IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

Digital Object Identifier 10.1109/ACCESS.2017.Doi Number

Keeping Host Sanity for Security of the SCADA Systems

Jae-Myeong Lee[1] and Sugwon Hong[2]

[1]Department of Computer Engineering, Myongji University, Yongin, South Korea; ljm9317kr@gmail.com
[2]Department of Computer Engineering, Myongji University, Yongin, South Korea; swhong@mju.ac.kr

Corresponding author: Sugwon Hong (e-mail: swhong@mju.ac.kr).

This work was supported by "Human Resources Program in Energy Technology" of the Korea Institute of Energy Technology Evaluation and Planning (KETEP), granted financial resource from the Ministry of Trade, Industry & Energy, Republic of Korea. (No. 20174030201790) And this research was also supported by Korea Electric Power Corporation. (Grant number: R18XA01).

ABSTRACT Cyber attacks targeting the Supervisory Control and Data Acquisition (SCADA) systems are becoming more complex and more intelligent. Currently proposed security measures for the SCADA systems come under three categories: *physical/logical network separation*, *communication message security*, and *security monitoring*. However, the recent malwares which were used successfully to disrupt the critical systems show that these security strategies are necessary, but not sufficient to defend these malwares. The malware attacks on the SCADA system exploit weaknesses of host system software environment and take over the control of host processes in the SCADA system. In this paper, we explain how the malware interferes in the important process logics, and invades the SCADA host process by using Dynamic Link Library (DLL) Injection. As a security measure, we propose an algorithm to block DLL Injection efficiently, and show its effectiveness of defending real world malwares using DLL Injection technique by implementing as a library and testing against several DLL Injection scenarios. It is expected that this approach can prevent all the hosts in the SCADA system from being taken over by this kind of malicious attacks, consequently keeping its sanity all the time.

INDEX TERMS SCADA security, Malware, DLL injection, Code injection, Host system security

I. INTRODUCTION

The Supervisory Control and Data Acquisition (SCADA), more broadly the Industrial Control System (ICS), is a system to monitor and control geographically distributed large-scale process field devices. The cyber attacks on the SCADA/ICS systems are considered severe threats for the critical industrial facilities like power grid due to their catastrophic impact on industry development and social safety. For this reason, to maintain the health and sanity of the SCADA/ICS system against cyber attacks poses daunting challenges.

All security measures which have been considered and published for the SCADA/ICS systems can be classified into three categories: *physical/logical network separation*, *communication message security*, and *security monitoring* [1]. Network separation is based on the concept of Defense in Depth [2, 3]. The logical separation is to separate a SCADA/ICS network into several segmented zones or domains depending on criticality or functionality.

The ISA/IEC standard [4] proposes the reference model of network segmentation for the industrial control network, which adopted the Purdue Enterprise Reference Architecture (PERA) [5]. The reference model, based on the concepts of zones and conduits, creates boundaries and defines connection points (conduits) between zones in order to effectively apply multiple layers of defense. Each zone constitutes a virtually separated network which is only connected to other zones via dedicated conduits. Neighboring zones should clearly define such conduits, only on which all information flows are exchanged and strict security policing is enforced. At the conduits, access control such as identification and authentication becomes major tasks.

Fig. 1 shows a multi-layered SCADA system architecture, which is revised for the purpose of more reflecting the real power SCADA system characteristics. So, the names of zones are different from those of IEC 62443 document [4]. The primary goal of network separation or segmentation is that when attackers try to penetrate deeper zones, it can reduce the probability of attack success, i.e. attacks are as isolated into a

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.2983179, IEEE Access

penetrated zone as possible, mitigating attack impacts. Firewall and intrusion prevention systems (IPS) are common equipment used for this purpose, and virtual private network (VPN) is a common network design technique used for deploying isolated networks. The network separation is the primary security strategy which the current SCADA/ICS systems rely on.

The final goal of attackers is to disrupt normal operation of the SCADA system, targeting process field devices at the layer 0 in Fig. 1. All the information related to operations in the SCADA system are delivered by messages which are specified by the SCADA communication protocols. Compromised command messages might invoke unintended operations, and also tempered response messages from field devices may well cause misunderstanding of current process states, resulting in fatal misjudgment on operations. For this reason, to keep the soundness of messages exchanged between all system components can be regarded as a logical next step for SCADA security. The communication message security aims at providing authenticity, integrity, and/or confidentiality of messages exchanged in the system. IEC 62351 standards, which is oriented for the IEC61850 protocols, specify the security measures for this purpose [6]. It goes without saying that the message security strategy can raise security level one step higher.

As in the case of IT networks, the security monitoring in the SCADA system is an indispensable security strategy. Intrusion detection system (IDS) is a main tool to implement network security monitoring. Network IDS is gaining interests as an efficient security approach in the SCADA system because the SCADA systems have very predictable traffic patterns and behaviors compared to IT networks, and IDS can be easily adapted without any major configuration change of current SCADA systems. We will explain in detail the recent research works for developing the SCADA-specific IDS in Section III.

Even though these three strategies give us a holistic approach to SCADA security, recent attacks on SCADA/ICS systems such as *Stuxnet*, *Black Energy 3*, *Triton*, and *Industroyer* make us realize that these strategies are necessary but not sufficient. The nature of these attacks is that once attackers penetrate the SCADA network, they take control of host systems by process manipulation, whether they are control servers or local device controllers [7]. Moreover, attackers use specially designed malware for SCADA to invade the core system of SCADA and achieve their attack goals [8,9]. Thus, either communication message security measures or security monitoring based on the network IDS can hardly detect these attacks, much less prevent them. For this reason, we need an additional measure which can keep host sanity under the threat of these sophisticated attacks.

In this paper, based on the analysis of the recent cyber attacks against SCADA/ICS systems, we explain how these SCADA-oriented malwares could interfere the important process logics, and exploit the code injection technique called *Dynamic Link Library (DLL) Injection*. As a defense measure, we propose a novel algorithm to block the DLL Injection attack efficiently, implement the algorithm in a form of DLL module, and show its efficacy by penetration testing.

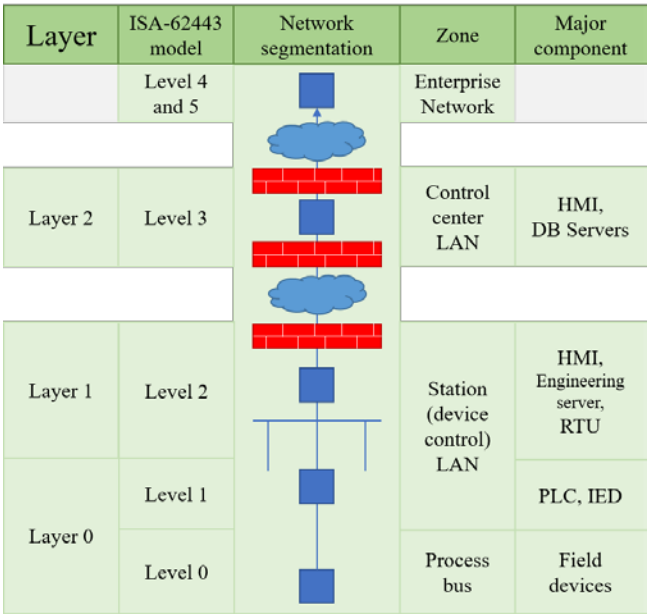

FIGURE 1. **Multi-layered SCADA network architecture**

This paper is organized as follows. In section II, we investigate major attacks on the SCADA/ICS systems and analyze the nature of attacks, highlighting how they penetrated the kernel of the system. In section III, we review the current research works on the security strategies for the SCADA system. The following section IV and V explain the DLL injection and API hooking, and then section VI explains the proposed an anti-DLL injection algorithm. In section VII, we show the experiment results against various attack scenarios.

II. MALWARE ATTACKS ON SCADA/ICS SYSTEMS

A. STUXNET

Stuxnet [10-13] is a computer worm, first uncovered in 2010, and it is known that this worm successfully sabotaged the nuclear plant. Its main goal was to disrupt normal operation by reprogramming Programmable Logic Controller (PLC).

Stuxnet employed a Dropper architecture. Dropper [14] is a trojan horse program for deploying malwares. Once dropper has been executed, the dropper extracts a wanted malware by itself, installs and/or executes it. All main activities of *Stuxnet* are programmed in a DLL module, and the dropper tries to inject this module into trusted process in order to execute malicious activities. Generally speaking, injecting DLL into the trusted process is to bypass the behavior-based security products and to hide its existence from monitoring programs, without any suspicion [15]. *Stuxnet* also installs a user-mode rootkit implemented in a DLL module on the system in order to avoid detection of security products like antiviruses. This

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.2983179, IEEE Access

module hides *Stuxnet* files by hooking Windows API functions getting list of files.

Once *Stuxnet* has been executed in a computer system, it tries to spread to other computers on the network through zero-day exploits and network file share functionality. At the same time, it also tries spreading by infecting WinCC database server through WinCC database server vulnerability, by infecting USB removable drives with a Windows' *AutoRun* configuration file named *autorun.inf*, and by infecting Step7 project files.

Main goal of *Stuxnet* is to infect a specific type of Siemens SIMATIC PLC devices. After *Stuxnet* arrives a system with WinCC/Step7 software installed, which is able to program PLC devices, through the steps described above, *Stuxnet* attempts to modify PLC program logics by altering Step7 control programs. PLC infection is done by substituting s7otbxdx.dll of SIMATIC, which handles block inputs and outputs, with a different file, and by replacing PLC code blocks with its own code blocks.

B. BLACK ENERGY 3

Black Energy 3 (*BE3*) is also one of well-known SCADA attack cases. It is the third version of Black Energy series. Black Energy was started with its first version *Black Energy 1* (*BE1*) in 2007, and its third version *BE3* was discovered in 2014. *BE3* was used to attack Ukraine power distribution system [16-19].

The first version *BE1* was a Hypertext Transfer Protocol (HTTP)-based botnet for Distributed Denial of Service (DDoS) attacks. Unlike other botnets, *BE1* does not communicate with a botnet master through Internet Relay Chat (IRC). Instead, *BE1* loads configurations from a MySQL database *db.sql* and executes the configured actions from it. The second version *Black Energy 2 (BE2)* was found in 2010. *BE2* was evolved into a modular framework unlike *BE1*, which aimed only at DDoS. *BE2* provides not only DDoS but also extended functionalities like espionage, fraud, stealing user credentials or key logger, scanning network, sending spam, with easily loadable attack-specific plugins. These plugins are downloaded from a command & control (C&C) server in an encrypted format. Furthermore, *BE2* also has a plugin which supports kill command to destroy the entire file system of the compromised system in order to disturb digital forensics.

The third version *Black Energy 3* was first discovered in 2014. *BE3* is a simplified version of *BE2*, and the first version that attempts to attack SCADA systems. *BE3*, similar to *Stuxnet*, also employed Dropper architecture. Dropper drops main DLL modules into a user-mode process such as svchost.exe (Service Host Process of Microsoft Windows). And *BE3* scans the internet for a specific Human-machine interface (HMI), i.e. the GE Intelligent Platforms HMI/SCADA (CIMPLICITY), and infects a target system by leveraging a directory traversal vulnerability of CimWebServer.exe (CIMPLICITY WebView component). After infection of the target system succeeds, *BE3* scans the network and local machines for data to leak.

C. HAVEX

Havex [20-21] is a Remote Access Trojan (RAT) that was discovered in 2013. *Havex* consists of two main components: a trojan horse and a command and control (C&C) server written in PHP. Specially, *Havex* malware has a network scanning module that scans SCADA/ICS devices on the network by leveraging the Open Platform Communications (OPC) and Distributed Component Object Model (DCOM). Similar to *Black Energy*, *Havex* is a cybercriminal tool for espionage against industrial systems, which is used by an Advanced Persistent Threat (APT) group.

D. INDUSTROYER

Industroyer [22, 23], also referred to as *Crashoverride* [24-26], is a malware discovered in 2016, and it is the first known malware which is specifically designed to attack power grids, especially focusing on the SCADA protocols such as IEC 101, IEC 104, and IEC 61850. *Industroyer* was used in the attack on a single transmission level substation of the power grid, which resulted in power outage. It leveraged OPC protocol to map the environment and select its targets similar to *Havex*. And it targeted the libraries and configuration files of HMIs. *Industroyer* replaces the Windows Notepad application with its trojanized version which includes a malicious shellcode.

E. TRITON/TRISIS

Triton [27], also called as *Trisis* [28] or *HatMan* [29], is a malware discovered in 2017. *Triton* was used to attack a petrochemical plant in 2017. It is written in Python and targeting the Schneider Electric's Triconex safety instrument system (SIS) controller, thus named as *Trisis*. *Triton* disguises itself as *TriStation* which is an administration tool for a SIS controller [30], and uploads new ladder logic to SIS controller in order to take control of the facilities. It is unclear how attackers can penetrate layer 1.

F. WANNACRY

WannaCry [31-33] is a ransomware found in 2017 and it was one of most successful ransomware attacks in history. This malware used zero-day exploits to spread to other Windows systems on the Internet. It is reported that several dozen computers of ICS were infected by this malware. *WannaCry* also injects a DLL module named *launcher.dll* into a system process *lsass.exe*. and this DLL module installs and executes the main executable file of ransomware which encrypts every file on local drives.

G. ATTACK ANALYSIS

Exploiting all imaginable attacks from primitive probing to manipulating an internal system, any sophisticated attackers or intruders have the ultimate goal, which is to achieve

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.2983179, IEEE Access

malfunction of primary field devices, consequently causing to disrupt normal operation and to degrade service quality, or to degenerate to the complete shut-down of system operation to the extreme. In the same vein, operators try to do the best to defend any possible threats to cause abnormal operations.

The attack model or scenario in the power system can be summarized into three steps. First, an attacker penetrates any vulnerable stations in the SCADA network, which corresponds to the penetration into layer 2 in Fig. 1. At the next stage, an attacker penetrates the station Local Area Network (LAN), i.e. layer 1, which is comprised of field device controllers and local HMIs. Finally, an attacker generates maliciously coded commands to the field devices and causes the power system into insecure or unsafe states. In rare cases, attackers can intrude into layer 1 directly, or even directly access into layer 0 from outside. However, all the attacks we investigated above follow the three stages of penetration. Once intruders get in the security zones of layer 2 or 1, data injection, interruption, interception, modification, fabrication, and Denial of Service (DoS) attacks are possible [34], because these attacks are directly related to compromise field devices, consequently causing malfunction in the power operation.

In Fig. 2 below, we point out major exploit tactics which were used in the well-known cases of successful malware attacks when they penetrated at each stage. As shown in the table, most malwares are using the *Code Injection* technique when they penetrate layer 1, which is the most critical security zone. *Code Injection* is an attack technique that injects an arbitrary code into another program in order to execute it on that process context. The code to inject is DLL in case of *Stuxnet*, *Black Energy 3* and *WannaCry*. It can also be a shellcode that *Industroyer* injected into a normal Windows program.

Penetration

	Layer 2	Layer 1	Layer 0
Stuxnet	• Infected USBs	• Propagates via: - Infected Step7 project files - OS/Software exploits • DLL Injection (Dropper, rootkit)	• DLL Injection (DLL replacement) • Replace PLC code blocks
Black Energy 3	• Infected Word/Excel documents • Social engineering to run malicious VBA macro	• Propagates via Web Server vulnerability of specific HMI • DLL Injection (Dropper)	• Opens circuit breakers to cut power • Replace PLC firmware • Destroy disk storages
Industroyer	• Social engineering	• Backdoor (RAT) • Shellcode Injection • Leverages OPC protocol	• Kills master process • Masquerades as new master • Wipe all datas
Triton	• Social engineering		• Replace ladder logic of SIS controller
WannaCry	• Stolen credentials (in case of earlier versions before May 12, 2017)	• Propagates via SMB protocol vulnerability • DLL Injection • Backdoor	• Encrypt files of local drives
Havex	• Cross-site Scripting (XSS)	• Infects web pages • Backdoor (RAT)	• Industrial espionage

FIGURE 2. Comparison of malware penetration methods

What is worth noting in this attack analysis is the attack vector, by which intruders gain access to a system in order to accomplish their attack goal. The system vulnerability to be exploited is vendor-specific software implementation flaws. It is known that most HMI devices in the SCADA systems are running under Microsoft Windows operating system [35, 36]. Thus, most cyber attacks on the SCADA exploited a vulnerable structure of Windows systems and OPC protocol.

III. REVIEW OF PROPOSED SECURITY STRATEGIES

A. NETWORK SECURITY MONITORING

In reality, network separation cannot guarantee complete prevention of illegal access, but can only decrease the possibility of unauthorized access and mitigate any disruptive incidents by illegal operations. While network separation is aimed to prevent unauthenticated and unauthorized access into the SCADA system, security monitoring intends to detect and reports any illegitimate behavior inside the SCADA system. For this reason, security monitoring is considered to be an integral part of security strategies. Compared to typical IT systems, the SCADA systems have predictable patterns of traffic flow between fixed network nodes. Furthermore, the communication in the SCADA system mostly relies on the standard protocols which correspond to the application layer protocols above the TCP/IP stack. This uniformity makes easy to derive rules that can be used to decide which behaviors are normal or not in the network. Moreover, security monitoring can be easily adapted to the current SCADA network without any significant change of system architecture and components.

The intrusion detection system (IDS) is a main tool to do network security monitoring. The anomaly detection among the IDS detection algorithms is to determine which observed events are to be identified as abnormal because it has significant deviation from normal behavior which is called ‘profile.’ Thus, the main task of designing SCADA-specific IDS is to derive the profiles which reflect all possible semantics of the SCADA system. The paper [37] classifies four kinds of information to derive profiles for SCADA-aware IDS: network flows, application protocols, process features, and data features for self-learning as shown in Fig. 3.

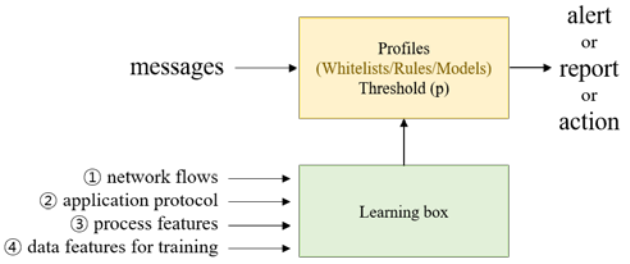

FIGURE 3. Framework of Intrusion Detection Systems for SCADA [37]

While a large portion of the traffic occurring in the IT network is involved in human actions, leading to dynamic traffic patterns, the SCADA network configuration is stable and has the fixed IP addressing schemes. Unless there is any abrupt change in the system configuration, intentional or

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.2983179, IEEE Access

unintentional, communication paths are deterministic since data exchanges take place between fixed nodes such as control servers, intelligent electronic device (IED), programmable logic controller (PLC), remote terminal unit (RTU), and other field devices, which are mostly machine-to-machine communication.

Considering the SCADA system is running on the TCP/IP stack and Ethernet-based LANs, the flow can be defined by the three address tuples of source and destination stations: (MAC address, IP address, TCP/UDP port). Then we can specify a list of allowable flows in the network, which is often called a whitelist. This list can be preconfigured based on the knowledge of legitimate nodes and communication in the network, and/or configured dynamically based on the result of monitoring traffic by switches in the network. The papers [38-41] elaborate the network flow-aware anomaly detection approach.

The SCADA networks in the power control system are operating based on the standard communication protocols such as IEC 61850, IEC 60870-5-104, DNP3.0, and Modbus. All the measured data, state information, and on/off command are delivered as the application protocol data unit (APDU) encapsulated in IP packets. The semantics of each fields in the APDU can be used to verify the validity of the incoming packets, and detect any anomalous communication. Furthermore, any correlated rules between different fields of the same packet or between the fields of subsequent packets are also utilized as effective criteria to decide whether incoming packets adhere to the logic of the application protocols.

In this way, the extracted APDU information of incoming packets is compared with profiles by simple matching or sometimes checking rules of a single packet or between the sequence of packets. This protocol-based approach is often called rule-based, model-based, or specification-based anomaly detection. The papers [42-49] derive the profile regarding IEC 61850 protocols, and the papers [50] propose IDS based on IEC 60870-5-104 and DNP3.0 protocols. The paper [51] proposes IDS relating to Modbus protocol, and the paper [52] proposes the framework to generate dynamic rules for multi-protocols.

The effectiveness of anomaly detection can be increased as we can derive profiles which are more aware of semantics of the target SCADA system. If we can extract normal/abnormal features from the changes of process-level states and field devices, we can enhance the performance of anomaly detection capability. The papers [53-58] propose and explain the process-aware anomaly detection approaches. This approach can also be combined with the protocol-aware monitoring, which is often touted as the deep packet inspection (DPI) [46, 48].

In recent years, many researchers have begun to focus on constructing SCADA-IDSs using machine learning and deep learning methods. Currently, few works have been done for deep learning-based IDSs focusing on the SCADA systems [59-65]. Some works are targeting the power SCADA systems [61, 63], while others are focusing on the different SCADA domains such as gas pipeline networks [59, 64], water treatment systems [62, 65], and heating control system [60].

All the works are using the data field information of the protocol data units (PDU) of the SCADA communication protocols such as Modbus [59, 60, 62] and DNP3.0 [61, 63], along with the TCP/IP header information. Together with the input data features, the deep learning models will affect the effectiveness of the detection model. The deep learning models to be used in the papers are varied: LSTM [59, 62, 65], ANN [60], CNN [61, 64], and RNN [63].

Machine and deep learning can provide significant benefits, since it is expected to offer capability of updating defense logic quick to respond new variants of attacks. However, model construction requires significant understanding of the problem being addressed, which involves analyzing the data collected to extract 'features' that are used to learn detection logics. The challenge of deep learning-based IDS lies in proper usage of data features for training and testing to reflect as much of semantics of the target SCADA system as possible, and reasonable validation to verify its usefulness of defense against attack variants.

B. COMMUNICATION MESSAGE SECURITY

As mentioned in section I, the goal of message security is to guarantee message integrity, message authentication, and/or message confidentiality. That is, message security prevents intruders from manipulating messages by modifying message contents, injecting false messages, or reusing old messages.

The IEC 62351 standards [66-69] are exemplary results of this effort. The IEC 61850 standards define the communication protocols for information exchange in the substations. In the substation automation systems based on IEC 61850, there are three kinds of message exchanges: between control devices such as IEDs inside a substation, between different substations, between substations and control centers. The IEC 62351 standards aim at developing security solutions for specific IEC 61850 communication protocols.

The IEC 62351 standards adopt the common IT security algorithms and protocols to derive security measures, such as crypto algorithms, keyed-hash message authentication code (HMAC), digital signature, and Transport Layer Security (TLS). TLS is the most favorite protocol for client-server application running over the TCP/IP stacks, since TLS provides all the flavors required for message security. In addition to technical capability, another factor to be considered is to meet the performance criteria for each application. The time-critical application based on the bare Ethernet communication may choose the light-weight message authentication protocols, avoiding heavy computing loads involving in encryption/decryption. For a practical reason, HMAC is preferable over digital signature which requires the public key crypto computation.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.2983179, IEEE Access

Aside from technical merit of the message security strategy, the main obstacle to adapt this strategy lies in implementation. Compared to IT systems, the long life span of the SCADA system hinders the deployment of IEC 62351-capable devices. Even though some transitional solution has been proposed, it is not expected that the situation of tardy adaption will be changed soon.

C. EVALUATION

We assume that attackers have gained access to the SCADA network by exploiting weakness of the perimeter defense systems such as firewall or any authentication mechanism. Our major concerns are how we can maintain the health and sanity of the SCADA system at this stage. Then, the question is whether the network security monitoring or the message security method such as IEC 62351 can detect any manipulation of communication messages by attackers. The answer, unfortunately, is not optimistic, considering the nature of the recent sophisticated attacks against SCADA/ICS systems which we analyzed in section II.

Main purpose of those security approaches is to check the integrity and authenticity of data communication, i.e. whether they comply with the underlying SCADA protocols and accompanying rules in the system. The network IDS, which is currently proposed in the context of the SCADA/ICS systems, intends to detect anomaly behavior based on traffic patterns or contextual mismatch of message contents which are specified in the standard protocols such as Modbus, DNP3.0 and IEC 61850. On the other hand, in the communication message security method, a receiving station checks the integrity, authenticity, and/or confidentiality of all arriving messages by deriving rules from its knowledge to be agreed upon with the other station beforehand.

However, the real attacks do not take place by way of the standardized and defined communication messages. In the real systems, there are other types of information or messages that are exchanged for configuration and operation related to various software, especially Operating System (OS). In most cases, attack vectors are vulnerability of the software underlying the host systems. Using this vulnerability, sophisticated attacks hijack or take control of host stations, control servers and eventually device controllers, once they penetrate the networks in one way or another. Once an attacker takes over the control of processes on the host in the SCADA system, it can masquerade its malicious messages as legitimate ones, consequently avoiding any possible detection by these methods.

As shown in the *Stuxnet*, the attack was realized by injecting malicious codes - DLL files - into legitimate process memory and executing the malicious codes in the context of a legitimate process. Once a code is injected into the target process, it has full access to the process memory and can manipulate code and data blocks of Programmable Logic Controllers (PLC), and eventually cause malfunction of field devices. For this reason, the authenticity and integrity check of the messages which are exchanged between control servers and local device controllers cannot detect these kinds of attacks.

In this regard, the process-aware security monitoring is considered to be more reasonable and realistic approach, since it can more likely detect possible attacks to cause abnormal process-level operations. It is a more effective approach than the other network monitoring approaches, keeping track of process states and decide the heath of the SCADA system based on information related to process state transition such as current or voltage changes.

If we want to find any security solutions on top of vendor-dependent platforms, we need to take special attention to the attack vectors which have been revealed in the recent incidents. From all incidents explained in this section, we can notice that every known malware attack on the SCADA system exploits weaknesses of host system environment. Therefore, it is important to prepare measures on this kind of attacks. In this paper, we pay attention to most commonly used attack method such as *Code Injection*, more specifically *DLL Injection*, and propose a solution to block it. It is expected that this approach can prevent all the hosts in the SCADA system from being taken over by this malicious attack, consequently keeping its sanity all the time.

IV. CODE INJECTION

When an attacker wants to force a running process to execute arbitrary actions, they leverage a technique called *Code Injection*. The term *code* in *Code Injection* not only indicates a shellcode, but also can indicate a *Dynamic Link Library (DLL) module* or *executable file* whose filename extension is .exe. And *Code Injection* indicates an attempt to inject that code into another running process in order to execute it on that process context.

But many attackers normally inject a DLL rather than a shellcode, because injecting shellcode has a number of restrictions. In case of shellcode injection, strings used by shellcode have to reside in shellcode and an attacker should use *LoadLibrary/GetProcAddress* API to call external functions when the referenced external library is not loaded yet. Thus, an attacker cannot directly use codes compiled from a C/C++ compiler. Therefore, many attackers prefer *DLL Injection* rather than *shellcode injection*. To make the problem simple, this work only focused on *DLL Injection*.

A. DYNAMIC LINK LIBRARY

Dynamic Link Library (DLL) is a Windows executable file type corresponding to *shared library* on Linux. DLL existed since the first version of Windows, and every Windows API is provided as a function included in a system DLL file such as *kernel32.dll*, *ntdll.dll*, *user32.dll*, and *gdi32.dll*. Fig. 4 shows export functions from *kernel32.dll* via *Dependency Walker* [70].

A DLL can export functions that can be used by other programs. When a program needs to call functions of DLL, a

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.2983179, IEEE Access

programmer lets linker make a connection to the DLL by providing symbolic information file (*.lib), or loads a DLL manually by calling *LoadLibrary* function and finds the location of function to call with *GetProcAddress* function. In the former case, Windows automatically loads the DLL and finds the location of function to call when the process starts, which is called as *static load*. And the latter is called as *dynamic load* [71].

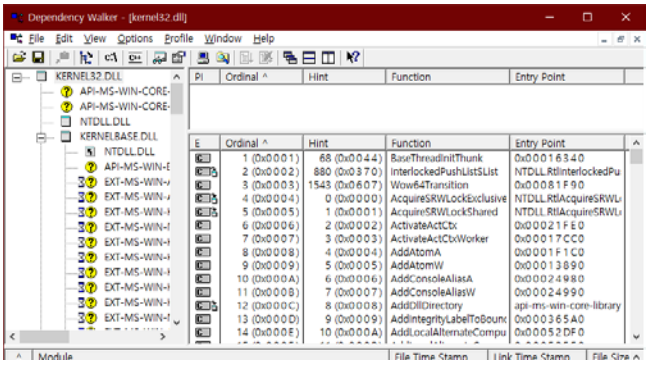

FIGURE 4. **Export functions of kernel32.dll**

DLL can also have an entry point named *DllMain*, which is a callback function called from the operating system as a notification whenever DLL is loaded or unloaded, or a new thread starts or exits [72]. A loaded DLL can execute its codes on the memory address space of its host process when its DllMain function is called or its function is called from the external [73]. In this way, *DLL Injection* can be used as an attack technique of Code Injection.

B. DLL INJECTION

DLL Injection is a Code Injection technique that forces another process running on the system to load an arbitrary DLL module. After it is first invented by Jeffrey Richter on his book [74] in 1995, many application software including antiviruses and malwares started to leverage DLL Injection technique and DLL Injection has become common.

The simplest approach for DLL Injection is replacing normal DLL file used by a target program with a new DLL file or putting a new DLL file in directory with higher priority than normal DLL file. This approach is used by *Stuxnet* to load Stuxnet DLL when a user opens infected Step7 project. However, this approach has a clear limitation in that DLL can only be injected before target process runs.

Thus, DLL Injection into running process is a more common strategy. There are three main approaches to inject a DLL into running process: *Remote Thread*, *Windows Hook*, *Asynchronous Procedure Call (APC)*.

C. DLL INJECTION VIA REMOTE THREAD

This approach is the original way published in Jeffrey Richter's book [74]. It leverages a Windows functionality of remote thread creation for another process. A remote thread can be created by calling *CreateRemoteThread* API [75].

```
HANDLE CreateRemoteThread(
    HANDLE hProcess,
    LPSECURITY_ATTRIBUTES lpThreadAttributes,
    SIZE_T dwStackSize,
    LPTHREAD_START_ROUTINE lpStartAddress,
    LPVOID lpParameter,
    DWORD dwCreationFlags,
    LPDWORD lpThreadId);
```

The argument *lpStartAddress* takes an address of the function to execute on target process context and a thread function should follow the next prototype [76]:

```
DWORD WINAPI ThreadProc(LPVOID lpParameter);
```

And this prototype is compatible with LoadLibraryA API [77] to load a DLL due to the same number of parameters,

```
HMODULE LoadLibraryA(LPCSTR lpLibFileName);
```

This approach uses address of *LoadLibraryA* API as a function pointer to a thread function. Fig. 5 summarizes this approach.

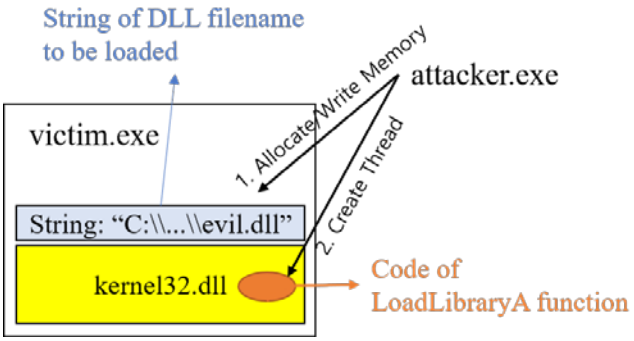

FIGURE 5. **DLL Injection by Remote Thread**

D. DLL INJECTION VIA WINDOWS HOOK

DLL can be also injected by Windows Hook functionality. Windows Hook is one of fundamental functions of Windows that allows applications to monitor a certain type of system events. SetWindowsHookEx API [78] is used for this.

```
HHOOK SetWindowsHookExA(
    int idHook,
    HOOKPROC  lpfn,
    HINSTANCE hmod,
    DWORD dwThreadId);
```

It is documented that, when other module's memory address is provided in the parameter *hmod*, Windows tries to load the specified module in the process where the event occurred if the module is not loaded yet [78]. Fig. 6 summarizes this approach.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.2983179, IEEE Access

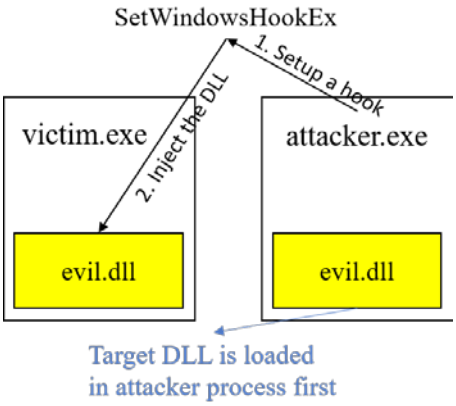

FIGURE 6. **DLL Injection via Windows Hook**

E. DLL INJECTION VIA APC

Asynchronous Procedure Call (APC) [79] is a Windows functionality that allows a thread to execute other tasks during alertable wait operations. QueueUserAPC API [80] is used for this purpose.

```
DWORD QueueUserAPC(
    PAPCFUNC pfnAPC,
    HANDLE hThread,
    ULONG_PTR dwData);
```

Like CreateRemoteThread API, QueueUserAPC takes a memory address of function to call parameter *pfnAPC* and the *pfnAPC* function should follow the prototype below [81]:

```
void Papcfunc(ULONG_PTR Parameter);
```

And this prototype is compatible with LoadLibraryA API, so this function can be used to call LoadLibraryA for DLL Injection. Fig. 7 summarizes this approach.

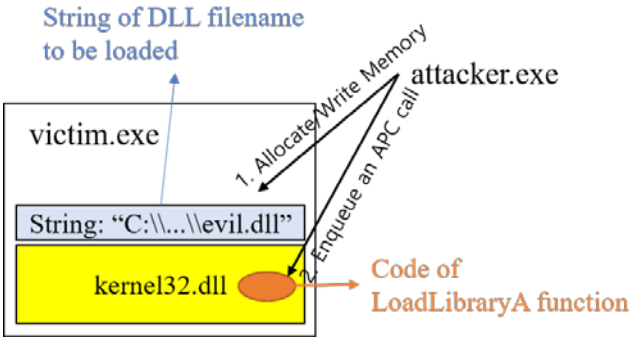

FIGURE 7. **DLL Injection via APC**

V. API HOOKING

In order to block DLL Injection attempts, it is necessary to monitor *LoadLibrary* function calls. In this paper, we suggest monitoring via API Hooking. API Hooking means a technique that intercepts function calls of specific Windows API.

In user mode processes, there are two main approaches to achieve API Hooking: *Import Address Table (IAT) Hooking*, *Inline Hooking*.

A. IAT HOOKING

When an application binds a DLL via the *static load* in order to use functions of that DLL, the linker constructs *Import Directory Table*, *Import Lookup Table*, and *Import Address Table* in the executable file [82].

Import Directory Table records information of referred external DLLs, and *Import Lookup Table* specifies DLL functions used by the program. *Import Address Table (IAT)* contains addresses of referred external DLL functions, and IAT entries are filled when Windows' Portable Executable (PE) loader loads an executable file.

Whenever the program calls a function of external DLL, the program calls the function by looking up the function address in IAT. Therefore, it is possible to redirect these function calls by replacing function address in IAT. This approach is called *IAT Hooking*.

Because any of special assembly language knowledge are not needed for hooking, it is the simplest way to hook an API. However, due to its mechanism, it is not possible to intercept API calls in case of applications using *dynamic load*. Thus, this paper chooses to use the second method, *Inline Hooking*.

B. INLINE HOOKING

Inline Hooking is a hooking technique that overwrites the prologue of a target function with a branch instruction like JMP in Intel x86, and this is the first designed by Hunt and Brubacher [83]. Fig. 8 shows the mechanism of Inline Hooking.

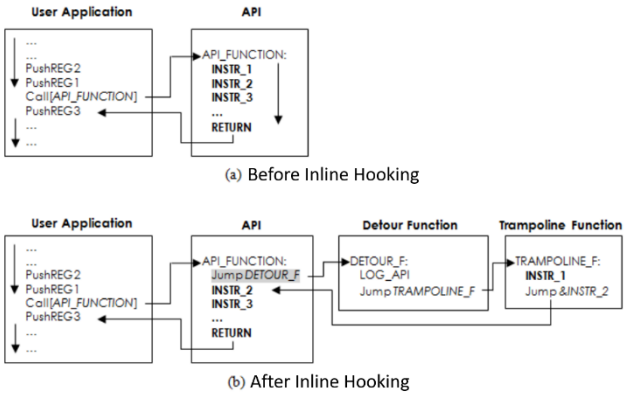

FIGURE 8. **Change of API call flow before/after Inline Hooking [84]**

In *Inline Hooking*, the replaced new function redirected from API is called *Detour Function*, and the code part that calls original function back from *Detour Function* is called *Trampoline Function*.

The API is hooked by overwriting the first part of the function with a JMP instruction jumping to *Detour Function*. If it is necessary to call the original function from *Detour Function*, then the performer makes *Trampoline Function*.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.2983179, IEEE Access

Trampoline Function is a function that consists of overwritten original first part of API and a JMP instruction jumping to where the hooking ends.

As an example, we are going to introduce *speedhack*. *Speedhack* is a game cheating tool using API Hooking, which manipulates the playing speed of game. In order to manipulate speed of game, *Speedhack* hooks a number of time-related API functions like *GetTickCount*. Fig. 9 and Fig. 10 show disassembly of *GetTickCount* API before/after applying speedhack functionality of *Cheat Engine 7.0* [85], with *OllyDbg* [86], and Fig. 11 shows schematic structure of its API Hooking of *GetTickCount* API.

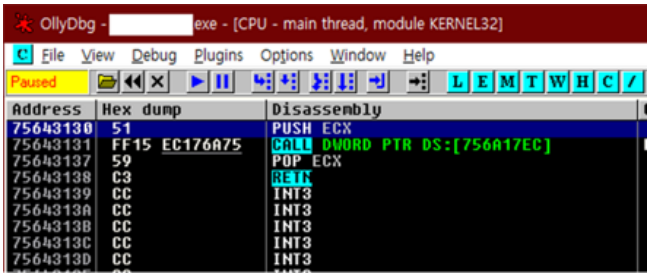

FIGURE 9. **GetTickCount API before applying speedhack**

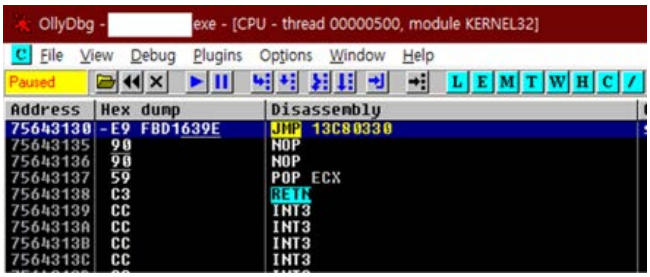

FIGURE 10. **GetTickCount API after applying speedhack**

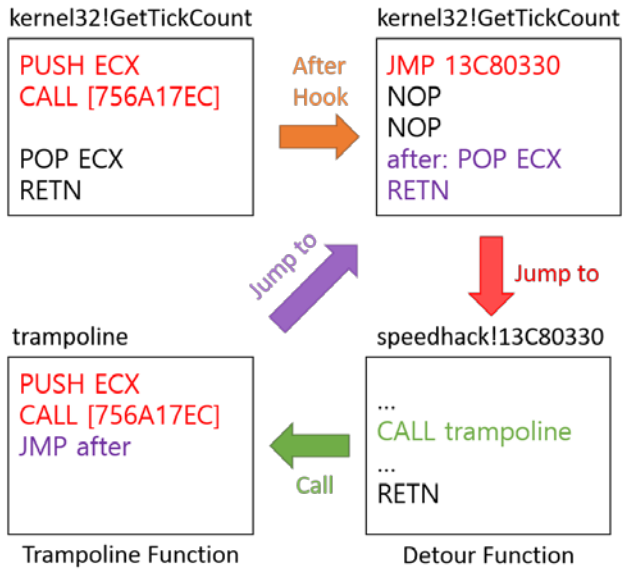

FIGURE 11. **Schematic structure of GetTickCount API with speedhack**

VI. ANTI-DLL INJECTION

In this paper, we propose an anti-DLL Injection system that hooks Windows API functions providing DLL loading functionality by using Inline Hooking and verifies the caller by checking the return address. Hooked API functions are *LoadLibraryA* (in *kernel32.dll*/*kernelbase.dll*), *LoadLibraryW* (in *kernel32.dll* and *kernelbase.dll*), *LoadLibraryExA* (in *kernel32.dll* and *kernelbase.dll*), *LoadLibraryExW* (in *kernel32.dll* and *kernelbase.dll*), and *LdrLoadDll* (in *ntdll.dll*).

In the aforementioned DLL Injection approaches like *Remote Thread*, *Windows Hook*, and *APC*, *LoadLibrary* API is called from a fixed location in operating system module. Table I shows the call location of each DLL Injection approach.

TABLE I
CALL LOCATIONS OF DLL INJECTION METHODS

Method	Call Location	Disassembly (Windows 10)
Remote Thread	kernel32.dll	CALL [756A1F88] *CALL ESI* PUSH EAX
Windows Hook	user32.dll	PUSH DWORD [EDI+1C] *CALL [768E4308]* MOV ESI, EAX
APC	ntdll.dll	MOV ECX, [EBP+8] *CALL [EBP+8]* MOV DWORD [EBP-4], -2

Normally we do not call *LoadLibrary* API by using *Remote Thread*, *Windows Hook*, and *APC*. It is normal to directly call *LoadLibrary* API from a code in the code memory of the program, i.e. *.text* section. Therefore, DLL Injection attacks can be blocked from checking caller address of *LoadLibrary* function calls.

The caller address of a function call can be acquired by checking return address in the stack frame and we will discuss it further in section VI-C.

A. SYSTEM PROPOSAL

Based on the above conclusions, we propose a system to block DLL Injection attacks. Our proposed solution consists of a number of algorithms and side effects between them, rather than a single algorithm, hence we classify our solution as a system. The overall process flow across the system that we propose is shown in Fig. 12.

In the initialization phase, the system prepares for monitoring of DLL load attempts. After the initialization phase, the system yields the execution flow to the target program, and makes the target program run.

When a DLL load attempt occurs, our detour function is called and our system decides whether the attempt is allowed. If the protected software wants to cancel protection, then our system enters the finalization phase.

In the finalization phase, our system cleans up the monitoring devices, i.e. *API hooking*.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.2983179, IEEE Access

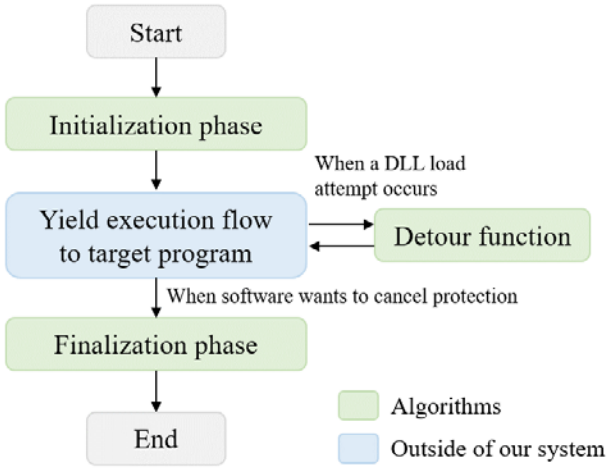

FIGURE 12. **Process flow across our anti-DLL injection system**

B. DETAILS OF ALGORITHMS

The following pseudocode in Fig. 13 describes the algorithm in the initialization phase.

Algorithm: Initialize Anti-DLL Injection system
Input: APIs = {(M, D) | M ∈ DLL loading APIs, D are detour functions corresponding to each M}
Function Initialize()
  create a new thread myself by using CreateRemoteThread
  record the caller address of thread procedure
  $M_{user32} \leftarrow$ get memory address of user32.dll
  If M_{user32} is a valid address **Then**
    $S_{user32} \leftarrow$ get size of image from M_{user32}
  End If
  enqueue an APC call by using QueueUserAPC
  record the caller address of APC procedure
  inline hook DLL loading APIs to monitor (APIs)
End Function

FIGURE 13. **Pseudocode of initialization of Anti-DLL Injection system**

At first, the system searches for caller addresses of three methods, i.e. *Remote Thread*, *Windows Hook*, and *APC*. For *remote thread* and *APC*, the algorithm records their caller addresses by using dummy function. And for *Windows Hook*, the algorithm gets memory address of *user32.dll* (M_{user32}) instead and obtains the size of image from M_{user32} (S_{user32}) to know the range of memory address of *user32.dll* that is $[M_{user32}, M_{user32} + S_{user32})$. After that, the system hooks DLL loading APIs such as *LoadLibraryA*, *LoadLibraryW*, *LdrLoadDll*, etc. to monitor them.

And the following pseudocode shown in Fig. 14 describes the algorithm in the finalization phase when the system closes. When the system is closing, the system unhooks DLL loading APIs.

Algorithm: Finalize Anti-DLL Injection system
Input: APIs = {M | M ∈ DLL loading APIs}
Function Finalize()
  unhook DLL loading APIs (APIs)
End Function

FIGURE 14. **Pseudocode of finalization of Anti-DLL Injection system**

And the following pseudocode in Fig. 15 describes the algorithm in the detour functions called whenever every DLL loading attempt arises.

Algorithm: Generic form of detour function
Input: $X, M_{RemoteThread}, M_{user32}, S_{user32}, M_{APC}, T$
  X = set of input arguments $\{X_1, X_2, \dots, X_n\}$
  $M_{RemoteThread}$ = recorded caller address of thread procedure
  M_{user32} = memory address of user32.dll
  S_{user32} = size of user32.dll image
  M_{APC} = recorded caller address of APC procedure
  T = corresponding trampoline function
Output: Y = function output
Function DetourFunction()
  $M_{caller} \leftarrow$ get caller address of detour function
  If $M_{caller} \in \{M_{RemoteThread}, M_{APC}\} \vee (M_{caller} \in [M_{user32}, M_{user32} + S_{user32}) \wedge M_{caller}$ has machine code pattern of DLL loader of Windows Hook) **Then**
    notify user of DLL Injection attack
    halt the execution
  End If
  If $M_{caller} \notin$ set of memory of every loaded module **Then**
    notify user of DLL Injection attack
    halt the execution
  End If
  $Y \leftarrow T(X)$
End Function

FIGURE 15. **Pseudocode of generic form of detour function**

When the detour function is called, in other words, a DLL loading attempt occurred, the function first gets the caller address of its own (M_{caller}). Then, the system compares the caller address M_{caller} with recorded memory addresses of three DLL Injection methods, i.e. Remote Thread, Windows Hook, and APC. In case of Windows Hook, instead of comparing directly with recorded caller address, the system compares M_{caller} with the range of memory address of *user32.dll* and hard-coded machine code pattern of DLL loader in Windows Hook. If the caller address is in any of those, the system recognizes it as a DLL Injection attack attempt, and halts its execution.

In addition, the system also determines whether the caller address is included in the set of memory of every loaded module. This guarantees that the loading attempt is genuine, more than only comparing with recorded caller addresses, because it is possible to avoid our protection by using newly allocated code memory that has CALL instructions for *LoadLibrary* and there are the cases using this method to call *LoadLibrary* like DLL Injection feature of Cheat Engine 7.0. Actually, this is *Shellcode Injection* rather than *DLL Injection*.

However, since using this approach to inject a DLL is common, it also addresses this DLL Injection problem. Fig. 16 shows disassembly and memory type of remote thread procedure when Cheat Engine 7.0 injects a DLL. Private memory type indicates that it is newly allocated memory.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.2983179, IEEE Access

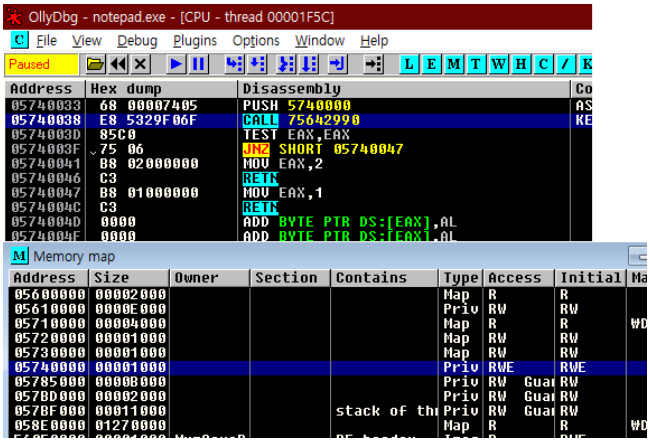

FIGURE 16. Disassembly and memory type of remote thread procedure of Cheat Engine 7.0 DLL Injection functionality

C. STACK FRAME ON IA-32

In this paper, we only focused on 32-bit Windows applications running on Intel Architecture 32-bit (IA-32) architecture which is the most common in SCADA HMI systems. Other architectures such as AMD64, IA-64 (Itanium) and ARM or other platforms like Linux are not considered yet.

Fig. 17 shows the stack memory on 32-bit Windows applications after a function is called [87].

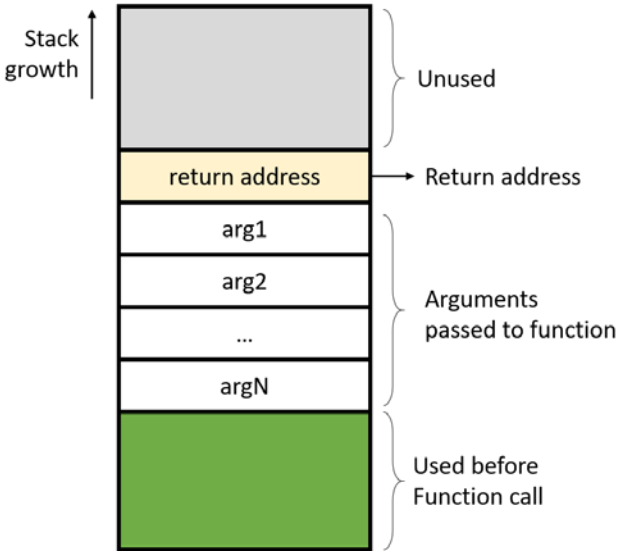

FIGURE 17. Stack memory after function call

In the memory, the return address to the caller is placed before the first argument, so it is possible to investigate the caller address of the function with a simple memory operation.

VII. EXPERIMENTS AND RESULTS

A. IMPLEMENTATION

We implemented the proposed Anti-DLL Injection system in a form of DLL module, using Visual C++ in Microsoft Visual Studio 2019. Instead of direct comparison between caller address and locations of every loaded module, we just determined the memory type of caller memory. We used *VirtualQuery* API [88] to obtain the memory type of a caller.

In Microsoft Windows, it is possible to distinguish whether a memory is from an executable image (MEM_IMAGE), or a newly allocated memory (MEM_PRIVATE), or a view of mapped file (MEM_MAPPED) by checking *Type* field in MEMORY_BASIC_INFORMATION structure. When a suspicious DLL loading attempt is found, instead of calling original DLL loading API, our system pretends that the attempt is failed because of error, to deceive the attacker.

B. WIN32/PARITE

Win32/Parite [89, 90] is a computer virus running under Microsoft Windows. This malware is first discovered in October 2001, and this is spreading by infecting other Windows executable files. This employed Dropper architecture. Once *Parite* runs, *Parite* injects its DLL module into a system process named *explorer.exe*, and starts the infection process inside the system process.

Fig. 17 shows the disassembly of the DLL Injection attack part of malware Win32/Parite. It shows that Parite tries DLL Injection by using WH_CALLWNDPROC Windows Hook.

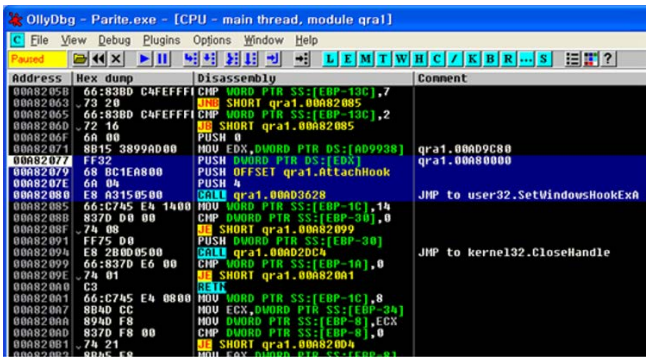

FIGURE 17. Disassembly of Parite's DLL Injection part

This experiment uses a Win32/Parite virus sample to prove that our implementation blocks DLL Injection attack from *Parite* virus. The experiment was done under a Windows XP environment.

Fig. 18 shows the list of loaded DLL modules in explorer.exe process before *Parite* runs.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.2983179, IEEE Access

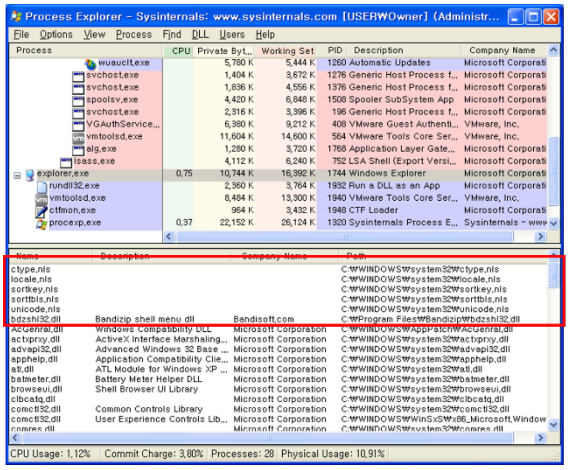

FIGURE 18. List of loaded DLLs in explorer.exe before Parite started

Fig. 19 shows the list of loaded DLL modules in explorer.exe process after *Parite* has been executed, without our implementation running.

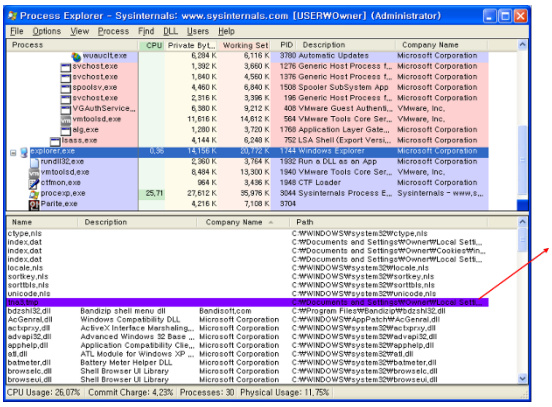

FIGURE 19. List of loaded DLLs in explorer.exe after Parite started (without protection)

As shown in Fig. 19, the virus sample *Parite.exe* is running and at the same time DLL from Parite *tna3.tmp* is injected into *explorer.exe* process (highlighted in the figure).

And the following Figure 20 shows the list of loaded DLL modules in *explorer.exe* process after *Parite* has been executed while our protection is working.

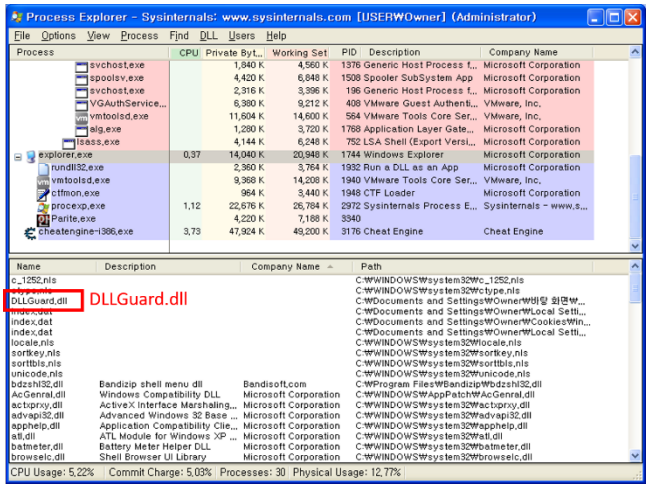

FIGURE 20. List of loaded DLLs in explorer.exe after Parite started (with protection)

As shown in Figure 20, virus sample *Parite.exe* is running but there is no DLL from *Parite* in *explorer.exe* process. It is clear that our implementation *DLLGuard.dll* blocked a DLL injection attempt from the *Parite* virus.

When running GMER [91] which is a rootkit detection tool, it finds that LoadLibraryA, LoadLibraryW, LoadLibraryExA, LoadLibraryExW, and LdrLoadDll functions are hooked by the system, which is shown in Fig. 21.

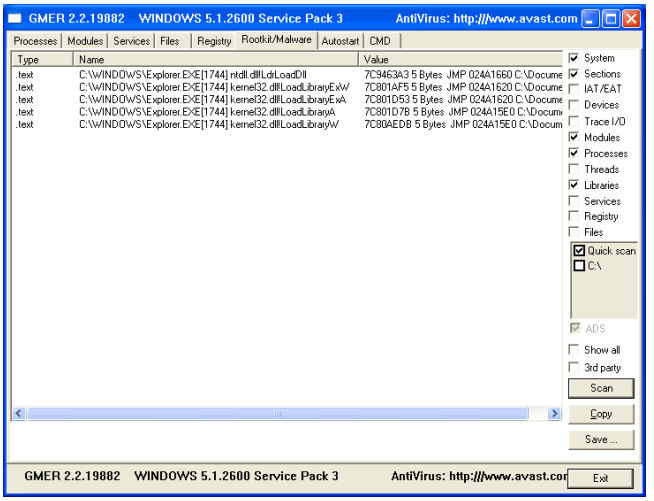

FIGURE 21. Rootkit scan result from GMER

C. INJECTALLTHETHINGS

InjectAllTheThings [92] is a DLL Injection tool that is developed by Rui Reis. It supports seven DLL Injection methods: Three methods using Remote Thread (*CreateRemoteThread*, *NtCreateThreadEx*, *RtlCreateUserThread*), Thread Execution Hijacking (*SetThreadContext*), Windows Hook (*SetWindowsHookEx*), APC (*QueueUserAPC*), Reflective DLL Injection.

We already explained *Remote Thread*, *Windows Hook*, *APC* in Section IV-C, IV-D, and IV-E. As for *Remote Thread*,

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.2983179, IEEE Access

this tool uses not only *CreateRemoteThread* API, but also *NtCreateThreadEx* API, and *RtlCreateUserThread* API. These APIs are doing the same operations. But, since the defense measure might only focus on some, not all, of them, the API which is not included could bypass the defense measure. For this reason, the tool includes all the APIs to test so that the security measure can be verified to work against all attacks.

Thread Execution Hijacking and *Reflective DLL Injection* are not discussed in the paper. DLL Injection using Thread Execution Hijacking calls *LoadLibrary* function by manipulating the context of a working thread.

Reflective DLL Injection is a recent DLL Injection technique near to generic code injection, which implements PE Loader directly in order to put DLL codes on target process memory. Our work did not consider about it yet.

We tried all the methods supported by InjectAllTheThings, and the result is shown in Table II.

TABLE II
ATTACK TESTING WITH INJECTALLTHETHINGS

Class	Used API	Attack Result
Remote Thread	CreateRemoteThread	Failed
Remote Thread	NtCreateThreadEx	Failed
Remote Thread	RtlCreateUserThread	Failed
Thread Execution Hijacking	SetThreadContext	Failed
Windows Hook	SetWindowsHookEx	Failed
APC	QueueUserAPC	Failed
Reflective DLL Injection		Unknown (abnormal termination)

As shown in Table II, our implementation successfully blocked all DLL Injection attempts from InjectAllTheThings.

VIII. CONCLUSION

In this paper, we pointed out a missing piece of the security strategies currently proposed for the SCADA/ICS systems, which is to keep host sanity in the system. Most attacks which targeted the SCADA/ICS systems were equipped with specifically designed malwares for the systems, and they leveraged vulnerabilities of the host system. This paper analyzed a number of malware attacks on the SCADA system and explained the most commonly used technique, *Code Injection*. This paper also proposed an approach to block the DLL Injection, and proved that our proposed solution works for real attack cases.

So far, our work only focused on *DLL Injection* on *Windows x86*. The future work of this work can be to extend this solution to more Code Injection attacks than only DLL Injection attacks. We may also combine *Machine Learning* techniques with this task in order to automatically detect Code Injection attack patterns.

REFERENCES

[1] S. Hong, "Cyber Security Strategies and their Implications for Substation Automation Systems," *Int'l J. of Smart Grid and Clean Energy*, vol. 8, no. 6, pp 747-756, Nov. 2019.
[2] Recommended Practice: Improving Industrial Control System Cybersecurity with Defense-in-Depth Strategies, DHS ICS-CERT, Sep. 2016.
[3] Guide to Industrial Control Systems (ICS) Security, NIST SP 800-82 Rev.2, May 2015.
[4] Industrial communication networks – Network and system security – Part 3-3: System security requirement and security levels, IEC 62443-3-3, Aug. 2013.
[5] What is PERA? [Online]. Available: http://www.pera.net/. Accessed on: Feb. 18, 2020.
[6] F. Cleveland, "IEC TC57 WG15: IEC 62351 Security Standards for the Power System Information Infrastructure,", IEC, June 2012.
[7] S. Hong and J.-M. Lee, "Direction of Security Monitoring for Substation Automation Systems," in *The 5th EECSS*, Lisbon, Portugal, August 2019.
[8] J. Slowik, "Evolution of ICS Attacks and the Prospects for Future Disruptive Events," [Online]. Available: https://dragos.com/wp-content/uploads/Evolution-of-ICS-Attacks-and-the-Prospects-for-Future-Disruptive-Events-Joseph-Slowik-1.pdf. Accessed on: Feb. 18, 2020.
[9] J. Weiss, "Control system cyberattacks have become more stealthy and dangerous – and less detectable," April 2019. [Online]. Available: https://www.controlglobal.com/blogs/unfettered. Accessed on: Feb. 18, 2020.
[10] N. Falliere, L. O. Murchu, and E. Chien, "W32.Stuxnet Dossier version 1.4," Symantec, Feb. 2011.
[11] E. Byres, A. Ginter, and J. Langill, "How stuxnet Spreads – a Study of Infection Paths in Best Practice Systems," Tofino Security, Feb. 2011.
[12] A. Matrosov, E. Rodionov, D. Harley, and J. Malcho, "Stuxnet Under the Microscope," Rev. 1.31, ESET, 2011.
[13] R. Langner, "To Kill a Centrifuge: A Technical Analysis of What Stuxnet's Creators Tried to Achieve," The Langner Group, Nov. 2013.
[14] H. Lau, "Trojan.Dropper," Symantec, Apr. 2012.
[15] A. Malik, "DLL Injection and Hooking," [Online]. Available: https://securityxploded.com/dll-injection-and-hooking.php. Accessed on: Feb. 18, 2020.
[16] CyberX, "BlackEnergy 3 – Exfiltration of Data in ICS Networks," ver. 1.0, May, 2015.
[17] ThreatSTOP, "Security Report: Black Energy," Feb. 2016.
[18] E-ISAC, "Analysis of the Cyber Attack on the Ukrainian Power Grid," Mar. 2016.
[19] R. Khan, P. Maynard, K. McLaughlin, D. Laverty, and S. Sezer, "Threat Analysis of BlackEnergy Malware for Synchrophasor based Real-time Control and Monitoring in Smart Grid," in *the 4th Int'l Symp. for ICS & SCADA Cyber Security Research*, 2016.
[20] NJ Cybersecurity & Communications Integration Cell (NJCCIC), "Havex," Aug. 2017. [Online]. Available: https://www.cyber.nj.gov/threat-profiles/ics-malware-variants/havex. Accessed on: Feb. 18, 2020.
[21] Cybersecurity and Infrastructure Security Agency (CISA), "ICS Alert (ICS-ALERT-14-176-02A)", Jun. 2014. [Online]. Available: https://ics-cert.us-cert.gov/alerts/ICS-ALERT-14-176-02A. Accessed on: Feb 17, 2020.
[22] ESET, "Win32/Industroyer: A new threat for industrial control systems," Jun. 2017. [Online]. Available: https://www.welivesecurity.com/wp-content/uploads/2017/06/Win32_Industroyer.pdf. Accessed on: Feb. 18, 2020.
[23] E-ISAC, "ICS Defense Use Case No.6: Modular ICS Malware," Aug. 2017. [Online]. Available: https://ics.sans.org/media/E-ISAC_SANS_Ukraine_DUC_6.pdf. Accessed on: Feb. 18, 2020.
[24] Dragos, "CRASHOVERRIDE: Analysis of the Threat to Electric Grid Operations," Jun. 2017. [Online]. Available: https://dragos.com/wp-content/uploads/CrashOverride-01.pdf. Accessed on: Feb. 18, 2020.
[25] J. Slowik, "Anatomy of an Attack: Detecting and Defeating Crashoverride," VB2018, October, 2018.
[26] J. Slowik, "CRASHOVERRIDE: Reassessing the 2016 Ukraine Electric Power Event as a Protection-Focused Attack," Dragos, Aug. 2019.

[27] B. Johnson, D. Caban, M. Krotofil, D. Scali, N. Brubaker, and C. Glyer, "Threat Research: Attackers Deploy New ICS Attack Framework TRITON and Cause Operational Disruption to Critical Infrastructure," FireEye, Dec. 2017. [Online]. Available: https://www.fireeye.com/blog/threat-research/2017/12/attackers-deploy-new-ics-attack-framework-triton.html. Accessed on: Feb. 18, 2020.

[28] Dragos, "TRISIS Malware: Analysis of Safety System Targeted Malware," 2017, [Online]. Available: https://dragos.com/wp-content/uploads/TRISIS-01.pdf. Accessed on: Feb. 18, 2020.

[29] Cybersecurity and Infrastructure Security Agency (CISA), "MAR-17-352-01 HatMan - Safety System Targeted Malware (Update B)," Mar. 2017. [Online]. Available: https://www.us-cert.gov/ics/MAR-17-352-01-HatMan-Safety-System-Targeted-Malware-Update-B. Accessed on: Feb. 18, 2020.

[30] A. D. Pinto, Y. Dragoni, and A. Carcano, "TRITON: The First ICS Cyber Attack," presented at the *Black Hat USA 2018*, Las Vegas, U.S., 2018.

[31] Kaspersky Lab ICS CERT, "WannaCry on industrial networks: error correction," Jun. 2017. [Online]. Available: https://ics-cert.kaspersky.com/reports/2017/06/22/wannacry-on-industrial-networks/. Accessed on: Feb. 18, 2020.

[32] A. McNeil, "How did the WannaCry ransomworm spread?", May 2017. [Online]. Available: https://blog.malwarebytes.com/cybercrime/2017/05/how-did-wannacry-ransomworm-spread/. Accessed on: Feb. 18, 2020.

[33] D.-Y. Kao, S.-C. Hsiao, and R. Tso, "Analyzing WannaCry Ransomware Considering the Weapons and Exploits," in *ICACT-TACT*, vol. 7, no. 2, pp. 1098-1107, Mar. 2018.

[34] S. East, J. Butts, M. Papa, and S. Shenoi, "A Taxonomy of Attacks on the DNP3 Protocol," in *Int'l Conf. on Critical Infrastructure Protection*, 2009.

[35] "Dated Windows software the weak link for SCADA systems," Bnamericas, 2017. [Online]. Available: https://www.bnamericas.com/en/news/dated-windows-software-the-weak-link-for-scada-systems. Accessed on: Feb. 18, 2020.

[36] D. Brandl, "Are Microsoft technologies still best for process control systems?", Control Engineering, 2014. [Online]. Available: https://www.controleng.com/articles/are-microsoft-technologies-still-best-for-process-control-systems/. Accessed on: Feb. 18, 2020.

[37] S. Hong, J. Lee, M. Altaha, and M. Aslam "Security Monitoring and Network Management for the Power Control Network," Int'l J. of Electrical and Electronic Engineering & Telecommunications (will be published) 2020. [Online]. Available: http://ants.mju.ac.kr/publication/P005-v4.pdf. Accessed on: Feb. 18, 2020.

[38] A. Lemay, J., Rochon, and J. Fernandez, "A Practical flow white list approach for SCADA systems," in *the 4th International Symposium for ICS & SCADA Cyber Security Research*, 2016.

[39] G. K. Ndonda and R. Sadre, "A Two-level Intrusion Detection System for Industrial Control System Networks using P4," in *ICS & SCADA*, 2018.

[40] R. Udd, M. Asplund, S. Nadjm-Tehrani, M. Kazemtabrizi, and M. Ekstedt, "Exploiting Bro for Intrusion Detection in a SCADA System," in *CPSS'16*, June 2016.

[41] R. Barbosa, "Anomaly Detection in SCADA Systems, A Network Based Approach," Ph.D. dissertation, University of Twente, April 2014.

[42] M. Ko and M. Jenkner, "Cybersecurity Defense System for Distributed Communication Network in IEC-61850 Power Substations," in *Cigre De Colloquium*, June 2019.

[43] J. Hong and C-C. Liu, "Intelligent Electronic Devices with Collaborative Intrusion Detection Systems," *IEEE Tran. On Smart Grid*, vol. 10, no.1, January 2019.

[44] M. Kabir-Querrec, "Cyber security of the smart grid control systems: intrusion detection in IEC 61850 communication networks," Ph.D. dissertation, University Grenoble, 2017. [Online] Available: https://hal.archives-ouvertes.fr/tel-01609230

[45] M. T. A. Rashid, S. Yussof, and Y. Yusoff, "Trust System Architecture for Securing GOOSE Communication in IEC 61850 Substation Network," in *International Journal of Security and its Applications*, vol. 10, no. 4, pp289-302, 2016.\

[46] Y. Yang, H.-Q. Xu, L. Gao, Y-B. Yuan, and K. McLaughlin, "Multidimensional Intrusion Detection System for IEC 61850 based SCADA Networks," *IEEE Transactions On Power Delivery*, vol. 32, no. 2, April 2017.

[47] J. Hong, C-C. Liu, "Integrated Anomaly Detection for cyber security of the substation," in *IEEE Tran. On Smart Grid*, vol. 5, no. 4, July 2014.

[48] Y. Yang, K. McLaughlin, S. Sezer, T. Littler, E. G. Im, B. Pranggono, and H. F. Wang, "Multiattribute SCADA-Specific Intrusion Detection System for Power Networks," *IEEE Tran. On Power Delevery*, vol. 29, no. 3, June 2014.

[49] Y. Yang, K. McLaughlin, L. Gao, S. Sezer, Y. Yuan, and Y. Gong, "Intrusion detection system for IEC 61850 based Smart Substations," in. *IEC Power and Energy Society General Meeting*, 2016.

[50] H. Lin, A. Slagell, C. D. Martina, Z. Kalbarczyk, and R. K. Lyer, "Adapting Bro into SCADA: Building a Specification-based Intrusion Detection System for the DNP3 Protocol," in *CSIIRW '12*, October 2012.

[51] S. Cheung, R. Dutertre, M. Fong, U. Lindqvist, and K. Skinner, "Using Model-based Intrusion Detection for SCADA Networks," in *the SCADA Security Scientific Symposium*, January 2007.

[52] J. Nivethan and M. Papa, "Dynamic Rule Generation for SCADA Intrusion Detection." In *IEEE Symposium on Technologies for Homeland Security*, May 2016.

[53] J. J. Chromik, A. Memke, and B. Haverkort, "Bro in SCADA: dynamic intrusion detection polices based on a system model," in *ICS & SCADA*, 2018.

[54] H. Lin, A. Slagell, Z. T. Kalbarczyk, P. W. Sauer, and R. K. Iyer, "Runtime Semantic Security Analysis to Detect and Mitigate Control-Related Attacks in Power Grids," *IEEE Tran. On Smart Grid*, vol. 9, no. 1, January 2018.

[55] J. Nivethan and M. Papa, "A SCADA Intrusion Detection Framework that Incorporates Process Semantics," In *CISRC*, 2016.

[56] J. J. Chromik, A. Remke, B. R. Haverkort, "What's under the hood? Improving SCADA security with process awareness," in *Joint Workshop on Cyber-Physical Security and Resilience in Smart Grids (CPSR-SG)*, 2016.

[57] H. Janicke, A. Nicholson, S. Webber, and A. Cau, "Runtime-Monitoring for Industrial Control Systems," *Electronics* 2015, 4, pp 995-1017.

[58] D. Hadziosmanovic, R. Sommer, and E. Zambon, "Through the Eye of the PLC: Towards Semantic Security Monitoring for Industrial Control Systems," in *ACSAC'14*, December 2014.

[59] R. L. Perez, F. Adamsky, R. Soua, and T. Engel, "Forget the Myth of the Air Gap: Machine Learning for Reliable Intrusion Detection in SCADA Systems," *EAI Endorsed Transactions on Security and Safety*, 2019.

[60] A. Hijazi and J-M. Flaus, "A Deep Learning Approach for Intrusion Detection System in Industry Network," in *The 1st int'l conference on Big Data and Cybersecurity intelligence*, Beirut, Lebanon, 2019.

[61] H. Yang, L. Cheng, and M. D. Chuah, "Deep-Learning-Based Network Intrusion Detection for SCADA Systems," in *IEEE conf. on Communication and Network Security (CNS): Workshops: CPS Sec: Internation Workshop On Cyber-Physical Systems Security*, 2019.

[62] J. Gao, L. Gan, F. Buschendorf, L. Zhang, H. Liu, P. Li, X. Dong, and T. Lu, "Omni SCADA Intrusion Detection Using Deep Learning Algorithms," *arXiv*, Aug. 2019.

[63] S. Kwon, H. Yoo, and T. Shon, "RNN-based Anomaly Detection in DNP3 Transport Layer," in *IEEE int'l Conf. on Communications, Control, and Computing Technologies for Smart Grids (SmartGridComm)*, 2019.

[64] J. Liu, L. Yin, Y. Hu, S. Lv, and L. Sun, "A Novel Intrusion Detection Algorithm for Industrial Control Systems Based on CNN and Process State Transition," in *IEEE 37th Int'l Performance, Computing and Communications*, 2018.

[65] J. Goh, S. Adepu, M. Tan, and L. X. Shan, "Anomaly Detection in Cyber Physical Systems using Recurrent Neural Networks," in *IEEE 18th Intl Symposium on High Assurance Systems Engineering*, 2017.

[66] Power systems management and associated information exchange - Data and communications security - Part 3: Communication network and system security – Profiles including TCP/IP, IEC TC57 WG15, IEC 62351-3, May 2018.

[67] Power systems management and associated information exchange - Data and communications security - Part 4: Profiles including MMS, IEC TC57 WG15, IEC 62351-4, November 2018.
[68] Power systems management and associated information exchange - Data and communications security - Part 5: Security for IEC 60870-5 and derivatives, IEC TC57 WG15, IEC 61850-5, 2013.
[69] Power systems management and associated information exchange - Data and communications security - Part 6: Security for IEC 61850, IEC TC57 WG15, IEC 62351-6, 2007.
[70] *Dependency Walker 2.2*. (2006). Microsoft. Accessed: Feb. 18, 2020. [Online]. Available: http://www.dependencywalker.com/
[71] Microsoft. *Link an executable to a DLL*. (2019). Accessed: Feb. 18, 2020. [Online]. Available: https://docs.microsoft.com/en-us/cpp/build/linking-an-executable-to-a-dll?view=vs-2019
[72] Microsoft. *DllMain entry point*. (2018). Accessed: Feb. 18, 2020. [Online]. Available: https://docs.microsoft.com/en-us/windows/win32/dlls/dllmain
[73] J. Richter, "Chapter 19: DLL Basics," in *Programming Applications for Microsoft Windows*, 4th ed., Redmond, WA, USA: MS Press, 1999.
[74] J. Richter, "Chapter 16: Breaking Through Process Boundary Walls," in *Advanced Windows: The Developer's Guide to the Win32 API for Windows NT 3.5 and Windows 95*, 4th ed., Redmond, WA, USA: MS Press, 1995.
[75] Microsoft. *CreateRemoteThread function*. (2018). Accessed: Feb. 18, 2020. [Online]. Available: https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-createremotethread
[76] Microsoft. *ThreadProc callback function*. (2018). Accessed: Feb. 18, 2020. [Online]. Available: https://docs.microsoft.com/en-us/previous-versions/windows/desktop/legacy/ms686736(v=vs.85)
[77] Microsoft. *LoadLibraryA function*. (2018). Accessed: Feb. 18, 2020. [Online]. Available: https://docs.microsoft.com/en-us/windows/win32/api/libloaderapi/nf-libloaderapi-loadlibrarya
[78] Microsoft. *SetWindowsHookExA function*. (2018). Accessed: Feb. 18, 2020. [Online]. Available: https://docs.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-setwindowshookexa
[79] Microsoft. *Asynchronous Procedure Calls*. (2018). Accessed: Feb. 18, 2020. [Online]. Available: https://docs.microsoft.com/en-us/windows/win32/sync/asynchronous-procedure-calls
[80] Microsoft. *QueueUserAPC function*. (2018). Accessed: Feb. 18, 2020. [Online]. Available: https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-queueuserapc
[81] Microsoft. *PAPCFUNC callback function*. (2018). Accessed: Feb. 18, 2020. [Online]. Available: https://docs.microsoft.com/ko-kr/windows/win32/api/winnt/nc-winnt-papcfunc
[82] Microsoft. *PE Format*. (2019). Accessed: Feb. 18, 2020. [Online]. Available: https://docs.microsoft.com/en-us/windows/win32/debug/pe-format
[83] G. Hunt and D. Brubacher, "Detours: Binary Interception of Win32 Functions," in the proceedings of the *Third USENIX Windows NT Symposium*, Jul. 1999.
[84] M. Shaid, S. Zainudeen and M. Maarof, "In memory detection of Windows API call hooking technique," in *Int'l conf. on Computer, Communications, and Control Technology (I4CT)*, 2015.
[85] *Cheat Engine 7.0*. (2019). Accessed: Feb. 18, 2020. [Online]. Available: https://www.cheatengine.org/
[86] *OllyDbg 1.1*. (2004). Accessed: Feb. 18, 2020. [Online]. Available: http://www.ollydbg.de/
[87] K. A. Monnappa, "Chapter 4: Assembly Language and Disassembly Primer," in *Learning Malware Analysis*, 1st ed. UK: Packt, 2018
[88] Microsoft. *VirtualQuery function*. (2018). Accessed: Feb. 18, 2020. [Online]. Available: https://docs.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-virtualquery
[89] Microsoft. *Win32/Parite*. (2010). Accessed: Feb. 18, 2020. [Online]. Available: https://www.microsoft.com/en-us/wdsi/threats/malware-encyclopedia-description?Name=Win32%2FParite
[90] AhnLab. *Virus Center - Win32/Parite.B*. (2010). Accessed: Feb. 18, 2020. [Online]. Available: https://global.ahnlab.com/site/securitycenter/viruscenter/virusList.do
[91] *GMER 2.2.19882*. (2016). Accessed: Feb. 18, 2020. [Online]. Available: http://www.gmer.net/
[92] *InjectAllTheThings*. (2020). Accessed: Feb. 18, 2020. [Online]. Available: https://github.com/fdiskyou/injectAllTheThings

JAE-MYEONG LEE (M'20) was born in Seoul, S. Korea on September 28, 1992. He received a Bachelor's degree in computer engineering from Myongji University, Korea, in 2018. Then he started his Master's degree program in computer engineering at Myongji University, Korea, from 2018. He is now researching about Machine Learning, Deep Learning, Computer Security, Smart Grid, and SCADA Security, under supervision of Prof. Sugwon Hong.

He was enthusiastic about computer programming from a very young age. In 2009, He was awarded Microsoft Most Valuable Professional (MVP) for Visual Basic by Microsoft, which was the youngest record in Korea.

SUGWON HONG was born in Incheon, S. Korea. He received the B.S. degree in physics from Seoul National University, Seoul, Korea, in 1979, and the M.S. and Ph.D. degrees in computer science from North Carolina State University, Raleigh, USA, in 1988 and 1992, respectively.

His employment experience includes Korea Institute of Science and Technology (KIST) Software Development Center; Korea Energy Economics Institute (KEEI); SK Innovation Co., Ltd. (formerly Korea Oil Company); Electronic and Telecommunication Research Institute (ETRI). Currently he is a professor in the department of computer engineering, Myongji University, Yongin, S. Korea, where he has been since 1995. His current research interest are cyber security and smart grid.