https://doi.org/10.33778/kcsa.2024.24.3.105

C언어 프로그래밍의 메모리 취약점에 대한 오류 감지 및 자동 수정 알고리즘★

서 연 경*, 전 상 훈**

요 약

프로그래밍 언어 교육은 컴퓨터의 기본적인 동작 원리를 학습하는데 도움을 주고, 논리적 사고력과 같은 여러 가지 문제 해결력을 길러주기 때문에 2015년도부터 초, 중, 고등학교 교육과정에 SW를 포함시켰다. 프로그래밍 언어는 다양하지만 그 중에서 C언어는 구성의 단순성, 하드웨어의 효율성, 오랜 역사가 있다는 장점으로 인해 다양한 분야에서 사용 중이다. 그러나 보안에 관련된 오류는 C언어를 사용할 때 매우 치명적이다. C언어의 보안 오류는 운영체제에서 문제를 발생시키고, 걷잡을 수 없이 확산되어 큰 피해를 입힐 수 있다. 따라서 본 논문에서는 C언어의 메모리 취약성 중 버퍼 오버플로우, 포인터 오류, 포맷 스트링, 정수 오버플로우 등 주요 네 가지 에러를 선정하고, 이들의 특징을 분석하여 오류 검출 및 수정 방법을 제시하는 '메모리 취약성 오류 검출 및 자동 수정' 알고리즘을 제안한다. 제안한 알고리즘을 통해 C언어 메모리 취약성의 심각한 오류와 피해가 줄어드는 것에 많은 도움을 줄 수 있을 것이다.

An Error Detection and Automatic Correction Algorithm for Memory-related Vulnerabilities in C language Programming

Yeon-Gyeong Seo*, Sanghoon Jeon**

ABSTRACT

Since 2015, programming has been included in school curricula to enhance computer literacy and problem-solving skills. C language, widely used for its simplicity, efficiency, and long history, poses significant security risks, particularly in memory vulnerabilities like buffer overflow, pointer errors, format strings, and integer overflow. These vulnerabilities can cause severe system issues and widespread damage. This paper proposes an "Error Detection and Automatic Correction of Memory Vulnerabilities (EDAC)" algorithm to detect and correct these errors, aiming to reduce the impact of C language memory vulnerabilities.

Key words : Programming, C language, Memory vulnerability, Error Detection, Automatic Correction

접수일(2024년 05월 31일), 수정일(1차: 2024년 06월 28일), 게재확정일(2024년 07월 09일)

★ 본 연구는 2021년도 정부(교육부)의 재원으로 한국연구재단의 지원을 받아 수행된 기초연구사업임 (No. 2021R1I1A1A0104094313)

* 수원대학교/정보보호학과 (주저자)

** 수원대학교/정보보호학과 (교신저자)

1. 서 론

2015년 9월부터 교육부는 SW 교육을 강화하겠다고 발표하였고, 2018년부터 초, 중, 고등학교 교육과정에 SW 교육을 포함시켰다[1]. 프로그래밍 언어 교육은 컴퓨터의 기본적인 동작 원리를 학습하는 데에 도움을 주고, 논리적 사고력과 문제 해결력을 길러주기 때문에 컴퓨터 교과과정에서 필수 영역으로 자리 잡고 있다[2].

프로그래밍 언어로는 C, C++, Java, Python 등이 있다. 그 중에서도 C언어는 현재까지 가장 널리 사용되는 프로그래밍 언어 중 하나이며, 다양한 프로그래밍 언어의 개발과 발전에 큰 도움을 주었다. 이는 구성의 단순성, 하드웨어 수준의 효율성, 오랜 역사 등의 장점으로 인해 다양한 분야에서 사용되고 있다[3]. 또한 C언어는 컴파일 언어로, 기계어에 가까운 코드로 변환되기 때문에 실행 속도가 빠를 뿐만 아니라 시스템 레벨에서의 프로그래밍이나 운영체제 개발에 적합한 언어이다[4]. 이러한 다양한 장점 덕분에 C언어는 대중적인 언어로 자리 잡게 되었다.

하지만, C언어는 성능 최적화를 최우선 목표로 설계되었기 때문에 최적화 관련 규약이 필요하다. 이러한 규약을 어기고 프로그램을 작성하게 되면 컴파일러가 경고 없이 컴파일되어 에러가 발생할 수 있다[5]. 보안과 관련된 C언어 에러는 주로 메모리 접근 오류로, 이는 C언어의 약한 타입 시스템, 통제되지 않는 타입 변환, 포인터의 무제한 사용, 배열 타입의 느슨한 처리 등 여러 가지 원인으로 발생한다[6]. 메모리 접근 오류는 포인터 변수를 잘못 사용하여 발생하며, 특정 운영체제에서 이러한 문제가 발생하면 빠르게 확산되어 심각한 피해를 야기할 수 있다. 해커들은 이를 이용해 다양한 공격기법을 만들어 내고 있으며, 공격의 방법과 강도는 더욱 심각해지고 있으며, 메모리 관련 오류는 오늘날까지도 여전히 매우 위협적인 취약점이다[7].

본 논문에서는 메모리 관련 C언어 보안 취약성 중에서 버퍼 오버플로우, 포인터, 포맷 스트링, 정수 오버플로우를 다루고, 이러한 취약점을 식별하고 자동으로 코드를 수정하는 자동 오류 검출 및 수정 알고리즘을 제시한다.

본 논문의 구성은 다음과 같다. Ⅱ장에서는 C언어 메모리 에러와 보안 방법에 관한 관련 연구를 기술한다. Ⅲ장에서는 C언어 메모리 에러를 자동으로 검출하는 제안 알고리즘을 설명한다. Ⅳ장에서는 해당 알고리즘을 실험과 사례 연구를 통해 검증한다. Ⅴ에서는 결론 및 향후 연구 계획을 제시한다.

2. C언어 메모리 에러와 보안 방법

C 언어는 강력하고 효율적인 프로그래밍 언어이지만, 보안 측면에서 몇 가지 중요한 문제가 있다. 특히 메모리 관련 오류는 보안에 큰 영향을 미쳐 프로그램의 안정성을 약화시키고 수정이 어려운 버그를 초래할 수 있다.

2.1 버퍼 오버플로우

버퍼 오버플로우는 배열이나 버퍼에 할당된 메모리를 초과하여 데이터를 쓸 때 발생한다[7]. 이는 메모리의 다른 부분에 침범하여 손상을 줄 수 있다. 또한 버퍼의 경계를 넘어서 데이터를 기록하면 다른 변수의 값에 영향을 미치거나 프로그램의 제어 흐름을 바꿀 수 있는데, 이는 소프트웨어의 심각한 보안 취약점으로 남는다. 특히 C언어로 프로그램을 작성할 때 버퍼의 경계를 확인하는 코드를 작성하는 것은 프로그래머의 몫이므로 프로그래머의 실수로 인해 버퍼 오버플로 취약점이 발생하기 쉽다[8]. 버퍼 오버플로우를 방지하기 위한 가장 중요한 방법은 '안전한 함수 사용'이다[9]. 특히 C언어에서는 문자열 처리와 관련된 함수들이 보안에 취약한 경우가 많으므로, 안전한 함수를 사용하는 것이 매우 중요하다.

2.2 포인터 오류

포인터 오류는 잘못된 포인터 조작으로 인한 메모리 오류를 의미한다. 이는 잘못된 주소 참조나 이미 해제된 메모리의 재참조와 같은 문제로

인해 예측할 수 없는 동작을 유발할 수 있다. 올바르게 해제되지 않은 메모리 블록을 가리키는 포인터는 메모리 누수를 일으켜, 시간이 흐를수록 시스템 리소스를 점차 소모하고 프로그램의 성능을 저하시킨다. 또한 포인터 자료형의 크기를 저하 시킬 수 있다[4]. 공격자는 메모리 오버플로우를 이용해 다른 메모리 영역을 조작하거나 악의적인 코드를 실행할 수 있는 메모리 공간을 확보할 수 있으며, 이는 공격의 영향을 확대한다[6]. 포인터 오류를 방지하기 위한 가장 중요한 방법은 '유효성 검사'이다. 포인터를 사용할 때 포인터가 유효한지 검사하여 NULL 포인터 역참조와 같은 오류를 방지해야 한다[10]. 따라서 포인터가 NULL인지 확인하고, NULL이 아닌 경우에만 포인터를 역참조하도록 코드를 작성해야 한다.

2.3 포맷 스트링

포맷 스트링은 특별한 문자열 지정자를 가진 문자열을 의미한다. 포맷 스트링에서의 취약점은 printf()등의 함수에서 문자열 입력 포맷에 따라 나타날 수 있다[11]. 사용자 입력을 무분별하게 처리하고 포맷 문자열을 무분별하게 사용하면 보안 문제가 발생할 수 있다. 이를 통해 공격자는 포맷 문자열을 통해 메모리 읽기 또는 쓰기와 같은 공격을 수행할 수 있다[12]. 이러한 공격은 프로그램의 메모리 레이아웃을 손상시키거나, 프로그램의 실행 흐름을 변경하거나, 프로그램이 비정상적으로 종료되도록 한다. 포맷 문자열을 방지하기 위한 가장 중요한 방법은 '사용자 입력을 차단'하는 것이다. 사용자가 입력한 문자열이 어떤 경우에도 포맷 스트링에 참여하지 못하게 프로그램을 작성한다면 포맷 스트링에 의한 보안 오류를 방지할 수 있다[13].

2.4 정수 오버플로우

정수 오버플로우는 변수가 표현 가능한 범위를 초과하여 값이 저장될 때 발생하는 문제로, 부호 있는 정수가 갑자기 음수로 변경되면 보안 문제를 일으킬 수 있다[11]. 이는 무결성 문제와 보안 취약점을 유발할 수 있다. 오버플로우가 발생할 때, 변수가 표현할 수 있는 범위를 넘어가면서 메모리의 다른 영역을 침범하는 현상이 일어난다. 이러한 오버플로우는 주로 버퍼 오버플로우로 알려져 있으며, 인젝션 공격과 같은 악의적인 코드의 실행으로 이어질 수 있다. 공격자는 오버플로우 취약점을 이용하여 중요한 데이터와 코드 영역을 손상시키고, 시스템에 대한 액세스와 제어권을 얻을 수 있다[14]. 특히 정수 오버플로우 취약점을 악용하여 공격자가 악의적인 코드를 삽입하고 실행하는 원격 코드 실행 공격은 매우 위험한 상황을 초래한다. 정수 오버플로우를 방지하는 가장 중요한 방법은 '입력 값을 검증'하는 것이다. 프로그램에서 입력 받은 값을 사용할 때, 해당 값이 예상 범위 내에 있는지 확인하여 정수 오버플로우를 방지할 수 있다[15]. 특히, 산술 연산이나 메모리 할당을 위해 입력 값을 사용할 때는 주의가 필요하다.

2.5 관련 에러 검출 알고리즘

정적 코드 분석 도구는 소프트웨어 개발 과정에서 코드 품질을 향상시키고 보안 취약점을 발견하는 데 중요한 역할을 한다. 주요 정적 코드 분석 도구인 Coverity Scan, Cppcheck, Snyk, 그리고 본 논문에서 개발한 EDAC 알고리즘에 대해 논하고, 이들의 기능과 특징을 비교한다.

Coverity Scan[16] 은 다양한 프로그래밍 언어와 플랫폼을 지원하는 종합적인 정적 코드 분석 도구이다. 대규모 프로젝트와 기업 환경에서 널리 사용되며, 코드 품질 향상과 보안 강화, 산업 표준 준수를 목표로 한다. 주요 특징으로는 종합적인 결함 탐지, 산업 표준 준수, 심층 분석, 보안 취약점 탐지가 있다. 이 도구는 정교한 데이터 흐름 분석과 제어 흐름 분석을 통해 복잡한 결함을 탐지하고, 다양한 결함 유형에 대해 상세한 보고서와 수정 제안을 제공한다. 특히 버퍼 오버플로우와 같은 보안 관련 결함을 중점적으로 다루며, MISRA, CERT C/C++ 등 다양한 코딩 표준을 준수할 수 있도록 지원한다.

Cppcheck[17]는 C와 C++ 코드에서 다양한 결함을 탐지하는 경량 정적 분석 도구이다. 사용 용이성과 경

량 분석, 커스터마이제이션에 중점을 두고 있으며, 개발 과정에서 빠른 피드백을 제공한다. 주요 특징으로는 코드 스타일 검사, 성능 문제 탐지, 메모리 누수 등 다양한 결함을 경량 분석 방식으로 탐지하고, 간단한 수정 제안을 제공하는 점이 있다. 또한, 사용자 정의 규칙을 추가하거나, 특정 유형의 결함을 더 중점적으로 분석하도록 설정할 수 있으며, 플러그인이나 스크립트를 통해 기능을 확장할 수 있다.

Snyk[18]는 오픈 소스 라이브러리, 종속성, 컨테이너, 코드, 인프라스트럭처 코드에서 보안 취약점을 탐지하고 수정하는 데 중점을 두고 있다. 주로 오픈 소스 보안 및 라이센스 컴플라이언스에 중점을 두며, DevSecOps 환경에서 널리 사용된다. 주요 특징으로는 취약점 데이터베이스를 기반으로 오픈 소스 라이브러리 및 종속성의 알려진 보안 취약점을 탐지하고, 컨테이너 이미지 및 인프라 코드의 보안 문제를 스캔하는 점이 있다. 또한, 보안 취약점에 대한 상세 보고서와 수정 가능한 패치를 제안하며, 취약한 라이브러리의 안전한 버전으로 업그레이드할 것을 권장한다.

EDAC 알고리즘은 메모리 취약성 오류 탐지에 특화된 도구로, 특히 버퍼 오버플로우, 포인터 오류, 포맷 스트링 취약점, 정수 오버플로우와 같은 메모리 관련 오류를 집중적으로 탐지한다. EDAC 알고리즘은 메모리 접근 패턴, 정수 연산, 포인터 조작, 포맷 스트링 사용 등을 집중적으로 분석하여 메모리 취약성 오류를 탐지하며, 메모리 관련 문제에 대한 상세 보고서와 수정 방법을 제공한다. 주로 내장형 시스템, 펌웨어 개발, 안전-critical 시스템 등 메모리 오류가 치명적인 문제를 야기할 수 있는 환경에서 유용하다.

EDAC 알고리즘과 Coverity Scan, Cppcheck, Snyk의 주요 차이점은 목적과 기능 면에서 나타난다. EDAC 알고리즘은 메모리 취약성 오류 탐지에 특화되어 있으며, 메모리 관련 문제를 정밀하게 분석한다. 반면, Coverity Scan은 다양한 유형의 코드 결함과 보안 취약점을 포괄적으로 탐지하여 종합적인 코드 품질과 보안을 보장한다. Cppcheck은 경량 분석 도구로, 개발 과정에서 빠른 피드백을 제공하고, 코드 스타일, 성능 문제, 메모리 누수 등 다양한 결함을 다룬다. Snyk은 오픈 소스 보안 및 종속성 관리에 특화되어 있어 보안 취약점 탐지와 수정에 강점을 가지고 있다.

<표 1> Comparison of four algorithms

Security Problem	Coverity Scan	Cpp check	Snyk	**EDAC**
Buffer Overflow	O	O	X	O
Pointer Errors	O	O	X	O
Format Strings	O	O	X	O
Integer Overflow	O	O	X	O

<표 1>는 Coverity Scan과 Cppcheck과 Snyk과 EDAC를 네 가지 오류에 따라 O/X로 비교한 표이다. 표를 보면 Snyk를 제외하면 다른 도구는 EDAC와 다를게 없어 보이지만 성능적인 부분에서 많은 차이점이 있다. EDAC는 성능적인 부분에서 네 가지 오류의 검출에 특화된 알고리즘이다.

Coverity Scan은 다양한 결함 유형을 포괄적으로 탐지하여 종합적인 코드 품질과 보안을 보장한다. 반면, EDAC 알고리즘은 메모리 취약성 오류 탐지에 특화되어 있으며, 메모리 관련 문제를 정밀하게 분석하고 수정 방법을 제시한다. Coverity Scan은 매우 광범위한 결함 탐지 기능을 가지고 있다. 이 도구는 메모리 오류, 논리적 결함, 자원 누수, 성능 문제 등을 포함하여 다양한 유형의 결함을 포괄적으로 탐지한다. 반면, EDAC 알고리즘은 메모리 취약성 오류에 중점을 두고, 메모리 접근 패턴, 정수 연산, 포인터 조작, 포맷 스트링 사용 등을 집중적으로 분석하여 버퍼 오버플로우, 포인터 오류, 포맷 스트링 취약점, 정수 오버플로우와 같은 메모리 관련 문제를 정밀하게 탐지한다. 또한 Coverity Scan은 대규모 소프트웨어 프로젝트와 기업 환경, 다양한 산업 분야에서 사용된다. 이 도구는 코드 품질과 보안을 종합적으로 강화하려는 프로젝트에 적합하다. 반면, EDAC 알고리즘은 내장형 시스템, 펌웨어 개발, 안전-critical 시스템 등 메모리 오류가 치명적인 문제를 야기할 수 있는 환경에 적합하다.

Cppcheck은 다양한 종류의 코드 결함을 포괄적으로 탐지하고 코드 품질을 개선하는 데 중점을 둔다.

코드의 스타일, 성능 문제, 메모리 누수 등을 포함한 여러 종류의 결함을 검사하여 개발자에게 빠른 피드백을 제공한다. 예를 들어, 코드 내에서 잘못된 포인터 사용이나 초기화되지 않은 변수를 식별하고, 메모리 누수가 발생할 수 있는 부분을 추적한다. 또한 코드의 복잡도를 분석하여 코드의 유지 보수성과 가독성을 개선하는 데 도움을 준다. 반면, EDAC 알고리즘은 메모리 취약성 오류를 특히 강조하여 보다 안전하고 신뢰할 수 있는 소프트웨어 개발을 지원한다. 메모리 접근 패턴, 정수 연산, 포인터 조작 등을 정밀하게 분석하여 프로그램이 실행될 때 발생할 수 있는 잠재적인 보안 문제를 사전에 예방하는 데 도움을 준다. 따라서 EDAC 알고리즘은 메모리 오류가 치명적인 시스템, 예를 들어 내장형 시스템이나 안전 관련 시스템에서 매우 유용하게 사용될 수 있다.

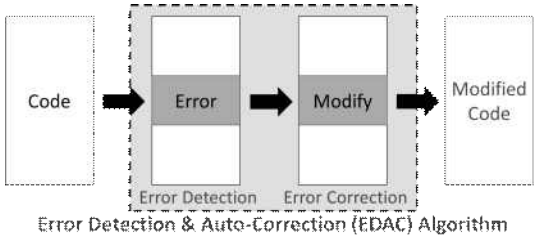
Snyk는 주로 오픈 소스 라이브러리, 컨테이너, 코드, 인프라스트럭처 코드(Infrastructure as Code)에서 보안 취약점을 식별하고 수정하는 데 중점을 둔 도구이다. Snyk은 버퍼 오버플로우, 정수 오버플로우, 포맷 스트링 취약점, 포인터 오류와 같은 특정 코드 결함을 직접적으로 탐지하는 도구는 아니므로 EDAC와 비교하기 어렵다.

3. 제안 자동 수정 알고리즘

3.1 메모리 취약성 오류 검출 및 자동 수정 알고리즘

본 논문에서는, 메모리 관련 주요 오류인 버퍼 오버플로우, 포인터 오류, 포맷 문자열, 정수 오버플로우를 자동으로 감지하고 수정하는 알고리즘을 제안한다. 이 알고리즘은 메모리 취약성 오류 검출 및 자동 수정(EDAC, Error Detection and Auto-Correction of Memory Vulnerabilities)으로 불리며, 메모리 오류를 자동으로 검출하고 수정하는 기능을 수행한다.

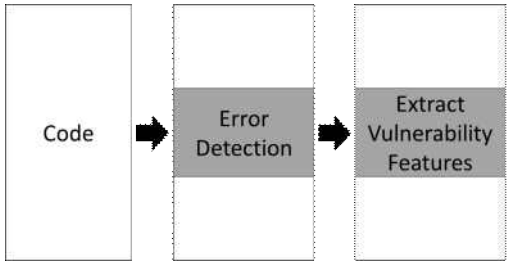
EDAC 알고리즘은 오류를 가진 코드를 가지고 크게 두 가지 단계를 거친 후 수정 방법을 제안하게 된다. 일반적인 정적 분석 도구가 코드 문제를 식별하고 보고하는 것에 그치는 반면, EDAC는 문제를 식별하고 적절한 수정 방안을 제공하여 직접 코드를 수정한다는 점에서 차별성이 있다. 이는 개발자들이 보다 효율적으로 문제를 해결할 수 있게 하는 도구이다. 단, 제공된 수정 코드는 최종 사용 전에 개발자의 검토가 필요하다.



(그림 1) Error Detection & Auto-Correction Algorithm

3.2 오류 검출부

오류 검출부에서는 주어진 코드의 오류 중 보안 취약성을 자동으로 검출한다. 버퍼 오버플로우, 포인터 오류, 포맷 문자열, 정수 오버플로우 각각의 오류마다 취약성이 다르기 때문에 각각의 대표적인 취약성 특징을 찾아낸다. 이 과정에서 EDAC는 오류를 검출에 중점을 두며, 검출된 오류에 따라 각각의 특징을 추출한다.



(그림 2) Error Detection Part

3.2.1 버퍼 오버플로우 특징 추출

가장 대표적인 버퍼 오버플로우는 배열 범위를 초과하여 데이터를 쓰거나 읽는 것이다. 예를 들어, strcpy 함수를 사용해 문자열을 복사할 때, 복사되는 문자열의 길이가 목표 버퍼의 크기를 초과하는 경우이다. 버퍼 오버플로우 특징을 추출하는

과정은 다음과 같다.

코드를 주의 깊게 검토하여 배열이나 버퍼를 사용하는 부분을 찾는다. 각 배열 또는 버퍼의 크기를 확인하고, 해당 배열이나 버퍼에 데이터를 쓰는 부분을 식별한다. 그 다음, 사용자의 입력이나 외부 입력을 받는 부분을 검토하여 입력 데이터의 유효성을 확인한다. 입력 데이터의 크기와 형식을 검사하여 유효성을 확인하고, 배열이나 버퍼에 쓰여질 데이터의 크기를 검사한다. 배열이나 버퍼를 사용하는 부분에서 해당 배열이나 버퍼의 인덱스를 확인하고, 배열 또는 버퍼의 크기를 초과하는 인덱스를 사용하는 부분을 찾는다. 루프나 반복문을 사용하여 데이터를 배열이나 버퍼에 쓰는 경우 해당 루프나 반복문의 조건을 검사하여 배열 또는 버퍼의 크기를 초과할 수 있는지 확인한다.

그 후, 정적 코드 분석 도구를 사용하여 코드를 검사하고 잠재적인 버퍼 오버플로우 취약점을 자동으로 감지한다. 이를 통해 발견된 취약점을 확인하고 수정할 수 있다. 마지막으로 메모리를 동적으로 할당하고 해제하는 부분을 검사하여 메모리 할당과 해제가 올바르게 이루어지는지 확인하고, 할당된 메모리 영역을 벗어나는 포인터 사용이나 메모리 누수가 발생하는지 확인한다.

3.2.2 포인터 오류 특징 추출

가장 대표적인 포인터 오류 중 하나는 Dangling Pointer이다. Dangling Pointer는 이미 해제된 메모리를 가리키는 포인터를 가리키는 경우 발생한다. 이러한 상황에서 포인터를 역참조하면 예상치 못한 결과가 발생할 수 있다. 포인터 오류의 특징을 추출하는 과정은 다음과 같다.

먼저 코드를 조사하여 포인터가 활용되는 위치를 파악한다. 이는 포인터가 메모리 주소를 참조하거나 변경하는 모든 부분을 포함한다. 코드에서 포인터가 NULL을 가리키는지 여부를 확인하고, 포인터가 참조하는 메모리 영역이 할당되었는지도 확인한다. 메모리를 동적으로 할당한 후에 포인터를 사용하기 전에 이를 검증한다. 할당된 메모리를 더 이상 사용하지 않을 때는 메모리를 해제한다. 또한, 포인터를 사용하여 배열이나 버퍼에 접근할 때 배열이나 버퍼의 경계를 초과하지 않는지 확인한다. 정적 코드 분석 도구를 활용하여 코드를 검사하고 잠재적인 포인터 오류 취약점을 자동으로 찾아낸다. 메모리 할당과 해제 부분을 검토하여 메모리 할당과 해제가 올바르게 이루어지는지를 확인하고 마지막으로 포인터 캐스팅을 적절하게 사용하는지도 확인한다.

3.2.3 포맷 문자열 특징 추출

printf() 함수에서 사용되는 포맷 문자열에 %s 변환 지정자를 사용할 때, 이 변환 지정자로부터 받은 문자열을 제어할 수 없는 외부 입력으로부터 받는 경우에 오류가 발생한다. 포맷 문자열의 특징을 추출하는 과정은 다음과 같다.

코드를 조사하여 printf(), sprintf(), fprintf() 등과 같은 포맷 문자열을 사용하는 함수 호출을 찾는다. 함수 호출에서 포맷 문자열이 하드코딩된 것인지, 아니면 사용자 입력으로부터 받은 것인지를 확인한다. 포맷 문자열이 하드코딩된 경우, 해당 문자열을 분석하여 취약점을 발견할 수 있는지 확인한다. 예를 들어, %s와 같은 사용자 입력 값을 받는 형식 지정자를 사용하는지를 검사한다. 그 다음, 사용자로부터 입력받은 포맷 문자열이 안전한지 확인하기 위해 입력의 유효성을 검증하고 포맷 문자열을 사용하는 함수에 전달되는 형식 문자열을 검사한다. 이를 통해 형식 문자열에 대한 적절한 검증이 이루어졌는지 확인한다. 마지막으로 정적 코드 분석 도구를 활용하여 코드를 검사하고 포맷 문자열 취약점을 자동으로 감지한다.

3.2.4 정수 오버플로우 특징 추출

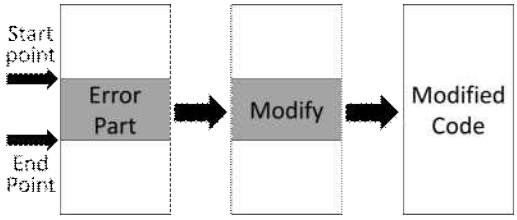
정수형 변수가 최대값을 초과하여 음수로 순환하는 경우는 주로 부호 있는 정수형(int)에서 발생한다. 정수 오버플로우의 특징을 추출하는 과정은 다음과 같다.

코드를 검토하여 산술 연산을 수행하는 부분을 찾는다. 이는 주로 덧셈, 뺄셈, 곱셈, 나눗셈 등의 연산을 포함한다. 연산에 사용되는 입력값과 변수들의 범위를 확인하고 이들이 각각의 데이터 타입의 범위를 벗어나는지를 확인하고 연산의 결과가

사용되는 부분을 확인하여 해당 결과가 변수나 데이터 타입의 범위를 초과하는지를 검사한다. 형변환이 올바르게 이루어지지 않거나, 변환된 값이 원하는 범위 내에 있는지를 확인한다. 조건문에서 변수나 연산 결과가 사용되는 부분을 검사하고 이들이 조건을 충족시키지 못하거나, 예상대로 동작하지 않을 수 있는지를 확인한다. 마지막으로 정적 코드 분석 도구를 활용하여 코드를 검사하고 정수 오버플로우 취약점을 자동으로 감지한다.

3.2 오류 수정부

오류 수정부에서는 오류 검출부에서 찾아낸 각각의 대표적인 취약성을 수정하는 과정이다. 코드 위치 파악, 코드 수정, 수정 방법 제안 순서로 진행되어 마지막에 자동으로 수정하게 된다. 이 과정에서 EDAC는 오류 수정부에서 받아온 오류의 특징들을 토대로 오류를 수정한다.



(그림 3) Error Correction Part

4. 실험 및 사례 연구

본 절에서는, EDAC 알고리즘의 구현 및 적용을 소개하고, 앞에서 소개했던 네 가지 메모리 관련 오류의 자동 수정의 실 예시를 알아본다.

4.1 EDAC 알고리즘

EDAC 알고리즘은 주어진 C 코드 파일에서 특정 유형의 오류(버퍼 오버플로우, 포인터 오류, 포맷 문자열 오류, 정수 오버플로우)를 자동으로 수정하여 새로운 파일에 저장한다. 각 오류 유형에 대한 전용 함수를 사용해 원본 코드에서 해당 오류를 찾아 수정하며, 수정된 파일을 생성하는 과정을 보여준다. 이 알고리즘의 작동 방식은 다음과 같다. 오류 파일 경로와 수정된 파일 경로를 설정하고 오류 파일을 읽기 모드로 열고 수정된 파일을 쓰기 모드로 연 다음 오류 파일을 한 줄씩 읽어와서 각 줄에 대해 에러에 맞는 수정 방법을 적용한다. 수정된 줄을 수정된 파일에 쓰고 파일을 닫는다. 마지막으로 완료 메시지가 출력되면 끝난다. (알고리즘 1)은 구현된 EDAC 알고리즘을 나타낸다.

1번째 줄에서 fix_buffer_overflow_error 함수는 주어진 문자열에서 strcpy를 strncpy로 대체하여 버퍼 오버플로우를 수정하고, 문자열에서 strcpy를 찾아 strncpy로 대체하고, NULL 문자를 추가하여 문자열의 끝을 표시한다.

10번째 줄에서 fix_format_string_error 함수는 포맷 스트링 오류를 검출하고 자동으로 수정하고, 문자열에서 %를 찾아 다음 문자가 s인지 확인하여 포맷 스트링 오류를 검출한다. 만약 다음 문자가 s가 아니라면, 해당 %를 NULL 문자로 대체하여 문자열을 종료한다.

18번째 줄에서 fix_integer_overflow_error 함수는 정수 오버플로우 오류를 검출하고 자동으로 수정한다. 문자열에서 int 형식의 최댓값을 찾아 INT_MAX로 대체합니다. 이는 int 형식의 최대 값이 INT_MAX로 정의된 경우에 해당한다.

27번째 줄에서 fix_pointer_error 함수는 포인터 오류를 검출하고 자동으로 수정한다. 문자열에서 " "를 찾아 다음 문자가 공백이나 개행 문자인지 확인하여 포인터 오류를 검출한다. 발견된 경우 해당 " "를 "_"로 대체한다.

EDAC를 통한 오류를 수정을 하기 위해서 오류 검출부의 과정을 거친다. 그 후 오류 수정부를 통해 수정 방법을 제안한다. 단, 수정된 코드를 완전히 신뢰하지 않고 수정 방법으로만 받아들인다.

Algorithm 1 EDAC Algorithm

```
1: function FixBufferOverflowError(line)
2:     while pos ← findSubstring(line,"strcpy") do
3:         replaceSubstring(line,pos,"strcpy","strncpy")
4:         for i ← length("strcpy") to length("strncpy") do
5:             pos[i] ← ''
6:         end for
7:     end while
8:     return line
9: end function
10: function FixFormatStringError(line)
11:     while pos ← findSubstring(line,"%") do
12:         if pos[1] ≠' s' then
13:             pos ← terminateStringAfter(line,pos)
14:         end if
15:     end while
16:     return line
17: end function
18: function FixIntegerOverflowError(line)
19:     while pos ← findSubstring(line,"2147483647") do
20:         replaceSubstring(line,pos,"2147483647","INT_MAX")
21:         for i ← length("2147483647") to length("INT_MAX") do
22:             pos[i] ← ''
23:         end for
24:     end while
25:     return line
26: end function
27: function FixPointerError(line)
28:     while pos ← findSubstring(line," * ") do
29:         if pos[1] =' ' or pos[1] =' ' then
30:             pos ← replaceChar(line,pos,' *',' _')
31:         end if
32:     end while
33:     return line
34: end function
```

(알고리즘 1) EDAC Algorithm

4.2 EDAC 실행

EDAC를 사용하려면 올바른 경로 입력이 필수적이며 오류 코드와 수정된 코드를 저장할 파일의 경로가 정확해야 한다. 예를 들어, C 드라이브의 'codeerror/buffer' 폴더에 'bufferoverflow.c' 파일이 있다면 경로는 'C:/codeerror/buffer/bufferoverflow/bufferoverflow.c'가 된다. 수정된 코드를 저장할 파일명이 'modify'이고, 이 파일도 C 드라이브에 있다면, 저장 경로는 'C:/modify/modify.c'가 된다. 수정된 코드를 'C:/modify/modify.c'에 저장하려면 경로를 정확히 입력해야 한다. 올바른 경로를 입력하고 실행하면, C 드라이브의 파일에 c코드 파일이 생긴다. 경로 오류 시 "Error: 오류 파일을 열 수 없습니다." 또는 "Error: 수정 파일을 생성할 수 없습니다." 메시지가 나타난다.

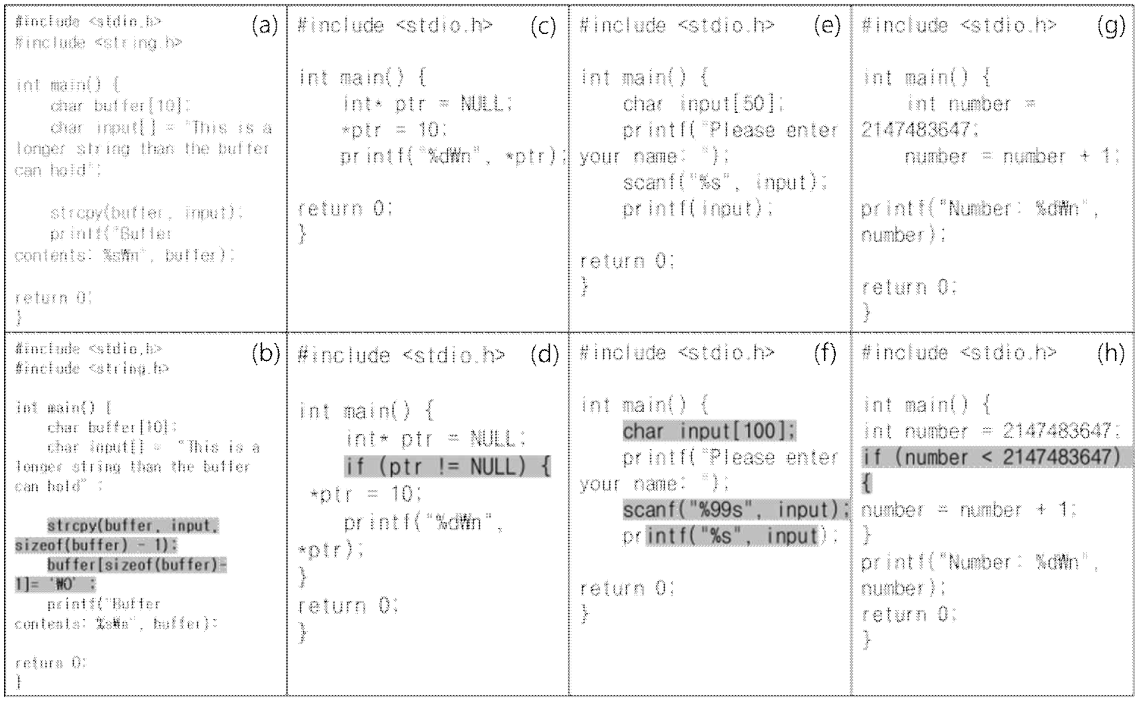
4.3 EDAC를 통한 오류 수정

(그림 4)는 왼쪽부터 버퍼 오버플로우, 포인터 오류, 포맷 문자열, 정수 오버플로우의 오류 코드와 수정 코드를 나타낸 사진이다. 위쪽 코드는 각 오류의 수정 전 코드이고 아래쪽 코드는 각 오류의 수정 후 코드이다.

(그림 4-(a))는 버퍼 오버플로우의 EDAC 적용한 예시를 다룬다. strcpy 함수를 사용해 input 배열의 내용을 buffer 배열로 복사할 때 발생하는 버퍼 오버플로우를 보여준다. buffer 배열의 크기를 초과하는 데이터가 복사되어 오버플로우가 발생한다. EDAC는 이를 감지하면 strncpy 함수를 사용하도록 제안하며, 수정된 코드를 저장한다. (그림 4-(b))에서는 strcpy 함수가 strncpy 함수로 변경된 것을 볼 수 있다. (알고리즘 1)의 두 번째 줄부터는 버퍼 오버플로우를 방지하기 위해 strncpy 함수를 사용하는 것으로 수정된다. strncpy 함수는 목표 버퍼의 크기를 지정해 문자열을 복사하고, 버퍼의 끝에 널 문자를 추가해 안전하게 버퍼 오버플로우를 방지한다.

(그림 4-(c))는 포인터 오류의 EDAC를 적용한 예시를 다룬다. 포인터를 NULL로 초기화한 후 그 포인터를 참조하려고 할 때 발생하는 오류를 보여준다. 이는 NULL 포인터 역참조 오류라고 한다. EDAC는 각 라인을 검사해 포인터 오류를 감지하면, NULL이 아닌 경우에만 역참조를 수행하도록 수정한다. 수정된 코드는 저장된다. (그림 4-(d))에서 수정된 부분에서는 포인터 ptr이 NULL이 아닌지 확인하는 부분이 추가된 것을 볼 수 있다. (알고리즘 1)의 28번째 줄부터는 포인터 오류를 방지하기 위해 ptr이 NULL이 아닌 경우에만 코드를 실행하도록 수정된다. 이는 NULL 포인터 역참조 오류를 방지한다.

(그림 4-(e))는 포맷 문자열 오류의 EDAC를 적용한 예시를 다룬다. 최대 49개의 문자를 저장할 수 있는 문자열 배열 input이 선언되어, 사용자가 입력한 %s와 같은 포맷 문자열이 printf 함수에 직접 전달될 때 발생하는 포맷 문자열 오류를 보여준다. 사용자가 의도하지 않은 형식 문자열을 입력하면 프로그램이 정상적으로 동작하지 않을



(그림 4) Error Code and Modified Code

수 있다. (그림 4-(f))의 수정된 부분에서는 첫 번째로 input 배열의 크기를 100으로 늘려 널 종료 문자를 위한 공간을 마련한 부분을, 두 번째로 scanf 함수 호출에서 %s 포맷 지정자 뒤에 최대 100자의 입력을 지정한 부분을 볼 수 있다. (알고리즘 1)의 11번째 줄부터는 printf 함수 호출 시 사용자 입력을 그대로 출력하는 대신 %s 포맷 지정자를 사용하도록 수정하는 방법을 제안한다.

(그림 4-(g))는 정수 오버플로우의 EDAC 적용한 예시를 다룬다. 정수형 변수 number가 초기값 2147483647을 가지며, 이는 int 형의 최댓값이다. 다음 줄에서 number에 1을 더해 정수 오버플로우가 발생하고, 그 결과 number에는 의도하지 않은 값이 저장된다. EDAC는 각 라인을 순회하며 정수 오버플로우를 감지하고, if 조건문을 사용해 int의 최댓값을 넘지 않도록 수정한다. (그림 4-(h))의 수정된 부분에서는 number 변수에 값을 할당하기 전에 if 조건문을 사용해 최댓값 2147483647과 비교한다. (알고리즘 1)의 19번째 줄부터는 number가 2147483647보다 작은 경우에만 증가 연산을 수행해 정수 오버플로우를 방지한다. 수정된 코드는 EDAC를 통해 저장된다.

5. 결 론

2015년부터 교육부가 SW 교육을 강화하기 시작했다. 프로그래밍 언어 교육은 논리적 사고력과 문제 해결력을 길러주기 때문에 초, 중, 고등학교 교육과정에서 필수 영역으로 자리 잡기 시작했다. 프로그래밍 언어 중 C언어는 널리 사용되는 언어로, 단순하고 효율적이지만 보안 취약점이 존재한다.

본 논문에서는 C언어의 메모리 관련 취약점과 이를 해결할 보안 방법을 연구하고, 오류를 감지하여 수정하는 알고리즘인 EDAC를 제안한다. EDAC는 코드를 읽고 취약점을 파악하여 오류의 종류를 식별하는 오류 검출 과정을 거치고, 이후 적절한 수정 방법을 찾아내어 코드를 수정하는 오류 수정 과정을 수행한다. 즉, EDAC는 버퍼 오버

플로우, 포인터 오류, 포맷 스트링 오류, 정수 오버플로우 등 네 가지 주요 오류 유형을 식별하고 자동으로 수정한다. EDAC는 검출하고자 하는 메모리 관련 오류에 집중하여 사용이 가능할 것이라는 점에서 기존의 정적 코드 분석 도구와 차이점이 있다. 기존의 정적 코드 분석 도구는 어떤 오류인지를 알려주는 선에서 끝나지만, EDAC는 오류를 발견하고 수정하는 과정을 자동화하여 개발자가 직접 코드를 수정하는 시간과 노력을 절약한다. 이는 특히 초보 개발자들이나 보안 전문 지식이 부족한 개발자들에게 유용하다. 또한 C언어는 단순하고 효율적이지만 메모리 관리에 취약점이 있는데, EDAC는 이러한 취약점을 효과적으로 탐지하고 보안을 강화하는 데 기여한다. 따라서 EDAC를 적용함으로써 소프트웨어의 보안성을 크게 향상시킬 수 있다.

제안된 연구는 메모리 관련 취약점 네 가지만을 다룬다는 점에서 한계가 있지만, 향후 후속 연구에서 많은 보안 관련 기능을 추가할 것이다. Coverity Scan의 취약점 탐지와 보고의 뛰어난 점과 보안 취약점의 다양한 유형 학습과 지속적인 개선과 같은 장점과 Cppcheck의 보안 관행과 최신 보안 취약성 이해를 기반으로 C언어의 다양한 메모리 취약성 문제를 해결할 수 있는 방법을 모색할 계획이다. 이를 통해 C언어 메모리 취약점으로 인한 피해를 줄이는 데 기여할 수 있을 것이다.

참고문헌

[1] Ryu Ji-hyun and Kim In-jeong, "A Case Study of Early Child Coding Education Program for Convergence Talent", Jonornal of the Korean Convergence Association Paper, vol. 10(8), pp. 129-135, Aug. 2019.

[2] Kim Haeng-im and Park Eun-kyung and Kim Hyun-joo and Bae Jong-min, "Integrated Visual-Based C Programming Environment for Beginners in Programming", Journal of the Jonornal of Computer Education Association, vol. 16, no. 6, pp. 111-120, Nov. 2013.

[3] Jeon Hyun-suk, Jeon Jong-kwang, and Kim Sung-sik, "Designing and operating problems for basic learning in C language", Journal of Korean Society of Computer Education, 18 (1), pp. 291-294, Jan. 2011.

[4] Park Si-hyung, Lee Je-min, and Kim Hyung - shin, "Compare Energy Efficiency by Programming Language in Internet of Things Devices", Journal of the Korean Information Society's academic presentation paper, 2016(12), pp. 334-336p, Dec. 2016.

[5] Heo Chung-gil, "the complex conventions of language C", Jonornal of Communications of the Korean Institute of Information Scientists and Engineers, 35(3), pp. 41-45, Mar. 2017.

[6] Pyo Chang-woo and Han Kyung-sook, "Coding Standards for Safe C Programs", Jonornal of information science journal, vol. 28, no. 2, pp. 48-50, Feb. 2010.

[7] Bonghan Kim, "Implementation of a purging agent for detecting buffer overflow vulnerabilities", Journal of the Korean Society of Convergence, 12(1), pp. 11-17, Jan. 2021.

[8] Kim Yu-il and Han Hwan-soo, "Status and Prospects of Static Analysis Tools for the Detection of Buffer Overflow", Journal of the Society of Information Protection, 16(5), pp. 45-54, Oct. 2006.

[9] Lee Min-jae and Han Kyung-sook, "New RAD: Solutions to Buffer Overflow Attacks", The Korean Society of Information Science, 32(2), pp. 979-981, Nov. 2005.

[10] Kim Hyun-soo, Kim Byung-man, and Bae Hyun-seop, "Detecting Null Pointer Accessibility Based on Assembly Code", Joneral of the Korean Society of Information Science, 36(2), pp. 41-44, Nov. 2009.

[11] Kim Ji-hong, Nam Ye-ji, and Um Young-ik, "Analysis of Attack and Defense Techniques according to Security Vulnerabilities", Journal

of the Korean Society for Information Science, pp. 1534-1535, Jun. 2014.

[12] Park Hyun-jin and Kim In-seok, "Effective Response to the Analysis and Evaluation of Vulnerabilities on Financial Institutions' Homepages", Journal of the Society for Information Protection, 27(4), pp. 885-895, Aug. 2017.

[13] Lee Hyung-bong, Cha Hong-jun, and Choi Hyung-jin, "the effect of format strings on program security in C language", Journal of the Journal of Information Processing Society, 8(6), pp. 693-702, Dec. 2001.

[14] Dietz. Will, Li. Peng, Regehr. John, and Adve. Vikram, "Understanding integer overflow in C/C++", Jonornal of ACM Transactions on Software Engineering and Methodology (TOSEM), vol. 25, no. 1, pp. 1-29, Jan. 2015.

[15] Kim Tae-yang and Park Jung-hoon, "Security of Software Development in C Language Compilers", a collection of papers at the Korean Software General Conference, 2017(12), pp. 1,623-1,625, Dec. 2017.

[16] Nasif Imtiaz, Brendan Murphy, and Laurie Williams, "How Do Developers Act on Static Analysis Alerts? An Empirical Study of Coverity Usage", 2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE), pp. 323-333, 2019.

[17] Pereira, Jose D'Abruzzo, and Marco Vieira. "On the use of open-source c/c++ static analysis tools in large projects", pp. 97-102, 2020.

[18] Sushma D, Nalini M K, R Ashok Kumar, MuraliKrishna Nidugala, "To Detect and Mitigate the Risk in Continuous Integration and Continues Deployments (CI/CD) Pipelines in Supply Chain Using Snyk tool", 2023 7th International Conference on Computation System and Information Technology for Sustainable Solutions (CSITSS), pp. 1-10, 2023.

〔저자소개〕

서 연 경 (Yeon-Gyeong Seo)
2022년 3월~현재 수원대학교
정보보호학과 학사과정
email : dasun031001@naver.com

전 상 훈 (Sanghoon Jeon)
2012년 2월: 경북대학교 IT대학 심화 전자공학 공학사
2014년 2월: 대구경북과학기술원 정보통신융합 공학전공 공학석사
2020년 8월: 대구경북과학기술원 정보통신융합 전공 공학박사
2020년 3월~8월: 한양대학교 산학협력단 선임연구원
2020년 9월~2022 9월: 한양대학교 의과대학 응급의학과 포닥연구원
2022년 10월~2023 9월: 한양대학교 의과대학 응급의학과 연구조교수
2023년 10월~현재: 수원대학교 지능형SW융합대학 정보보호학과 조교수
email : shjeon@suwon.ac.kr