Journal of The Korea Institute of Information Security & Cryptology ISSN 1598-3986(Print)
VOL.29, NO.2, Apr. 2019 ISSN 2288-2715(Online)
https://doi.org/10.13089/JKIISC.2019.29.2.365

PLC용 uC/OS 운영체제의 보안성 강화를 위한 실행코드 새니타이저*

최 광 준,[†] 유 근 하, 조 성 제[‡]
단국대학교

Executable Code Sanitizer to Strengthen Security of uC/OS Operating System for PLC*

Gwang-jun Choi,[†] Geun-ha You, Seong-je Cho[‡]
Dankook University

요 약

PLC(Programmable Logic Controller)는 안전 지향 제어시스템(safety-critical control system)을 위한 실시간 임베디드 제어 애플리케이션들을 지원하는 고신뢰성의 산업용 디지털 컴퓨터이다. PLC의 실시간 제약조건을 만족시키기 위하여 uC/OS 등의 실시간 운영체제들이 구동되고 있다. PLC들이 산업제어 시스템 등에 널리 보급되고 인터넷에 연결됨에 따라, PLC 시스템을 대상으로 한 사이버 공격들이 증대되고 있다. 본 논문에서는, 통합 개발 환경(IDE)에서 개발된 프로그램이 PLC로 다운로드 되기 전에 실행 코드를 분석하여 취약성을 완화시켜주는 "실행코드 새니타이저(sanitizer)"를 제안한다. 제안기법은, PLC 프로그램 개발 중에 포함되는 취약한 함수들과 잘못된 메모리 참조를 탐지한다. 이를 위해 취약한 함수 DB 및 이상 포인터 연산과 관련된 코드 패턴들의 DB를 관리한다. 이들 DB를 기반으로, 대상 실행 코드 상에 취약 함수들의 포함 여부 및 포인터 변수의 이상 사용 패턴을 탐지·제거한다. 제안 기법을 구현하고 실험을 통해 그 유효성을 검증하였다.

ABSTRACT

A PLC (Programmable Logic Controller) is a highly-reliable industrial digital computer which supports real-time embedded control applications for safety-critical control systems. Real-time operating systems such as uC/OS have been used for PLCs and must meet real-time constraints. As PLCs have been widely used for industrial control systems and connected to the Internet, they have been becoming a main target of cyberattacks. In this paper, we propose an execution code sanitizer to enhance the security of PLC systems. The proposed sanitizer analyzes PLC programs developed by an IDE before downloading the program to a target PLC, and mitigates security vulnerabilities of the program. Our sanitizer can detect vulnerable function calls and illegal memory accesses in development of PLC programs using a database of vulnerable functions as well as the other database of code patterns related to pointer misuses. Based on these DBs, it detects and removes abnormal use patterns of pointer variables and existence of vulnerable functions shown in the call graph of the target executable code. We have implemented the proposed technique and verified its effectiveness through experiments.

Keywords: Execution code sanitizer, Programmable logic controller, uC/OS, Vulnerable function, Pointer misuse

Received(01. 14. 2019), Modified(04. 10. 2019), Accepted(04. 10. 2019)

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

† 주저자, hwancheon@gmail.com
‡ 교신저자, sjcho@dankook.ac.kr(Corresponding author)

I. 서 론

PLC (Programmable Logic Controller)는 실시간 요구조건을 가지며 높은 신뢰성을 요구하는 제어 공정(control process)에 사용되는 컴퓨터 기반의 단일 프로세서 장치이다[1]. PLC는 산업 자동 시스템의 주요 부분으로서, 대표적으로 공정제어, SCADA(Supervisory Control and Data Acquisition) 시스템, 원자력 발전소 등 산업 제어 시스템 등에 적용되고 있다. 표준 ICS3-1978에 기술되어 있는 정의에 따르면 PLC는 "디지털 또는 아날로그 입출력 모듈을 통해 로직(logic), 시퀀싱(sequencing), 타이밍, 카운팅, 연산과 같은 특수한 기능을 수행하기 위해 프로그램이 가능한 메모리를 사용하고 다양한 종류의 기계나 프로세서를 제어하는 디지털 동작 수행 전자 장치"를 의미한다. 프로그래밍이 가능한 PLC가 기존의 전자회로로 구성된(hard-wired) 제어장치에 비해 더 높은 유연성을 제공함에 따라, 산업 시스템의 로직을 실행하는 대부분의 제어 요소(control element)들이 PLC로 대체되고 있다. 산업 제어 시스템에서 PLC는 입출력 장치(예: 밸브, 스위치, 센서) 또는 다른 PLC에 연결되어, 공정에 따른 입력과 출력을 모니터링한다.

최근 IoT 기술의 발전에 따라 PLC는 모터 제어, 펌핑 시스템(pumping system), 시스템 모니터링 등과 같은 애플리케이션에도 널리 사용되고 있다. Siemens S7 시리즈를 포함한 최근의 PLC제품들은 인터넷으로 연결되어 원격으로 접근될 수도 있다. 인터넷에 연결된 PLC가 산업 제어 시스템 등의 주요 하부구조(critical infrastructures)에서 중요한 역할을 수행함에 따라 점차적으로 사이버 공격의 주요 대상이 되고 있다[2, 3, 4, 5]. 2010년 이란 원자력시설 우라늄 원심분리기 가동을 중단시킨 스턱스넷(Stuxnet)[6]을 시작으로 2011년 제어시스템 정보 수집 및 유출을 목적으로 한 두큐(Duqu)[7], 2012 년 중동 국가의 전력제어시스템을 손상시킨 플레이머(Flamer)[8] 등의 악성코드가 그 대표적인 사례이다. 스턱스넷의 경우 원전 원심분리기를 제어하는 PLC의 통합 관리 도구를 대상으로 한 공격이었으며, 이를 통해 악성 명령을 PLC에 전달하였다[6].

PLC용 운영체제로는 VxWorks, Microware OS-9, QNX Neutrino, Linux, Windows CE, uC/OS 등의 임베디드 실시간 운영체제(Real-Time Operating System: RTOS)를 사용한다[9, 10]. 그러나 [9]와 [10]의 연구에서 볼 수 있듯이 PLC 운영체제들이 취약점을 가지고 있으며, 이러한 취약점을 제거 또는 완화할 필요가 있다. 산업 제어 시스템의 PLC를 대상으로 한 사이버 공격으로 인한 산업시설의 가동중단 및 오작동은 공공의 안전 및 환경을 위협하는 심각한 결과를 야기할 수 있다. 따라서 제어시스템에서 PLC 자체에 존재하는 취약점뿐만 아니라 이를 관리하는 도구에서 PLC로 전송하는 명령 또는 다운로드하는 프로그램에 대한 보안성을 강화하는 연구가 수행되어야 한다.

본 논문에서는 uC/OS 기반의 PLC를 대상으로 개발된 실행 코드를 분석하여 PLC의 오작동 및 취약성을 사전에 탐지및 필터링하는 "실행코드 새니타이저(Executable Code Sanitizer)"를 설계하고 구현한다. Micrium의 uC/OS는 소형 마이크로 커널 RTOS이다. uC/OS는 마이크로프로세서용 우선순위 기반의 멀티태스킹 운영체제로, 주로 C 프로그래밍 언어로 작성되었다. 현재, uC/OS는 PLC 또는 산업제어시스템의 임베디드 운영체체로 사용되고 있다[11, 12].

본 논문에서 제안하는 실행코드 새니타이저는 통합개발환경(IDE)에서 컴파일된 기계어 코드(machine code)를 PLC로 다운로드하기 전에 정적으로 분석하여 취약한 함수나 포인터의 사용을 식별하여 제거할 수 있도록 한다. 실행코드 새니타이저는 "함수 새니타이저(Function sanitizer)"와 "포인터 새니타이저(Pointer sanitizer)" 2개의 모듈로 구성된다. 함수 새니타이저 모듈은 실행 코드로부터 호출 그래프(call graph)를 생성하고 PLC에 악영향을 줄 수 있는 표준 라이브러리 함수나 코딩규칙 매뉴얼에서 허용하지 않는 함수들을 탐지하여 보안성을 강화한다. 포인터 새니타이저 모듈은 실행코드를 역어셈블(disassemble)한 후 포인터 변수에 대한 이상 연산(anomaly operation) 패턴을 탐지하고, 소스코드와 심볼 정보를 이용하여 보안 및 안전에 위협이 될 수 있는 메모리 값 설정이나 변경을 탐지한다.

본 논문의 구성은 다음과 같다. 2장에서는 관련연구에 대해 기술한다. 3장에서는 제안기법에 대해 상세히 설명하며, 4장에서는 실험을 통해 제안기법의 유용성과 타당성을 보인다. 5장에서는 결론 및 향후 연구에 대해 기술한다.

II. 관련 연구

2.1 Address sanitization 기법

AddressSanitizer(ASan) 프레임워크[13]는 개발자가 작성한 소스코드와 함께 주요 메모리 영역을 표시하는 stub코드를 넣고 컴파일하여 동적으로 프로그램의 취약성을 점검하는 프레임워크이다. ASan 프레임워크의 핵심 프로세스는 소스코드를 컴파일 할 때 컴파일러 레벨에서 중요 메모리 구역을 표시하는 stub 코드를 삽입하는 것이다. 컴파일된 프로그램은 실행 시에 Memory shadowing과 Redzone 기법을 통해 메모리 위/변조를 탐지 및 차단하는 방식으로 동작한다. 이 기법을 통해 메모리의 주요 영역(heap, stack, 전역변수)의 잘못된 접근 및 use-after-free, use-after-return 등의 주요 메모리 공격에 대해 공격의 원인을 파악하고 잘못된 접근을 차단한다.

ASan을 이용하여 10개월 동안의 테스팅 단계에서 Chromium 브라우저 및 3rd-파티 라이브러리에서 300개 이상의 버그를 탐지하였다. ASan 기법을 64-비트 리눅스에서 SPEC CPU2006 벤치마크를 사용하여 성능을 평가한 결과, CPU2006에서 평균 73%의 속도저하(slowdown)가 발생하였고, 인스트루멘테이션(instrumentation)으로 인해 메모리 사용량은 평균 3.37배 증가하였다.

이 기법은 모든 메모리 영역을 관리하는 것이 아니기 때문에 제시한 메모리 영역 외에서 발생하는 취약성에 대해서는 한계점이 존재한다. 또한 프로세스의 메모리 영역을 shadow memory 및 Redzone 기법을 통해 관리하여, ulimit이나 정적 라이브러리 사용 등의 프로세스 관리 기법을 사용할 수 없다. 이러한 한계점으로 인해 ASan 기법은 저전력 또는 low-end 시스템을 지향하는 임베디드 환경에서는 적합하지 않다. 본 논문에서는 임베디드 환경에 적합한 실행코드 새니타이저 기법을 제안하며, 실행 시 추가 오버헤드도 발생하지 않는다.

2.2 임베디드 실행파일의 코딩표준 준수 여부 분석

Venkitaraman 등[14]은 임베디드 환경의 실행파일이 어셈블리 레벨에서 코딩 표준을 준수(compliance with coding standards)했는지 검사하는 연구하였다. TI XDAIS (Texas Instruments' Express DSP Algorithm Interoperability Standard)는 DSP 프로세서용 TI TMS320 패밀리를 대상으로 DSP 코드를 위한 요구사항들을 정의한다. 이는 3rd-파티 DSP 소프트웨어 벤더의 의해 개발된 프로그램 코드가 준수해야 하는 표준을 명시하고 있으며, 그 표준에는 34개의 규칙과 15개의 가이드라인이 정의되어 있다.

Venkitaraman 등[14]은 임베디드 실행파일을 역어셈블(disassemble)하여 함수 단위의 흐름 그래프(flow-graph based on function)로 표현하고, 작성된 흐름 그래프를 이용하여 하드 코딩된 포인터(hard-coded pointers)가 존재하는지 검토한다. 하드 코딩된 포인터는 TI XDAIS 표준의 3번째 규칙에 해당된다. 하드 코딩된 포인터 변수는 소스 코드에서 고정된 메모리 주소를 할당한 변수로 소스코드 상에서 메모리에 직접 접근할 수 있는데, 대부분의 저성능 임베디드 장치 또는 uC/OS 운영체제는 단일 주소체계로 구성되어 있기 때문에 이러한 포인터변수는 Pointer alias 등의 안전성 또는 보안에 취약한 문제를 일으킬 수 있다. 일반적으로 임베디드 표준 코딩 규칙 가이드에 따르면, 이러한 포인터 변수의 사용은 지양되어야 한다[15, 19].

코딩표준 준수 여부 확인 기법은 코딩 표준규칙에 대한 것으로, 보안 측면보다는 안전(safety) 측면에 중점을 두고 있다. 이에 반해 본 논문은 안전 측면보다는 보안 취약점 완화에 중점을 둔 연구를 수행한다. 또한, 하드 코딩된 포인터뿐만 아니라 버퍼 오버플로 취약점도 고려한다.

2.3 uC/OS 운영체제 방어 기법

Sikiligiri의 연구[16]는 Altera Nios II softcore 프로세서 및 uC/OS 운영체제에서 코드 주입 공격(code injection attack), 즉 스택 기반 버퍼 오버플로 공격이 가능함을 보이면서, 그 공격을 방어할 수 있는 기법을 제시하였다. 코드 주입 공격을 방어하는 기법 중의 하나는, 스택 영역에 주입된 코드가 실행되지 못하도록 스택 영역을 실행 불가능한 공간으로 설정하는 것이다. 그러나 이 방어 기법은 가상 메모리 지원 하에서 가능한 방법으로, 가상 메모리를 지원하지 않는 uC/OS에서는 그대로 적용할 수 없다.

[16]에서는 함수 콜 스택을 위해 특정 전용 SRAM 메모리를 사용하여 버퍼 오버플로 공격을 방

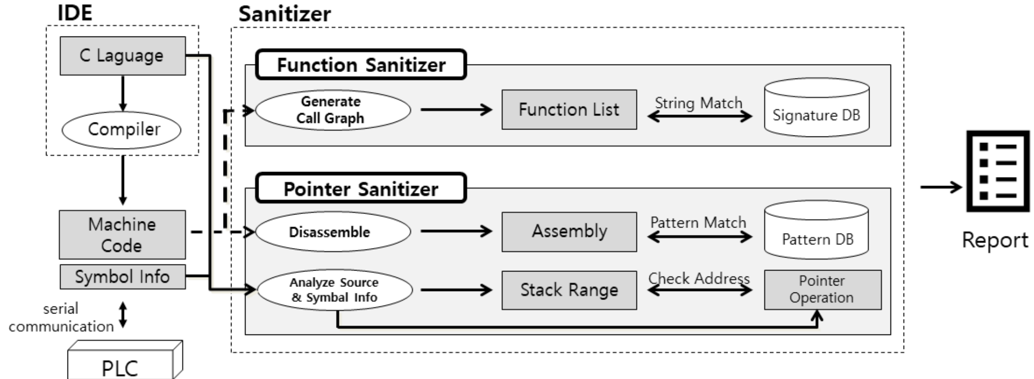


Fig. 1. Overview of the proposed sanitizer

어하는 하드웨어 기반 기법을 제안하였다. 그 전용 SRAM 메모리를 '읽기/쓰기'만 가능한 영역으로 지정하여, 공격자가 스택에 공격 코드를 주입하더라도 공격 코드를 명령 버스(instruction bus)가 접근할 수 없게 구현하였다.

하지만 [16]에서 제안한 기법을 사용하기 위해서는 함수 콜 스택을 저장하기 위한 별도의 하드웨어 지원이 필요하다는 제약조건이 있다. 이에 반해, 본 논문에서는 별도의 하드웨어 지원을 요구하지 않으며, 또한 버퍼 오버플로 공격 외에 포인터 연산 관련 취약점도 고려한다.

III. 제안 기법

본 논문에서 제안하는 "실행코드 새니타이저(Executable Code Sanitizer)"는 기계어 코드(machine code, 실행코드)가 PLC로 다운로드 되기 전, 바이너리 파일을 정적으로 분석하여 PLC 프로그램의 보안성을 강화하는 방안을 제시한다. Fig.1.과 같이 제안기법은 함수 새니타이저(Function Sanitizer)와 포인터 새니타이저(Pointer Sanitizer) 2개의 모듈로 구성된다. 함수 새니타이저 모듈은 기계어 코드로부터 콜 그래프(call graph)를 생성하고, PLC 보안에 악영향을 줄 수 있는 라이브러리 함수나 코딩규칙 매뉴얼에서 허용하지 않는 취약한 함수를 탐지하여 보안성을 강화한다. 포인터 새니타이저 모듈은 기계어 코드를 역어셈블하여 포인터 변수의 이상 연산(anomaly operation) 패턴을 탐지하고, 추가적으로 소스코드와 심볼정보를 이용하여 불필요한 메모리 값 변경을 탐지한다.

3.1 함수 새니타이저

함수 새니타이저는 분석 대상 기계어 코드로부터 콜 그래프를 생성하고, 생성된 콜 그래프로부터 코드에 사용된 함수 명 및 함수 간 호출 관계를 파악한다. 즉, 콜 그래프는 실시간 태스크들이 실제 호출하는 함수들의 호출 관계를 그래프 형태로 나타내 주므로, 코드 내에 사용된 취약한 함수를 탐지하는데 매우 용이하다. 본 제안기법은 사용자가 개발한 실제 동작에 관련 된 태스크의 함수 리스트를 시그니처 DB(signature database)와 비교하여 안정성을 검토한다. 시그니처 DB는 개발 매뉴얼에서 허용하지 않는 표준 함수나 보안 관점에서 취약한 표준 함수 명을 시그니처로 관리한다.

Table 1.은 보안 및 안전에 악영향을 줄 수 있는 함수들의 리스트로 "Buffer Overflow 취약 함수"와 "Memory 취약성 관련 함수"로 구성되어 있다. [17, 18, 19]에 따르면, Buffer Overflow에 취약한 함수로 'strcat()', 'strcpy()', 'strncpy()', 'sprintf()', 'vsprintf()', 'gets()', 'fscanf()', 'scanf()'를 제시하고 있다. 또한, memory 조작과 관련된 함수로 'calloc()', 'free()', 'malloc()', 'realloc()', 'memcpy()', 'memset()', 'memmove()'를 제시하고 있다.)

함수 새니타이저를 통해 취약한 함수 사용을 탐지한 이후, 콜 그래프를 통해 어느 위치에 어떤 함수가 사용되었는지 파악할 수 있다. 따라서 함수 새니타이저 결과와 개발자 소스코드를 손쉽게 비교할 수 있으

Table 1. Vulnerable Functions in safety and security

Functions vulnerable to buffer overflow	Memory-related functions
strcat, strcpy, strncpy, sprintf, vsprintf, gets, fscanf, scanf	calloc, free, malloc, realloc, memcpy, memset, memmove

며, 개발자의 실수 혹은 악의적인 행위로 불필요한 함수가 포함되었는지 또는 IDE 자체의 결함으로 인해 소스코드에 존재하지 않은 함수가 포함되어있는지 판단할 수 있는 근거를 제공할 수 있다.

3.2 포인터 새니타이저

임베디드 RTOS인 uC/OS의 경우 단일 주소 공간(single address space)으로 구성되어 있어, 포인터 변수에 직접 주소를 입력하고 포인터 연산을 통해 데이터 영역의 메모리 주소 값을 변조할 수 있다. 이와 같은 행위는 일반적으로 임베디드 표준 코딩 규칙[15, 19, 20]이나 PLC 프로그램 매뉴얼에서 제한하고 있는 사항이다. 본 논문에서 제시하는 포인터 새니타이저는 먼저 기계어 코드를 대상으로 수행하고, 필요한 경우 소스 코드 분석을 통해 필요한 정보를 추출하여 수행한다.

포인터 새니타이저에서는 포인터 변수에 직접 주소를 하드코딩하여 할당하는 명령 패턴을 탐지한다. 기계어 코드를 사용하여 포인터 변수의 이상 연산(anomaly operation) 여부를 탐지하기 위해서는 기계어 코드에서 포인터 변수의 사용을 확인할 필요가 있다. 이를 위하여 바이너리 코드를 역어셈블하여, 어셈블리 수준에서 포인터 변수의 이상 사용 패턴을 확인한다. 추가적으로, 포인터 새니타이저의 유효성과 정확도를 개선하기기 위해 소스코드와 심볼 정보를 통해 각 태스크의 스택의 범위를 계산하고 각 태스크에 할당된 주소 범위를 벗어난 포인터 연산을 탐지하는 기법도 제안하여 적용한다.

패턴 DB는 포인터 변수 연산을 탐지하는데 필요한 코드 패턴을 관리한다. 패턴 DB에 유지할 시그니처를 생성하기 위해 본 기법을 적용하려는 PLC의 CPU용 어셈블리 사용 패턴을 분석한다. 본 제안기법에서 탐지하고자 하는 패턴은 포인터 변수에 하드코딩된 주소 값을 할당하는 명령(들)과 포인터 변수가 가리키는 주소번지의 값을 변경하는 명령(들)으로 구성된다. 즉, 포인터 새니타이저는 어셈블리(또는 기계어) 코드들의 조합으로 이루어진 패턴을 사용한다. 어셈블리 코드만을 사용해서는 하드코딩된 메모리 주소 값을 식별하는 것이 쉽지 않은 경우에는 소스코드를 분석하여 하드코딩된 메모리 주소 값을 식별한다.

포인터 새니타이저는 uC/OS 소스코드와 심볼 정보를 사용하여 태스크들의 스택 위치와 크기를 계산한다. 이후 태스크의 소스코드 내부에 포인터 변수가 있는지를 확인하고, 포인터 변수에 메모리 주소를 하드코딩 하는지 확인한다. 이후 포인터 변수에 입력되는 주소 값이 해당 태스크의 스택 범위를 벗어나는 경우 이를 탐지한다.

IV. 실험 및 구현

4.1 실험 환경

본 논문에서 제안한 기법을 검증하기 위한 실험은 TI(Texas Instrument)사의 TMS320C2000계열 DSP인 TMS320F28335가 장착된 실험용 보드와 개발 IDE로 TI사에서 제공하는 IDE인 CCS 6.0(Code Composer Studio)[20]을 사용하였다.

4.2 실험을 위한 테스트 프로그램 작성

4.2.1 보안에 취약한 표준함수가 포함된 프로그램

본 논문의 제안기법을 실험하기 위해, 'strcpy()' 함수를 포함하는 간단한 소스코드를 작성하였고, 정상적으로 컴파일 되어 기계어 코드가 생성되어 실험용 보드에 다운로드 하였다. 'strcpy()'는 버퍼 오버플로 취약점을 갖는 대표적인 함수로 보안문제 때문에 시큐어 코딩 관점에서도 사용을 금하고 있다. 아래 Fig.2.는 'strcpy()' 함수가 포함 된 직접 작성한 uC/OS 소스코드 중 일부를 보여준다.

Fig.2.에서 버퍼 오버플로의 위험이 존재하는 부분은 (1) vul_func 함수 내부의 strcpy() 함수이다. (2)부분에서 선언 된 문자열배열 보다 긴 문자열이 strcpy()함수의 두 번째 인자로 들어갈 경우 버퍼 오버플로가 발생하여 개발자가 의도하지 않은 동작이 일어날 수 있다. (1)의 vul_func 함수를 호출하는 (3)부분에서 함수 인자로 매우 긴 문자열을 입력하여 서비스 거부(Denial of Service) 공격이

```
#include <string.h>
void vul_func(char *a)          ------------- (1)
{
          char buf[10];       -------------- (2)
          strcpy(buf, a);
          return;
}
static void App_Task1 (void *p_arg)
{
    while (1) {
          vul_func("Input"); ------------- (3)
        OSTimeDlyHMSM(0, 0, 0, 300);
    }
}
int main(void)
{
   OSTaskCreateExt(APP_Task1, ... );
   OSStart();
   return 0;
}
```

Fig. 2. Example source codes for uC/OS with a strcpy function call

```
void print_normal()  ------------ (1)
{
          printf("Normal\n");
          return;
}
void print_hacked()  ------------ (2)
{
          printf("Hacked!\n");
          return;
}
static  void  App_Task1 (void *p_arg)
{
   void (*a)();         ------------ (3)
   a = print_normal;
   while (1) {
        a();
        OSTimeDlyHMSM(0, 0, 0, 300);
    }
}
static  void  App_Task2 (void *p_arg)
{
   int *ptr;            ------------ (4)
   ptr = 0xe640;  /* Hard-coded a's address
in App_Task1 */
   *ptr = 0xb3c0; /* Change a's value of
App_Task1 into the address of print_hacked()
*/
    while (1) {
        OSTimeDlyHMSM(0, 0, 0, 300);
    }
}
```

Fig. 3. Example source codes with pointer manipulation

가능하다.

4.2.2 포인터 연산이 포함된 프로그램

uC/OS의 경우 단일 주소 공간 체계이므로, 포인터 변수를 통해 임의 데이터 영역의 메모리 값을 손쉽게 변조 가능하다. 예로, 실험에서 하나의 태스크(App_Task2)가 다른 태스크(App_Task1)의 함수 포인터 변수를 조작하는 소스코드를 작성하였다(Fig.3. 참조). 이러한 상황은 임베디드 표준 코딩 규칙에서 금지하는 연산이며, MS Windows나 Linux와 같은 범용 운영체제에서는 허용되지 않는 연산이다.

일반적으로 하나의 태스크가 다른 태스크의 변수를 읽기/쓰기를 수행하지 말아야하지만, 포인터 변수(Fig.3.의 (4))에 다른 태스크에서 선언된 함수 포인터 변수(Fig.3.의 (3))의 주소를 하드코딩하여 조작함으로써, 최종적으로 함수 포인터 변수 a의 값을 print_hacked()로 변조시킬 수 있다. Fig.4.는 다른 태스크에 의해 변조된 함수 포인터 변수의 호출 결과를 보여준다. 이를 통해 함수포인터의 값을 변조하여 의도하지 않은 함수를 호출하거나 임의 영역의 메모리 값을 변조하는 공격이 가능하다.

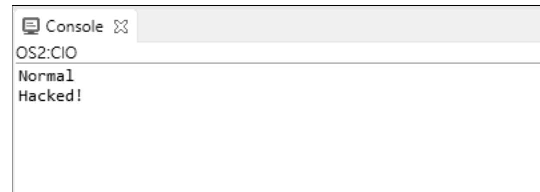
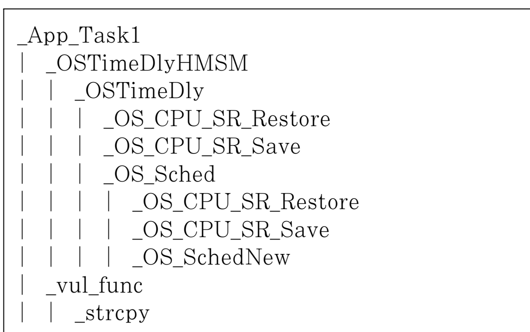


Fig. 4. Memory hacking result by pointer manipulation

4.3 새니타이저의 구현

4.3.1 함수 새니타이저

함수 새니타이저 구현을 위해 콜 그래프를 생성하는데, 콜 그래프는 태스크가 동작하면서 호출하는 모든 함수의 이름과 호출 관계를 Fig.5.와 같은 형태로 보여준다. 따라서 함수 새니타이저는 취약한 함수 사용 여부를 시그니처 DB와 비교하여 확인할 수 있으며, 이를 바탕으로 코드 분석도 수행한다. 본 실험에서는 COFF 실행파일 포맷의 기계어 코드를 정적



```
_App_Task1
| _OSTimeDlyHMSM
| | _OSTimeDly
| | | _OS_CPU_SR_Restore
| | | _OS_CPU_SR_Save
| | | _OS_Sched
| | | | _OS_CPU_SR_Restore
| | | | _OS_CPU_SR_Save
| | | | _OS_SchedNew
| _vul_func
| | _strcpy
```

Fig. 5. A call graph of a uC/OS task

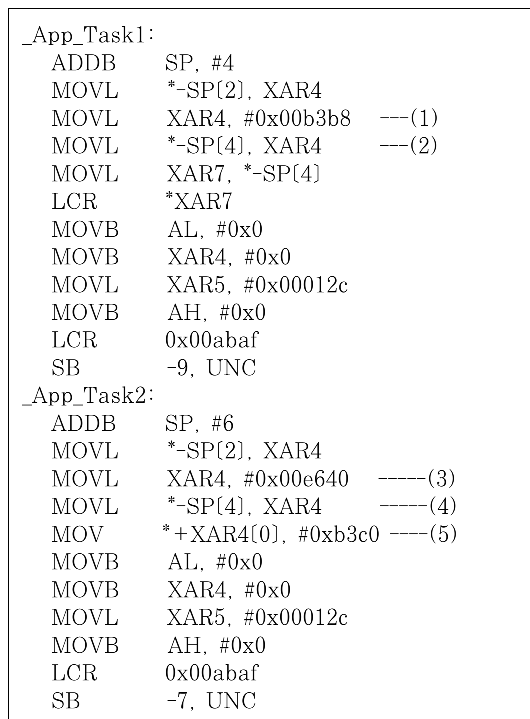
으로 분석하여 함수 호출 그래프를 출력해주는 TI사 도구인 CG-XML (code generation tools XML processing scripts)을 사용했다.

Fig.5.는 Fig.2.의 소스코드로부터 생성된 기계어 코드에 대한 함수 콜 그래프 중 태스크의 호출 순서를 보여준다. Fig.5.를 확인하면 태스크 수행 시에 호출하는 모든 함수명을 추출할 수 있다. 함수 새니타이저는 시그니처 DB와 추출한 함수명들을 문자열 비교하여 버퍼 오버플로에 취약한 'strcpy()' 함수가 태스크에 포함 되어있는 것을 탐지하여 사용자에게 경고해준다.

4.3.2 포인터 새니타이저

포인터 새니타이저를 구현하기 위해 기계어 코드를 역어셈블하여 어셈블리 코드를 분석한다. Fig.6.은 Fig.3.의 태스크 소스코드에 대응되는 어셈블리 코드이다. XAR은 범용레지스터, SP는 스택포인터를 뜻한다. Fig.6.의 (1)과 (2)는 함수포인터 변수 a에 함수명으로 주소를 할당하는 것을 나타낸다. (3)과 (4)는 포인터 변수 ptr에 App_Task1의 변수 a의 주소를 하드코딩을 하는 것을 나타낸다. (5)는 포인터 변수 ptr이 가리키는 메모리 주소의 값을 print_hacked()로 변경하는 것을 나타낸다.

Fig.6.에 나타난 (3)과 (4)의 어셈블리 코드가 포인터 새니타이저에서 검출하려는 패턴이다. 그러나 (3)과 (4)의 어셈블리 코드와 유사한 패턴은 전체 어셈블리 코드에서 많이 탐지되어 포인터 새니타이징을 위한 패턴으로 적절하지 않다. 예를 들어 (1)과 (2)의 경우 포인터 변수에 하드코딩을 하는 소스코드는 아니지만 (3)과 (4)의 어셈블리 코드와 유사함을 확인할 수 있다. 따라서, 포인터 변수가 가리키는

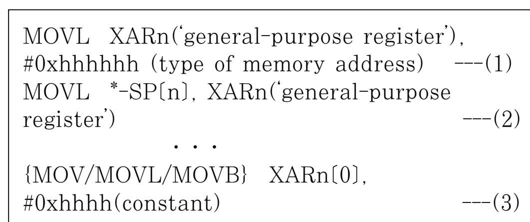


```
_App_Task1:
   ADDB     SP, #4
   MOVL     *-SP[2], XAR4
   MOVL     XAR4, #0x00b3b8   ---(1)
   MOVL     *-SP[4], XAR4     ---(2)
   MOVL     XAR7, *-SP[4]
   LCR      *XAR7
   MOVB     AL, #0x0
   MOVB     XAR4, #0x0
   MOVL     XAR5, #0x00012c
   MOVB     AH, #0x0
   LCR      0x00abaf
   SB       -9, UNC
_App_Task2:
   ADDB     SP, #6
   MOVL     *-SP[2], XAR4
   MOVL     XAR4, #0x00e640   -----(3)
   MOVL     *-SP[4], XAR4     -----(4)
   MOV      *+XAR4[0], #0xb3c0 ----(5)
   MOVB     AL, #0x0
   MOVB     XAR4, #0x0
   MOVL     XAR5, #0x00012c
   MOVB     AH, #0x0
   LCR      0x00abaf
   SB       -7, UNC
```

Fig. 6. Assembly codes for Fig.3

메모리 주소의 값을 변경하는 (5)의 어셈블리 코드를 추가하여, (3)~(5)의 패턴을 포인터 새니타이저의 탐지 패턴 DB에 추가하였다. 이 패턴을 이용한 실험에서 해당 코드 패턴을를 탐지하는 것을 확인하였다.

Fig.7.은 포인터 변수를 사용한 메모리 값을 변조하는 패턴을 나타낸 것이다. Fig.7.의 (2)와 (3)사이 있는 빈 줄(empty line)은, 포인터 변수에 메모리 주소를 할당하고 나서 포인터 변수가 가리키는 메모리 주소 값을 변조하기 전에, 임의의 어셈블리 코드가 수행되는 상황도 고려하고 있음을 보인다. (2)번 명령 다음에 메모리 값을 변조하는 시점에 따라



```
MOVL  XARn('general-purpose register'),
#0xhhhhhh (type of memory address)   ---(1)
MOVL  *-SP[n], XARn('general-purpose
register')                            ---(2)
          . . .
{MOV/MOVL/MOVB}  XARn[0],
#0xhhhh(constant)                     ---(3)
```

Fig. 7. Example code pattern for the pointer sanitizer

다른 어셈블리 코드가 삽입될 수 있다. (3)의 경우 두 번째 피연산자에 따라 연산자가 MOV, MOVL, MOVB 등으로 변할 수 있다.

Fig.6.과 Fig.7.의 예에서 볼 수 있는 바와 같이, 어셈블리 코드만으로 하드코딩된 주소 값을 식별하는 것은 오탐지(false negative) 가능성이 있다. 그러나 Fig.3.의 소소 코드를 분석한다면 App_Task2에서만 하드코딩된 주소 값을 사용한다는 것을 쉽게 파악할 수 있다. 따라서 소스코드 분석을 동시에 이용한다면 안전하지 않은 포인터 연산 검출의 정확성을 높일 수 있다. 소스코드와 심볼 정보를 사용한 포인터 새니타이저를 구현하기 위해서는 각 태스크의 스택의 위치와 크기를 확인하는 것이 필요하다.

Fig.8.은 uC/OS 코드 중 태스크를 생성하는 함수의 원형이다. Fig.8.에서 태스크를 생성 시 사용되는 첫 번째 인자(App_Task2)은 태스크 코드의 시작점을 가리키는 포인터이며, 세 번째 인자(App_TaskPostStk[])는 태스크 스택의 시작위치를 가리키는 포인터이다. App_TaskPostStk[] 인자가 가리키는 위치는 심볼 정보를 통해 메모리 주소를 확인할 수 있다. 추가로 스택의 크기를 지정하는 인자(APP_CFG_TASK_STK_SIZE)의 값을 사용하면, 태스크의 스택 범위를 확인할 수 있다.

Fig.9.는 컴파일된 기계어 코드에 대한 심볼 정보의 일부이다. 이를 통해 태스크 스택의 시작위치를 구해, 스택의 범위를 계산할 수 있다. Fig.9.의 정보를 이용하여 App_Task2의 스택 범위는 0xe040 ~ 0xe1c0임을 확인할 수 있다. 이 정보를 이용하여 Fig.3.의 App_Task2에서 포인터 변수 ptr에 메모리 주소 값을 직접 하드코딩 하는 경우는

```
#define  APP_CFG_TASK_STK_SIZE          384u
OSTaskCreateExt(
App_Task2,                                -----(1)
(void     *)0,
(CPU_STK *)&App_TaskPostStk[0],           -----(2)
(INT8U     )APP_CFG_TASK_PEND_PRIO,
(INT16U    )APP_CFG_TASK_PEND_PRIO,
(CPU_STK *)&App_TaskPostStk
[APP_CFG_TASK_STK_SIZE - 1u],
(INT32U    )APP_CFG_TASK_STK_SIZE,  ----(3)
(void     *)0,
(INT16U    )(OS_TASK_OPT_STK_CHK |
 OS_TASK_OPT_STK_CLR));
```

Fig. 8. Function prototype for task creation in uC/OS

```
address      name
----------    ----------
0000e040    _App_TaskPostStk
0000e340    _App_TaskStartStk
0000e640    _App_TaskPendStk
```

Fig. 9. Example: symbol table information

App_Task2의 스택범위가 아니므로 이를 탐지하여 사용자에게 경고해준다.

4.4 분석 및 논의

제안 기법을 기존 연구들과 비교 평가한 결과를 Table 2.로 나타내었다. Address Sanitization 프레임워크[13] 연구는 주요 메모리 영역에 stub 코드를 삽입하여 잘못된 접근 및 공격을 파악하고 차단할 수 있지만, stack과 heap 영역에만 한정되고 stub 코드 삽입으로 인해 메모리 사용량이 증가하는 만큼 임베디드 시스템에는 적합하지 않다. 또한 보안에 취약한 함수의 호출 여부 및 호출 관계를 알 수 없고, 동적 분석이기 때문에 입력 값이나 실행 시나리오에 따라 코드 커버리지가 달라질 수 있다.

임베디드 실행파일 코딩 표준 준수 여부 분석 기법[14]은 실행파일을 역어셈블하여 함수 단위의 흐름 그래프를 이용하여 하드코딩된 포인터 연산의 오용을 탐지할 수 있고 정적 분석으로 코드 커버리지가 넓다. 그러나 하드코딩된 포인터 연산 외의 부적절한 메모리 접근이나 취약한 함수를 고려하지 않은 한계가 있다.

uC/OS 운영체제 방어 기법[16]은 함수 콜 스택을 사용해 취약함수 호출 여부 및 호출 관계를 파악할 수 있다. 그러나 포인터 연산의 오용이나 잘못된 메모리 접근을 탐지할 수 없다. 콜 스택을 임베디드 장치 내부의 별도 공간에 저장함으로써 메모리 사용량이 증가하며 새로운 하드웨어 도입을 필요로 한다. 또한, 데이터나 힙 영역의 버퍼 오버플로 공격이나 부적절한 포인터 연산 등에 대한 탐지를 할 수 없다.

반면 제안 기법은 정적 분석으로 코드 커버리지가 넓고, 임베디드 기기에 바이너리 코드가 적재되기 전 사용자 컴퓨터에서 콜 그래프 생성 및 분석하므로 실행 오버헤드 및 추가적인 메모리 사용이 발생하지 않는다. 함수 새니타이저는 취약한 함수들의 사용 여부와 호출 관계를 파악할 수 있으며, 포인터 새니타이

Table 2. Comparison of Proposed Technique and Existing Techniques

	ASan 〔13〕	Coding Standard Compliance 〔14〕	Defending uC/OS against buffer overflows 〔16〕	Proposed technique
Analysis of vulnerable function calls	X	X	O	O
Detection of misused pointer operation	X	O	X	O
Detection of unsafe memory access	O	O	O	O
Code coverage	Low	High	Low	High
Memory overhead of embedded devices	High	Low	High	Low
No hardware change	X	O	O	O

저를 이용하여 포인터 연산의 오남용 여부도 탐지할 수 있다. 스택 영역뿐만 아니라 데이터나 힙 영역에서 발생할 수 있는 버퍼 오버플로 및 부적절한 포인터 연산 관련 취약점도 탐지할 수 있다.

V. 결 론

본 논문에서는 uC/OS를 사용하는 PLC의 보안성을 강화하기 위해, PLC로 직접 다운로드 될 기계어 코드를 검사하여 관리자/개발자가 의도하지 않은 취약한 라이브러리 함수와 포인터 연산을 탐지하는 실행코드 새니타이저를 제안하고 구현하였다. 본 제안기법은 개발자의 실수로 부적합한 함수와 포인터 연산을 사용하거나, 악성코드에 감염된 IDE가 악성 라이브러리 함수를 임의로 추가하는 경우에도 사전 탐지하여, PLC의 보안성을 강화할 수 있다.

본 논문의 제안기법은 표준 라이브러리 함수명과 포인터 연산의 특정 패턴만을 시그니처로 사용하기 때문에, 악성 행위를 위해 공격자가 만든 사용자 정의 함수 및 코드의 경우는 고려하지 않아 탐지하지 못한다는 한계점을 갖고 있다. 포인터 새니타이저의 경우에는 패턴을 정밀하게 설정하지 않으면 오탐지의 가능성이 존재한다. 이러한 한계점을 극복하기 위하여, 향후에는 공격자에 의해 추가된 악의적 함수나 코드를 탐지하여 필터링 하는 연구와 uC/OS에 제어 흐름 무결성(control flow integrity)기법을 적용하여 PLC 동작 중에 공격을 탐지하는 연구 수행할 계획이다.

References

〔1〕 Ephrem Ryan Alphonsus and Mohammad Omar Abdullah, “A review on the applications of programmable logic controllers (PLCs),” Renewable and Sustainable Energy Reviews vol. 60, pp. 1185-1205, 2016.

〔2〕 Naoum Sayegh, Ali Chehab, Imad H. Elhajj and Ayman Kayssi, “Internal security attacks on SCADA systems,” Third International Conference on Communications and Information Technology(ICCIT), pp. 22-27, Jun. 2013.

〔3〕 Ali Abbasi, Ghost in the PLC: stealth on-the-fly manipulation of programmable logic controllers’ I/O, CTIT Technical Report Series, (TR-CTIT-16-02), University of Twente, 2016.

〔4〕 Do-Yeon Kim, “Cyber security issues imposed on nuclear power plants,” Annals of Nuclear Energy vol. 65, pp. 141-143, 2013.

〔5〕 G. P. H. Sandaruwan, P. S. Ranaweera and Vladimir A. Oleshchuk, “PLC security and critical infrastructure protection,” IEEE 8th International Conference on Industrial and Information Systems, pp. 81-85, Aug, 2013.

〔6〕 Nicolas Falliere, “Liam O Murchu and Eric Chien, W32.Stuxnet Dossier,”

White paper, Symantec Corporation, Security Response, vol. 5. no. 6, pp. 1-69, Feb, 2011.
〔7〕 Eric Chien, Liam O Murchu and Nicolas Falliere, “W32.Duqu: The Precursor to the Next Stuxnet,” The 5th USENIX Workshop on Large-Scale Exploits and Emergent Threats, 2011.
〔8〕 Candid Wueest, “Targeted Attacks Against the Energy Sector,” Symantec Security Response, pp. 1-29, Jan, 2014.
〔9〕 Stevan A. Milinkovic and Ljubomir R. Lazic, “Industrial PLC security issues,” 20th Telecommunications Forum (TELFOR), pp. 1536-1539, Jan. 2012.
〔10〕 Kwang Hyun Park and Jae Wook Jeon, “Embedded Operating Systems: Windows CE, Embedded Linux, pSOS, uC/OS,” 2003 International Conference on Control, Automation and Systems (ICCAS 2003), pp. 1976-1981, Oct. 2003.
〔11〕 Dong Yulin and Zheng Chunjiao, “Design and research of embedded PLC development system,” 3rd International Conference on Computer Research and Development, vol. 3, pp. 226-228, May 2011.
〔12〕 Sampat S. Pawar and P.C. Bhaskar, “Design and Development of ARM based Real-Time Industry Automation System using GSM,” International Research Journal of Engineering and Technology (IRJET), Vol. 2, No. 5, pp.800-805, Aug, 2015.
〔13〕 Konstantin Serebryany, Derek Bruening, Alexander Potapenko and Dmitry Vyukov, “AddressSanitizer: A Fast Address Sanity Checker,” USENIX Annual Technical Conference, pp. 309-318Jun. 2012.
〔14〕 Ramakrishnan Venkitaraman and Gopal Gupta, “Static program analysis of embedded executable assembly code,” Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems, pp. 157-166, Sep, 2004.
〔15〕 Software Reliability Enhancement Center, Technology Headquarters and Information-technology Promotion Agency, ESCR(Embedded System development Coding Reference) 〔C language edition〕 Ver.3.0, Information-technology Promotion Agency(IPA), Mar. 2018.
〔16〕 Amjad Basha M Sikiligiri, “Buffer overflow attack and prevention for embedded systems,” Doctoral dissertation of Science in Computer Engineering, University of Cincinnati, 2011.
〔17〕 Haugh Eric and Matt Bishop, “Testing C Programs for Buffer Overflow Vulnerabilities,” The 10th Annual Network and Distributed System Security Symposium(NDSS), Feb, 2003.
〔18〕 Robert Seacord, Secure Coding in C and C++: Secure Coding in C and C++ (SEI Series in Software Engineering) 2nd Ed, Addison-Wesley Professional, Mar. 2013.
〔19〕 Motor Industry Software Reliability Association, “MISRA-C:2012 Guideline for the use of the C language in critical systems,” MIRA Limited, ISBN 978-1-906400-10-1, 2012.
〔20〕 Texas Instruments, TMS320C6000 Code Composer Studio Tutorial, Literature Number SPRU301C, Texas Instruments Publishers, 2000.

〈저자소개〉

최 광 준 (Gwang-jun Choi) 학생회원
2015년 8월: 단국대학교 컴퓨터학과 졸업
2019년 2월: 단국대학교 컴퓨터학과 석사
〈관심분야〉 침입탐지, 임베디드 시스템

유 근 하 (Geun-ha You) 학생회원
2018년 2월~현재: 단국대학교 소프트웨어학과 학사과정
〈관심분야〉 시스템 보안, 시스템소프트웨어, 임베디드 시스템

조 성 제 (Seong-je Cho) 종신회원
1989년: 서울대학교 컴퓨터공학과 졸업
1991년: 서울대학교 컴퓨터공학과 공학석사
1996년: 서울대학교 컴퓨터공학과 공학박사
1997년 3월~현재: 단국대학교 컴퓨터학과/소프트웨어학과 교수
〈관심분야〉 시스템 보안 및 악성코드 분석, 소프트웨어 보증, 시스템소프트웨어, 임베디드 소프트웨어 등