**KIPS Trans. Comp. and Comm. Sys.**
Vol.12, No.9 pp.283~290
ISSN: 2287-5891 (Print), ISSN 2734-049X (Online)


https://doi.org/10.3745/KTCCS.2023.12.9.283

# Efficient Stack Smashing Attack Detection Method Using DSLR

Do Yeong Hwang† · Dong-Young Yoo††

## ABSTRACT

With the recent steady development of IoT technology, it is widely used in medical systems and smart TV watches. 66% of software development is developed through language C, which is vulnerable to memory attacks, and acts as a threat to IoT devices using language C. A stack-smashing overflow attack inserts a value larger than the user-defined buffer size, overwriting the area where the return address is stored, preventing the program from operating normally. IoT devices with low memory capacity are vulnerable to stack smashing overflow attacks. In addition, if the existing vaccine program is applied as it is, the IoT device will not operate normally. In order to defend against stack smashing overflow attacks on IoT devices, we used canaries among several detection methods to set conditions with random values, checksum, and DSLR (random storage locations), respectively. Two canaries were placed within the buffer, one in front of the return address, which is the end of the buffer, and the other was stored in a random location in-buffer. This makes it difficult for an attacker to guess the location of a canary stored in a fixed location by storing the canary in a random location because it is easy for an attacker to predict its location. After executing the detection program, after a stack smashing overflow attack occurs, if each condition is satisfied, the program is terminated. The set conditions were combined to create a number of eight cases and tested. Through this, it was found that it is more efficient to use a detection method using DSLR than a detection method using multiple conditions for IoT devices.

Keywords : Stack Smashing, Buffer overflow, Canary, Random Value, XOR Bit

# DSLR을 이용한 효율적인 스택스매싱 공격탐지 방법

황 도 영† · 유 동 영††

## 요 약

최근 IoT 기술이 꾸준하게 발전되면서 의료 시스템, 스마트 TV 시계 등에서 많이 활용되고 있다. 소프트웨어 개발의 66%가 메모리 공격에 취약한 C 언어를 통해 개발되고 C 언어를 사용하는 IoT 기기에 위협적으로 작용한다. 스택스매싱 오버플로 공격은 사용자가 정의한 버퍼 크기보다 큰 값을 삽입하여 반환 주소가 저장된 영역을 덮어쓰게 하여 프로그램이 정상적으로 동작하지 못하게 한다. 메모리 가용량이 적은 IoT 기기는 스택스매싱 오버플로 공격에 취약하다. 또한, 기존의 백신 프로그램을 그대로 적용하게 되면 IoT 기기가 정상적으로 동작하지 못한다. 연구에서는 IoT 기기에 대한 스택스매싱 오버플로 공격을 방어하기 위해 여러 탐지 방법 중 카나리아를 사용하여 각각 무작위 값, 체크썸, DSLR(무작위 저장 위치)로 조건을 설정했다. 2개의 카나리아를 버퍼 내에 배치하여 하나는 버퍼의 끝인 반환 주소 앞에 배치하고 나머지 하나는 버퍼 내 무작위 위치에 저장했다. 이는 고정된 위치에 저장된 카나리아 값은 공격자가 위치를 예측하기 쉬우므로 무작위한 위치에 카나리아를 저장하여 공격자가 카나리아의 위치를 예측하기 어렵게 했다. 탐지 프로그램 실행 후 스택스매싱 오버플로 공격이 발생 후 각 조건을 만족하게 되면 프로그램이 종료된다. 설정한 조건을 각각 조합하여 8가지 경우의 수를 만들었고 이를 테스트했다. 이를 통해 IoT 기기에는 다중 조건을 사용한 탐지 방법보다 DSLR을 이용한 탐지 방법을 사용하는 것이 더 효율적이라는 결과를 얻었다.

키워드 : 스택스매싱 공격, 버퍼 오버플로, 카나리아, 무작위 값, XOR 비트

## 1. 서 론

사물 인터넷(IoT) 기술은 현재까지 꾸준하게 발전되면서 많은 곳에서 활용되고 있다. 특히 환자나 의사의 위치를 실시간으로 알아야 하는 의료 시설에 큰 도움이 되고 있다. Raspberry Pi 기반의 IoT 기기에서 블루투스, WiFi, RFID 등 무선 네트워크를 활용해 GPS 기술로 환자나 의료진의 위치를 추적할 수 있다. 하지만 IoT 장치를 사용하는 시스템의 테스트 및 구축 시 비용 증가로 인해 IoT 보안 기능 구현을 포함하지 않고 있다. 이는 IoT 기기를 통해 수집하는 의료 데이터가 저장된 시스템을 외부 공격에 그대로 노출된다. 보안이 적용되지 않은 IoT 기기는 데이터 유출 및 위조, 무단 원격 접근, 서비스 거부 공격, 스택스매싱 오버플로 등의 공격을 받을 수도 있다

† 준 회 원 : 홍익대학교 산업융합형동과정 석사과정
†† 종신회원 : 홍익대학교 소프트웨어융합과 부교수
Manuscript Received : May 30, 2023
First Revision : July 21, 2023
Accepted : August 11, 2023
* Corresponding Author : Dong-Young Yoo(ydy@hongik.ac.kr)

[1]. NIST 보고서[2]에서는 전체 취약점 중 77%는 운영체제가 아닌 소프트웨어 애플리케이션에서 발생하고, 66%가 메모리 공격에 취약한 C 언어를 통해 개발된 소프트웨어에서 취약점이 발생하게 되고 C 언어를 사용하는 IoT 장치의 취약점으로 이어진다. IoT 펌웨어 및 스마트 시계, 스마트 TV와 같은 IoT 장치에서 사용되는 공개된 코드를 통해 개발된 네트워크 연결 관리자 소프트웨어 등 스택스매싱 오버플로 공격에 노출되어 있다[3]. IoT 기기를 사용하여 시스템을 구축한 사례에서 발생 가능한 스택스매싱 오버플로 공격을 탐지하기 위해 무작위 값과 XOR 비트, DSLR 방법을 이용했다. 스택스매싱 오버플로 공격에 대한 IoT 기기의 취약성을 알아보기 위해 Raspberry Pi와 Ubuntu OS 환경에서 테스트를 진행했다. 2장에서는 스택스매싱 오버플로와 공격 탐지메커니즘 종류 및 카나리아를 활용한 탐지메커니즘에 연구하고 3장에서는 공격탐지를 위한 환경 구축 및 공격 성공 시 메모리의 상태 변화 등을 서술하고 4장에서는 구축한 환경에서 테스트한 실험결과에 대해 분석하였으며 5장에서는 결론과 향후 과제를 도출했다. 본 연구를 통해 스택스매싱 오버플로 공격을 탐지하는 방법 중 IoT 기기에 효율적인 공격탐지 방법이 될 것으로 기대한다.

## 2. 관련 연구

### 2.1 스택스매싱 오버플로 공격

스택스매싱 오버플로 공격은 C 프로그래밍에서 버퍼의 경계를 자동으로 확인하지 않아 버퍼 복사 작업 중 버퍼에 인접한 영역을 덮어쓸 수 있다. Fig. 1은 메시지는 공격자가 입력한 일련의 데이터를 가리키며 공격 데이터는 쉘코드를 포함하여 80byte 이상이다. strcpy() 함수 실행 후 모든 공격 데이터

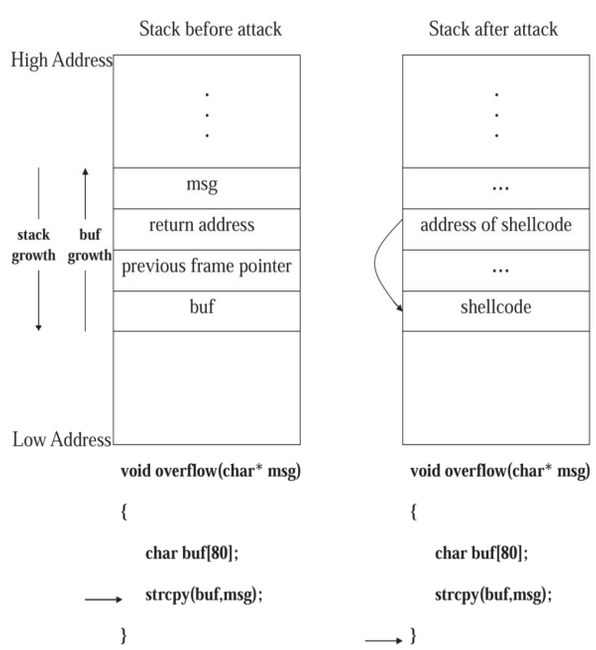


Fig. 1. Stack Smashing Attack[4]

는 스택에 복사되면서 버퍼의 경계를 검사하지 않는다. 이때, return address는 공격 데이터로 덮어지고 프로그램의 제어권이 쉘코드로 넘어가게 된다[4].

스택스매싱 오버플로 공격은 사용 중인 서비스를 중지시키고 공격자가 시스템에 침입하도록 하는 공격 방법이다. 응용 프로그램 메모리 영역의 취약점을 기반으로 한다. 스택스매싱 오버플로 공격은 버퍼의 크기보다 긴 문자열을 버퍼에 강제로 삽입하여 오버플로를 발생시킨다. 이로 인해 버퍼 내에 있는 호출 함수 및 포인터의 반환 주소를 파괴하여 제 역할을 못하게 할 수 있다. 공격자는 기계어 명령어를 문자열로 주입하여 반환 주소를 공격 코드의 주소로 변경하여 실행을 유도하여 공격자가 원하는 대로 정보를 변경하고 궁극적으로 시스템에서 권한 있는 셸을 제어한다[5]. 스택스매싱 오버플로 공격은 데이터가 메모리의 버퍼에 저장되는 버퍼 크기를 확인하지 않는 문제를 악용한다. 공격자는 배열의 끝부분에 저장된 프로그램 상태를 임의로 변경할 수 있다. 스택스매싱 오버플로 공격은 스택에 할당된 버퍼를 공격하여 공격 코드 주입과 반환 주소 변경을 시도한다. 공격 코드 주입 공격은 실행 가능한 입력 문자열, 취약한 함수 사용 등 단순한 코드 또는 shell 파일로 실행된다. 반환 주소 변경의 공격은 반환 주소를 공격 코드로 가리키도록 변경하여 함수가 반환될 때, 호출된 위치로 점프하지 않고 공격 코드로 점프하게 된다. ShadowStack 방식은 반환 주소를 저장하기 위해 사용되는 전용 메모리 공간을 예약하여 함수가 프로세스에 의해 실행될 때마다 반환 주소는 원래의 스택과 ShadowStack에 저장된다. 반환 주소를 공격하는 스택스매싱 오버플로가 발생한 경우, ShadowStack에 저장된 주소의 복사본과 비교하여 공격을 탐지한다[6]. 스택스매싱 오버플로 공격은 버퍼의 경계 검사가 없는 점을 공격하여 발생하게 된다. 버퍼의 경계를 지나 데이터를 추가하면 스택에 저장된 정보를 덮어쓰게 되고 오버플로가 발생하게 되고 반환 주소가 변경되어 함수가 제대로 반환되지 않아 프로그램이 동작하지 않게 된다. 공격자는 이를 악용하여 스택 프레임에 포함된 변수, 포인터 및 반환 주소를 변형할 수 있다. 반환 주소를 덮어쓰는 스택스매싱 공격을 방어하기 위해 스택을 읽기만 가능하게 만들고 버퍼에 새로운 메모리 공간을 할당하여 기존의 반환 주소를 저장하여 보호한다[7].

IoT 장치에서 사용되는 ConnMan은 Wi-Fi, 이더넷, Bluetooth와 같은 플러그인을 사용하여 IoT 장치 연결을 관리할 수 있다. ConnMan은 IoT 장치에서 사용되는 네트워크 응용 프로그램으로 스택 기반의 오버플로 취약점을 포함하고 있어 공격자가 IoT를 원격으로 손상시켜 제어할 수 있다. 2017년 8월을 기점으로 취약점이 패치되었지만 1.34 이하 버전에는 취약점이 존재한다. 따라서, IoT의 제한된 자원에 효율적인 경량화된 탐지 기법이 필요하다[3].

### 2.2 스택스매싱 오버플로 공격 탐지메커니즘

스택스매싱 오버플로 공격은 스택과 힙 영역에서 수행되며 스택 영역에서는 실행 중인 프로그램의 제어 흐름이 변경되도

록 함수 프레임의 제어 정보를 덮어쓴다. 제어 정보에는 반환 주소, 저장된 프레임 포인터 등이 있다. 힙 영역에서는 프로그래머의 메모리 관리 오류로 힙 메타데이터 덮어쓰기, 이중 해제 등이 발생한다. 버퍼 오버플로 공격을 방어하기 위해 알고리즘 중심 및 키 기반 기술로 분류된다. 알고리즘 기술은 스택 스매싱 오버플로 공격에 방어하기 위해 특정 알고리즘에 의존하고 키 기반 기술은 비밀 키를 사용하여 버퍼 오버플로 공격을 실행하지 못하도록 한다. 공격의 대상인 스택을 보호하기 위한 예방 기술로 코드 검사, 주소 범위 확인, 스택 실드, 난독화 기술 등의 기술과 키 기반의 보호 메커니즘에서 스택스매싱 오버플로 공격을 사전에 탐지하기 위해 스택 메모리 내의 반환 주소 앞에 카나리아를 배치하여 스택스매싱 오버플로 공격을 탐지하는 방법이 있다[8]. 네트워크 시스템에서 스택스매싱 오버플로 공격의 원인이 되는 패킷 처리 기능을 악용하는 방법과 OS 기반 라우터에서 사용 가능한 스택 보호 메커니즘에 관하여 설명했다. 네트워크 프로세스에서 스택스매싱 오버플로 공격은 운영체제의 스택을 덮어쓰고 제어 흐름을 악성 코드로 전환하여 메모리 액세스 오류, 프로세스 충돌 등의 원인으로 프로그램이 오작동 될 수 있다. 스택스매싱 오버플로 공격으로부터 보호하기 위해 호출 스택에 임의의 값을 추가하고 함수가 반환되기 전 무작위 스택 카나리아 보호 방법을 통해 스택의 무결성을 검사한다[9]. 메모리 손상으로 이어지는 취약점을 악용하는 스택 손상 및 문자열 공격을 사용하는 방법과 방어하기 위한 stack canary, ASLR(address space layout randomization) 방법을 제안했다. 버퍼 오버플로 공격은 반환 조수를 덮어 스택 카나리아 값을 덮어쓰게 되므로 스택 카나리아 및 ASLR 기능을 통해 공격을 사전에 방지한다. 또한, 메모리 손상 취약성을 완화하는 방법은 이전으로 복구하는 방법이다. 정적 분석, 컴파일러 수정 및 소프트웨어 사용 전 소스 코드 감지, 삭제 등 메모리 취약성을 방지할 수 있는 기술을 사용할 수 있고 다른 방법으로 데이터 유형에 따라 데이터 세그먼트의 데이터를 분리하고 보호 페이지를 통해 분리 영역을 격리하는 방법과 DEP(Data Execution Prevention) 메커니즘을 사용하여 특정 메모리 위치만 실행 권한을 갖도록 허용하는 방법도 있다[10]. DSLR(Data Structure Layout Randomization)은 프로그램 데이터 구조(함수에서 선언된 구조체, 클래스 및 스택 변수)를 무작위화하는 방법이다. 스택스매싱 오버플로를 시도하는 공격자는 프로그램 버퍼가 함수 포인터 또는 반환 주소에 인접해 있다는 것을 알고 있다. 공격 시 데이터 구조 및 프로토콜 메시지의 구조를 알고 있으면 퍼즈 공간을 줄이고 공격 침투 속도를 높일 수 있다. 공격자가 프로그램의 정확한 데이터 구조 레이아웃을 얻지 못하도록 막기 위해 레이아웃과 데이터 구조를 무작위화하여 공격을 방어한다[11].

스택스매싱 오버플로 공격으로부터 서버를 보호하기 위해 UNIX 기반의 보호 기술에 대하여 분석했다. 서버 내 악성 프로그램이 스택에 있는 버퍼에 할당된 양보다 더 많은 데이터를 사용할 때 공격이 발생한다. 서버를 보호하기 위한 보호 기술로 NX/DEP, ASLR, SSP를 적용했다. 메모리 영역은 읽기 전용으로 쓰기 및 실행 명령이 불가능하다. 하지만 실행 권한 강제로 가져오는 악성 코드를 삽입하여 메모리 영역의 실행 권한을 얻을 수 있다. NX/DEP 기술은 메모리 영역에 코드를 성공적으로 삽입해도 쓰기 기능과 실행 기능을 동시에 사용할 수 없게 한다. ASLR 기술은 프로그램 코드 또는 라이브러리의 주소를 알아내어 프로세스를 충돌시킬 수 있으므로 프로세스가 메인 메모리에 로드될 때마다 운영체제는 프로세스의 다양한 영역(스택, 힙 영역 등)을 호출하여 침입을 방지한다. SSP 기술은 스택에 저장된 반환 주소를 덮어쓰는 시도를 스택에 배치된 카나리아 값이 함수 끝에서 확인하여 초기의 값과 일치하지 않을 때 프로그램이 중단된다[12].

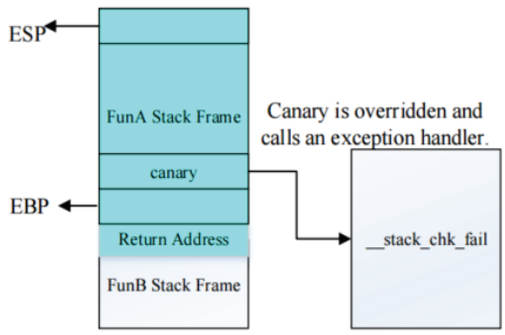


Fig. 2. Stack Overflow State in Canary[13]

Fig. 2에서 Linux 시스템은 스택스매싱 오버플로 방어 수단으로 카나리아 보호 방법을 사용한다. 함수가 스택에 추가될 때 EBP 앞에 임의의 숫자가 추가된다. 이때, 난수를 카나리아라고 한다. 함수가 반환되기 전에 스택의 카나리아가 이전에 생성된 카나리아 값과 일치하는지 비교한다. 일치하지 않으면 스택스매싱 오버플로가 발생한 것으로 판단하고 __stack_chk_fail 함수를 호출하여 예외 처리를 수행한다[13].

### 2.3 카나리아 탐지 방법

기존의 스택 카나리아 값은 프레임 포인터(FP)와 반환 주소 사이에 배치하는 구조다. Window Canary는 스택 카나리아 방식과 유사하게 반환 주소가 수정되지 않도록 보호하기 위해 canary 단어를 지역변수와 반환 주소 사이에 배치하여 버퍼 오버플로 공격을 탐지한다. 또한, 카나리아 단어를 처리하기 위해 오버플로 및 언더플로 예외를 사용하게 되면 카나리아 단어를 배치/확인하고 레지스터를 복원하는 사이에 시간이 흘러 공격자가 반환 주소를 조작할 수도 있다. 공격을 방지하기 위해 카나리아 단어 배치/확인 및 반환 주소 배치/확인 과정은 동시에 발생해야 한다. 공격을 감지하고 카나리아 단어가 변조가 감지되면 프로그램은 레지스터를 준비하기 위해 인터럽트 컨텍스트를 종료하기 전에 다른 위치로 점프하게 된다[14].

스택 카나리아는 스택스매싱 오버플로 공격에 대해 사용되는 탐지 기술 중 하나다. 시스템 호출 과정에서 공격자가 커널 스택 카나리아를 변경하여 카나리아 정보를 얻을 수 없

도록 프로그램을 구현하여 ARM64 및 x86_64 Linux 커널용과 Android HiKey960 Linux 커널에서 테스트했다. 사전에 공격자는 커널 메모리를 읽기 기능과 Meltdown, Specter 같은 메모리 취약점 공격을 실행하여 커널 내 모든 메모리 정보를 읽을 수 있다는 조건을 부여했다. 추가로, linux 커널 난수 생성기를 통해 무작위 카나리아 값을 생성하고 공격자는 다음 난수를 예측할 수 없다. 카나리아 불일치 확인을 위해 스택의 모든 카나리아를 배치하여 사용 및 업데이트하게 되면 오버헤드 발생이 커지게 되므로 모니터 카운터 레지스터(PMC) 시스템 레지스터를 무작위 소스로 사용하여 PMC 레지스터에서 값을 가져온 후 소수의 명령어로 카나리아 업데이트 로직을 구현하여 탐지했다[15]. 버퍼 오버플로 취약점을 악용한 스택 스매싱 오버플로 공격을 무작위 카나리아 값으로 탐지하는 RCR(Random Canaries Repository) 접근 방식을 제안했다. RCR 방식은 취약한 응용 프로그램을 보호하기 위해 프로그램이 실행하는 동안 무작위 카나리아 값 저장소를 생성한다. 카나리아 값을 포함하는 저장소는 세 개의 복사본을 생성하고 복사본을 읽기 전용 메모리에 저장한다. 함수가 호출되면 반환 주소와 프레임 포인터를 스택으로 push 후 무작위 카나리아 값이 설정된다. 이를 통해, RCR은 낮은 오버헤드를 사용하여 스택스매싱 오버플로 공격을 방지하고 공격자가 카나리아 값을 추정하기 더 어렵게 만들었다[16].

서버에서 스택스매싱 오버플로 공격은 시스템 내 스택에 할당된 버퍼를 공격하여 이를 감지하기 위해 스택 프레임의 반환 주소 뒤에 카나리아 값을 배치한다. 부모와 자식 프로세스는 모두 같은 카나리아 값을 가지게 된다. 서버 프로세스의 분기된 서버에서 공격자는 자식 프로세스에 무차별 대입 공격을 통해 카나리아 값을 예측한다. 단일 바이트를 오버플로 카나리아의 바이트 값을 x로 덮어쓴다. x가 정확하면 서버가 충돌하지 않고 8개의 카나리아 바이트가 일치할 때까지 계속된다. 무차별 대입 공격을 막기 위해 DiffGuard 프로세스를 제안했다. DiffGuard는 자식 프로세스에서 실행 명령 직전에 부모 프로세스로부터 상속받은 카나리아 값을 업데이트 후 실행하게 하여 바이트 단위 무차별 대입 공격을 실행 불가능하게 한다[17]. 안드로이드 OS 환경에서는 스택스매싱 오버플로 공격을 방어하는 SSP 기술의 효율성이 떨어진다고 설명한다. 구글 플레이 스토어에 악성 애플리케이션 업로드, 카나리아 값 노출 등의 원인으로 공격자가 오버플로 공격을 통해 반환 주소를 변경하고 악성 코드를 실행하는데 사용될 수 있다. 안드로이드 환경에서 발생하는 스택스매싱 오버플로 공격을 방어하기 위해 카나리아 값을 버퍼 앞에 배치했다. 무작위로 발생한 참조 카나리아 값은 함수 프레임에 복제된다. 함수 실행 후 완료 시 프레임 카나리아 값을 참조 카나리아 값과 비교하여 오버플로 공격이 발생했는지 탐지한다. 카나리아 단어가 고정된 값일 경우 예측을 통해 값을 추출하기 쉬우므로 참조 카나리아 값을 무작위로 생성하여 구현했다[18].

Ubuntu에 배포된 방어 메커니즘 기능을 분석하기 위해 ARM 기반 RISPE-ARE(Runtime Intrusion Prevention Evaluator)를 구현하여 테스트했다. ARM 아키텍처에 대한 850개의 버퍼 오버플로 공격을 공격 방법, 취약 함수 등 다양한 조합에 따라 방어율을 ARM 기반 시스템인 라즈베리 파이에서 진행했다. RIPE-ARM에 850개의 버퍼 오버플로 공격을 사용하여 라즈베리 파이에 적용된 방어 메커니즘을 평가했다. 보호 기능이 없을 때, 전체 중 144의 공격이 성공했고 카나리아와 DEP 기능을 하나씩 사용하여 방어하게 되면 각각 95번, 12번의 공격이 성공했다. 마지막으로 카나리아와 DEP 두 기능 모두 활성화했을 경우 850개의 공격 중 10가지 공격만 성공했다. 또한, 비교 평가를 위해 우분투 시스템의 방어 메커니즘에서도 테스트하여 전체 중 60가지의 버퍼 오버플로 공격 방법만 성공했다[19].

## 3. 연구 내용

본 연구에서는 버퍼의 크기를 모르는 공격자가 무작위로 버퍼 크기를 삽입하여 스택스매싱 오버플로 공격을 시도하는 것을 가정하여 2개의 카나리아를 버퍼에 무작위 값으로 생성하고 공격 전과 후의 값을 비교하여 스택스매싱 공격을 탐지했다. 첫 번째 카나리아는 버퍼의 마지막 주소값인 7ffff7dd0530에 배치해서 버퍼의 크기보다 더 큰 값이 입력되는지를 탐지한다. 두 번째 카나리아는 프로그램이 실행될 때마다 무작위로 버퍼 내에 저장되어 스택스매싱 공격 시도가 있는지 탐지한다. 2가지 카나리아 모두 공격을 탐지할 경우, 실행한 프로그램을 강제로 종료시킨다.

버퍼의 끝에 배치된 카나리아는 함수 포인터나 반환 주소에 인접하기 때문에 공격자가 저장된 위치를 예측할 수 있다. 무작위 값으로 공격을 시도하는 공격자는 공격 시 탐지되지 않는 범위에 카나리아가 있는 것을 알 수 있고 계속 공격을 시도하여 범위를 줄이고 카나리아가 저장된 위치를 알아내어 이를 우회하여 공격할 수도 있다. 공격에 성공한 공격자는 데이터 및 프로토콜 메시지 구조를 알 수 있다. 스택스매싱 오버플로 공격을 통해 구조를 파악한 공격자는 해당 시스템 또는 IoT 기기에 악성 프로그램 침투 테스트에서 퍼즈 공간을 줄이고 테스트 속도를 높일 수 있다. 공격자가 스택스매싱 오버플로 공격을 통해 데이터 구조를 파악하지 못하도록 DSLR 방법을 카나리아 탐지 방법에 적용했다. 고정된 위치에서 공격을 탐지하게 되면 위치가 예측되고 카나리아를 우회할 수 있다. 공격자가 카나리아가 저장된 위치를 예측하지 못하도록 탐지 프로그램이 실행될 때마다 무작위 위치에 카나리아가 배치되도록 했다. 공격자는 매번 변경되는 카나리아 위치를 예측하기 어려워 데이터 구조 레이아웃 등을 파악하지 못하게 할 수 있다. 하지만 DSLR 방법은 데이터 구조 정의가 공용 라이브러리를 통해 정의된 경우에는 무작위화할 수 없고 네트워크 통신에 사용되는 데이터 구조의 경우, 무작위화되면 통신 당사자 간 구조가 달라 서로 데이터를 이해하지 못하는 단점이 있다.

```
test@ubuntu:~/test2$ xxd test11.c
00000000: 2369 6e63 6c75 6465 203c 7374 6469 6f2e  #include <stdio.
00000010: 683e 0a23 696e 636c 7564 6520 3c73 7464  h>.#include <std
00000020: 6c69 622e 683e 0a23 696e 636c 7564 6520  lib.h>.#include
00000030: 3c6d 656d 6f72 792e 683e 0a23 696e 636c  <memory.h>.#incl
00000040: 7564 6520 3c73 7472 696e 672e 683e 0a23  ude <string.h>.#
00000050: 696e 636c 7564 6520 3c74 696d 652e 683e  include <time.h>
00000060: 0a0a 766f 6964 2066 756e 6374 696f 6e28  ..void function(
00000070: 6368 6172 202a 7374 7229 207b 0a20 0a20  char *str) {. .
00000080: 2020 2020 6368 6172 2062 7566 6665 725b      char buffer[
00000090: 3230 5d3b 0a20 2020 2020 6d65 6d73 6574  20];.     memset
000000a0: 2862 7566 6665 722c 2027 4127 2c20 7369  (buffer, 'A', si
000000b0: 7a65 6f66 2873 7472 2929 3b0a 2020 2020  zeof(str));.
000000c0: 2073 7472 6370 7928 6275 6666 6572 2c20   strcpy(buffer,
000000d0: 7374 7229 3b0a 0a7d 0a0a 696e 7420 6d61  str);..}..int ma
000000e0: 696e 2876 6f69 6429 207b 0a0a 2020 2020  in(void) {..
000000f0: 696e 7420 6920 3d20 303b 0a20 2020 2073  int i = 0;.    s
00000100: 7261 6e64 2874 696d 6528 4e55 4c4c 2929  rand(time(NULL))
00000110: 3b0a 2020 2020 696e 7420 6d61 785f 7369  ;.    int max_si
00000120: 7a65 203d 2028 7261 6e64 2829 2025 2033  ze = (rand() % 3
00000130: 3029 202b 2031 303b 0a20 2020 2063 6861  0) + 10;.    cha
00000140: 7220 6c61 7267 655f 7374 7269 6e67 5b6d  r large_string[m
00000150: 6178 5f73 697a 655d 3b0a 0a20 2020 2070  ax_size];..    p
00000160: 7269 6e74 6628 226c 6172 6765 5f73 7472  rintf("large_str
00000170: 696e 6720 7369 7a65 203a 2025 6c6e 6e22  ing size : %d\n"
00000180: 2c20 6d61 785f 7369 7a65 293b 0a0a 2020  , max_size);..
00000190: 2020 666f 7228 693d 303b 2069 203c 206d    for(i=0; i < m
000001a0: 6178 5f73 697a 653b 2069 2b2b 297b 0a20  ax_size; i++){.
000001b0: 2020 2020 2020 206c 6172 6765 5f73 7472          large_str
000001c0: 696e 675b 695d 203d 2027 4127 3b0a 2020  ing[i] = 'A';.
000001d0: 2020 7d0a 2020 2020 6675 6e63 7469 6f6e    }.    function
000001e0: 286c 6172 6765 5f73 7472 696e 6729 3b0a  (large_string);.
000001f0: 0a20 2020 2072 6574 7572 6e20 303b 0a7d  .    return 0;.}
00000200: 0a                                       .
```

Fig. 3. Stack Smashing Attack Code Memory Dump Screen

Fig. 3은 스택스매싱 공격 코드를 Hex dump를 통해 16진수로 분석한 화면이다. 선행 연구에서 strcpy() 함수가 버퍼의 경계를 검사하지 않고 실행되어 프로그램에 취약하다는 것을 알아냈다. main 함수에서 무작위 버퍼 값을 생성하기 위해 rand() 함수를 사용하여 10에서 30 사이의 난수를 발생시킨다. 발생한 난수는 max_size 변수에 저장되고 srand() 함수에 의해 프로그램을 실행할 때마다 값이 변하게 된다. for 문에서 무작위로 발생한 max_size 값만큼 large_string[i] 배열에 문자 A를 저장하고 function 함수를 호출한다. 함수 function에서 크기가 20인 문자형 배열 buffer를 생성하고 memset 함수를 사용하여 입력받은 str 크기만큼 배열 buffer에 문자 A를 저장한다.

스택스매싱 오버플로 공격을 발생시키기 위해 strcpy 함수를 사용하여 str에 저장된 문자열을 배열 buffer에서 복사한다. 이때, 배열 buffer 크기를 확인하지 않고 실행되기 때문에 스택스매싱 오버플로가 발생한다.

스택스매싱 오버플로 공격을 사전에 탐지하고 차단하기 위해 카나리아 값을 활용하여 테스트를 진행했다. 공격자는 사용자의 버퍼 크기를 모르고 무작위 값을 생성하여 공격하는 상황을 가정했다. 공격자는 스택스매싱 오버플로를 발생시켜 프로그램을 동작하지 못하게 하는 목적도 있지만, 반환 주소 정보를 탈취하여 자신의 악성 코드 주소로 변형할 수 있다. 공격자는 무작위 값으로 스택스매싱 오버플로 공격을 시도하여 버퍼의 경계를 알아내어 반환 주소가 어디에 저장되어 있는지 알아낸다. Fig. 4는 공격을 탐지하기 위해 카나리아 값을 무작위로 생성하여 버퍼 끝과 버퍼 내에 저장했다. 버퍼 끝에 저장

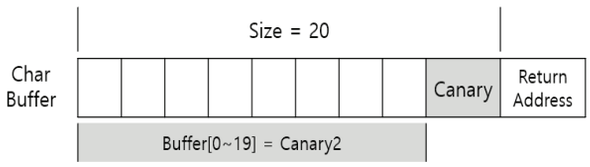


Fig. 4. Character Array Buffer Structure

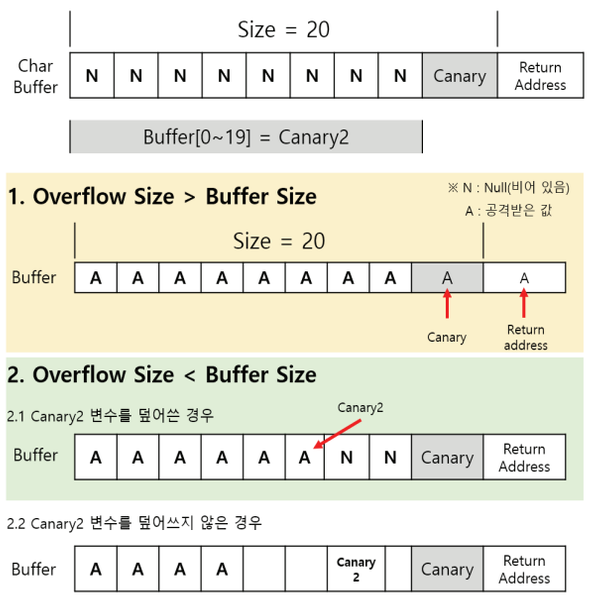


Fig. 5. Stack Smashing Overflow Attack Attempt

한 카나리아 값은 버퍼의 경계를 넘어서 반환 주소를 덮는 공격을 탐지하고 차단하기 위해 배치했다. 나머지 카나리아 값은 버퍼 내 공간에 무작위 배열에 저장되어 버퍼 오버플로 공격 시도가 있는지 탐지한다. 공격자는 무작위 값으로 공격하기 때문에 오버플로 공격이 버퍼의 경계를 넘지 못하는 경우도 발생한다. 그럴 경우, 버퍼 내의 값만 변경되고 배열 끝에 있는 카나리아가 탐지하지 못한다. 그래서 카나리아를 하나 더 배치하여 공격 시도가 있는지 탐지하도록 했다.

Fig. 5는 무작위 값으로 스택스매싱 오버플로 공격을 시도했을 때의 경우를 나타낸다. 버퍼의 크기보다 오버플로 크기가 더 클 경우, 카나리아 값과 반환 주소가 저장된 영역까지 값을 덮어쓰게 된다. 이때, 카나리아 영역에서 값이 변조가 일어났음을 확인 후 사용자에게 알리고 프로그램을 종료한다. 다른 경우는 버퍼의 크기보다 오버플로 크기가 더 작을 경우, Canary 변수가 공격을 탐지하지 않고 무작위 버퍼 공간 내 생성된 Canary2 변수가 공격을 탐지한다. Canary2 변수의 저장 위치와 오버플로 공격 크기에 따라 탐지하는 경우가 다르다. 오버플로 공격 크기가 Canay2 변수가 저장된 배열보다 클 때, 공격이 발생했음을 알리고 저장된 배열보다 작을 때 공격이 발생하지만 이를 탐지하지는 못한다.

## 4. 연구 결과

앞서 설명한 연구 내용을 증명하기 위해 스택스매싱 오버플로 공격탐지 코드를 Raspberry Pi와 Ubuntu 16.04 LTS 환경에 구현했다.

스택스매싱 오버플로 공격을 탐지하기 위해 Fig. 6과 같이 3가지 탐지 조건을 부여했다. 첫 번째는, 버퍼의 크기보다 오버플로 크기가 더 큰 경우, 공격 이전에 생성된 무작위 카나리아 값과 공격 이후 카나리아가 저장된 버퍼의 값과 비교하여 탐지한다. 두 번째는, XOR 비트를 활용해 공격 이후 Canary 변수의 값과 Canary2 변수의 값이 같으면 0, 다르면 1을 출력하고 출력된 값이 1일 때, 스택스매싱 오버플로 공격을 탐지한다. 마지막으로 버퍼의 크기보다 오버플로 크기가 작을 때, 공격 이전에 생성된 Canary2 변수의 값과 Canary2 변수가 저장된 버퍼의 값과 비교하여 공격 시도가 있음을 탐지한다. 제안한 3가지 조건을 조합하여 8가지 경우의 수를 실험하였다.

스택스매싱 오버플로 공격이 탐지되는 경우를 다음과 같이 분류했다. [Case 0]의 경우, 모든 보호 기능을 해제하고 스택스매싱 오버플로 공격을 진행했다. 아무런 보호 기능 없이 공격에 쉽게 노출된다. [Case 1]은 무작위로 생성한 카나리아 값이 공격으로 인해 값이 변형되는지를 탐지했다. 버퍼의 크기를 넘는 공격을 받을 때 탐지가 되고 실험에서는 50회의 공격 중 26회의 공격을 탐지했다. [Case 2]는 두 개의 카나리아 값을 XOR 연산을 통해 값이 일치하는지 확인했다. XOR 연산 결과에 대한 예외 처리를 하지 않아 값이 일정하게 나오지 않았기 때문에 50회의 공격 중 10회의 공격만을 탐지했다. [Case 3]은 DSLR을 이용하여 버퍼 내에 무작위로 배치했다. [Case 1]과 반대로 버퍼의 크기를 넘지 않을 때 무작위로 배치된 카나리아 값이 공격 이후 변형되는지 탐지했다. 공격 크기와 카나리아의 위치에 따라 탐지가 될 수도 안될 수도 있다. 실험에서 50회의 공격 중 36회의 공격을 탐지했다. [Case 4]는 무작위 값과 checksum, [Case 5]는 무작위 값과 DSLR, Case 6은 checksum과 DSLR로 탐지했다. [Case 7]은 무작위 값, checksum, 무작위 위치로 모든 조건을 사용하여 공격을 탐지하도록 했다. 오버헤드는 탐지 로직이 독립적으로 실행하진 않았으며, [Case 1~7]의 실험과 같이 갈수록 오버헤드 값이 커지는 것을 확인할 수 있다. (Fig. 7)은 [Case 7]의 탐지 결과 화면이다. [Case 1]에서 Canary 변수값이 일치하지 않았고 [Case 3]에서 Canary2 변수값도 일치하지 않아 스택스매싱 공격을 탐지할 수 있었다. 총 50회의 스택스매싱 공격을 시도 후 이를 탐지한 테스트 결과는 Table 1과 같다. 스택스매싱 공격탐지 테스트 전의 예상 결과는 모든 Case가 공격탐지율 50% 이상으로 예측했다. 하지만 [Case 2]의 경우, 20%밖에 탐지하지 못했고 [Case 1]의 경우, 탐지율이 50%를 넘었지만, 결과가 우수하다고는 할 수 없다. 마지막 [Case 3]은 3가지 조건 중 탐지율 72%라는 결과를 얻어, 테스트한 조건 중 가장 효율적인 것으로 확인되었다.

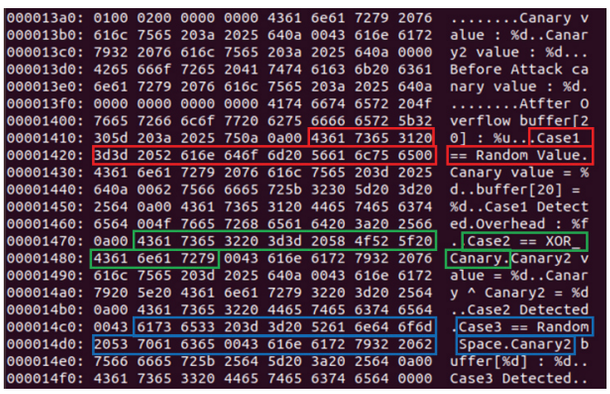


Fig. 6. Stack Smashing Overflow Attack Detection Conditions

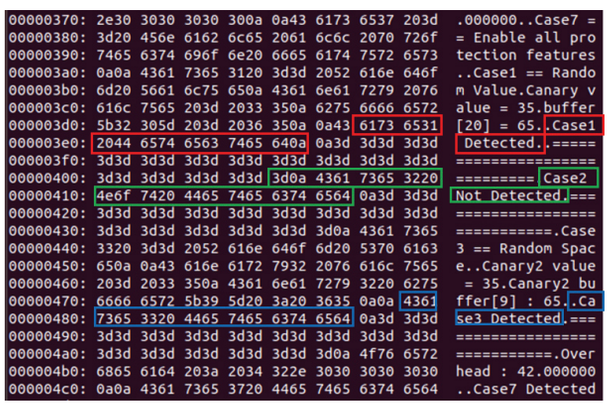


Fig. 7. Detection Screen after all Protection is Enabled

Table 1. Stack Buffer Overflow Test Results

| | Runtime Overhead(s) | Number of detections | Number of defenses |
|---|---|---|---|
| Case 0 | 0 | 0% | 0% |
| Case 1 | 0.220 | 52% | 52% |
| Case 2 | 0.280 | 20% | 20% |
| Case 3 | 0.368 | 72% | 72% |
| Case 4 | 0.469 | 72% | 72% |
| Case 5 | 0.566 | 72% | 72% |
| Case 6 | 0.683 | 72% | 72% |
| Case 7 | 0.775 | 72% | 72% |

Fig. 8은 카나리아의 무작위 위치에 따른 탐지 횟수를 분포도로 나타냈다. 가로축은 카나리아가 저장된 버퍼의 위치이고 세로축은 스택스매싱 공격을 시도한 공격 크기다. 원의 크기는 탐지 횟수를 나타내며 원의 크기가 클수록 탐지 횟수가 많다. 총 200번의 공격 시도 중 탐지한 횟수는 140회이며 나머지 60회는 공격을 탐지하지 못했다. 이는 카나리아 위치가 뒤쪽에 있고 공격 크기가 위치값 보다 작았기 때문에 탐지가 안된 것으로 추정한다. Fig. 8을 통해 카나리아가 버퍼의 앞쪽에 배치될수록 탐지 횟수가 많아지고 더 효율적으로 탐지되는 것을 알 수 있다.

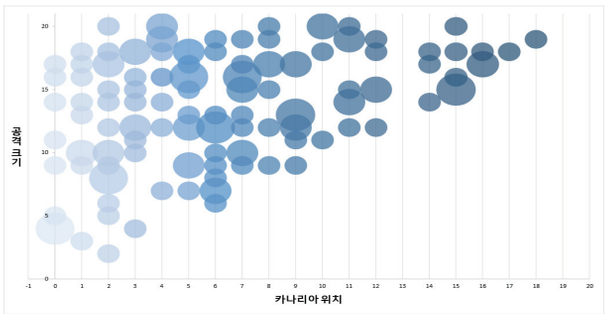


Fig. 8. Detection Distribution Plot by Canary Location

## 5. 결 론

본 연구에서는 IoT 기기에서 발생할 수 있는 스택스매싱 오버플로 공격을 효율적으로 탐지하는 방법을 연구했다. IoT 기기는 PC보다 상대적으로 성능이 낮아, 기기에서 성능 저하가 발생하지 않는 공격탐지 방법이 요구되고 있어, 본 연구에서는 기존에 연구된 탐지 방법 중 카나리아 탐지 방법을 선택했다. 카나리아 탐지 방법은 반환 주소 앞에 배치되어 스택스매싱 오버플로 공격으로부터 반환 주소가 변형되지 못하게 했다. 이는 공격자가 지속해서 공격하여 카나리아 위치를 예측할 수 있다. 카나리아 위치를 파악한 공격자는 이를 우회하여 공격을 시도할 수도 있다. 따라서, 공격자가 카나리아 위치를 예측할 수 없게 실험을 통하여 효율적인 스택스매싱 공격탐지 방법을 확인하였다.

IoT 기기에서 스택스매싱 오버플로 취약점과 탐지율을 높이기 위해 무작위 값과 Checksum을 활용해 8가지 경우를 테스트했다. 스택스매싱 오버플로 공격을 탐지하는 대표적인 방법인 카나리아 값을 활용했다. 탐지 효율성을 측정하기 위하여, 탐지 조건을 추가하여 공격을 탐지하면 탐지율을 높일 수 있을 것이라 예상하였으나. 2가지 이상의 탐지 조건을 부여하여 탐지할 경우, 탐지 효율성이 증가하지 않았다. 따라서, IoT 기기에서는 2가지 조건의 스택스매싱 오버플로 공격을 탐지하는 것이 효율성을 최대화 할 수 있는 방법으로 판단된다. 향후 연구는 카나리아 탐지 영역을 우회하거나, 인증을 회피할 수 없는 해시 함수 등을 활용해서 탐지 능력을 향상할 계획이다.

## References

[1] C. Bradley, S. El-Tawab, and M. H. Heydari, "Security analysis of an IoT system used for indoor localization in healthcare facilities," *In 2018 Systems and Information Engineering Design Symposium*, pp.147-152, 2018.

[2] P. Black, M. Badger, B. Guttman, and E. Fong, "Dramatically reducing software vulnerabilities: Report to the white house office of science and technology policy," *No. NIST Internal or Interagency Report (NISTIR) 8151 (Draft), National Institute of Standards and Technology*, 2016.

[3] K. V. English, I. Obaidat, and M. Sridhar, "Exploiting memory corruption vulnerabilities in connman for iot devices," *In 2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pp.247-255, 2019.

[4] C. M. Chen, S. M. Chen, W. C. Ting, C. Y. Kao, and H. M. Sun, "An enhancement of return address stack for security," *Computer Standards & Interfaces*, Vol.38, pp.17-24, 2015.

[5] H. Etoh and K. Yoda, "ProPolice: Protecting from stack-smashing attacks," *Technical Report, IBM Research Division*, Tokyo Research Laboratory, 2000.

[6] R. S. Ferreira and F. Vargas, "ShadowStack: A new approach for secure program execution," *Microelectronics Reliability*, Vol.55, pp.2077-2081, 2015.

[7] D. Jiang and J. Mai, "A new approach against stack overrun: Separates the stack to two parts," *In 2011 First International Conference on Instrumentation, Measurement, Computer, Communication and Control*, pp.441-444, 2011.

[8] N. R. Kisore, "A qualitative framework for evaluating buffer overflow protection mechanisms," *International Journal of Information and Computer Security*, Vol.8, No.3, pp.272-307, 2016.

[9] P. Wu and T. Wolf, "Stack protection in packet processing systems," *In 2014 International Conference on Computing, Networking and Communications (ICNC)*, pp.53-57, 2014.

[10] H. M., Gisbert and I. Ripoll, "On the effectiveness of nx, ssp, renewssp, and aslr against stack buffer overflows," *In 2014 IEEE 13th International Symposium on Network Computing and Applications*, pp.145-152, 2014.

[11] Z. Lin, R. D. Riley, and D. Xu, "Polymorphing software by randomizing data structure layout," *In Detection of Intrusions and Malware, and Vulnerability Assessment: 6th International Conference*, DIMVA 2009, Como, Italy, Vol.6, pp.107-126, 2009.

[12] Q. Zhou, H. Dai, L. Liu, K. Shi, J. Chen, and H. Jiang, "The final security problem in IOT: Don't count on the canary!," *In 2022 7th IEEE International Conference on Data Science in Cyberspace (DSC)*, pp.599-604, 2022.

[13] N. Huang, S. Huang, and Z. Deng, "Automatic detection of stack overflow attack in canary," *In 2018 Eighth International Conference on Instrumentation & Measurement, Computer, Communication and Control (IMCCC)*, pp.1418-1423, 2018.

[14] K. Lehniger and P. Langendörfer, "Window Canaries: Rethinking stack canaries for architectures with register windows," *IEEE Transactions on Dependable and Secure Computing*, 2022.

[15] J. Sun, X. Zhou, W. Shen, Y. Zhou, and K. Ren, "PESC: A per system-call stack canary design for Linux kernel," *In Proceedings of the Tenth ACM Conference on Data and Application Security and Privacy*, pp.365-375, 2020.

[16] D. A. H. Shehab and O. A. Batarfi, "RCR for preventing stack smashing attacks bypass stack canaries," *In 2017 Computing Conference*, pp.795-800, 2017.

[17] J. Zhu, W. Zhou, Z. Wang, D. Mu, and B. Mao, "Diffguard: Obscuring sensitive information in canary based protections," *In Security and Privacy in Communication Networks: 13th International Conference*, SecureComm 2017, Vol.13, pp.738-751, 2018.

[18] H. Marco-Gisbert and I. Ripoll-Ripoll, "SSPFA: Effective stack smashing protection for Android OS," *International Journal of Information Security*, Vol.18, No.4, pp.519-532, 2019.

[19] S. Zhou and J. Chen, "Experimental evaluation of the defense capability of arm-based systems against buffer overflow attacks in wireless networks," *In 2020 IEEE 10th International Conference on Electronics Information and Emergency Communication (ICEIEC)*, pp.375-378, 2020.

**황 도 영**

https://orcid.org/0009-0009-0565-0233
e-mail : hdy@g.hongik.ac.kr
2020년 동의대학교 컴퓨터공학부(학사)
2021년 ~ 현 재 홍익대학교
산업융합형동과정 석사과정
관심분야 : IoT 취약점 탐지, 보호 메커니즘

**유 동 영**

https://orcid.org/0000-0002-8231-5203
e-mail : ydy@hongik.ac.kr
2011년 고려대학교 컴퓨터학과(박사)
2014년 ~ 2018년 숭실대학교
정보과학대학원 겸임교수
2000년 ~ 2019년 한국인터넷진흥원 단장
2021년 ~ 현 재 홍익대학교 소프트웨어융합과 부교수
관심분야 : 정보보호, 융합보안, IoT, 블록체인 등