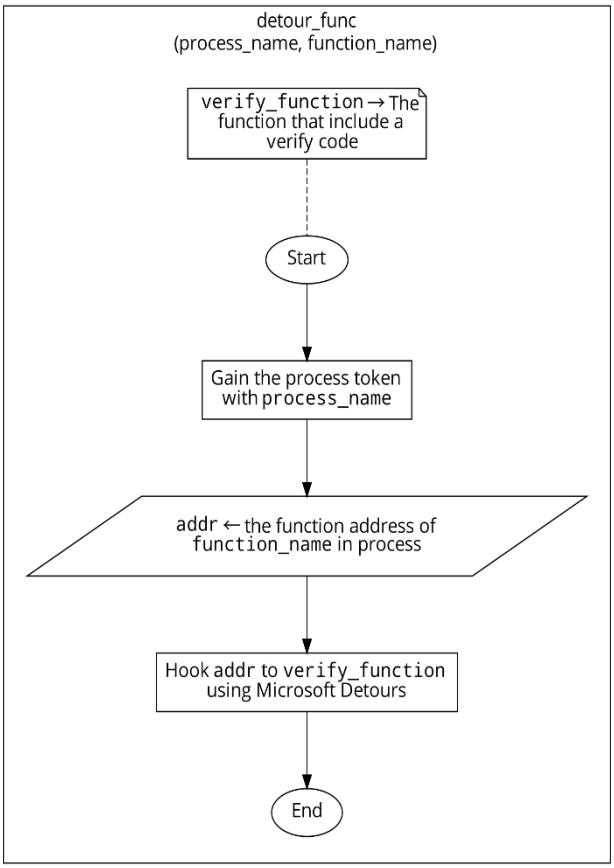
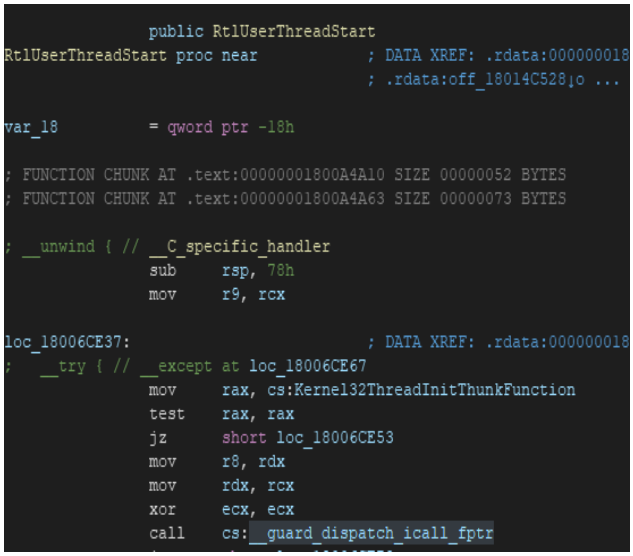


Figure 10. Flow chart for target process of API Hooking

5.3 Proposal of Verification Method for Blocking Based on Whitelist Chain to Prevent DLL Injection Execution

This section describes a method of additionally verifying whether the first caller address is valid when the thread in the EWS process is executed. If it uses the *DllBase* and *SizeOfImage* fields among the nodes constituting the whitelist chain, it can find the DLL corresponding to this address area. For example, if there is no DLL address corresponding to the area of this address through the node data constituting the chain, this is a DLL that is not recorded in the whitelisted chain, and the hooked API intentionally causes the return value to contain an error DLL injection can be blocked by applying a verification routine. Additionally, calling the LoadLibrary function through a thread created separately from the process is a basic routine for DLL injection, and if executed, the initial call point will point to the address of the thread creation function of the Windows native DLL, not the address of the process area, It was verified that the thread creation API (nt!RtlUserThreadStart) points to the RDX (x86 equals edx) register as shown in Figure 11.



```
                public RtlUserThreadStart
RtlUserThreadStart proc near            ; DATA XREF: .rdata:000000018
                                        ; .rdata:off_18014C528↓o ...

var_18          = qword ptr -18h

; FUNCTION CHUNK AT .text:00000001800A4A10 SIZE 00000052 BYTES
; FUNCTION CHUNK AT .text:00000001800A4A63 SIZE 00000073 BYTES

; __unwind { // __C_specific_handler
                sub     rsp, 78h
                mov     r9, rcx

loc_18006CE37:                          ; DATA XREF: .rdata:000000018
;   __try { // __except at loc_18006CE67
                mov     rax, cs:Kernel32ThreadInitThunkFunction
                test    rax, rax
                jz      short loc_18006CE53
                mov     r8, rdx
                mov     rdx, rcx
                xor     ecx, ecx
                call    cs:__guard_dispatch_icall_fptr
```

Figure 11. nt!RtlUserThreadStart, thread starting branch and assembly code in ntdll.dll (x86-64 based)

These routines are deliberately called through a thread unless they are performed in the EWS process code area. If the proposed method is written in code, it is as shown in Figure 12. At this time, if a routine is suspected of DLL injection, it is designed to cause an error intentionally by inducing a stub function call. At this time, for 32 bits, rcx and rdx register can be replaced with ecx and edx register, respectively. Here, the stub function does not perform any function, but only performs "*ret 0*", and is an empty function that does not mean much to execution. Through our proposed verification routine, we can induce a stub function to be executed without executing a function written by the malicious.

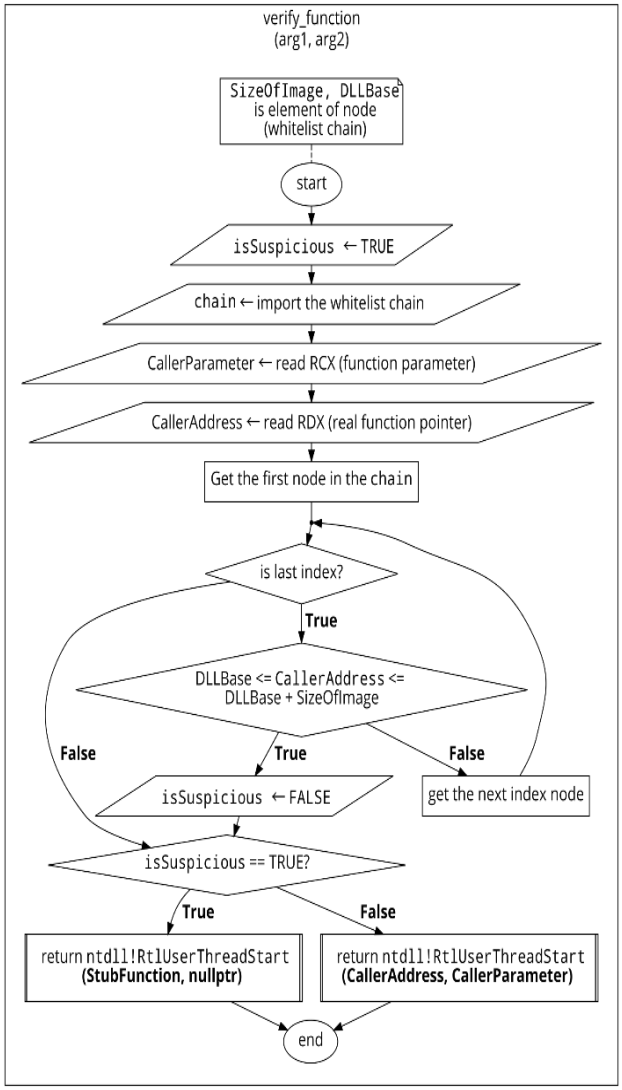


Figure 12. Flow chart for verifying nt!RtlUserThreadStart API call using the data of whitelist chain

6 Whitelist Chain Based DLL Injection Detection & Experiment Result for Blocking Verification

We constructed an experiment environment as describes in Table 6 to verify the methodology for Sections 5.1 and 5.2. PLC and EWS are connected through the same network, and EWS targets TIA Portal v15 developed by Siemens, which is widely used in the ICS field, and EWS developed by Siemens becomes an attack site for Stuxnet malware and is a representative victim of DLL injection. The testbed operation consists of updating the values of the outputs (Q0.0-Q0.7) of the PLC Siemens S7-1200 (Q0.0-Q0.7) once a second on a virtual HMI using the built-in WinCC software in TIA Portal v15. The programming environment was written based on Visual Studio 2019, C++17, and uses a datasheet based on std::vector. In this case, that means there is no restriction to create a whitelist chain even if the datasheet is different depending on the user.

Table 6. Configuration of the experiment environment

Category	Description
CPU	Ryzen 7 2700x
OS	Windows 10 Pro 64bit Build 18362
Compile Ver.	VS2019 v142 (Visual C++)
RAM	16GB
PLC	Siemens S7-1200 1214C
EWS	Siemens TIA Portal v15
Datasheet	std::vector (C++17)

```
index=[154]: DLL node (DLLBase, TimeStamp) -> 00007ffa3ddc0000 | 5a282860 | C:\Program Files\Siemens\Automation\Portal V15\Bin\Siemens.Automation.Portal.ComServer.dll
index=[155]: DLL node (DLLBase, TimeStamp) -> 00007ffa376b0000 | 5a2827cf | C:\Program Files\Siemens\Automation\Portal V15\Bin\Siemens.Automation.Portal.Com.dll
index=[156]: DLL node (DLLBase, TimeStamp) -> 000001e452060000 | 5a2827e4 | C:\Program Files\Siemens\Automation\Portal V15\Bin\Siemens.Automation.ObjectFrame.Kernel.dll
index=[157]: DLL node (DLLBase, TimeStamp) -> 00007ffa50880000 | 5c7d2109 | C:\Windows\Microsoft.NET\Framework64\v2.0.50727\System.EnterpriseServices.Thunk.dll
index=[158]: DLL node (DLLBase, TimeStamp) -> 00007ffa37c70000 | 5f52ac18 | C:\WINDOWS\assembly\NativeImages_v4.0.30319_64\System.Deployment\b0d1e152f58b75210b19a746efe1a99b\System.Deployment.ni.dll
index=[159]: DLL node (DLLBase, TimeStamp) -> 000001df25520000 | 5a282ab5a | C:\Program Files\Siemens\Automation\Portal V15\Bin\Siemens.Simatic.Hwcn.Interpreter.Gsd.GSDCtrlWrapper.dll
index=[160]: DLL node (DLLBase, TimeStamp) -> 00007ffa2cad0000 | 576adb7f | C:\Program Files\Siemens\Automation\Portal V15\Bin\System.Data.SQLite.dll
index=[161]: DLL node (DLLBase, TimeStamp) -> 00007ffa65bd0000 | 5c7a1de8 | C:\WINDOWS\assembly\NativeImages_v4.0.30319_64\System.Transactions\fb74bdf2ed5f668682fcce5e11360d0c\System.Transactions.ni.dll

index=[162]: DLL node (DLLBase, TimeStamp) -> 00007ffa65b80000 | 5c7a1de8 | C:\WINDOWS\Microsoft.Net\assembly\GAC_64\System.Transactions\v4.0_4.0.0.0__b77a5c561934e089\System.Transactions.dll
index=[163]: DLL node (DLLBase, TimeStamp) -> 00007ffa5ded0000 | 5c7a1629 | C:\WINDOWS\assembly\NativeImages_v4.0.30319_64\System.Ente96d83b35#\9b80b054a8f281de0a24b52b4e42c6ba\System.EnterpriseServices.ni.dll
index=[164]: DLL node (DLLBase, TimeStamp) -> 00007ffa5de70000 | 5c7a1411 | C:\WINDOWS\assembly\NativeImages_v4.0.30319_64\System.Ente96d83b35#\9b80b054a8f281de0a24b52b4e42c6ba\System.EnterpriseServices.Wrapper.dll
index=[165]: DLL node (DLLBase, TimeStamp) -> 00007ffa62b60000 | 5c7a1411 | C:\WINDOWS\Microsoft.Net\assembly\GAC_64\System.EnterpriseServices\v4.0_4.0.0.0__b03f5f7f11d50a3a\System.EnterpriseServices.Wrapper.dll
index=[166]: DLL node (DLLBase, TimeStamp) -> 00007ffa5c630000 | 5f2a06c8 | C:\WINDOWS\assembly\NativeImages_v4.0.30319_64\System.Web\27668d857e87add5e794a6489eb5f9f7\System.Web.ni.dll
index=[167]: DLL node (DLLBase, TimeStamp) -> 000001df2b780000 | 5a282c12 | C:\Program Files\Siemens\Automation\Portal V15\Bin\Siemens.Simatic.Hwcn.BusinessLogic.Extensions.dll
index=[168]: DLL node (DLLBase, TimeStamp) -> 00007ffa62be0000 | 5c7a23fb | C:\WINDOWS\assembly\NativeImages_v4.0.30319_64\System.Runt19c51595#\e086decf7372afdb892814019cc5608e\System.Runtime.Caching.ni.dll
index=[169]: DLL node (DLLBase, TimeStamp) -> 00007ffa56b20000 | 5c7a241f | C:\WINDOWS\assembly\NativeImages_v4.0.30319_64\System.Management\7bdb01d70f7f2aabb86f8ca19cc4ff28\System.Management.ni.dll
index=[170]: DLL node (DLLBase, TimeStamp) -> 00007ffa6bc10000 | fd05cf31 | C:\WINDOWS\system32\wbem\wmiutils.dll
index=[171]: DLL node (DLLBase, TimeStamp) -> 00007ffa70300000 | 6af5c06d | C:\WINDOWS\SYSTEM32\wbemcomn.dll
index=[172]: DLL node (DLLBase, TimeStamp) -> 00007ffa6d540000 | f4f4f290 | C:\WINDOWS\system32\wbem\wbemprox.dll
index=[173]: DLL node (DLLBase, TimeStamp) -> 00007ffa52eb0000 | 5c7a15c8 | C:\Windows\Microsoft.NET\Framework64\v4.0.30319\wminet_utils.dll
index=[174]: DLL node (DLLBase, TimeStamp) -> 00007ffa6bd60000 | f9651a38 | C:\WINDOWS\system32\wbem\wbemsvc.dll
index=[175]: DLL node (DLLBase, TimeStamp) -> 00007ffa6be10000 | 65ead981 | C:\WINDOWS\system32\wbem\fastprox.dll
index=[176]: DLL node (DLLBase, TimeStamp) -> 00007ffa6aee0000 | d9e0f2cc | C:\WINDOWS\SYSTEM32\amsi.dll
index=[177]: DLL node (DLLBase, TimeStamp) -> 00007ffa88920000 | 1c3a0614 | C:\WINDOWS\SYSTEM32\dhcpcsvc6.DLL
index=[178]: DLL node (DLLBase, TimeStamp) -> 00007ffa88c90000 | 3c1facb4 | C:\WINDOWS\SYSTEM32\dhcpcsvc.DLL
index=[179]: DLL node (DLLBase, TimeStamp) -> 00007ffa89730000 | 3f83c149 | C:\WINDOWS\SYSTEM32\WINNSI.DLL
index=[180]: DLL node (DLLBase, TimeStamp) -> 00007ffa881d0000 | 0ec7710e | C:\WINDOWS\System32\fwpuclnt.dll
index=[181]: DLL node (DLLBase, TimeStamp) -> 00007ffa89720000 | 6a2bea39 | C:\Windows\System32\rasadhlp.dll
index=[182]: DLL node (DLLBase, TimeStamp) -> 00007ffa15f10000 | 5a0178f5 | C:\Program Files\Common Files\Siemens\Automation\Simatic OAM\bin\S7scpcomm64.dll
index=[183]: DLL node (DLLBase, TimeStamp) -> 000001df28720000 | 59b1658d | C:\Program Files\Common Files\Siemens\Automation\Simatic OAM\bin\S7ONTCP.DLL
index=[184]: DLL node (DLLBase, TimeStamp) -> 00000000a130000 | 4df2bcac | C:\WINDOWS\SYSTEM32\MSVCR100.dll
index=[185]: DLL node (DLLBase, TimeStamp) -> 00007ffa65a50000 | 59a7fd5f | C:\Program Files\Common Files\Siemens\Automation\Simatic OAM\bin\sn_regbase.dll
index=[186]: DLL node (DLLBase, TimeStamp) -> 00007ffa518a0000 | 59f31a0b | C:\Program Files\Siemens\Automation\Portal V15\Bin\OMSp_ASOM.dll
index=[187]: DLL node (DLLBase, TimeStamp) -> 00007ffa5b6a0000 | 5f4ff370 | C:\WINDOWS\assembly\NativeImages_v4.0.30319_64\System.Security\aaaee07878a18fd29c7bb885bdf1ee9c\System.Security.ni.dll
index=[188]: DLL node (DLLBase, TimeStamp) -> 00007ffa20d60000 | 5a282f80 | C:\Program Files\Siemens\Automation\Portal V15\Bin\Siemens.Simatic.Lang.MC7Codegenerator.dll
index=[189]: DLL node (DLLBase, TimeStamp) -> 00007ffa20a60000 | 57061e55 | C:\Program Files\Siemens\Automation\Portal V15\Bin\s7blpdlx.dll
index=[190]: DLL node (DLLBase, TimeStamp) -> 00000000636e0000 | 4df2cfda | C:\WINDOWS\SYSTEM32\mfc100.dll
index=[191]: DLL node (DLLBase, TimeStamp) -> 00007ffa7a3c0000 | af424cac | C:\WINDOWS\SYSTEM32\MSIMG32.dll
index=[192]: DLL node (DLLBase, TimeStamp) -> 00007ffa8cad0000 | 7cd7bb0f | C:\WINDOWS\system32\dwmapi.dll
index=[193]: DLL node (DLLBase, TimeStamp) -> 000001df27eb0000 | 4d5f13f1 | C:\WINDOWS\SYSTEM32\MFC100KOR.DLL
```

Figure 14. Output node information connected to whitelist chain

For DLL injection test, we used the SecurityXploded Remote DLL Injector, InjectAllTheThings program [30]. These are injection experiment programs using CreateRemoteThread related functions and QueueUser APC, SetWindowsHookEx, and SetThreadContext functions that can create remote jobs for injection. When the process is first loaded, The DLL implemented based on the methodology according to Figure 9, loaded into the process as shown in Figure 13. The result of the loaded dll as shown in Figure 14, an environment in which the whitelist chain and API verification function can be operated while residing in the EWS process was established. In other words, it plays a role of protection for the process through well-intentioned DLL injection.

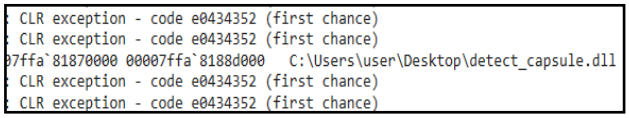


Figure 13. Resides in the process and loads the DLL for verification and chain creation

6.1 Whitelist Chain Based DLL Injection Detection Technique Experiment and Results

The whitelist chain stores, and records DLL module information based on Ldr data while executing the task at 0.1 second intervals for a total of 8 hours. As also shown in Figure 13, a DLL containing information gathering routines is executed to create the whitelist chain. Additionally, to minimize the performance impact of the EWS parent process, we created a separate thread to do. It can be designed to act in a separate thread space from the parent process to collect the process of the DLL being onloaded into the EWS. This approach is like the technique used by DLL injection, but the difference is that it creates a whitelist chain, it is a well-intentioned work to fulfill the preceding work for defense. As a result, the process correctly created the whitelist node for the module information of 193 DLLs used for work, and the CPU occupancy used for creation was 6-8% on average and the RAM occupied about 35MB. If it reads the whitelist chain and print all the connected nodes, it is as shown in Figure 14, these shows that the *PEB-LDR* data was read, the whitelist chain was created normally. When a malicious DLL is loaded by injection while the DLL is monitoring, it can be verified that it is success-fully detected as shown in Figure 15.

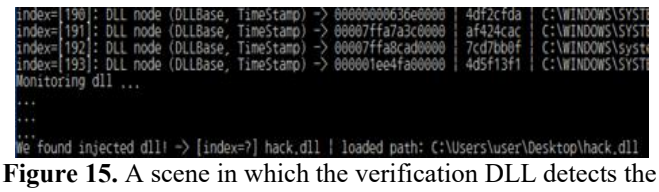


Figure 15. A scene in which the verification DLL detects the loading of a malicious DLL by DLL injection

6.2 Whitelist Chain Based Blocking Technique Experiment and Result of Prevent DLL Injection Execution

We conduct an experiment to verify whether actual DLL injection can be blocked by utilizing both the whitelist chain based on the design technique proposed in Section 5.1 and the DLL injection blocking technique proposed in Sections 5.2 and 5.3. Previously, through Section 5.3, we added a verification routine for Windows Native API (NTAPI), which must be called for creating a remote thread, and NTAPI for writing memory to a remote process, and a verification routine for RTL functions that are called when creating a thread. The main purpose is to experiment to see if these verification routines work correctly. We envisioned an EWS process to include a validation routine in nt!RtlUserThreadStart to determine whether it is called from a valid address or not. Furthermore, we performed caller address verification for low-level API such as LdrLoadDll. Previously, we checked the functions including various suffixes for each API [2, 4], so proceeded to target the low-level API. As a result of the experiment, it was verified that the routine process works correctly for the main API for injection as shown in Figure 16. The figure shows the stack up to the routine that verifies that when LoadLibrary API is called from EWS process, being called from a valid address. Before calling the API, we observed that the call was entered correctly into the validation function we implemented separately.

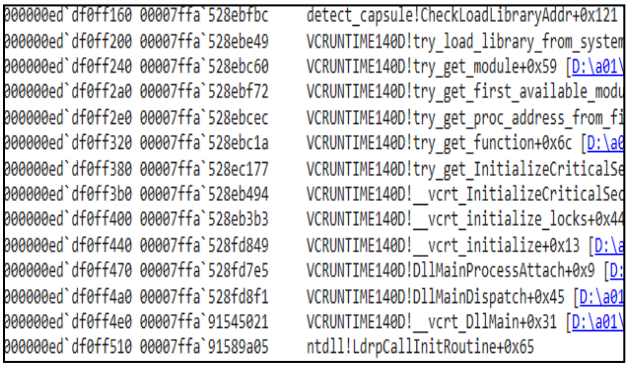


Figure 16. Scene where verification code is called before calling LoadLibrary API in new thread of EWS process

In addition, when attempting to load the DLL forcibly by the injector, it was verified that the thread creation in the EWS process failed as shown in Figure 17. The EWS process was not terminated and operated normally, the CPU occupancy used for detection is on average 10-15% and the RAM occupancy is about 60MB, which means that the DLL implementing the defense technique proposed by us normally blocks malicious DLL execution by the DLL injection technique of external malicious code in the EWS process. Additionally, the process does not terminate and operates normally, the results described in Table 7, it means that DLL injection is called through a remote thread to ensure stable operation and shows that the verification code inserted at the thread creation call point is effective in blocking. In a separate, the main goal of the SetWindowsHookEx function is to hook the message, the LoadLibrary function is included only in the call category for operating the main code [9]. In our experiment, when trying to hook using the function the exploit tool causes freezed, it succeeded in blocking, but essentially more detailed verification is required to determine whether it led to hooking. At this time, we considered technical compatibility, the purposed technology referenced specific

NTAPI and *LDTE* data to detect DLL injection attacks. LDTE and LDR data have been supported since Windows XP, which means that legacy OS can be technically used [26-27]. In generally, ICS environments except for the overhaul period, legacy OSs such as Windows XP and Server 2003 are still widely used. In the other word, it can satisfy the backward compatibility of the specific NTAPI and *LDTE* data we utilized.

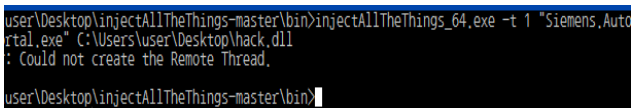


Figure 17. Sense of example when DLL injection was attempted

Table 7. Configuration of the experiment environment

API Name	Block Result (Reason)
CreateRemoteThread	Success (Failed to create thread)
SetWindowsHookEx	Unknown (Freezed the expolt tool)
SetThreadContext	Success (Failed to create thread)
QueueUserAPC	Success (Failed to create thread)

7 Conclusion Future Works

As the experiement results, we verified the DLL injection attack detection method for EWS used in ICS in IoT environment. The presented method also contributes to the safety operation of ICS along with device and platform security technologies in the IoT environment that have been previously worked on. Compared with other studies, our proposed DLL injection blocking method does not require various types of training data for DLL injection detection. In addition, it is possible to protect DLL information used in the EWS process by establishing a whitelist chain based on *PEB-LDR* data through periodic DLL information recording in consideration of ICS operation characteristics. For the preceding steps to present the technology, we analyzed symbol information to analyze PEB structure information, and described how to utilize the specific fields containing DLL information in LDR structure. These analysis results can be widely applied when searching for traces of DLLs that can be utilized in a specific process or referenced when accessing them, or when building tasks that require status monitoring. On the other hand, we targeted the main attack types targeting the Windows OS such as DLL injection attacks, but if EWS is software running on Unix/Linux OS, it can be compromised similar attack types such as .so file injection. Therefore, it is necessary to study defense techniques targeting platforms other than Windows OS.

Currently, it is publicly known that malicious actions through DLL injection as well as attacks that intentionally inject code in a DLL or inject code into process memory are possible. Although the above method detects based on DLL injection attacks that target processes, protection against code injection attacks that target processes or DLLs has not been verified. These types of attacks need to address pre-requirements that need to be viewed broadly on a binary or code basis rather than a file information. For this reason, in the future, we will propose a technology that automatically recognizes DLLs used in EWS and automatically creates whitelist data and plans to conduct technology research that can protect not only DLL injection but also code unit injection. Besides, to create a whitelist chain through the above method, there is a hassle of periodically recording in a pure environment in advance. Since EWS used by ICS in IoT environment is still actively used for long-term monitoring and management of SCADA systems, it is necessary to devise various detection techniques to prepare for fatal attacks such as DLL injection to guarantee the normal function of EWS in the future. To overcome these shortcomings, we will devise a technique that can detect suspicious behaviors such as DLL injection without separate prior procedures such as whitelisting.

Acknowledgments

This work was supported by the Nuclear Safety Research Program through the Korea Foundation of Nuclear Safety (KoFONS) using the financial resource granted by the Nuclear Safety and Security Commission (NSSC) of the Republic of Korea (No. 2106058, 45%), Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (NRF-2020R1A2C1012187, 45%), and the Gachon University research fund of 2021(GCU-202106330001, 10%).

References

[1] A. Moradbeikie, K. Jamshidi, A. Bohlooli, J. Garcia, X. Masip-Bruin, An IIoT Based ICS to Improve Safety Through Fast and Accurate Hazard Detection and Differentiation, *IEEE Access*, Vol. 8, pp. 206942-206957, November, 2020.

[2] A. Shahzad, Y.-G. Kim, A. Elgamoudi, Secure IoT Platform for Industrial Control Systems, *2017 International Conference on Platform Technology and Service (PlatCon)*, Busan, South Korea, 2017, pp. 1-6.

[3] A. Hansson, M. Khodari, A. Gurtov, Analyzing Internet-connected industrial equipment, *2018 International Conference on Signals and Systems (ICSigSys)*, Bali, Indonesia, 2018, pp. 29-35.

[4] M. H. Alquwatli, M. H. Habaebi, S. Khan, Review of SCADA Systems and IoT Honeypots, *2019 IEEE 6th International Conference on Engineering Technologies and Applied Sciences (ICETAS)*, Kuala Lumpur, Malaysia, 2019, pp. 1-6.

[5] J. Ferreira, J. N. Soares, R. J. Goncalves, C. Agostinho, Management of IoT Devices in a Physical Network, *2017 21st International Conference on Control Systems and Computer Science (CSCS)*, Bucharest, Romania, 2017, pp. 485-492.

[6] D. Kushner, The real story of stuxnet, *IEEE Spectrum*, Vol. 50, No. 3, pp. 48-53, March, 2013.

[7] J. Lee, S. Hong, Keeping Host Sanity for Security of the SCADA Systems, *IEEE Access*, Vol. 8, pp. 62954-62968, March, 2020.

[8] A. O. A. El-Mal, M. A. Sobh, A. M. B. Eldin, Hard-Detours: A new technique for dynamic code analysis, *Eurocon 2013*, Zagreb, Croatia, 2013, pp. 46-51.

[9] A. Klein, I. Kotler, *Windows Process Injection in 2019*, August, 2019, Available Online: https://i.blackhat.com/USA-19/Thursday/us-19-Kotler-Process-Injection-Techniques-Gotta-Catch-Them-All-wp.pdf.

[10] C.-W. Park, J.-W. Son, H.-K. Hwang, K.-C. Kim, Detection of Systems Infected with C&C Zeus Through Technique of Windows API Hooking, *Asia-Pacific Journal of Multimedia Services Convergent with Art, Humanities, and Sociology*, Vol. 5, No. 2, pp. 297-304, April, 2015.

[11] J. Berdajs, Z. Bosnic, Extending applications using an advanced approach to DLL injection and API hooking, *Software Practice and Experience*, Vol. 40, No. 7, pp. 567-584, June, 2010.

[12] H.-M. Sun, Y.-H. Lin, M.-F. Wu, API Monitoring System for Defeating Worms and Exploits in MS-Windows System, *Information Security and Privacy (ACISP 2006)*, Melbourne, Australia, 2006, pp. 159-170.

[13] S. Z. M. Shaid, M. A. Maarof, In memory detection of Windows API call hooking technique, *2015 IEEE 2015 International Conference on Computer, Communications, and Control Technology (I4CT 2015)*, Kuching, Malaysia, 2015, pp. 294-298.

[14] F. Mira, A Review Paper of Malware Detection Using API Call Sequences, *2019 2nd International Conference on Computer Applications & Information Security (ICCAIS)*, Riyadh, Saudi Arabia, 2019, pp. 1-6.

[15] M. K. Shankarapani, S. Ramamoorthy, R. Movva, S. Mukkamala, Malware detection using assembly and API call sequences, *Journal in Computer Virology*, Vol. 7, No. 2, pp. 107-119, May, 2011.

[16] S. Yusirwan, Y. Prayudi, I. Riadi, Implementation of Malware Analysis using Static and Dynamic Analysis Method, *International Journal of Computer Applications*, Vol. 117, No. 6, pp. 11-15, May, 2015.

[17] S. I. Bae, E. G. Im, Unpacking Technique for In-memory Malware Injection Technique, *Korean Institute of Smart Media*, Vol. 8, No. 1, pp. 19-26, March, 2019.

[18] S. Karnouskos, Stuxnet Worm Impact on Industrial Cyber-Physical System Security, *IECON 2011 - 37th Annual Conference of the IEEE Industrial Electronics Society*, Victoria, Australia, 2011, pp. 4490-4494.

[19] C.-I. Fan, H.-W. Hsiao, C.-H. Chou, Y.-F. Tseng, Malware Detection Systems Based on API Log Data Mining, *2015 IEEE 39th Annual Computer Software and Applications Conference*, Taichung, Taiwan, 2015, pp. 255-260.

[20] J.-H. Hwang, S.-B. Hwang, H.-G. Kim, J.-H. Ha, T.-J. Lee, Malware Analysis Based on Section, DLL, *Journal of the Korea Institute of Information Security and Cryptology*, Vol. 27, No. 5, pp. 1077-1086, October, 2017.

[21] J.-H. Ha, S.-J. Kim, T.-J. Lee, Feature Extraction using DLL/API Statistical Analysis and Malware Detection based on Machine Learning, *The Journal of Korean Institute of Communications and Information Sciences*, Vol. 43, No. 4, pp. 730-739, April, 2018.

[22] W. Matsuda, M. Fujimoto, T. Mitsunaga, Detection of Malicious Tools by Monitoring DLL Using Deep Learning, *Journal of Information Processing*, Vol. 28, pp. 1052-1064, December, 2020.

[23] P. Forsgren, Requirements specification in SCADA/EMS/DMS procurement projects, *Fourth International Conference on Power System Control and Management (Conf. Publ. No. 421)*, London, UK, 1996, pp. 226-230.

[24] S. Ghosh, S. Sampalli, A Survey of Security in SCADA Networks: Current Issues and Future Challenges, *IEEE Access*, Vol. 7, pp. 135812-135831, July, 2019.

[25] S. Choi, T. Chang, C. Kim, Y. Park, x64Unpack: Hybrid Emulation Unpacker for 64-bit Windows Environments and Detailed Analysis Results on VMProtect 3.4, *IEEE Access*, Vol. 8, pp. 127939-127953, July, 2020.

[26] Microsoft, *PEB (winternl.h) - Win32 apps | Microsoft Docs*, December, 2018, https://docs.microsoft.com/en-us/windows/win32/api/winternl/ns-winternl-peb, retrieved in February, 2021.

[27] Microsoft, *PEB_LDR_DATA (winternl.h) - Win32 apps | Microsoft Docs*, December, 2018, https://docs.microsoft.com/en-us/windows/win32/api/winternl/ns-winternl-peb_ldr_data, retrieved in February, 2021.

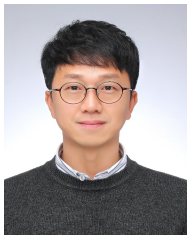
[28] Microsoft, *Microsoft public symbol server - Windows drivers | Microsoft Docs*, April, 2018, https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/microsoft-public-symbols, retrieved in February, 2021.

[29] K. Sharif, S. R. M. Zeebaree, L. M. Haji, R. Zebari, Performance Measurement of Processes and Threads Controlling, Tracking and Monitoring Based on Shared-Memory Parallel Processing Approach, *2020 3rd International Conference on Engineering Technology and its Applications (IICETA)*, Najaf, Iraq, 2020, pp. 62-67.

[30] R. C. B. Hink, K. Goseva-Popstojanova, Characterization of Cyberattacks Aimed at Integrated Industrial Control and Enterprise Systems: A Case Study, *2016 IEEE 17th International Symposium on High Assurance Systems Engineering (HASE)*, Orlando, FL, USA, 2016, pp. 149-156.

Biographies

Junwon Kim received his M.S. degree in Information Security Engineering from Gachon University, South Korea, in 2022, and the B.S. degree in Information Security Engineering from Soonchunhyang University, South Korea, in 2020. His research interests include ICS security, CPS security, Anomaly Detection, Deep Packet Inspection, Vulnerability Analysis, and Information Security.



Jiho Shin received his M.S. degree in Digital Forensic from Korea University, South Korea, in 2015, and the Ph.D. Degree in Information Security Engineering from Soonchunhyang University, South Korea, in 2022. He is currently a research officer with Science and Technology Research Division, Police Science Institute of Korean National Police University. His research interests include Digital Forensics, Cybercrime Response, OT security, Industrial Control System, and Information Security.

Jung Taek Seo received the M.S. degree in Computer Engineering from Ajou University, South Korea, in 2001, and the Ph.D. degree in Information Security Engineering from Korea University, South Korea, in 2006. He worked for National Security Research Institute as a senior researcher. He is currently an Associate Professor with the Department of Computer Engineering, Gachon University. His research interests include CPS security, ICS cybersecurity, smart grid security, nuclear power plant security, smart factory security, smart city security, and automotive cybersecurity.