KIPS Tr. Comp. and Comm. Sys.
Vol.6, No.1 pp.43~50 pISSN: 2287-5891



https://doi.org/10.3745/KTCCS.2017.6.1.43

# A Study on the Remove Use-After-Free Security Weakness

Yong Koo Park† · Jin Young Choi††

## ABSTRACT

Use-After-Free security problem is rapidly growing in popularity, especially for attacking web browser, operating system kernel, local software. This security weakness is difficult to detect by conventional methods. And if local system or software has this security weakness, it cause internal security problem. In this paper, we study ways to remove this security weakness in software development by summarize the cause of the Use-After-Free security weakness and suggest ways to remove them.

**Keywords : Use-After-Free, Security Weakness, Vulnerability, Dangling Pointer, Nullification**

# 소프트웨어 개발단계 Use-After-Free 보안약점 제거방안 연구

박 용 구† · 최 진 영††

## 요 약

최근 컴퓨터 시스템 내부에 존재하는 웹 브라우저, 운영체제 커널 등에서 Use-After-Free 보안문제가 지속적으로 발생하고 있다. 해당 보안약점은 기존의 보안약점 탐지방법으로 제거하기 어려우며 소프트웨어 내부에 해당 보안약점이 존재할 경우 내부 보안에 심각한 영향을 미친다. 본 논문에서는 소프트웨어 개발 과정에서 해당 보안약점을 제거하기 위한 방안을 연구하였다. 이 과정에서 해당 보안약점의 발생 원인을 정리하고 이를 제거하기 위한 방법을 제시한다.

**키워드 : Use-After-Free, 보안약점, 취약점, 허상포인터, 무효화**

## 1. 서 론

현대 사회에서 많은 사람들에 의해 사용되고 있는 컴퓨터 시스템은 개인이나 특정 단체에 의하여 지속적으로 공격대상이 되어왔다[1]. 특히, 해당 시스템에서 동작하는 커널이나 웹 브라우저, 문서 편집용 소프트웨어 등의 보안약점을 이용한 공격들은 소프트웨어의 정상 동작을 방해하여 계정의 권한 상승 및 제어 흐름 변경과 같은 행동을 일으켜 내부 정보 유출과 같은 보안 문제를 일으킨다.

컴퓨터 시스템 공격에 사용되는 보안약점은 2000년대 후반까지 버퍼(스택, 힙) 영역을 대상으로 진행되었다[2]. 이에 비하여 2000년대 후반부터 최근까지 Use-After-Free (UAF) 보안약점을 이용한 공격이 주를 이루고 있다. 2008년 이후 UAF를 이용한 공격은 매년 약 2배씩 성장하고 있다[3]. UAF의 경우 69%가 웹 브라우저에서 발생하였으며 21%가 운영체제 내부에서 발생하였다[4]. 해당 분야에서 90%의 발생빈도를 보이는 이유는 웹 브라우저 혹은 운영체제와 관련된 소프트웨어 및 커널 부분은 복잡한 데이터 구조와 효율적인 메모리 관리를 위해 메모리 오염 문제에 취약한 C/C++ 언어로 작성되어 있기 때문이다.

UAF 보안약점은 기존의 버퍼 오버플로우와 같은 보안문제에 비하여 탐지하기 어렵다는 특징이 있다[5]. 기존의 보안약점의 경우 패턴분석 등과 같은 방법을 이용하여 정적·동적 분석하여 상당부분 제거할 수 있다. 하지만 UAF의 경우 소프트웨어 내부에서 메모리를 할당하거나 해제하는 과정에서 문제가 발생하기 때문에 실행환경, 방법, 입력 값 등 복합적인 분석이 요구되었다. 따라서 기존의 정적·동적 분석 방법으로 확인하기에 어려움이 있다.

본 논문에서는 UAF 보안약점이 다른 보안약점과 마찬가지로 개발과정에서 프로그래머의 코딩 실수로 발생한다는 점에 집중하였다[4]. 따라서 UAF를 통한 보안문제가 발생하

† 준 회 원 : 고려대학교 정보보호대학원 정보보호학과 석사과정
†† 정 회 원 : 고려대학교 정보보호대학원 정보보호학과 교수
Manuscript Received : September 26, 2016
First Revision : October 12, 2016
Accepted : October 31, 2016
* Corresponding Author : Jin Young Choi(choi@formal.korea.ac.kr)

Table 1. Classification of Analysis about UAF Attack or Detection

| Analysis | Software Development Process | | | | |
|---|---|---|---|---|---|
| | Requirement | Design | Coding | Testing | Maintenance |
| Feist [6] | | | | | O |
| Zhang [4] | | | | O | |
| Lee [7] | | | | O | |
| Xu [8] | | | | O | |
| Caballero [3] | | | | | O |
| This Paper | | | O | | |

는 원인을 파악하고 개발과정에서 해당 원인을 제거할 수 있는 방안을 연구하였다.

본 논문은 총 5부분으로 구성되어 있다. 2장에서는 분석하고자 하는 UAF 보안약점에 대하여 기존에 진행된 연구와 개발과정에서 해당 보안약점 제거 방법 연구의 필요성을 제시하였다. 3장에서는 실제 웹 브라우저에서 발생한 UAF 보안문제를 통하여 발생 원인을 분석하고 종류별로 정리하였다. 또한 해당 원인을 개발과정에서 제거하기 위한 방법으로 2 레벨 트리구조와 알고리즘을 언급하였다. 4장에서는 제안하는 제거 방법을 실험하였다. 실험을 위하여 간략화한 실제 보안문제를 사용하였으며 논문에서 제안하는 방법으로 보안약점을 제거할 수 있는지 확인하였다. 5장에서는 본 논문의 결론 및 향후에 진행하고자 하는 연구에 대하여 언급하였다.

## 2. 연구 배경

### 2.1 관련 연구

시스템 내부에 존재하는 소프트웨어 혹은 커널을 대상으로 UAF 보안약점 탐지 및 제거에 관한 논문은 2013년 Josselin Feist, Laurent Mounier와 Marie-Laure Potet가 바이너리 코드를 대상으로 해당 보안약점을 탐지할 수 있는 방법을 제시하였다[6]. 해당 방법의 경우 바이너리 코드를 중간언어 형태로 변환하고 이를 그래프 형태로 표시하였으며 메모리 영역을 할당하고 해제하고 재사용하는 과정을 직접 확인하기 때문에 분석속도가 느리다는 단점이 있다. 2015년 Bin Zhang, Bo Wu, ChaoFeng 외 2명이 제시한 방법의 경우 유효하지 않은 포인터 사용 탐색을 통한 UAF 탐지방법을 제시하였다[4]. 2015년 Byoungyoung Lee, Chengyu Song, Yeongjin Jang 외 4명이 제안한 방법의 경우 기존에 존재하는 컴파일러에 UAF 보안약점 탐지 방법 추가하여 오픈소스로 제공되는 소프트웨어를 재 컴파일하고 실행하여 UAF 보안문제를 억제하였다[7]. 2015년 Wen Xu, Juanru Li, Junliang Shu 외 1명이 제시한 방법의 경우 리눅스 커널에 존재하는 UAF 보안약점을 공격하기 위한 방법을 제시하였다[8]. 2012년 Juan Caballero, Gustavo Grieco, Mark Marron 외 1명이 제시한 방법의 경우 UAF와 Double-Free 버그를 탐지하기 위하여 오염된 입력 값 분석을 통한 포인터 추적기법을 사용하였다[3].

국내의 경우 기존에 존재하는 보안약점을 정적·분석 방법 등으로 탐지하고 제거하기 위한 연구는 지속하여 왔지만, 앞서 언급한 UAF 보안약점에 관한 연구는 미미한 실정이다.

기존에 진행된 연구들을 소프트웨어 개발 방법론적 관점에서 각각의 단계에 적용하면 Table 1과 같다. Testing의 경우 소스코드가 존재하는 과정에서 분석을 진행하였을 경우, 그리고 Maintenance는 소스코드가 없는 상태에서 분석을 진행하였을 경우로 분류하였다.

### 2.2 개발단계 UAF 보안약점 제거 방법의 필요성

소프트웨어 및 컴퓨터 시스템 보안의 관점에서 개발단계에 UAF 보안약점을 제거하는 것은 다음의 2가지의 이유로 필요하다.

첫째, 내부 보안문제의 경우 소프트웨어 개발이 끝난 후 운영 및 유지·보수 단계에서 탐지하고 제거할 경우 개발 단계에서 보안문제를 해결할 때에 비하여 최대 30배의 비용이 필요하다[9]. 보안문제를 일으키는 대부분의 내부 보안약점의 경우 개발단계에서 프로그래머의 코딩실수로 발생한다. 따라서 UAF 보안약점을 개발단계에서 제거할 경우 경제적 관점에서 많은 비용을 절약할 수 있으므로 이에 대한 노력이 필요하다.

둘째, 개발자들을 대상으로 UAF 보안약점의 존재를 인식시키고 해당 보안약점을 제거하도록 노력하게 한다. 소프트웨어 내부에서 발생하는 보안문제로 인하여 안전한 소프트웨어 개발의 중요성이 증가하였다. 개발단계가 끝나고 유지·보수 단계에서 테스트 등을 통한 보안약점 탐지 및 제거의 경우 한계가 존재한다. 만약 내부 개발자들이 발생할 수 있는 보안문제를 인지하고 개발단계에서 각각의 보안약점을 제거하였다면 이를 통해 개발된 소프트웨어의 경우 높은 신뢰도를 얻을 수 있다.

소프트웨어 개발 과정에서 코딩실수로 발생하는 보안문제의 경우 개발이 끝난 후 유지·보수 단계에서 파악하기 어렵다는 특징이 있다. 만약 개발 단계에서 존재하는 보안약점을 인지하고 이를 제거하는 노력을 적용할 경우 개발 과정의 경제적 관점 그리고 결과물인 소프트웨어의 질적인 면에서 장점이 있다. 따라서 개발 단계에서 UAF 보안약점을 제거하기 위한 방법의 연구가 필요하다.

## 3. UAF 보안약점 원인 및 제거 방안

### 3.1 UAF 보안약점 발생 원인

보안약점을 제거하기 위해서는 해당 보안약점이 발생하는 원인을 파악하고 이를 억제해야 한다. 본 논문에서는 UAF 보안약점으로 인해 발생한 보안문제 중에서 CVE-2012-4792로 등록된 iExplorer Cbutton 문제를 참고하였다[10-12].

이 보안문제의 경우 iExplorer 버전 6, 7, 8에서 발생하며 UAF 보안약점을 사용하였다. 공격자는 해당 보안약점을 바탕으로 원격지에서 사용자의 시스템에 보안문제를 일으킬 수 있었으며 이 과정에서 제어흐름 변경을 통하여 원하는 코드를 실행할 수 있었다. Fig. 1의 경우 해당 보안문제를 정리해놓은 개념도이다[12].

CVE-2012-7492 보안문제는 Cbutton 객체가 메모리 영역에서 CollectGarbage()함수에 의하여 해제한 상태에서 공격자가 Cbutton객체가 할당되었던 영역을 다른 값으로 덮어쓴 뒤에 해당 객체를 참조하던 포인터를 재사용하는 과정에서 발생한다.

CVE-2012-4792를 통해 확인해본 결과 UAF를 통한 취약점의 경우 특정 메모리 영역을 하나 이상의 포인터가 참조하는 상태에서 메모리 영역을 해제하고 이 영역을 다른 값을 덮어쓴 상태에서 포인터를 재사용할 때 발생한다는 것을 알 수 있다. Fig. 2는 UAF 보안약점이 발생하는 원인을 보여준다[12].

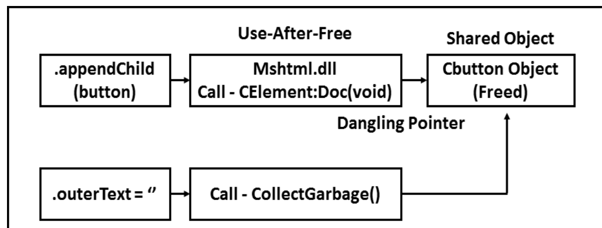
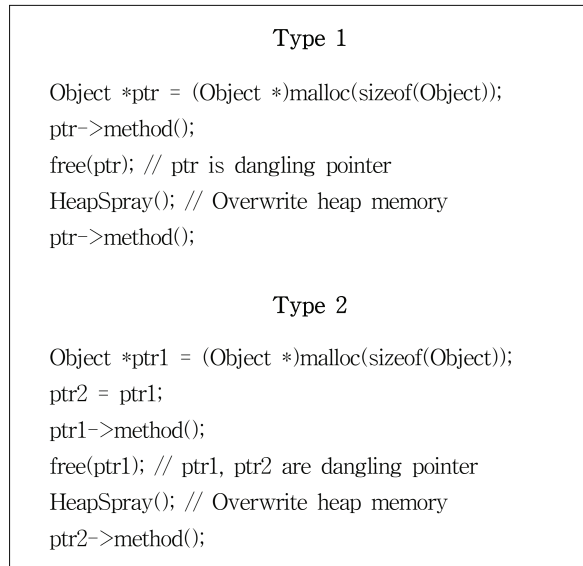



Fig. 1. CVE-2012-4792 Attack Graph

UAF 보안약점의 원인은 2개의 타입으로 나눠진다. 첫 번째 타입은 하나의 포인터가 특정 메모리 영역을 참조하고 있다가 할당 해제 후 사용될 때 문제가 된다. 이때 HeapSpray()는 해제된 메모리 영역을 다른 값으로 덮어쓰는 공격기법 중에 하나이다. 이에 비하여 두 번째 타입은 둘 이상의 포인터가 특정 메모리 영역을 공유하여 참조하는 상황에서 개발자가 실수로 메모리 영역 해제한 후 포인터를 사용할 때 문제가 된다.

UAF 보안약점의 경우 메모리를 할당하고 해제하는 과정에서 해제된 메모리 영역을 참조하는 포인터의 존재 때문에 문제가 된다. 이 포인터를 허상 포인터(Dangling Pointer)라고 한다. Fig. 2의 타입 1에서는 ptr이 허상 포인터이며 타입 2에서는 ptr1, ptr2가 허상포인터이다.

소스코드 내부에 허상 포인터가 존재한다고 해서 모두 시



Type 1

```
Object *ptr = (Object *)malloc(sizeof(Object));
ptr->method();
free(ptr); // ptr is dangling pointer
HeapSpray(); // Overwrite heap memory
ptr->method();
```

Type 2

```
Object *ptr1 = (Object *)malloc(sizeof(Object));
ptr2 = ptr1;
ptr1->method();
free(ptr1); // ptr1, ptr2 are dangling pointer
HeapSpray(); // Overwrite heap memory
ptr2->method();
```

Fig. 2. UAF Security Weakness

스템 보안과 관련된 것은 아니다. 공격자가 허상 포인터를 통해 공격을 수행하기 위해서 허상 포인터가 참조하고 있는 메모리 영역을 다른 값으로 덮어 쓸 수 있어야 한다. 결국 허상 포인터는 임의의 영역을 참조하는 포인터가 있을 때 해당 영역이 해제된 상황에서 내부 주소 값을 무효화하지 않아 계속 해당 메모리 영역을 참조하는 포인터라고 정의할 수 있다[7]. 그리고 안전하지 않은 허상 포인터의 경우 소스코드 내부에 존재하는 허상 포인터에 대하여 사용자로 인해 해당 허상 포인터가 참조하는 메모리 영역이 읽거나 쓸 수 있을 경우 안전하지 않은 포인터라고 정의할 수 있다[7].

결국 UAF 보안약점을 개발단계에서 제거하기 위해서는 안전하지 않은 허상 포인터를 무효화(Nullification) 해줘야 한다.

### 3.2 UAF 보안약점 제거 방안

소프트웨어를 개발하는 과정에서 객체 관리, 메모리 관리 등을 위하여 많은 수의 포인터가 사용된다. 메모리 사용이 끝나고 나서 해당 메모리 영역을 해제하게 되면 1개 이상의 포인터가 허상 포인터가 된다. 이렇게 생성된 허상 포인터는 UAF 보안약점의 원인이 될 수 있기 때문에 무효화를 해주어야 한다.

기존의 개발과정에서 UAF 보안약점을 제거하기 위한 방법의 경우 메모리 영역을 해제할 때 발생하는 허상 포인터를 개발자가 일일이 무효화해주는 방식을 사용하였다[13]. 하지만 해당 방법은 Type 1의 허상 포인터에만 부분적으로 적용할 수 있는 방법이며 Type 2와 같이 특정 메모리 영역을 공유하는 포인터의 개수가 많아질수록 개발자가 해당 포인터를 모두 기억하고 관리할 수 없기 때문에 무효화하지 않는 실수가 발생하게 되고 결국 소스코드 내부에 안전하지 않은 허상 포인터가 존재하게 되어 프로그램의 보안성을 떨어뜨린다. 따라서 본 논문에서는 2레벨 트리구조와 알고리

즘을 이용한 보안약점 제거방법을 제안한다.

2레벨 트리구조를 사용할 경우 메모리 영역과 그 메모리 영역을 참조하는 참조 포인터를 노드형태로 간단하게 구현할 수 있다는 장점이 있으며 특정 메모리 노드와 해당 메모리 영역을 참조하는 포인터에 대한 노드에 대하여 기존의 탐색기법을 쉽게 적용할 수 있다는 장점이 있다.

Fig. 3은 특정 메모리 영역을 참조하는 모든 포인터를 추적하기 위해 사용한 2레벨 트리구조를 나타낸 것이다. 루트 노드를 시작으로 레벨 1의 노드들은 소스코드 내부에서 할당되는 메모리 영역에 대한 정보를 저장하고 있다. 해당 정보는 메모리 영역에 대한 베이스 주소와 메모리 영역을 참조하는 포인터들의 링크 정보, 다른 메모리 영역의 정보를 저장한 노드의 링크 정보로 구성되어 있다. 레벨 2의 노드들은 특정 메모리 영역을 참조하고 있는 포인터의 정보가 저장되어 있다. 해당 노드의 경우 포인터 변수의 주소와 다른 포인터의 정보가 저장되어 있는 노드의 링크 정보를 담고 있다.

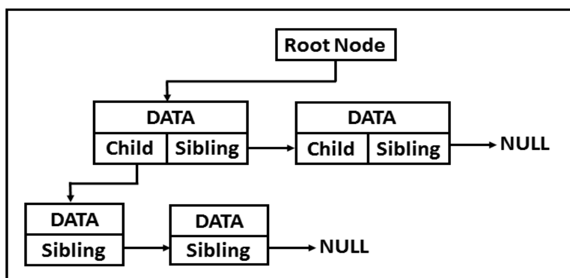



Fig. 3. 2 Level Tree Data Structure

Fig. 4는 할당된 메모리 영역과 해당 메모리 영역을 참조하는 포인터들을 예시로 나타내고 해당 예시를 본 논문에서 제안하는 2레벨 트리구조로 나타낸 것이다.

이 트리구조를 기반으로 하여 메모리를 할당하고 해제하는 과정에서 발생하는 허상 포인터를 추적하고 무효화해주기 위해 Fig. 5 알고리즘을 제안한다.

Fig. 5의 (a) 알고리즘은 메모리를 할당할 때 사용하며 할당된 메모리 영역의 정보를 저장하는 역할을 한다. 이 알고리즘은 메모리 영역의 노드와 참조 포인터 노드를 생성하고 이를 링크로 연결해주는 방식으로 동작한다. Level 1 노드와 Level 2 노드를 각각 생성하고 함수 파라미터로 넘어온 변수의 주소를 이용하여 Level 1 노드에서는 메모리 영역의 주소가 저장되고 Level 2 노드에서는 변수의 주소가 저장한다. Level 1의 메모리 주소의 경우 해당 메모리 영역을 참조하는 포인터를 추적할 때 사용하며 Level 2 노드의 변수 주소의 경우 참조하는 메모리 영역이 해제되었을 때 해당 포인터를 무효화하는 과정에서 사용한다. 그 후에 Level 1 노드의 Child 값으로 Level 2 노드를 추가해 준다. 그리고 나서 Level 1 노드를 Root 노드에 추가해주는데 이 과정에서 Root 노드의 Child 노드들의 Sibling 끝 부분에 추가된다. 이 과정에서 넓이 우선 탐색(Breadth First Search) 알고리즘을 사용하였다.

Fig. 5의 (b) 알고리즘은 특정 메모리 영역을 두 개의 포인터가 공유하는 상황에서 사용하게 된다. lPtr에는 공유 포인터의 정보가 들어가 있으며 rPtr에는 참조하는 메모리의 정보가 들어가게 된다. 먼저 lPtr이 기존의 트리 내부에 Level 2 노드로 존재하게 되면 다른 메모리 영역을 참조하고 있던 포인터가 rPtr이 참조하는 메모리로 참조를 바꾸는 것으로 판단하여 트리구조 내부에 lPtr의 정보를 저장하고 있는 노드를 삭제한다. 이 과정에서 lPtr이 참조하던 메모리 주소를 포함한 Level 1 노드를 찾고 해당 메모리를 참조하는 포인터들의 정보가 저장되어 있는 Level 2 노드 확인하는 과정에서 각각 넓이 우선 탐색 알고리즘을 적용하였다. 이를 통하여 기존에 존재하는 lPtr 노드를 삭제한다. 그리고 나서 새롭게 lPtr 정보를 포함하는 Level 2 노드를 생성한다. 새롭게 생성된 lPtr 노드의 경우 rPtr이 참조하는 메모리 영역의 주소 정보를 저장하고 있는 Level 1 노드의 Child 중에서 가장 끝부분 Sibling으로 포함된다.

Fig. 5의 (c) 알고리즘은 특정 메모리 영역이 해제될 때 사용한다. 먼저 ptr이 참조하는 메모리 주소를 이용하여 Level 1 노드 중에서 해당 메모리 주소를 저장하고 있는 노드를 찾는다. 그 후에 해당 노드의 Child를 대상으로 무효화를 진행

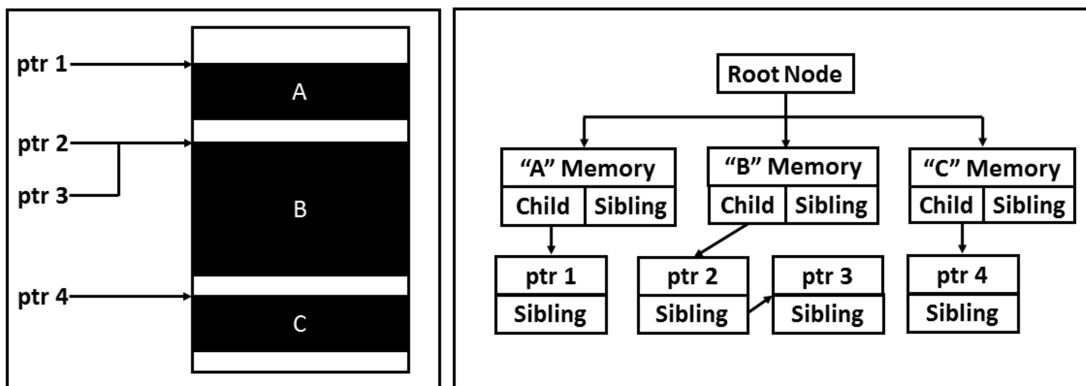



Fig. 4. Example of 2 Level Tree Structure

```
Algorithm (a). Add Object

def addObj(ptr) :
    level1Node = createLevel1Node()
    level2Node = createLevel2Node()
    level2Node.addInfo() # insert memory address
    level1Node.addInfo() # insert pointer address
    level1Node.addChild = level2Node
    rootNode.addChild(level1Node)

Algorithm (b). Share Object

def shareObj(lPtr, rPtr) : # lPtr = rPtr
    if(nodeTree.level2NodeFind(lPtr)) :
        nodeTree.deleteLevel2Node(lptr)
    level1Node = nodeTree.findLevel1Node(rPtr)
    level2Node = createTreeNode2(lPtr)
    level1Node.addChild(level2Node)

Algorithm (c). free Object

def freeObj(ptr) :
    level1Node = nodeTree.findLevel1Node(ptr)
    for pointer in level1Node.childList :
        pointer = Nullification # ptr = NULL
    nodeTree.deleteLevel1Node(level1Node)
```

Fig. 5. Algorithm for Dangling Pointer Nullification

한다. 이 과정에서 Child 노드와 Sibling 노드들에 대하여 가지고 있는 포인터 정보를 이용하여 해당 포인터를 각각 무효화해주며 Sibling 값이 NULL이 될 때까지 진행하는 방법으로 해당 메모리를 참조하는 모든 포인터를 무효화해준다. 그 후에 마지막으로 해당 메모리 영역의 정보를 저장하고 있는 Level 1 노드를 삭제하면서 알고리즘을 종료하게 된다.

본 논문에서 제안한 레벨 2 트리구조와 알고리즘을 활용하여 실제 UAF 보안약점으로 인한 보안문제가 존재하는 예시에 적용하여 보안약점을 제거하는지 확인한다.

## 4. 실험 및 결과

본 논문에서 제시하는 UAF 보안약점 제거 방법이 특정 메모리 영역을 참조하는 허상 포인터를 추적하고 무효화해주는지 확인한다.

해당 실험을 위하여 CVE-2011-4130에 보고된 보안문제를 간략화한 예시와 CWE(Common Weakness Enumeration)에서 제공하는 예시를 수정하여 사용하였다[6, 14].

예시 중에서 CVE-2011-4130 예시를 소스코드로 나타내면 Fig. 6과 같다. 해당 소스코드는 힙 메모리 영역에서 UAF

```
int cmp(void) {
    return *p_global >= MIN && *p_global <= Max;
}

void index_user(int *p) {
    int *p_global_save;
    p_global_save = p_global;
    p_global = p;
    if(cmp() >= 0) {
        printf("The secret is less than 50\n");
        p_global = p_global_save;
        return ;
}

int main(int argc, char *argv[]) {
    int *p_index, *p_pass;
    if(argc != 2) {
        printf("Mode Easy = 1, Mode Hard ! = 1\n");
        return 0;
    }
    p_global = (int *)malloc(sizeof(int));
    *p_global = SECRET_PASS;
    if(atoi(argv[1]) == MODE_EASY) {
        p_index = (int *)malloc(sizeof(int));
        printf("Give a number 0 ~ 100\n");
        scanf("%d", p_index);
        index_user(p_index);
        free(p_index);
    }
    else
        printf("Good Luck!\n");
    p_pass = (int *)malloc(sizeof(int));
    printf("Give the secret\n");
    scanf("%d", p_pass);
    if(*p_pass == *p_global)
        printf("Congrats!\n")
    else
        printf("Sorry...\n");
    return 0;
}
```

Fig. 6. UAF Security Problem Example (CVE-2011-4130)

보안약점으로 인하여 검증 무효화 보안문제를 가지고 있다.

UAF 보안약점이 존재하는 소스코드에 본 논문에서 제안하는 제거방법을 적용하기 위하여 메모리 영역을 할당하고 공유하고 해제하는 구간에 코드를 추가하였다.

Fig. 7은 보안약점이 존재하는 소스코드 내부에 추가한 코드의 일부이다. 메모리 영역이 할당될 때에는 Fig. 9(a)와 같이 addObj 함수를 사용하여 트리구조에 할당된 메모리 영역의 데이터를 입력하였다. 그리고 할당된 메모리 영역을 하나 이상의 포인터가 공유할 경우 Fig. 7(b) shareObj 함수를 통하여 메모리 영역을 참조하는 포인터들의 정보를 관리하였다. 마지막으로 특정 메모리 영역이 해제될 경우 해당 메모리 영역을 참조하는 포인터는 허상 포인터가 되어 UAF

(a) Allocate Memory

```
p_global = (int *)malloc(sizeof(int));
addObj(&p_global);
```

(b) Share Pointer

```
p_global_save = p_global
shareObj(&p_global_save, &p_global);
```

(c) Deallocate Memory

```
free(p_index);
freeObj(&p_index);
```

Fig. 7. Apply UAF Security Weakness Remove Method

보안약점의 원인이 될 수 있으므로 Fig. 7(c)와 같이 freeObj 함수를 통하여 해제된 메모리 영역을 참조하고 있던 모든 허상 포인터를 무효화해준다.

UAF 보안약점이 존재하는 소스코드에 본 논문에서 제안하는 방법을 적용하지 않았을 때와 적용했을 때 인증우회 직전의 메모리 구조와 해당 메모리를 참조하는 포인터를 나타내면 Fig. 8과 같다. 실제 CVE 예시의 경우 인증과정에서 p_pass와 p_global 값을 비교하게 되어 인증우회가 발생하는데 p_global 대신에 p_index로 코드를 수정해도 인증우회가 발생한다는 것을 확인하였다.

문제가 있는 예제들에 본 논문에서 제안하는 방법을 적용했을 때 결과는 Table 2와 같다.

Table 2. Result of Apply UAF Security Weakness Remove Method

| Example | CVE Example (CVE-2011-4103) | CWE Example (Demonstrative Example 1) |
|---|---|---|
| Problem | Authentication Bypass | Using Dangling Pointer in Function |
| Result | Block Bypass (Program Terminate) | Block Method (Program Terminate) |

먼저 CVE 예제의 경우 허상 포인터가 참조하는 메모리 영역이 다른 값으로 변형되면서 인증우회가 발생하는 문제를 가지고 있었다. 해당 보안문제를 해결하기 위하여 개발단계에서 해당 예제 소스코드를 개발한다고 가정하고 본 논문에서 제안하는 보안약점 제거방법을 적용하였다. 그 결과 소스코드 내부에 존재하는 허상 포인터인 p_index와 p_global에 대하여 개발자가 해당 포인터를 기억하고 있을 필요없이 알고리즘 호출을 통하여 자동으로 무효화되는 것을 확인하였다. 결국 사용자의 입력 값과 p_global의 값이 달라지기 때문에 인증우회 보안문제가 발생하지 않았으며 해당 부분에서 허상 포인터를 사용했음을 인지할 수 있었다.

CWE 예제의 경우 소프트웨어 내부에 존재하는 허상 포인터가 참조하는 메모리가 다른 값으로 할당된 상태에서 strcpy와 같이 보안문제를 일으킬 수 있는 함수에 해당 허상 포인터가 사용되면서 문제가 발생하였다. 이 문제를 해결하기 위하여 본 논문에서 제안한 방법을 적용하였을 때 strcpy에 사용되는 허상 포인터가 무효화되었기 때문에 개발자는 해당 함수에 허상 포인터를 사용했음을 확인하고 개발단계에서 소스코드를 수정할 수 있었다.

본 논문에서 제안하는 방법을 이용하여 개발단계에서 UAF 보안약점을 제거할 경우 예제를 대상으로 MCC (Modified Cyclomatic Complexity)를 측정한 결과 최대 복잡도가 평균 17% 증가하는 것을 확인하였으며 평균 복잡도는 18.7% 상승하였다. 또한 프로그램 실행과정에서 평균 14.9%의 오버헤드가 발생하였다. 자동화된 방법으로 UAF 보안약점을 탐지하거나 보안문제 발생을 탐지할 경우 실행과정에서 1~270% 정도의 오버헤드를 보였으며 평균 53.1%의 오버헤드를 보였다[7]. 그리고 분석 과정에서 주석 혹은 바이너리 형태의 명령들이 추가되었다. 프로그램의 성능에 영향을 미치는 부분에서 포인터가 많이 사용되거나 사용되지 않는 것으로 인하여 도구 및 방법에 따라 오버헤드의 차이를 보였다. 다만, 자동화된 도구의 경우 프로그램 내부에 존재하는 전역변수 형태, 지역변수 형태, 함수 파라미터 형태 등 모든 포인터를 추적하기 때문에 실행과정에서 많은 오버헤드가 발생

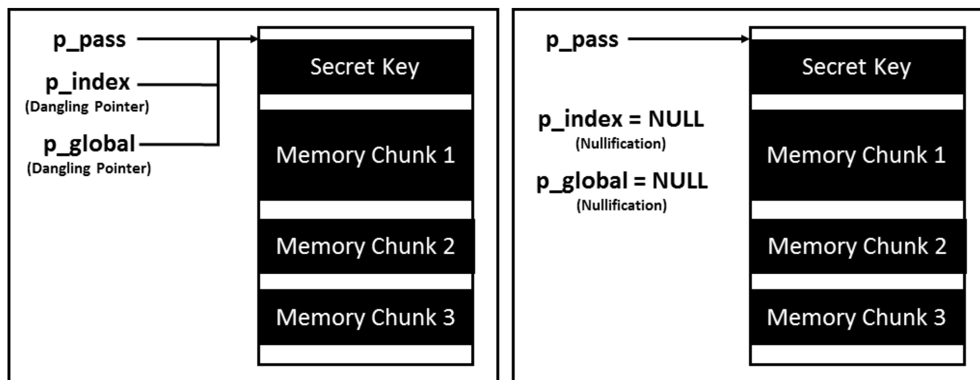



Fig. 8. Apply Before and After Remove Method

하였다. 하지만 개발과정에서 UAF 보안약점 제거 방법을 적용할 경우 소프트웨어 내부에 존재하는 모든 포인터를 추적하는 것이 아니라 사용자의 입력 값 등으로 인하여 UAF 보안문제가 발생할 수 있는 함수 등에만 선택적으로 적용할 수 있기 때문에 자동화된 탐지도구에 비하여 프로그램 실행 간 발생할 수 있는 오버헤드가 작아질 것으로 판단된다.

## 5. 결론 및 향후 연구

본 논문에서는 최근까지 웹 브라우저 및 운영체제에서 가장 많이 일어나는 UAF 보안약점에 대하여 실제 예제를 통하여 발생 원인을 정리하였다. 이를 통하여 기존의 연구와는 다르게 개발과정에서 해당 보안약점을 제거하기 위한 방법을 제안하였다.

실험과정에서 기존에 존재하는 방법 대신에 트리 구조를 이용한 방법을 소스코드에 적용하였다. 그 결과 개발단계에서 UAF 보안약점에 원인이 되는 허상포인터를 소프트웨어 내부에서 관리할 수 있었으며 해당 포인터에 대한 무효화를 통하여 보안 문제 발생을 억제하는 효과가 있음을 확인하였다.

향후 연구에서는 본 논문에서 제시하는 UAF 보안약점 제거 방법을 활용하여 시큐어 코딩 규칙 개발하여 제안한다. 현재 국내에 적용되고 있는 시큐어 코딩 규칙의 경우 UAF 보안약점을 제거하기 위한 방법이 누락되어 있다. 따라서 해외에서 적용되고 있는 시큐어 코딩 규칙과 본 논문에서 제시하는 방법을 간소화하여 시큐어 코딩 규칙을 개발한다. 또한 기존에 존재하는 정적·분석도구를 대상으로 해당 논문에서 제안하는 UAF 분석방법을 적용하여 소스코드가 존재하는 소프트웨어를 대상으로 테스팅 및 유지·보수 단계에서 UAF 보안 약점을 탐지하고 억제할 수 있는지 확인한다.

메모리 영역을 대상으로 보안문제가 지속적으로 발생하고 있지만 이를 유지·보수 과정에서 정적·동적 분석방법으로 모두 해결하기는 어려움이 있다. 따라서 개발과정에서 이런 보안 문제를 제거할 수 있도록 지속적인 관심과 연구가 필요하다. 또한 기존에 존재하는 방법들에 대하여 성능 및 편의성에 대한 분석을 통하여 방법들의 질을 높일 필요가 있다.

## References


[1] L. Szekeres, M. Payer, T. Wei, and D. Song, “Sok: Eternal war in memory,” in *Security and Privacy (SP), 2013 IEEE Symposium on,* pp.48-62, 2013.

[2] Breno Cunha, Perspectives on exploit development and cyber attacks [Internet], http://blog.tempest.com.br/breno-cunha/perspectives-on-exploit-development-and-cyber-attacks.html.

[3] J. Caballero, G. Grieco, M. Marron, and A. Nappa, “Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities,” in *Proceedings of the 2012 International Symposium on Software Testing and Analysis,* pp.133-143, 2012.

[4] B. Zhang, B. Wu, C. Feng, X. Zhang, and C. Tang, “Statically detect invalid pointer dereference vulnerabilities in binary soft ware,” in *2015 IEEE International Conference on Progress in Informatics and Computing (PIC)*, pp.390-394, 2015.

[5] Mark Yason, Use-After-Frees : That pointer may be pointing to something bad[Internet] https://securityintelligence.com/use-after-frees-that-pointer-may-be-pointing-to-something-bad.

[6] J. Feist, L. Mounier, and M. L. Potet, “Statically detecting use after free on binary code,” *Journal of Computer Virology and Hacking Techniques*, Vol.10, No.3, pp.211-217, 2014.

[7] B. Lee, C. Song, Y. Jang, T. Wang, T. Kim, L. Lu, and W. Lee, “Preventing Use-after-free with Dangling Pointers Nullification,” in *NDSS*, 2015.

[8] W. Xu, J. Li, J. Shu, W. Yang, T. Xie, Y. Zhang, and D. Gu, “From collision to exploitation: Unleashing use-after-free vulnerabilities in linux kernel,” in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security,* pp.414-425, 2015.

[9] G. Tassey, “The economic impacts of inadequate infrastructure for software testing,” *National Institute of Standards and Technology, RTI Project*, *7007*(011). 2002.

[10] CVE-2012-4792 [Internet], http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2012-4792.

[11] Red Alert, CVE-2012-4792 Microsoft Internet Explorer CButton Object Use-After-Free Vulnerability, 2013.

[12] H.M. Kim, “*Windows System Hacking Guide : Bug Hunting and Exploit*,” SECU BOOK, Goyang-City, Gyunggi Province, 2016.

[13] CERT : MEM01-C [Internet], https://www.securecoding.cert.org/confluence/pages/viewpage.action?pageId=440.

[14] CWE-416 : Use After Free [Internet], https://cwe.mitre.org/data/definitions/416.html.

**박 용 구**

e-mail : ygpark@formal.korea.ac.kr

2015년 서울시립대학교 수학과, 컴퓨터과학과(학사)

2015년 ~ 현 재 고려대학교 정보보호대학원 정보보호학과 석사과정

관심분야 : Software Security Analysis, Reverse Engineering, Bug Detection & Automatic Tools development

**최 진 영**

e-mail : choi@formal.korea.ac.kr
1982년 서울대학교 전산학(학사)
1986년 Drexel University 전산학(석사)
1993년 Universit of Pennsylvania 전산학 (박사)
1996년~현 재 고려대학교 정보보호대학원 정보보호학과 교수
관심분야 : Formal Method, Secure Software Engineering, Software Assurance, SDN