논문 2020-1-8 http://dx.doi.org/10.29056/jsav.2020.06.08

PLC용 uC/OS-II 운영체제 기반 펌웨어에서 발생 가능한 취약점 패턴 탐지 새니타이저

한승재*, 이건용**, 유근하*, 조성제*†

A Sanitizer for Detecting Vulnerable Code Patterns in uC/OS-II Operating System-based Firmware for Programmable Logic Controllers

Seungjae Han*, Keonyong Lee**, Guenha You*, Seong-je Cho*†

요 약

산업제어 시스템에서 많이 사용되는 PLC(Programmable Logic Controller)는 마이크로 컨트롤러, 실시간 운영체제, 통신 기능들과 통합되고 있다. PLC들이 인터넷에 연결됨에 따라 사이버 공격의 주요 대상이 되고 있다. 본 논문에서는, 데스크톱에서 개발한 uC/OS-II 기반 펌웨어를 PLC로 다운로드 하기 전, 펌웨어 코드의 보안성을 향상시켜 주는 새니타이저를 개발한다. 즉, PLC용 임베디드 펌웨어를 대상으로 버퍼의 경계를 넘어선 접근을 탐지하는 BU 새니타이저(BU sanitizer)와 use-after-free 버그를 탐지하는 UaF 새니타이저(UaF sanitizer)를 제안한다. BU 새니타이저는 대상 프로그램의 함수 호출 그래프와 심볼 정보를 기반으로 제어 흐름 무결성 위배도 탐지할 수 있다. 제안한 두 새니타이저를 구현하고 실험을 통해 제안 기법의 유효성을 보였으며, 기존 연구와의 비교를 통해 임베디드 시스템에 적합함을 보였다. 이러한 연구결과는 개발 단계에서 의도하지 않은 펌웨어 취약점을 탐지하여 제거하는데 활용할 수 있다.

Abstract

As Programmable Logic Controllers (PLCs), popular components in industrial control systems (ICS), are incorporated with the technologies such as micro-controllers, real-time operating systems, and communication capabilities. As the latest PLCs have been connected to the Internet, they are becoming a main target of cyber threats. This paper proposes two sanitizers that improve the security of uC/OS-II based firmware for a PLC. That is, we devise BU sanitizer for detecting out-of-bounds accesses to buffers and UaF sanitizer for fixing use-after-free bugs in the firmware. They can sanitize the binary firmware image generated in a desktop PC before downloading it to the PLC. The BU sanitizer can also detect the violation of control flow integrity using both call graph and symbols of functions in the firmware image. We have implemented the proposed two sanitizers as a prototype system on a PLC running uC/OS-II and demonstrated the effectiveness of them by performing experiments as well as comparing them with the existing sanitizers. These findings can be used to detect and mitigate unintended vulnerabilities during the firmware development phase.

한글키워드 : 프로그램 가능한 논리 제어기, uC/OS-II, 새니타이저, 임베디드 펌웨어, 버퍼 언더플로, 해제 후 사용

keywords : PLC, uC/OS-II, Sanitizer, Embedded firmware, Buffer underwrite, Use-after-Free

* 단국대학교 컴퓨터학과
** 단국대학교 응용컴퓨터공학과
† 교신저자: 조성제(email: sjcho@dankook.ac.kr)
접수일자: 2020.05.22. 심사완료: 2020.06.02.
게재확정: 2020.06.19.

1. 서 론

최근 스마트 팩토리, 공장 자동화, 모터 제어,

www.kci.go.kr

펌핑 시스템, 항공우주 등의 산업 공정이나 기반 시설 제어에 산업 제어 시스템(Industrial Control Systems, ICS)을 적용하는 사례가 증가하고 있다. 산업 제어 시스템은 감시제어 및 데이터 취득(Supervisory Control and Data Acquisition, SCADA) 시스템, 분산 제어 시스템(Distributed Control System, DCS), 프로그램 가능한 논리 제어기(Programmable Logic Controller, PLC) 등으로 구성된다. 이들 중 PLC는 시스템 입력과 요구사항에 기반을 두어 물리적인 컴포넌트들을 관리하고 제어하도록 프로그램된 임베디드 장치다. 즉, PLC는 프로그램 입력에 따라 순차적으로 연산을 수행하면서 PLC에 연결된 액츄에이터(actuator), 센서 등을 제어 및 관리한다[1][2].

제어 시스템을 구성하는 PLC의 경우, 특정 임무를 수행하는 태스크의 수행 시작과 완료(마감시한)에 대한 실시간성을 보장해야 한다. 따라서 PLC에는 실시간 운영체제(Real-Time Operating Systems: RTOS)들이 주로 사용된다. 대표적인 PLC 운영체제(OS)로는 VxWorks, Microware OS-9, QNX Neutrino, RTLinux, Windows Embedded Compact (Windows CE), uC/OS-II 등이 있다.

최근 PLC는 개발자/운영자에게 효율성 및 편의성을 제공하기 위해 ICT 기술(TCP/IP, 웹서버, FTP, 원격 관리 서비스 등)과 결합됨에 따라, 네트워크를 이용한 산업 제어 시스템 및 PLC에 대한 사이버 공격이 증가하고 있다[3]. 대표적인 사례로 이란 원자력시설 우라늄 원심분리기 가동을 중단시킨 Stuxnet, 제어 시스템에 대한 정보 수집 및 유출을 목적으로 한 Duqu, 전력제어 시스템을 손상시킨 Flame, 특정 조직의 시스템 데이터와 금융정보를 수집하는 Gauss, 중동 석유 회사의 약 3만 대의 컴퓨터를 파괴하고, 정보를 유출한 Shamoon등이 있다[4-6]. 이러한 산업 제어 시스템의 PLC를 대상으로 한 공격은 산업시설의 가동중단 및 오작동을 유발하여, 대규모 정전사태, 전산망 마비, 원전 오염물질 유출 등으로 국가적 재난을 일으킬 수 있다[3][7].

한편 PLC의 최하위 수준 프로그래밍 추상화 계층은 펌웨어(firmware)로, PLC 펌웨어가 악의적으로 위·변조될 경우 산업 제어 시스템에 대한 완전한 통제를 공격자에게 넘겨주게 된다[8]. 따라서 산업 제어시스템에서 PLC 자체에 대한 보안뿐만 아니라, EWS (Engineering Work Station)에서 개발되어 PLC에 다운로드 되는 프로그램인 펌웨어에 대한 보안성을 향상시킬 수 있는 연구가 수행되어야 한다.

본 논문에서는 uC/OS-II 운영체제 기반의 PLC에 다운로드 되는 펌웨어를 대상으로 바이너리 정적 분석을 통해 스택(stack) 및 힙(heap) 영역에 대한 보안 취약점(vulnerability)을 미리 능동적으로 탐지 및 필터링하는 "코드 새니타이저(Code Sanitizer)"를 제안한다. 제안하는 코드 새니타이저는 "BU 새니타이저(Buffer Underwrite Sanitizer, Buffer Underflow Sanitizer)"와 "UaF 새니타이저(Use-After-Free Sanitizer)" 2개의 모듈로 구성된다. 두 개의 모듈 모두 펌웨어 코드를 대상으로 역어셈블(disassemble)하여 어셈블리 언어 수준에서 탐지한다.

"BU 새니타이저"는, 버퍼 언더플로(underflow) 취약점과 연관된 코드 패턴을 식별·탐지한다. 더욱 효과적인 방어를 위해, 펌웨어 코드에 대한 함수 콜 그래프(call graph)와 함수에 대한 심볼 정보를 이용하여 제어 흐름 무결성(control flow integrity, CFI) 위배 여부도 판단한다. "UaF 새니타이저"는, 메모리 정보를 유출할 수 있는 Use After Free 취약점과 관련된 코드 패턴을 탐지한다. 제안한 새니타이저 기법으로 PLC 펌웨어의 취약점을 개선하여, PLC에 대한 보안 위협을 줄일 수 있다.

본 논문의 구성은 다음과 같다. 2장에서는 관

www.kci.go.kr

련 연구에 대해 기술한다. 3장에서는 제안한 새니타이저 기법을 설명하며, 4장에서는 버퍼 언더플로와 UaF 취약점이 존재하는 경우 악용될 수 있음을 실험을 통해 증명한다. 5장에서는 구현과 실험을 통해, 제안 기법의 유효성과 타당성을 보이며, 기존 새니타이저 기법들과 비교 및 분석한다. 마지막으로 6장에서 결론 및 향후 연구에 대해 기술한다.

2. 관련연구

새니타이저는 특정 프로그램에 대해 추가적인 검사를 수행하면서 인스트루먼테이션(instrumentation)하게 해 주는 컴파일 프레임워크(compilation framework)이다[9]. 새니타이저는 테스팅 단계에서 보안 정책 위배와 관련된 소프트웨어 결함을 찾는 데 도움을 준다. 새니타이저는 메모리 안전(memory safety) 또는 타입 안전(type safety)과 같은 저수준 보안 위배를 탐지한다.

2.1 Address sanitization 기법

AddressSanitizer (*ASan*)[10]는 소스코드의 메모리 버그를 감지하기 위한 도구이다. C/C++로 작성된 프로그램은, 스택과 힙 영역에서 발생할 수 있는 버퍼 오버플로/언더플로, Use After Free, Double-free, invalid free, 메모리 누수 등의 버그에 노출될 수 있다. 이러한 버그들을 예방하고 메모리 안전을 보장하기 위해, AddressSanitizer는 컴파일러 수준에서 중요 메모리 영역을 표시하는 코드를 삽입하고, 프로그램 실행 시 메모리 객체(object) 주변에 red-zone을 설정하여 메모리에서 발생할 수 있는 오류(버그)의 원인을 파악하고 잘못된 접근을 차단한다.

Red-zone은 메모리 접근의 유효성을 확인하기 위해, 메모리 객체들 사이에 삽입된 특별한 공간이다. 이 red-zone을 관리하기 위해 shadow byte라는 자료구조를 유지하며, 메모리 접근 시 대상 메모리 주소의 유효 여부가 점검된다. 또한, 인스트루멘테이션을 적용하여, 메모리 보호 목적으로 설정된 특수한 값이 변경되지 않고 유지되는지를 보장한다.

ASan을 이용하여 Chromium 브라우저 및 3rd-party 라이브러리에서 300개 이상의 버그를 탐지하였다. ASan 기법의 성능을 평가한 결과, 평균 73%의 속도저하가 발생하였고, 인스트루멘테이션으로 인해 메모리 사용량은 평균 3.37배 증가하였다.

2.2 Thread sanitization 기법

스레드(threads) 간의 데이터 경쟁(data race)은 둘 이상의 스레드가 동시에 공유 메모리에 접근하면서 쓰기 연산을 수행할 때 발생하는 상황이다. 데이터 경쟁은 데이터 손상 또는 세그멘테이션 결함(Segmentation fault)을 초래할 수 있다. *ThreadSanitizer* (*TSan*) 프레임워크[11]는 프로그램 실행 중에 동적으로 메모리 접근 및 동기화 이벤트를 감시하여, 스레드 간의 데이터 경쟁을 탐지한다. 동기화 이벤트란 두 개의 스레드가 임계 구역(critical section), 즉 공유 메모리에 동시에 진입하려고 시도하는 것을 말하며, Dynamic Annotation API를 이용하여 데이터 경쟁 발생 여부를 탐지하였다. 이를 위해, 전역 변수와 힙 영역 변수들의 쓰기 연산을 감시하고, 어떤 스레드가 값을 변경했는지를 기록한다.

TSan은 테스팅 단계에서 Chromium 브라우저 및 3rd-party 라이브러리에서 수십 개의 데이터 경쟁을 탐지하였지만, 성능 평가 결과 5배에서 15배의 속도 저하와 메모리 사용량 오버헤드가 발생하였다[9].

www.kci.go.kr

2.3 Memory sanitization 기법

MemorySanitizer (*MSan*) 프레임워크[12]는 C / C++에서 초기화되지 않은 메모리 사용을 감지하는 동적 도구이다. Bit precise shadow memory를 활용하여 초기화되지 않은 메모리 영역을 읽어오는 행위를 탐지한다. 또한, 컴파일러에 Shadow propagation 기술을 삽입하여 초기화되지 않은 메모리 복사(copy)에 대한 잘못된 탐지를 최소화한다. 이 기법을 통해 초기화되지 않은 메모리의 출처를 추적하여 오류 메시지를 생성할 수 있다.

MSan은 할당된 객체들의 상태를 유지하기 위해, 중량의 프로그램 변환(heavy-weight program transformation) 기법을 사용하였다[9]. 메모리 할당 연산들과 초기화되지 않은 메모리를 읽는 행위를 감시한다. MSan는 3배의 속도저하와 2배의 메모리 사용량 오버헤드를 유발하였다.

2.4 PLC용 기존 실행코드 새니타이저

ASan, TSan, MSan 등의 기법들은 시스템 속도 저하 및 많은 양의 메모리 사용량 오버헤드를 유발하므로, 저성능 실시간 임베디드 시스템에 적용할 수 없다. 이에 실행코드 새니타이저(Executable Code Sanitizer)[13]가 제안되었으며, 함수 새니타이저(function sanitizer)와 포인터 새니타이저(pointer sanitizer)로 구성된다.

함수 새니타이저는 먼저 버퍼 오버플로 및 메모리 관련 취약점이 존재하는 라이브러리 함수 15개를 선정하였다. 실행코드가 PLC에 다운로드되기 전에 선정된 함수들의 사용 여부를 식별한다. 포인터 새니타이저는 포인터 변수에 메모리 주소가 하드 코딩(hard coding)되어 할당되는 패턴을 탐지한다. 대상 패턴의 예는, 하드 코딩된 주솟값을 할당하는 명령과 포인터 변수가 가리키는 주소의 값을 변경하는 명령이다. 또한, 소스코드와 심볼 정보를 이용하여 태스크의 스택의 범위를 계산하고 각 태스크에 할당된 스택의 주소 범위를 벗어난 포인터 연산을 탐지하는 기법도 제안하였다.

본 연구는 기존 [13]의 연구와 같은 환경에서, 기존에 다루지 않은 오류를 탐지하기 위해 버퍼 언더플로 새니타이저와 UaF 새니타이저를 제안하는 것이다.

3. PLC용 코드 새니타이저

실시간 임베디드 장치의 특징으로는, 제한된 저장소, 저용량 메모리, 낮은 연산능력, 저전력 요구, 실시간성 만족 등이 있다. 본 논문에서는 데스크톱 수준의 보안기법 및 메모리 관리 서비스를 적용하기 어려운 실시간 임베디드 환경에서 적용 가능한 효율적인 코드 새니타이저를 제안한다. 본 논문의 코드 새니타이저는 그림 1과 같이 BU 새니타이저와 UaF 새니타이저로 구성된다. 이 두 가지 기법은, 4장에서 후술할 취약점이 악용되는 사례를 통해 도출된 코드 패턴을 반영하였으며, 이는 스택과 힙에서 발생할 수 있는 이상 행위 패턴과 연관이 있다.

3.1 버퍼 언더플로 새니타이저

본 논문에서 제안하는 *버퍼 언더플로 새니타이저*(Buffer Underwrite Sanitizer, Buffer Underflow Sanitizer, BU 새니타이저)는 펌웨어 코드를 역어셈블하여 생성된 어셈블리 코드를 대상으로 수행된다. 구체적으로 버퍼에 할당된 범위(경계, boundary)를 벗어나 버퍼 하위 메모리 영역의 데이터를 참조 및 조작하는 비정상 코드 패턴을 탐지한다.

www.kci.go.kr

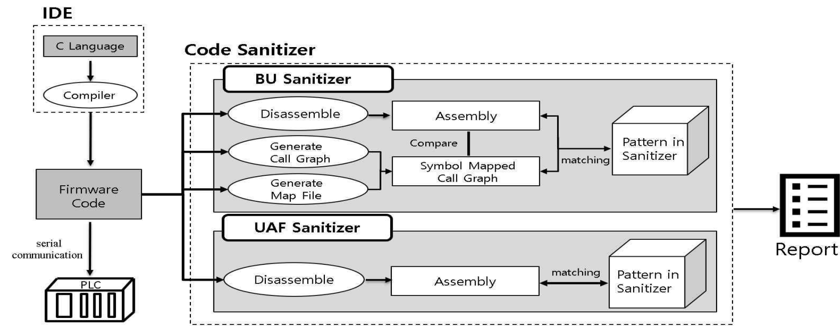

그림 1. 코드 새니타이저 개요
Fig. 1. Overview of Code Sanitizer

기존 인텔 x86 구조에서 잘 알려져 있는, 스택 영역의 버퍼 오버플로(buffer overflow) 취약점은 스택에 할당된 버퍼의 공간을 초과하는 데이터를 입력하여 버퍼보다 상단에 있는 프로그램의 반환 주소(return address)를 조작 가능하다. 이 취약점을 이용하여 궁극적으로 공격자가 원하는 악성 코드도 실행할 수 있다. 본 논문의 BU 새니타이저에서 탐지하려는 코드 패턴은 인텔 x86 구조의 버퍼 오버플로 취약점 코드와 차이가 있다.

본 실험 환경에서 함수 프롤로그(prologue)는 함수 호출 시 스택이 쌓이는 방향이 낮은 주소에서 높은 주소로 향하기 때문에 반환 주소는 버퍼보다 낮은 주소에 위치한다[14]. 본 논문에서는 버퍼 언더플로 취약점을 악용하여 공격자가 원하는 함수를 실행시키는 패턴을 보이고, 이 패턴을 BU 새니타이저가 탐지해야 할 코드 유형으로 선정하여 프로그램 개발 또는 테스트 단계에서 능동적으로 탐지함을 목표로 한다. 더불어, 고려한 패턴을 공격자가 우회할 경우를 대비해, 콜 그래프(call graph)와 함수의 심볼 정보를 이용하여 반환 주소를 조작하는 제어 흐름 무결성(CFI) 위배를 탐지하는 기법도 제안한다.

3.2 UaF 새니타이저

UaF(Use-After-Free)는 힙 영역에서 동적으로 할당한 메모리를 해제(free)하고 재사용(use)할 때 발생할 수 있는 취약점을 말한다. 힙 영역에서 이전에 해제된 메모리를 참조(reference)하여 수행 중인 프로그램을 크래시(crash) 시키거나 유효한 데이터를 손상시킬 수도 있다.

UaF 새니타이저(Use after Free Sanitizer)도 컴파일된 펌웨어 코드를 역어셈블하여 생성된 어셈블리 코드를 대상으로 수행한다. 어셈블리 코드를 분석해서 메모리 블록(Memory block)에 초기화되지 않은 데이터를 읽어오는 패턴을 탐지한다. 힙 영역에서 메모리 블록은 미리 만들어진 분할(partition)의 할당 단위로, 프로그래머가 동적으로 요청하여 할당받을 수 있다. 사용한 메모리 블록이 해제되어 해당 분할에 반납한 이후, 그 메모리 블록 할당이 요청될 경우, 이전에 반납한 메모리 블록의 영역을 할당받아 남아있는 데이터가 유출될 수도 있다.

본 논문의 실험 환경인 uC/OS-II에서는 프로그램에서 할당받은 메모리 블록을 해제(반환)하고자 할 때, 해제할 메모리 블록의 앞 2바이트만 초기화하는 특징이 있다. 따라서 이전에 해제된 메모리 블록을 재할당할 경우, 해제되기 전의 일부 내용을 알 수 있는 부분적인 UaF 취약점이 존재한다. uC/OS-II에서 이러한 메모리 블록 해제 기법을 알고 있는 공격자라면, 해제된 메모리

www.kci.go.kr

블록을 가리키는 블록 포인터의 2바이트 뒤의 문자열을 읽거나 출력하는 명령을 통하여, 해제된 메모리 블록의 내용을 파악할 수 있다.

4. 실험 환경 및 테스트 프로그램 구성

제안한 두 가지 기법을 TI(Texas Instruments)사의 TMDSDOCK28335 보드에 Micrium의 uC/OS-II를 설치하여 구현하였다. 개발 IDE (Integrated Development Environment)로 TI사에서 제공하는 CCS 9.1(Code Composer Studio 9.1)을 사용하였다. uC/OS-II에서 사용하는 자료형은 일반적인 C언어에서 사용하는 자료형과 차이점이 존재한다. 전체적인 실험 환경을 표 1에, uC/OS-II에서 사용하는 자료형을 표 2에 정리하였다.

표 1. 실험 환경
Table 1. Experimental environment

Board	TMSDOCK28335
MCU	32bit, 150Mhz
Architecture	Havard
OS	uC/OS-II
IDE	Code Composer Studio 9.1.0
Compiler	C2000
Language	ISO C Standard
Number of Tasks	2

표 2. uC/OS-II에서의 자료형
Table 2. Data types in uC/OS-II

Type Definition	uC/OS-II Type
unsigned char	INT8U
signed char	INT8S
unsigned short	INT16U
signed short	INT16S
unsigned long	INT32U
signed long	INT32S
float	FP32
long double	FP64

본 논문의 제안기법을 실험하기 위해 BU 및 UaF 취약점이 존재하는 프로그램을 작성하여 실험용 보드에 다운로드 하였다.

4.1 스택 기반 버퍼 언더플로 취약점

본 절에서는 공격자가, 스택 기반 버퍼 언더플로 취약점을 이용하여 특정 함수의 반환 주소를 조작하고 공격자가 지정한 함수를 호출할 수 있는 예시를 보인다. 그림 2에는 버퍼 언더플로 취약점이 존재하는 소스코드 예시가 나타나 있다.

```
INT16S y;
void Function_Caller()        --------- (1)
{
    void Function_Calee(INT16S);

    Function_Calee(-4);
    puts("End Function");
}
void Function_Callee(INT16S x) ---- (2)
{
    INT16S buffer[10]= {65,65,65,65,65,
                    65,65,65,65,65};-- (3)
    y=41615;          -------------- (4)
    buffer[x] = y;           --------- (5)
}
void Attacker_Function()  --------- (6)
{
    puts("Hacked!\n");
    return;
}
static  void  User_Task1 (void *p_arg)
{
   while(DEF_TRUE){
       Function_Caller();
       OSSemPend( AppTaskObjSem,
                  0,       &os_err);
   }
}
```

그림 2. 함수 반환 주소 조작이 가능한 소스코드 예
Fig. 2. An example of source code for manipulating the return address of a function

www.kci.go.kr

그림 2와 같이 사용자가 생성한 태스크(User_Task1)에서 사용자 정의 함수(Function_Caller)를 호출하면, MCU(Micro Controller Unit)는 User_Task1의 반환 주소를 저장하고 있는 RPC(Return Program Counter) 레지스터를 참조하여 반환 주소를 스택에 쌓는다. 이후, Function_Caller의 반환 주소를 RPC 레지스터에 저장한다. 이와 같은 방법으로 Function_Callee 함수를 호출할 때, Function_Caller 함수의 반환 주소를 스택에 저장하고, RPC 레지스터에 Function_Callee 함수의 반환 주소를 저장한다. 소스코드에는 Attacker_Function 함수를 호출하는 코드는 존재하지 않는다. 본 실험 환경에서, 각 함수의 심볼 주소와 반환 주소는 표 3을 통해 확인할 수 있다.

표 3. 각 함수별 심볼 주소 및 반환 주소
Table 3. Symbol address and return address for each function

Function	Function's symbol address	Return address
User_Task1	0xA294	0x8D55
Function_Caller	0xA26E	0xA298
Function_Callee	0xA277	0xA272
Attacker_Func	0xA28F	0x9A38

스택에 쌓인 반환 주소는 그림 3을 통해 확인할 수 있다. 그림 3은 CCS의 디버깅 모드를 통해 스택 메모리의 값을 확인하였으며, 그림 2의 (4) 지점에 중단점(breakpoint)을 설정하였다.

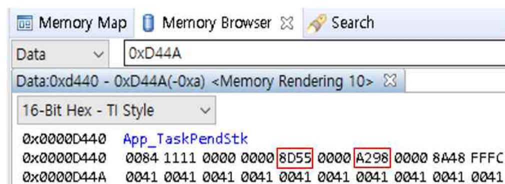

그림 3. 스택에 존재하는 함수의 반환 주소
Fig. 3. Return address of the function on the stack

그림 3을 통해 0xD444 주소에 User_Task1의 반환 주소인 0x8D55와 0xD446 주소에 Function_Caller의 반환주소인 0xA298이 스택에 쌓여 있음을 확인할 수 있다. Function_Callee의 반환 주소 0xA272는 현재 RPC 레지스터에 저장되어 있다.

Function_Callee 함수(그림 2의 (2))에서 크기가 10인 부호 있는 정수형(signed short) 배열 버퍼가 존재하며, 해당 함수의 인자(argument) 값과 (4), (5)를 통해 Function_Caller의 반환 주소 조작이 가능하다. 부호 있는 정수형 변수 x는 Function_Caller의 반환 주소가 저장된 위치를 가리키는 인덱스(index)이며, 변수 y는 Attacker_Function의 심볼 주소를 갖고 있다. 본 실험에서 BU 취약점을 통해 Function_Caller의 반환 주소가 저장된 위치에 Attacker_Function의 심볼 주소인 0xA28F로 조작 가능함을 확인하였다(그림 4 참고).

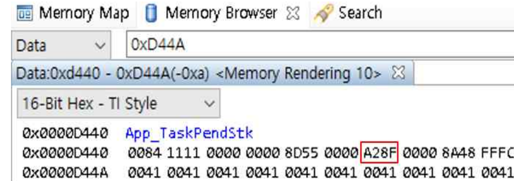

그림 4. 스택 언더플로를 통한 반환주소 조작
Fig. 4. Manipulating return address through stack underflow

Function_Caller 함수의 실행이 끝나면, RPC 레지스터는 해당 위치에서 조작된 반환 주소(0xA28F)를 적재한다. 이후, PC(Program Counter)는 RPC 레지스터에 적재된 주소를 가져와 분기한다. 결과적으로, 그림 5와 같이 RPC 레지스터가 조작된 반환 주소를 가져와 Attacker_Function 함수를 실행시킬 수 있음을 확인하였다.

www.kci.go.kr

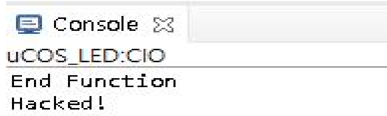

그림 5. 반환 주소 조작 결과
Fig. 5. Result of Return address manipulation

4.2 힙 기반 UaF 취약점

본 절에서는 UaF 취약점을 이용하여, 메모리에 초기화되지 않은 데이터를 읽어올 수 있는 예시를 보이고 그 과정에 대하여 설명한다. 그림 6은 UaF 취약점이 존재하는 소스코드 예시이다

uC/OS-II에서의 동적 메모리 할당은 일반적으로 메모리의 단편화(fragmentation)로 인해 단일 연속 메모리 공간을 확보하지 못하는 문제가 존재한다. 따라서 ANSI C 표준에서 사용하는 malloc, free 함수는 권장하지 않는다[15]. 이에 대한 대안으로 표 4와 같은 메모리 관리 서비스를 지원한다.

표 4. uC/OS-II 메모리 관리 서비스
Table 4. uC/OS-II memory management services

메모리 관리 함수	역할
OSMemCreate()	파티션 생성
OSMemGet()	메모리 블록 할당
OSMemPut()	메모리 블록 반환

메모리 관리 서비스를 이용하기 위해서 그림 6의 (1)과 같이 전역 변수로 선언한 2차원 배열을 OSMemCreate()를 통해 분할(partition, 파티션)로 구성하였다. 그림6의 (2)에서 메모리 블록을 할당받는 OSMemGet()을 통하여 포인터 변수 buf1은 첫 번째 메모리 블록을 할당받는다. 첫 번째 메모리 블록을 할당받은 후, (3)의 OSMemPut()을 통하여 할당 받은 메모리 블록을 반환하고, (4)에서 buf3은 반환된 메모리 블록을 uC/OS-II에 요청하여 할당받는다. 즉, buf3은 buf1이 반환했던 메모리 블록과 같은 블록을 할당받게 되므로 UaF 취약점이 발생한다.

CCS의 디버깅 모드를 통하여 그림 6의 코드를 실행시킨 후, 메모리의 데이터를 확인한 결과가 그림 7에 나타나 있다.

```
OS_MEM *partptr;
INT8U   part[3][20];-----------------(1)

static  void  App_TaskTest (void *p_arg)
{
  CPU_INT08U  err;
  INT8U *buf1;
  INT8U *buf2;
  INT8U *buf3;
  int i;
  INT8U buff1[5] = {"1','2','3','4',"5'};
  INT8U buff2[5] = {'6','7','8','9','0'};

  while (DEF_TRUE){
   buf1 = OSMemGet(partptr, &err);---(2)
   buf2 = OSMemGet(partptr, &err);

     if (buf1 != (INT8U *)0) {
        for(i=0;i<sizeof(buff1);i++){
        *(buf1+i)=buff1[i];
         }
        puts(buf1);
      }
     if (buf2 != (INT8U *)0) {
        for(i=0;i<sizeof(buff2);i++){
        *(buf2+i)=buff2[i];
         }
        puts(buf2);
      }
    err = OSMemPut(partptr, (void*)buf1);--(3)

     buf3 = OSMemGet(partptr, &err);--(4)
     if (buf3 != (INT8U *)0) {
        puts((buf3+2));           ---------(5)
      }
     break;
    }
}
```

그림 6. uC/OS-II의 메모리 관리 서비스를 이용한 메모리 블록 재할당 소스코드 예
Fig 6. An example of source code for reallocating a memory block using uC/OS-II's memory management services

www.kci.go.kr

buf1이 할당받은 메모리 블록에 데이터를 기록하고 나서 해제하였음에도 불구하고, 블록의 첫 2바이트의 1과 2 부분만 초기화되었고 나머지 데이터인 3, 4, 5는 남아 있는 것을 볼 수 있다. 위 결과를 통해 이전에 해제된 메모리 블록을 할당받을 경우, 할당된 메모리 블록의 첫 2바이트를 제외한 나머지 내용은 그대로 남아 있음을 알 수 있었다. 즉 uC/OS-II에서 이전에 해제된 메모리 블록이 다시 할당될 때, UaF 취약점이 존재함을 확인하였다.

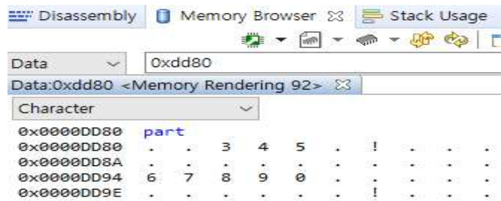

그림 7. [그림 6] 코드의 메모리 내용
Fig 7. Memory content of the code in [Fig 6]

5. 제안 기법 구현 및 실험

5.1 BU 새니타이저

그림 2의 소스코드를 컴파일하면 기계어 코드로 이루어진 COFF(Common Object File Format) 실행파일이 생성된다. 이를 TI사에서 제공하는 CG-XML(Code Generation tools XML processing scripts)을 이용하여 역어셈블한 결과는 그림 8과 같다.

XAR은 범용레지스터, SP는 스택 포인터를 의미한다. 그림 8의 (A)는 Function_Callee를 호출하는 코드이며, 0x00a277은 Function_Callee의 주소이다. (B)에서는 Function_Callee의 인자값을 스택에 저장한다. (C)는 배열 초기화 과정을 나타내며, 그림 2의 (3)에 대응한다. 0x00e2d0은 배열을 초기화하는 값인 65가 저장된 주소이다. 그림 8의 (D)에 나타난 RPT(repeat) 연산자(opcode)는 다음 명령어인 (E)의 반복 횟수를 나타낸다. 횟수는 0부터 시작하기 때문에 (E)에 해당하는 명령어를 10번 반복함을 의미한다. (E)를 통해 배열 초기화 값인 65를 부호 있는 정수형 배열 buffer에 전달한다.

```
_Function_Caller:
    MOV      AL, #0xfffc
    LCR      0x00a277           -------------(A)
    MOVL     XAR4, #0x00e200
    LCR      0x00a1fb
    LRETR

_Function_Callee:
    ADDB     SP, #12
    MOV      *-SP[11], AL       ---------(B)
    MOVZ     AR4, SP
    SUBB     XAR4, #10
    MOVZ     AR4, @AR4
    MOVL     XAR7, #0x00e2d0 ------(C)
    RPT      #9                 ---------------(D)
    ||PREAD  *XAR4++, *XAR7 ------(E)
    MOVW     DP, #0x350
    MOV      @0x3c, #0xa28f     ------(F1)
    MOVZ     AR4, SP
    SUBB     XAR4, #10
    MOVZ     AR7, AR4
    SETC     SXM                -----------(G)
    MOVL     ACC, XAR7          ---------(H1)
    ADD      ACC, *-SP[11]      ---------(H2)
    MOVL     XAR7, ACC          ---------(H3)
    MOV      AL, @0x3c          ---------(F2)
    MOV      *+XAR7[0], AL      --------(I)
    SUBB     SP, #12
    LRETR
```

그림 8. [그림 2]에 대한 어셈블리 코드
Fig. 8. Assembly code of the code in [Fig.2]

(F1)에서는 전역 변수인 y의 주소 0x3c에 Attacker_Function의 주소인 41,615(0xA28F)를 저장한다. (G)의 SXM(Sign-extension mode bit)는 부호 확장 여부를 결정하는 비트이며, SETC 연산자를 통해 활성화된다[14]. 이를 통해 그림 2의 (5)에서 버퍼의 인덱스로 음수가 들어갈 경우,

www.kci.go.kr

0xfffc의 정수형 값인 65,532(1111 1111 1111 1100)를 음수로 판단할 수 있다. 즉, 버퍼의 인덱스인 -4를 나타내기 위한 어셈블리 코드이다.

(H1)을 통해 XAR7 (부호 있는 정수형 배열의 위치, 0xD44A)을 ACC(Accumulator)에 옮긴 다음, (H2) 명령어를 통해 ACC는 Function_Caller의 반환 주소(0xA298)가 저장된 위치인 0xD446 (0xD44A+(-4)=0xD446)을 가진다. (H3) 명령은 ACC 레지스터의 값(0xD446)을 XAR7 레지스터로 전달한다. 이후 (F2)에서 ACC 레지스터의 하위 16bit인 AL에 Attacker_Function의 주소를 저장한다. 마지막으로, (I) 명령을 통해 Function_Caller의 반환 주소가 저장된 위치에 Attacker_Function의 주소를 덮어씌운다.

그림 8의 (G)는 그림 2의 부호 있는 정수형(INT16S) 변수 x를, 부호가 없는 정수형(INT16U)으로 선언하였을 경우의 어셈블리 코드에는 존재하지 않는 패턴임을 확인하였다(그림 9 참조). 배열의 인덱스를 나타내는 변수 y를 부호 없는 정수형으로 선언할 경우 버퍼 언더플로 취약점이 발생하지 않는다. 따라서 (G)패턴은 버퍼 언더플로 취약점을 이용할 경우 나타날 수 있는 패턴이다. SXM가 활성화된 이후에는 그림 8의 (H1), (H2)와 같은 MOV나 ADD 연산자(opcode)를 포함하여 SUB 연산자가 나타날 수 있으며, 이는 개발 코드에 따라 다르다. 하지만 피연산자(Operand)의 목적지 주소는 ACC 레지스터를 사용한다[14]. 따라서 BU 새니타이저가 검출하려는 패턴으로 그림 8의 (B)~(I)를 선정하였다. 의미를 정리하면 다음과 같다.

(C)~(E)는 배열에 데이터를 저장하는 패턴이다. (B),(G),(H1)~(H3)은 함수의 인자값이 음수이며, 해당 인자값을 배열의 인덱스로 활용한다. 배열이 할당된 위치의 낮은 주소 즉, 이전에 호출된 함수들의 반환 주소가 저장된 영역을 가리킬 수 있는 패턴이다.

```
_Function_Callee:
    ADDB        SP, #12
    MOV         *-SP[11], AL
    MOVZ        AR4, SP
    SUBB        XAR4, #10
    MOVZ        AR4, @AR4
    MOVL        XAR7, #0x00e2d0
    RPT         #9
    ||PREAD     *XAR4++, *XAR7
    MOVW        DP, #0x350
    MOV         @0x3c, #0xa28f
    MOVZ        AR4, SP
    MOVZ        AR0, *-SP[11]
    SUBB        XAR4, #10
    MOVZ        AR4, @AR4
    MOV         AL, @0x3c
    MOV         *+XAR4[AR0], AL
    SUBB        SP, #12
    LRETR
```

그림 9. unsigned short를 사용한 소스코드에 대한 어셈블리 코드

Fig. 9. Assembly code of the source code using unsigned short

(F1)~(F2), (I)는 할당된 버퍼보다 낮은 주소에 있는 반환 주소를 임의로 호출하려는 함수의 심볼 주소로 덮어쓰는 것을 의미한다. 따라서 BU 새니타이저의 탐지 대상 패턴을 “배열의 인덱스를 음수로 사용하여 함수의 심볼 및 데이터를 해당 인덱스 위치에 저장한다”로 정의한다.

BU 새니타이저에 의해 식별된 패턴 형태는 그림 10과 같다.

그림 10을 통해 버퍼 기반 언더플로 탐지가 가능하며, 4장에서 작성한 테스트 프로그램이 갖는 취약점과 버퍼를 활용한 유사 방식의 공격을 탐지할 수 있다.

하지만 공격자는 BU 새니타이저의 탐지를 우회하여 함수의 반환 주소를 변경할 가능성이 존재한다. 이에 제어 흐름 무결성 위배 여부를 탐지할 수 있는 기법을 추가로 제안한다. CG-XML을 이용하여 펌웨어 코드의 콜 그래프를 생성한 결과는 그림 11과 같다.

www.kci.go.kr

```
MOV       *-SP[n], AL
MOVL      XARn('general-purpose register'),
#0xhhhh(type of memory address)
RPT       #n(constant)
||PREAD   *XARn++, *XARn
MOV       @0xhhhh, #0xhhhh
SETC      SXM
MOVL      ACC, XARn
ADD|SUB|MOV      ACC, *-SP[n]
MOVL      XARn, ACC
MOV       AL,  @0xhhhh
MOV       *+XARn[0], #0xhhhh
```

그림 10. BU 새니타이저에서 탐지할 코드 패턴
Fig. 10. Code pattern to be detected by the BU sanitizer

```
_User_Task1
|  _Function_Caller
|  |  _Function_Callee
|  |  _puts
|  _OSSemPend
|  |  _OS_CPU_SR_Restore
|  |  _OS_CPU_SR_Save
|  |  _OS_EventTaskRemove
|  |  _OS_EventTaskWait
|  |  _OS_Sched
```

그림 11. 펌웨어 코드의 콜 그래프 생성
Fig. 11. Call graph generation of firmware code

또한, CCS에서 소스코드를 컴파일할 때 생성되는 맵 파일(map file)을 통해 펌웨어 코드에 존재하는 함수의 심볼 주소를 그림 12와 같이 확인할 수 있다.

```
000083de  _OS_EventTaskWait
00008462  _OS_EventTaskRemove
000085e8  _OS_Sched
00009995  _OSSemPend
0000a1fb  _puts
0000a2bb  _OS_CPU_SR_Save
0000a2c0  _OS_CPU_SR_Restore
0000a26e  _Function_Caller
0000a277  _Function_Callee
...
0000dd80  _part1       ---------(1)
0000ddc0  _OSFlagTbl ---------(2)
```

그림 12. 펌웨어 코드의 맵 파일
Fig. 12. Map file of firmware code

콜 그래프와 심볼 주소를 대응시켜, 그림 13과 같은 결과를 나타낼 수 있다. 그림 13의 정보를 활용하여 그림 8의 (F1, F2, I)를 통해 임의의 함수를 실행시키는 행위를 탐지할 수 있다. 예로, 그림 8의 (F1, F2, I)는 그림 13에서 제시한 심볼 주소가 대응된 콜 그래프에 존재하지 않는 Attacker_Function의 심볼 주소인 0xA28F를 사용한다. 따라서 정상적인 콜 그래프에 해당하는 심볼 주소가 아니므로 제어 흐름 무결성이 위배됨을 탐지할 수 있다.

```
_User_Task1
|  _Function_Caller              [0xa26e]
|  |  _Function_Callee           [0xa277]
|  |  _puts                      [0xa1fb]
|  _OSSemPend                    [0x9995]
|  |  _OS_CPU_SR_Restore         [0xa219]
|  |  _OS_CPU_SR_Save            [0xa214]
|  |  _OS_EventTaskRemove        [0x8462]
|  |  _OS_EventTaskWait          [0x83de]
|  |  _OS_Sched                  [0x85e8]
```

그림 13. 콜 그래프 함수에 심볼 주소 매핑
Fig. 13. Symbol address mapping for call graph functions

5.2 UaF 새니타이저

앞서 그림 6과 그림 7을 통하여 uC/OS-II에서 제공하는 메모리 관리 서비스에서 UaF 취약점이 존재함을 확인하였다. 즉, 그림 6의 (5)와 같이, 같은 위치에 재할당되는 블록 포인터의 위치를 임의로 조작하는 행위를 통하여 공격자는 메모리 블록에 남아있는 데이터를 얻을 수 있다.

그림 14는 그림 6의 소스코드 실행 결과이다. 그림 14의 첫 번째와 두 번째 줄의 결과를 통해 각각의 블록에 있는 데이터를 확인하였다. 하지만 세 번째 줄의 결과는 첫 번째 줄의 결과에서 사용한 블록 포인터 변수(buf1)와 다른 포인터 변수(buf3)를 사용하였음에도 불구하고 블록 포

www.kci.go.kr

인터의 위치를 임의로 조작하여, 첫 번째 블록에 초기화되지 않은 데이터를 출력한다.

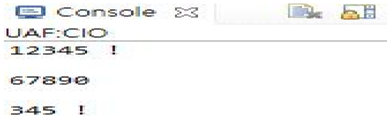

그림14. [그림 6] 소스코드 실행 결과
Fig 14. Result of executing the code of Figure 6

그림 6의 (5)를 역어셈블한 결과는 그림 15와 같다.

```
puts((buf3+2))
MOVB        ACC, #2------------------(1)
ADDL        ACC, *-SP[8]-------------(2)
MOVL        XAR4, ACC----------------(3)
LCR         0x00a2d5
```

그림 15. 초기화 되지 않은 문자열을 읽어오는 어셈블리 코드
Fig 15. Assembly code of reading a string that is not initialized

그림 15의 (1)을 통해 ACC에 상수 2를 저장하고, (2)에서 해당 스택 포인터가 가리키는 buf3의 주소와 ACC에 저장해 놓은 값인 2를 더하며, (3)에서는 (1), (2)를 통해 얻은 메모리 블록 주소를 XAR4 범용 레지스터로 읽어온다. UaF 새니타이저에서 탐지하려고 하는 패턴은 그림 16과 같으며, 해당 코드 패턴을 이용하여 UaF 취약점 탐지가 가능하다.

```
MOVB        ACC, #2      -----------(1)
ADDL        ACC, *-SP[##]
MOVL        XAR##, ACC
```

그림 16. UAF 새니타이저에서 탐지할 코드 패턴
Fig 16. A code pattern to be detected by the UaF sanitizer

그림 16의 패턴을 이용한 UaF 새니타이저는 uC/OS-II의 메모리 블록 해제 특징을 이용하여 초기화되지 않은 데이터 전체를 읽는 행위만을 탐지할 수 있다.

이 경우, 그림 16의 (1)에서 ACC에 대입되는 상수에 따라 초기화되지 않은 데이터를 부분적으로 읽어올 수 있는 취약점이 여전히 존재한다. 이러한 한계점을 극복하려는 방안은 다음과 같다. 그림 12의 맵 파일에서 파티션의 주소(그림 12의 (1), 0xdd80)와 다음 심볼의 시작 주소(그림 12의 (2), 0xddc0)의 차이를 계산하면 파티션의 크기를 파악할 수 있다. 파티션의 크기(0xddc0 - 0xdd80 = 0x40)를 이용하여 그림 16의 (1)을 "MOVB ACC, #n(n≤Partition Size)"와 같은 패턴으로 UaF 새니타이저에 적용하면 파티션 내부의 데이터를 부분적으로 읽어오는 행위를 탐지하여 앞의 한계점을 보완할 수 있다.

5.3 분석 및 논의

제안 기법을 관련 연구들과 비교 평가한 결과를 표 5로 나타내었다. 표에는 기존 새니타이저들이 지원하는 주요 기능이 나열되어 있다. 메모리 오류를 탐지하는 ASan[10]은 2배의 속도 저하(slowdown)를, 초기화되지 않은 읽기 연산을 탐지하는 MSan[12]은 3배의 성능 저하를, 스레드 간의 데이터 경쟁 여부를 탐지하는 TSan[11]은 5배~15배의 속도 저하와 메모리 사용량 오버헤드를 유발한다. 따라서 ASan, MSan, TSan 등의 기법은 저성능 실시간 임베디드 시스템에 적용하기 매우 어렵다.

Choi 등이 제안한 ECSan(Executable Code Sanitizer) 기법[13]은 표준 라이브러리 함수명과 포인터 연산의 특정 패턴만을 시그니처로 활용하기 때문에 악성 행위를 위해 공격자가 만든 사용자 정의 함수 및 코드의 경우는 탐지하지 못하는

www.kci.go.kr

한계점을 갖고 있다. 또한, 선정한 포인터 연산의 특정 패턴은 전체 어셈블리 코드에서 많이 나타날 수 있어, 오탐을 유발할 수 있다.

본 논문에서 제안한 새니타이저는 버퍼 언더플로와 Use-After-Free의 취약점을 악용한 실제 패턴을 식별 및 선정하였다. BU 새니타이저의 경우, SXM과 같은 명령어를 이용한 탐지는 일반적인 어셈블리 코드와 차별성이 있음을 확인하였다. 또한, 함수 심볼 정보와 대응된 콜 그래프를 통해 제어 흐름 무결성 위배 여부를 탐지할 수 있으므로, 공격자가 주입한 함수나 악성 코드를 임의로 실행하는 행위도 탐지 가능하다.

표 5. 제안 기법과 기존 기법 비교
Table 5. Comparison of Proposed Technique and Existing Techniques

	ASan [10]	TSan [11]	MSan [12]	ECSan [13]	Proposed technique
Out-of bounds	O	X	X	O	O
misused pointer operation	X	△	X	O	X
Uninitialized reads	X	△	O	X	△
Use after free	O	△	△	X	O
Double-free	O	X	△	X	X
Control Flow Integrity	X	X	X	△	O
Memory overhead	Medium	High	High	Low	Low
Computation overhead	Medium	High	High	Low	Low

본 새니타이저는 IDE에서 개발한 프로그램의 바이너리 코드가 PLC와 같은 임베디드 장치로 다운로드 되기 전에 해당 취약점 패턴을 탐지한다. 따라서 프로그램을 개발하는 호스트(EWS)의 테스팅 단계에서 약간의 오버헤드만을 유발한다.

결과적으로 제안기법은 임베디드 장치에 추가적인 하드웨어 모듈이 필요하지 않으며, 메모리 및 실행 시간 오버헤드가 발생하지 않아 저성능 임베디드 시스템에 적합하다.

6. 결 론

본 논문에서는 uC/OS-II를 구동하는 PLC에 다운로드 되는 펌웨어 코드에 대한 보안성 향상을 위한 코드 새니타이저를 제안하였다. 먼저, 스택 영역의 버퍼 언더플로 취약점과 힙 영역의 UaF 취약점, 그리고 이들 취약점을 이용한 악용 사례를 보였다. PLC 펌웨어가 실제 운용되기 전 개발 또는 테스트 단계에서, 이러한 취약점들을 미리 탐지하여 보안 위협을 방어할 수 있는 BU 새니타이저와 UaF 새니타이저를 구현하고 실험하였다. BU 새니타이저는 버퍼 언더플로 취약점 패턴과 제어 흐름 무결성 위배 여부를 탐지한다. UaF 새니타이저는 UaF 취약점 패턴을 탐지한다. 본 논문에서 개발한 기법은 개발자 실수로 발생할 수 있는 소프트웨어 취약점을 사전에 미리 탐지하여, PLC의 보안성을 강화할 수 있다.

제안기법은 특정 패턴을 탐지하기 때문에 알려지지 않은 취약점 및 식별된 패턴 탐지 우회를 통한 공격에 대해서는 방어할 수 없다는 한계점을 갖고 있다. 이러한 한계점을 극복하기 위해 악용 가능한 취약점 및 다양한 테스트 케이스를 발굴하고 여러 상황에서 발생할 수 있는 패턴을 기반으로 취약점 탐지 연구를 심층적으로 수행할 계획이다.

본 연구는 산업통상자원부(MOTIE)와 한국에너지기술평가원(KETEP)의 지원을 받아 수행한 연구과제임.(NO. 20171510102080)

www.kci.go.kr

참 고 문 헌

[1] K. C. Kwon and M. Lee, "Technical review on the localized digital instrumentation and control systems", Nuclear engineering and technology, vol. 41, no.4, pp.447-454, May 2009. https://doi.org/10.5516/net.2009.41.4.447

[2] S. Karanasios and D. Allen, "ICT for development in the context of the closure of Chernobyl nuclear power plant: An activity theory perspective", Information Systems Journal, vol.23, no.4, pp.287-306, February 2013. https://doi.org/10.1111/isj.12011

[3] V. M. Igure, S. A. Laughter, and R. D. Williams, "Security issues in SCADA networks", computers & security, vol.25, no.7, pp.498-506, October 2006. https://doi.org/10.1016/j.cose.2006.03.001

[4] N. Falliere, L. O. Murchu, and E. Chien, "W32.Stuxnet Dossier (Version 1.4)", White paper, Symantec Security Response, February 2011. https://www.wired.com/images_blogs/threatlevel/2011/02/Symantec-Stuxnet-Update-Feb-2011.pdf

[5] E. Chien, L. O. Murchu, and N. Falliere, "W32. Duqu: the precursor to the next stuxnet", Proceedings of the 5th USENIX Workshop on Large-Scale Exploits and Emergent Threats, 25-27; SAN JOSE, US, April 2012. https://www.usenix.org/conference/leet12/workshop-program/presentation/chien

[6] C. Wueest, "Targeted attacks against the energy sector (Version 1.0)", White paper, Symantec Security Response, January 2014. https://www.symantec.com/content/dam/symantec/docs/security-center/white-papers/targeted-attacks-against-engery-sector-14-en.pdf

[7] A. Sajid, H. Abbas, and K. Saleem, "Cloud-assisted IoT-based SCADA systems security: A review of the state of the art and future challenges", IEEE Access, vol.4, pp.1375-1384, March 2016. https://doi.org/10.1109/access.2016. 2549047

[8] Z. Basnight, J. Butts, J. Lopez Jr, and T. Dube, "Firmware modification attacks on programmable logic controllers", International Journal of Critical Infrastructure Protection, vol.6, no.2, pp. 76-84, June 2013. https://doi.org/10.1016/j.ijcip.2013.04.004

[9] M. Payer, Software Security - Principles, Policies, and Protection, Free ebook, pp. 56-58, April 2019. https://nebelwelt.net/SS3P/softsec.pdf

[10] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A Fast Address Sanity Checker", Proceedings of the 2012 USENIX Annual Technical Conference, 13-15; Boston, US, June 2012. https://dl.acm.org/doi/10.5555/2342821.2342849

[11] K. Serebryany and T. Iskhodzhanov, ThreadSanitizer: data race detection in practice, Proceedings of the workshop on binary instrumentation and applications, 62-71; New York, US, December 2009. https://doi.org/10.1145/1791194.1791203

[12] E. Stepanov and K. Serebryany, "MemorySanitizer: fast detector of uninitialized memory use in C++", 2015 IEEE/ACM International Symposium on Code Generation and Optimization(CGO), 7-11; San Francisco, US, February 2015. https://doi.org/10.1109/cgo.2015.7054186

[13] 최광준, 유근하, 조성제, "PLC용 uC/OS 운영체제의 보안성 강화를 위한 실행코드 새니타이저", 정보과학회논문지, vol.29, no.2, pp.365-375, April 2019. https://doi.org/10.13089/JKIISC.2019.29.2.365

[14] Texas Instruments, TMS320C28x CPU and Instruction Set - Reference Guide, Literature Number: SPRU430F, Texas Instruments Incorporated, April 2015.

www.kci.go.kr

http://www.ti.com/lit/ug/spru430f/spru430f.pdf?&ts=1590046187628
[15] https://doc.micrium.com/display/osiidoc/Memory+Management/, February 28, 2020.

저 자 소 개

한승재(Seungjae Han)

2019.2 단국대학교 소프트웨어학과 졸업
2019.3-현재 단국대학교 컴퓨터학과 석사
<주관심분야> 시스템 보안, 임베디드 보안

이건용(Keonyong Lee)

2015.3-현재 단국대학교 응용컴퓨터공학과 학사과정 재학
<주관심분야> 시스템 보안, 임베디드 보안

유근하(Geunha You)

2019.2 단국대학교 소프트웨어학과 졸업
2019.3-현재 단국대학교 컴퓨터학과 석사
<주관심분야> 시스템 보안, 임베디드 보안

조성제(Seong-je Cho)

1989년 서울대학교 컴퓨터공학과 공학사
1991년 서울대학교 컴퓨터공학과 공학석사
1996년 서울대학교 컴퓨터공학과 공학박사
2001년 미국 University of California, Irvine 객원 연구원
2009년 미국 University of Cincinnati 객원 연구원
1997년 3월~현재 단국대학교 소프트웨어학과/컴퓨터학과 교수
<주관심분야> 컴퓨터보안, 소프트웨어 지적재산권 보호, 시스템 소프트웨어, 스마트폰 보안 등

www.kci.go.kr