Journal of The Korea Institute of Information Security & Cryptology ISSN 1598-3986(Print)
VOL.28, NO.4, Aug. 2018 ISSN 2288-2715(Online)
https://doi.org/10.13089/JKIISC.2018.28.4.919

동적 기호 실행을 이용한 힙 메모리 OOB 취약점 자동 탐지 방법*

강 상 용,† 박 성 현, 노 봉 남‡
전남대학교 정보보안협동과정

Automated Method for Detecting OOB Vulnerability of Heap Memory Using Dynamic Symbolic Execution*

Sangyong Kang,† Sunghyun Park, Bongnam Noh‡
Interdisciplinary Program of Information Security, Chonnam National University

요 약

OOB(Out-Of-Bounds)는 힙 메모리에서 발생하는 취약점 중 가장 강력한 취약점 중 하나이다. OOB 취약점을 이용하면 Array의 길이를 속여 해당 길이만큼의 메모리를 읽기 혹은 쓰기가 가능하기 때문에 공격자는 기밀 정보에 대한 무단 액세스를 악용할 수 있다. 본 논문에서는 동적 기호 실행과 쉐도우 메모리 테이블을 활용하여 힙 메모리에서 발생하는 OOB 취약점을 자동으로 탐지하는 방법을 제안한다.

먼저, 힙 메모리 할당 및 해제 함수 후킹을 통해 쉐도우 메모리 테이블을 구축한다. 이후 메모리 액세스가 발생할 때, 쉐도우 메모리를 참조하여 OOB가 발생할 수 있는지를 판단하고, 발생 가능성이 존재할 경우 크래시를 유발하는 테스트케이스를 자동으로 생성한다. 제안하는 방법을 활용할 경우, 취약한 블록 탐색에 성공한다면 반드시 OOB를 유발하는 테스트케이스를 생성할 수 있다. 뿐만 아니라 전통적인 동적 기호 실행과는 다르게 명확한 목표 지점을 설정하지 않더라도 취약점 탐색이 가능하다.

ABSTRACT

Out-Of-Bounds (OOB) is one of the most powerful vulnerabilities in heap memory. The OOB vulnerability allows an attacker to exploit unauthorized access to confidential information by tricking the length of the array and reading or writing memory of that length. In this paper, we propose a method to automatically detect OOB vulnerabilities in heap memory using dynamic symbol execution and shadow memory table.

First, a shadow memory table is constructed by hooking heap memory allocation and release function. Then, when a memory access occurs, it is judged whether OOB can occur by referencing the shadow memory, and a test case for causing a crash is automatically generated if there is a possibility of occurrence. Using the proposed method, if a weak block search is successful, it is possible to generate a test case that induces an OOB. In addition, unlike traditional dynamic symbol execution, exploitation of vulnerabilities is possible without setting clear target points.

Keywords: Dynamic Symbolic Execution, Software Vulnerability, Heap Memory Vulnerability, Out-of-bounds

Received(05. 08. 2018), Modified(06. 26. 2018), Accepted(06. 26. 2018)
* 이 논문은 ETRI부설연구소의 2017년도 위탁연구과제로 수행한 연구결과입니다.
† 주저자, scytelezz@gmail.com
‡ 교신저자, bbong@jnu.ac.kr(Corresponding author)

I. 서 론

1.1 연구 목표

OOB(Out-Of-Bound)는 힙 메모리에서 발생하는 취약점 중 가장 강력한 취약점 중 하나이다. OOB 취약점을 이용하면 Array의 길이를 속여 해당 길이만큼의 메모리를 읽기 혹은 쓰기가 가능하기 때문에 공격자는 기밀 정보에 대한 무단 액세스를 악용할 수 있다. 민감한 데이터에 대한 액세스 권한이 있는 제품의 보급이 증가함에 따라 잠재적으로 악용될 가능성이 있는 시스템이 늘어나고, 자동화된 소프트웨어 검사 도구가 더 많이 필요하게 되었다[1-2]. DARPA는 최근 소프트웨어 테스팅 자동화에 대한 연구의 중요성을 보여주기 위해 수백만 달러에 달하는 자금을 지원했다[3].

잠재적인 버그를 찾아내는 현재의 대표적인 기술은 퍼징(fuzzing)과 기호 실행(symbolic execution)이다. 각 기술은 장단점을 지니는데, 퍼징의 경우 불완전성을 지닌다. 이러한 요소를 피하기 위해 기호 실행이 고안되었지만, 기호 실행 기술은 버그를 탐색하기 위해서 실제 크래시를 발생하기까지의 경로를 모두 탐색해야 한다는 단점이 있다. 결국 타깃 소프트웨어의 깊은 곳까지의 탐색이 필수적이며, 이는 경로 폭발 문제를 유발할 가능성이 농후하다.

본 논문에서는 동적 기호 실행(dynamic symbolic execution)과 쉐도우 메모리 테이블(shadow memory table)의 활용을 통해 힙 메모리에서 발생하는 OOB 취약점을 자동으로 탐지하는 방법을 제안한다. 먼저, 힙 메모리 할당 및 해제 함수 후킹을 통해 쉐도우 메모리 테이블을 구축한다. 이후 메모리 액세스가 발생할 때, 쉐도우 메모리를 참조하여 OOB가 발생할 수 있는지를 판단하고, 취약점 발생 가능성이 존재할 경우 크래시를 유발하는 테스트케이스를 자동으로 생성한다.

제안하는 방법의 효율성을 검증하기 위해 CVE에 등록된 기존 취약점의 코드 스니핏을 활용했다. 실험을 통해 퍼징 및 전통적인 동적 기호 실행보다 효율적으로 힙 메모리 OOB 취약점 탐색이 가능함을 입증하였다.

1.2 연구 개요

본 논문에서 제안하는 핵심 개념은 쉐도우 메모리 테이블을 활용한 동적 기호 실행 기술에서 기반한다. 다음은 특정 입력값(int x 〉 11)이 전달되었을 때 힙 오버플로우를 발생시키는 예제이다.

Fig. 1과 같은 바이너리에서 힙 메모리 OOB 취약점을 자동 탐지하기 위해 전통적으로 퍼징 기법을 주로 사용해왔다. 만약 소스코드가 존재한다면 Asan[4]과 같은 CTI(compile time instrumentation) 도구를 적용시켜 크래시 로그를 생성한다. 하지만 이와 같은 방법으로 OOB를 탐지한다면 퍼징 기법의 불확실성으로 인해 크래시를 발견하지 못할 확률이 존재한다.

Fig. 2는 Fig. 1의 바이너리의 DFG(data flow graph)를 요약한 것이다. 입력값의 조건이

```
1  #include <stdio.h>
2  #include <ctype.h>
3
4  int isNum(char* input)
5  {
6    if(0 <= atoi(input) && atoi(input) <= 100)
7      return 1;
8    else
9      return 0;
10 }
11
12 int main(int argc, char* argv[])
13 {
14   long size =65, argv1 =0;
15   char *buf1;
16   char *buf2;
17
18   buf1 = (char*)malloc(size);
19   buf2 = (char*)malloc(size);
20
21   if(argc >= 2 && isNum(argv[1]))
22   {
23     argv1 = atoi(argv[1]);
24   }
25   else
26     argv1 =0;
27
28   buf2[size-1] = ' ₩0';
29   memset(buf2, '2', size-1);
30
31   buf1[size-1] = ' ₩0';
32   memset(buf1, '1', size-1+(argv1-10));
33   return 0;
34 }
```

Fig. 1. Example of heap overflow

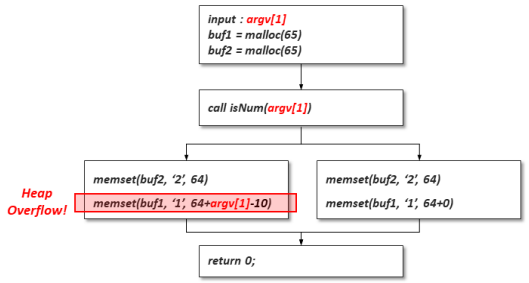

Fig. 2. Data flow graph of example

int x 〉 11일 경우 32번 라인에서 힙 오버플로우가 발생한다. 만약 퍼저가 11보다 작은 정수를 생성했다고 가정해보자. 이 경우, 해당 바이너리의 취약한 블록을 지나쳤음에도 불구하고 크래시가 발생하지 않는다. 즉, 코드 커버리지를 100% 달성했음에도 크래시를 발견하지 못하는 상황이 발생한다.

전통적인 동적 기호 실행(conventional dynamic symbolic execution)은 코드 커버리지를 기준으로 퍼징에 비해 높은 효율을 보인다[5]. 하지만 매실행마다 타깃 지점을 설정해야하며, 그렇지 않더라도 에러 탐지를 위한 모델을 매번 생성해주어야 한다. 따라서 분석 대상 바이너리에 대한 상세 분석이 필수적이다. 만약 추가 분석 없이 전통적인 기호 실행 방법을 이용한다면, 퍼징 기법과 마찬가지로 커버리지를 달성했음에도 원하는 결과를 얻지 못할 수 있다. Table 1은 기존 방식을 이용했을 때 취약한 블록 탐색에 성공했음에도 불구하고, 생성된 테스트케이스에 올바르지 않은 결과가 섞이게 되는 예시다. 퍼징의 경우 무작위 입력만을 생성할 뿐 경로 제약조건을 전혀 고려하지 않는 블랙박스 검사(black-box testing)이기 때문에 이러한 결과가 나타난다. 동적 기호 실행은 퍼징과는 달리 경로 제약조건을 계산하며 탐색을 수행한다. 하지만 타깃 블록에 도달하는 입력을 생성하기 위한 제약조건만을 수집하기 때문에 취약점을 트리거하는 조건이 아닌

Table 1. Validation result of example

case	fuzzing	conventional DSE	proposed method
1	AA (X)	0 (X)	**40 (O)**
2	0x00 (X)	10 (X)	**36 (O)**
3	BB (X)	**34 (O)**	**18 (O)**
4	**0x100 (O)**	**35 (O)**	**36 (O)**
5	FF (X)	**66 (O)**	**24 (O)**

베이직블록 수준에서의 도달성만을 다룬다. 만약, 실제 상용 소프트웨어와 같은 복잡한 바이너리를 대상으로 할 경우 상황은 더 나빠질 수 있다.

본 논문에서 제안하는 힙 메모리 취약점 자동 탐지 방법은 퍼징 기법과 전통적인 동적 기호 실행의 문제점을 극복한다. 힙 메모리에서 발생하는 모든 액세스에 대한 모니터링을 통해 쉐도우 메모리 테이블을 작성한다. 이를 이용해 32번 라인의 경우와 같은 메모리 쓰기가 발생했을 때, OOB가 발생할 수 있는 제약조건을 생성한 뒤 삽입한다. 그리고 해당 제약조건을 해결함으로써 테스트케이스를 생성한다. 즉, 제안하는 방법을 활용할 경우, 취약한 블록 탐색에 성공한다면 반드시 OOB를 유발하는 테스트케이스를 생성할 수 있다. 뿐만 아니라 전통적인 동적 기호 실행과는 다르게 명확한 목표 지점을 설정하지 않더라도 취약점 탐색이 가능하며, 최소한의 탐색으로 테스트케이스를 생성하기 때문에 경로 폭발 문제를 간접적으로 완화시키는 효과도 기대할 수 있다.

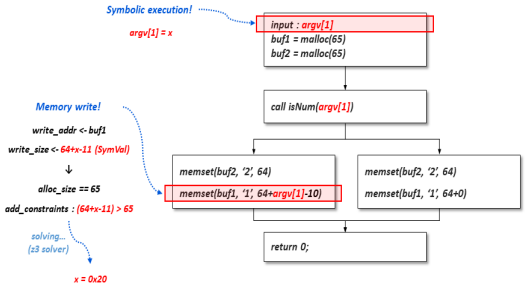

Fig. 3. Proposed idea

II. 힙 메모리 취약점 탐지를 위한 동적 address sanity check 적용 방법

제안하는 힙 메모리 취약점 탐지 기법은 동적 기호 실행(dynamic symbolic execution)과 address sanity check 기술을 응용한다. 먼저, 힙 메모리에 할당 및 해제가 발생했을 때마다 해당 영역을 쉐도우 메모리 테이블(shadow memory table)에 매핑(mapping)한다. 이 후, 해당 메모리 영역에 대한 모니터링을 통해 읽기 및 쓰기 등의 액세스를 확인하고, 액세스 크기의 심볼릭 마킹 여부를 확인한다. 만약 액세스 크기가 심볼릭 변수라면, 해당 힙 메모리 영역에서 취약점이 발생할 수 있는 여지가 있다고 판단하고, 크래시를 유발할 수 있는 제

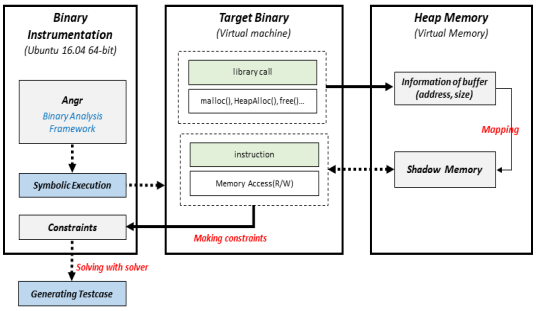

Fig. 4. An overview of our approach and toolset

약조건을 생성한 뒤 강제로 삽입한다. 이 때, 삽입된 제약조건을 포함한 현재 state의 제약조건을 해결자(solver)를 이용해 계산하여 테스트케이스를 생성한다.

2.1 힙 메모리 할당 및 해제 함수 후킹

힙 메모리 할당 함수(malloc 등) 및 할당 해제 함수(free 등)를 후킹하여 모니터링을 수행한다. 이를 통해 힙 메모리가 할당 및 해제될 경우 해당 정보를 쉐도우 메모리 테이블에 기록한다. 이와 같이 수집한 힙 메모리에 할당된 주소 및 크기 정보는 추후 해당 정보를 OOB를 유발하는 제약조건 생성에 사용된다.

2.2 쉐도우 메모리 테이블 매핑 전략

쉐도우 메모리 테이블은 메모리 주소 공간 8바이트를 9개의 상태 시퀀스를 나타내는 1바이트로 매핑한다. 쉐도우 메모리는 Fig. 5와 같이 초기 상태코드 0으로 설정되어있으며, 메모리 할당이 발생할 경우 상태코드 1로 설정한다. 이 때 첫 비트와 마지막 비트를 0으로 설정하여 각 할당 영역의 구분자로 사용한다.

예를 들어, 8바이트의 힙 메모리가 할당되었다면,

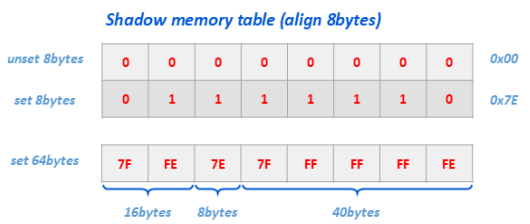

Fig. 5. Memory mapping rules

첫 비트와 마지막 비트는 0으로 설정하고, 나머지 6비트는 1로 설정하여 01111110(2) 즉, 0x7E를 쉐도우 메모리 테이블에 기록된다. Fig. 5와 같이 16바이트, 8바이트, 40바이트의 메모리가 할당될 경우, 쉐도우 메모리 테이블에는 0x7F, 0xFE, 0x7E, 0x7F, 0xFF, 0xFF, 0xFF, 0xFE가 기록된다. 앞에서 언급한 것처럼, 첫 비트와 마지막 비트를 0으로 설정했기 때문에 세 개 각각의 영역을 구분하는 것이 가능하다.

2.3 힙 메모리 액세스 모니터링

힙 메모리 액세스가 발생할 경우, 해당 영역의 취약 여부를 판단하기 위해서는 이전에 할당된 크기 정보가 필요하다. 이와 같은 정보는 쉐도우 메모리 테이블을 참조함으로써 얻을 수 있다.

예를 들어 Fig. 6과 같은 쉐도우 테이블이 생성되었다고 했을 때 특정 주소에 동적으로 결정된 크기의 쓰기가 발생했다고 가정하자. 해당 영역은 0x7F가 기록되어 있기 때문에 0xFE까지의 크기 즉, 32바이트가 할당되었음을 확인할 수 있다. 만약 0x7E가 기록되어 있다면 8바이트의 크기가 할당된 영역이라고 판단할 수 있다.

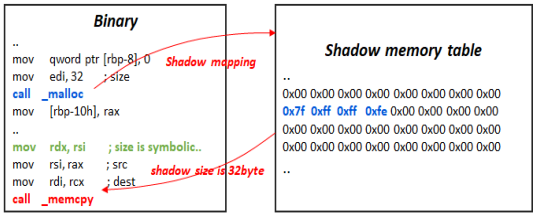

Fig. 6. Obtaining information from shadow memory table

2.4 제약조건 생성 및 삽입

2.3절에서 쉐도우 메모리 테이블을 통해 액세스가 발생한 영역에 할당된 크기를 획득 가능함을 보였다. 이전에 할당된 크기를 *alloc_size*라고 할 때, 액세스가 발생한 크기 *access_size* 보다 작을 경우를 제약조건으로 추가한다. 즉, *alloc_size* 〈 *access_size*의 식을 제약조건으로 생성한 뒤 현재 state에 강제로 삽입한다.

만약 쉐도우 메모리 테이블을 통해 얻어온 *alloc_size*가 0이라면, 해당 주소는 이미 할당 해제

된 메모리 영역이다. 즉, 이 경우 use-after-free가 발생했다고 판단한다. 만약 *alloc_size* 〈 *access_size*에 해당하는 조건이 추가됨으로 인해 현재 state의 제약 조건이 해결 불가능한 조건이 되었다면, 이는 OOB가 발생할 수 없음을 의미한다. 다시 말해, 경계값 검증이 충분히 이루어졌다고 판단할 수 있기 때문에 삽입된 조건을 다시 삭제한다. 그렇지 않고, 제약조건 해결이 가능하다면, 그 시점에서 탐색을 종료하고 테스트케이스를 생성한다. 이렇게 생성된 테스트케이스는 OOB를 발생시키는 콘크리트 값이다.

III. 구 현

본 논문에서 제안한 도구는 파이썬 기반의 기호 실행 도구인 Angr(A binary Analysis Framework)[6]를 확장하여 구현하였다. 주요 구성 요소들은 모두 파이썬으로 구현하였으며, Angr의 플러그인 형태로 동작한다.

3.1 쉐도우 메모리 매핑 라이브러리

Table 2는 쉐도우 메모리 테이블을 생성하는 알고리즘이다. 해당 함수는 힙 메모리 할당 함수가 호출될 때마다 호출된다. 힙 메모리 8바이트는 쉐도우 메모리 1바이트로 매핑되며, 3.2절에서 언급한 바와 같이 첫 비트와 마지막 비트를 0으로 설정하기 위해 8바이트의 할당 크기를 기준으로 매핑 방식이 다르게 동작한다. 할당되지 않은 쉐도우 메모리 영역은 0으로 초기화되어 있으며, 할당 해제될 경우에도 해당 영역을 0으로 설정한다.

3.2 힙 메모리 모니터링 라이브러리

힙 메모리에 대한 액세스가 발생할 경우, Table 3의 알고리즘과 같이 쉐도우 메모리를 참조한다. 참조 영역에 기록된 값이 0x7E라면, 0111 1110(2)으로 할당 크기가 8바이트 이하임을 알 수 있다. 만약 0x7F 즉, 0111 1111(2) 값이 기록되어 있다면 할당 크기는 8바이트 이상이며, 0xFE(1111 1110) 값으로 표기되는 마지막 위치를 찾을 때까지 루프를 반복하게 된다. 이와 같이, 인덱스를 통해 할당된 주소의 마지막 위치를 확인한 뒤 시프트 연산을 통해 할당된 힙 버퍼의 크기를 획득한다.

Table 2. Shadow memory mapping Algorithm

Algorithm1 Shadow Memory Mapping

INPUT *addr:* allocated or freeed memory address
size: allocated or freed size
sig: called system call signal ➤ *sig* = {*1, 2*}
OUTPUT *kHeapShadow*: shadow memory table ➤ $T = \{l_0, l_1, l_2, \ldots, l_n\}$
DECLARE *kHeapBeg, kHeapEnd*: first and last heap memory location
indexBeg, indexEnd: first and last shadow memory index
BEGIN
IF NOT *addr* is over *kHeapbeg* **AND** under *kHeapEnd* **THEN**
return 0 ➤ this address does not belong to heap memory
indexBeg ← *(addr – kHeapBeg)* >> 3
indexEnd ← *(addr – kHeapBeg + size* - 1*)* >> 3
L ← shadow index list ➤ $L = \{l_{indexBeg}, l_{indexBeg+1}, \ldots, l_{indexEnd}\}$
IF *sig* is *1* **THEN** ➤ called allocation function
IF *indexBeg* is *indexEnd* **THEN** ➤ *size* is under 8bytes
kHeapShadow[*indexBeg*] ← *0x7E* ➤ 0111 $1110_{(2)}$
ELSE THEN ➤ *size* is over 8bytes
kHeapShadow[*indexBeg*] ← *0x7F* ➤ 0111 $1111_{(2)}$
kHeapShadow[*indexEnd*] ← *0xFE* ➤ 1111 $1110_{(2)}$
WHILE *index* in *L* **THEN** ➤ except first and last
kHeapShadow[*index*] ← *0xFF* ➤ 1111 $1111_{(2)}$
IF *sig* is *2* **THEN** ➤ called free function
WHILE *index* in *L* **THEN** ➤ include first and last
kHeapShadow[*index*] ← *0x00*
return *kHeapShadow*
END

3.3 제약조건 삽입 및 테스트 케이스 획득

Table 4는 제약조건 삽입을 통해 OOB를 발생시키는 테스트케이스를 생성하기 위한 알고리즘이다. 동적 기호 실행 과정에서 메모리 액세스를 발생시키는 콜백 함수를 추적한다. 해당 함수에 전달되는 크기 값(*size*)의 심볼릭 여부를 판단하고, 제약 조건을 삽입한다. 이 때, 액세스가 발생한 주소(*addr*)에 할당된 버퍼의 크기(*allocatedSize*)보다 액세스 크기(*size*)가 더 큰 값을 갖도록 제약조건(*size* 〉

allocatedSize)을 삽입한다. 만약 해당 제약조건의 해결이 가능하다면, 힙 OOB를 발생시키는 테스트케이스를 획득할 수 있다.

Table 3. Obtaining allocated size Algorithm

Algorithm2 Obtaining allocated size

INPUT *addr:* accessed memory address
kHeapShadow: shadow memory table ➤ $T = \{l_0, l_1, l_2, ..., l_n\}$
OUTPUT *allocatedSize*: allocated memory size ➤ referenced shadow table
DECLARE *kHeapBeg, kHeapEnd*: first and last heap memory location
indexBeg, indexEnd: first and last shadow memory index
BEGIN
indexBeg ← *(addr – kHeapBeg)* >> 3
indexEnd ← *(kHeapEnd – kHeapBeg)* >> 3
L ← shadow index list ➤ $L = \{l_{indexBeg}, l_{indexBeg+1}, ..., l_{indexEnd}\}$
WHILE *index* in L **THEN** ➤ include first and last
IF *kHeapShadow*[*index*] is *0x00* **THEN**
return *0* ➤ not allocated area
allocatedSize ← *allocatedSize* + 1
IF *kHeapShadow*[*index*] is 0x7E or 0xFE **THEN**
allocatedSize ← *allocatedSize* << 3
return *allocatedSize*
END

Table 4. Obtaining OOB testcases Algorithm

Algorithm3 Obtaining OOB testcases

INPUT *addr:* accessed memory address
size: accessed memory size
allocatedSize: allocated memory size
constraints: before constraints
OUTPUT *constraints*: after constraints
DECLARE *kHeapBeg, kHeapEnd*: first and last heap memory location
indexBeg, indexEnd: first and last shadow memory index
BEGIN
IF *size* is *NOT 0* and *symbolic variable* **THEN**
IF *allocatedSize* is *0* **THEN**
return *constraints* ➤ use-after-free vulnerability
IF *allocatedSize* is upper *0* **THEN**
return *constraints* ← *constraints* + (*size* > *allocatedSize*)
➤ heap OOB vulnerability
END

IV. 평 가

제안하는 방법의 효과를 확인하기 위해 바이너리 데이터셋에 대한 평가를 수행한다. 기존에 사용되었던 퍼징을 이용한 취약점 탐색 및 전통적인 DSE 기술 방식과의 비교를 통해 효율성을 검증한다.

4.1 실험 환경(Experiment Setup)

실험에 사용된 호스트 시스템은 Intel Core i7-3770 CPU 3.0GHz와 16GB RAM 사양의 16.04 Ubuntu 64bit 운영체제이다. 취약점 점검을 위해 사용된 동적 기호 실행 도구는 Angr이며, 동적 분석 및 계측(instrumentation)을 위해 파이썬 모듈을 활용했다. 이러한 파이썬 모듈을 활용해 라이브러리 후킹 및 심볼릭 수식 적용 등을 수행하였으며, 동적 계측 과정에서 쉐도우 메모리를 구축하고 후킹된 라이브러리를 통해 이를 관리했다.

4.2 실험(Experiments)

4.2.1 CVE-2016-2385 취약점 재연

CVE-2016-2385는 Kamailio 4.3.4(Linux SIP Server)에서 전달되는 검증되지 않은 입력 메시지를 사용자가 제어함으로써 발생하는 힙 오버플로우 취약점이다[7].

msg는 sip_msg 구조체 포인터이며, Kamailo에서 처리하는 SIP 패킷이다. buf는 패킷의 내용을 저장하는 버퍼인데, memcpy를 호출하는 과정에서 len이 버퍼의 크기보다 클 경우 오버플로우가 발생할 수 있다.

```
1   Int encode_msg(struct sip_msg *msg, char
    *payload,int len)
2   {
3      ...
4      /*now we copy the actual message after
    the headers-meta-section*/
5      memcpy(&payload[j],msg->buf,msg->len);
6      LM_DBG("msglen = %d,msg starts at
    %dWn",msg->len,j);
7      j=htons(j);
8      ...
9   }
```

Fig. 7. encode_msg.c(CVE-2016-2385)

```
[!] allocated size  3200
[!] allocated address: 0xc40008c0 alloc_size: 3200
[!] called malloc! shadow memory mapping!

0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x7f 0xff 0xff 0xff 0xff 0xff
0xff ... 0xfe 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
...
```

Fig. 8. Shadow memory table(CVE-2016-2385)

```
[!] size is symbolic!
[!] succeeded in getting shadow_size:  3200
[!] added constraints (size > shadow_size)

...kecm38140pkcskcs230p...
...kecm38540pkcskcs230p...
...0z0pj8100r42803k9szz...
...kecm38589pkcskcs230p...
...kecm38578pkcskcs230p...
...3dke38589pkhsh3v23pp...
...0z0pj8100r42802k9szb...
...
```

Fig. 9. Generating testcase(CVE-2016-2385)

실험을 위해 해당 취약점의 코드 스니핏을 활용했다. Fig.7에서 나타난 것과 같이 상수로 정의되어 있는 3200바이트 크기의 힙이 할당되며, 이에 상응하는 쉐도우 메모리 테이블이 Fig.8과 같이 구성된다.

이후 해당 영역에 대한 액세스가 발생할 때, 할당된 3200보다 큰 크기의 액세스를 발생시키는 제약조건을 삽입하고, 테스트케이스를 추출한다. 실험 결과 Fig.9와 같이 이에 영향을 미치는 영역이 정상적으로 변조되었음을 확인할 수 있었고, 생성된 테스트케이스가 실제로 힙 오버플로우를 발생시키는 것을 확인할 수 있었다.

4.2.2 평가

제안하는 방법의 평가를 위해 과거 취약점 발생사례가 있는 소프트웨어를 활용했다. 대상 소프트웨어들은 사용자 입력값에 의해 액세스 크기가 조절 가능하며, 이로 인해 힙 오버플로우가 발생한다. 퍼징 및 전통적인 기호 실행과의 비교를 수행했으며, 취약점이 발생하는 영역을 최초로 커버하는 시점을 기준으로 테스트케이스를 추출했다. 실험 결과, Table 5에서 나타난 것처럼 제안하는 도구를 이용해 힙 오버플로우를 발생시키는 정확한 테스트케이스를 추출할 수 있음을 확인할 수 있었다.

V. 관련 연구

5.1 ASan(AddressSanitizer)

ASan(Address Sanitizer)은 C/C++로 작성된 프로그램의 메모리 에러를 탐지하기 위한 도구이다. ASan은 경계값 밖의 접근(out-of-bounds accesses)을 탐지하고, 힙 메모리에서 발생하는 use-after-free 버그 탐지 또한 가능하다. 뿐만 아니라 힙 버퍼 오버플로우, 스택 버퍼 오버플로우 및 메모리 누수 등의 메모리 에러 또한 탐지가 가능하다.

해당 도구는 컴파일러 계측 모듈과 malloc 및 free 함수를 대체하는 런타임 라이브러리로 구성되어 있다. 이러한 런타임 라이브러리를 통해 할당된 영역의 주위 메모리를 오염시키며, 할당 해제된 메모리는 격리 상태로 대체한다. 프로그램의 모든 메모리

Table 5. Evaluation of proposed tool

Target SW	Vulnerability	Fuzzing (crash testcase)		Conventional DSE (crash testcase)		Propsed tool (crash testcase)	
		normal	crash	normal	crash	normal	crash
Kamailo 4.3.4	CVE-2016-2385 (heap overflow)	O	X	O	X	O	O
libgd 2.1.1	CVE-20163074 (heap overflow)	O	X	O	X	O	O
Make 3.8.1	N/A (heap overflow)	O	X	O	X	O	O

Table 6. Compare tools that detect memory error

	Asan	Memcheck	Dr.Memory	Proposed tool
Technology	CTI	DBI	DBI	DBI+DSE
Heap out-of bounds	O	O	O	O
Stack out-of bounds	O	X	X	extendable
Global out-of bounds	O	X	X	O
Use-after-free	O	O	O	O
Use-after-return	△	X	X	extendable
Uninitialized reads	X	O	O	X

접근이 컴파일러에 의해 특수한 형태로 변형되며, 이를 이용해 쉐도우 메모리를 구성한다.

ASan은 Valgrind(Memcheck)[15], Dr. Memory[16] 등이 스택이나 전역 변수의 범위를 벗어나는 버그를 찾을 수 없다는 단점을 극복했다. 현재 퍼징 등의 기술과 결합해 메모리 에러 탐지에 유용하게 사용되지만, 본 논문에서 제안하는 방법과는 달리 C/C++로 작성된 소스코드를 대상으로만 적용이 가능하다는 한계점이 존재한다.

5.2 동적 기호 실행(dynamic symbolic execution)

동적 기호 실행은 최초에 소스 코드 기반으로 동작하도록 설계되었지만 현재, 바이너리를 대상으로 동작이 가능하도록 확장되었다. 이에 대한 관심이 높아지면서 SAGE[8]를 필두로 S2E[9], FuzzBALL[10], Mayhem[11], Driller[12]와 같은 바이너리 기반 동적 기호 실행 도구들이 등장했다. 뿐만 아니라 Dowser[13]와 같은 해당 기술을 활용한 연구도 있었다. Dowser는 크래시를 유발하는 적절한 입력을 생성한다는 측면에서 본 논문의 목표와 유사하다. 하지만 최초 샘플 테스트케이스가 필요하다는 것과 버퍼 오버플로우만이 탐지 가능하다는 한계점을 가지고 있다.

Dowser와 유사하게 BuzzFuzz[14] 또한 샘플 입력 테스트케이스에 오염 추적(taint-tracking)을 적용하여 미리 정의된 공격지점을 검사한다. 즉, Dowser와 마찬가지로 사전 정보가 필요하다는 단점이 있다.

VI. 결론 및 향후 연구

본 논문에서 동적 기호 실행과 쉐도우 메모리 테이블을 활용하여 힙 메모리에서 발생하는 OOB 취약점을 자동으로 탐지하는 방법을 제안하였다. 제안하는 방법을 통해 취약한 블록 탐색에 성공한다면 반드시 크래시를 유발하는 테스트케이스를 생성할 수 있음을 증명했다. 뿐만 아니라 전통적인 동적 기호 실행과는 다르게 명확한 목표 지점을 설정하지 않더라도 취약점 탐지가 가능하며, 최소한의 탐색으로 테스트케이스를 생성함으로써 경로 폭발 문제도 간접적으로 완화시킬 수 있었다.

본 연구를 통해 힙 메모리 취약점 분석의 자동화가 가능함을 보였고, 소프트웨어 안전성 검증 자동화 연구 분야의 발전에 기여할 수 있을 것으로 기대한다.

References

[1] Secunia. "Resources vulnerability review 2017". http://secunia.com/resources/vulnerability-review/introduction/.

[2] C. Details. "Vulnerability distribution of CVE security vulnerabilities by type". http://www.cvedetails.com/vulnerabilities-by-types.php.

[3] DARPA. "Cyber Grand Challenge". https://www.darpa.mil/program/cyber-grand-challenge

[4] SEREBRYANY, K., BRUENING, D., POTAPENKO, A., AND VYUKOV, D. "AddressSanitizer: A fast address sanity checker". In Proceedings of USENIX Annual Technical Conference. 2012.

[5] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley. "Enhancing symbolic execution with veritesting". In Proceedi

ngs of the International Conference o n Software Engineering (ICSE), pages 1083 - 1094. ACM, 2014.

〔6〕 Y. Shoshitaishvili, R. Wang, C. Hause r, C. Kruegel, and G. Vigna. "Firmalic e - automatic detection of authentica tion bypass vulnerabilities in binary f irmware". In Proceedings of the Symp osium on Network and Distributed Sy stem Security (NDSS), 2015.

〔7〕 CVE Details, "The ultimate security v ulnerability datasource", https://www. cvedetails.com/cve/CVE-2016-2385/.

〔8〕 P. Godefroid, M. Y. Levin, and D. Mo lnar. SAGE: Whitebox fuzzing for sec urity testing. Communications of the ACM, 55(3):40 - 44, 2012.

〔9〕 V. Chipounov, V. Kuznetsov, and G. Candea. S2E: A platform for in-vivo multi-path analysis of software syste ms. In Proceedings of the Sixteenth I nternational Conference on Architectu ral Support for Programming Languag es and Operating Systems, ASPLOS X VI, pages 265 - 278. ACM, 2011.

〔10〕 D. Caselden et al. Transformation-aw are Exploit Gen- eration using a HI-C FG. Tech. rep. University of Cal- ifor nia, Berkeley, 2013.

〔11〕 S. K. Cha, T. Avgerinos, A. Rebert, a nd D. Brumley. Unleashing Mayhem o n binary code. In Proceedings of the I EEE Symposium on Security and Priv acy, 2012.

〔12〕 N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Sh oshitaishvili, C. Kruegel, and G. Vign a, "Driller: Augmenting fuzzing throu gh selective symbolic execution," in N DSS'16. Internet Society, pp. 1 - 16. 2 016.

〔13〕 I. Haller, A. Slowinska, M. Neugschw andtner, and H. Bos. Dowsing for ove rflows: A guided fuzzer to find buffer boundary violations. In Proceedings of the USENIX Security Symposium, 201 3.

〔14〕 V. Ganesh, T. Leek, and M. Rinard. Taint-based directed whitebox fuzzin g. In Proceedings of the International Conference on Software Engineering (ICSE), 2009.

〔15〕 Nicholas Nethercote and Julian Sewar d. "Valgrind: A framework for heavyw eight dynamic binary instrumentation ". In Proc. of the ACM SIGPLAN Conf erence on Programming Language Des ign and Implementation (PLDI '07), p ages 89 - 100, June 2007

〔16〕 Derek Bruening and Qin Zhao. "Practi cal memory checking with Dr. Memor y". In Proc. of the International Symp osium on Code Generation and Optimi zation (CGO '11), pages 213 - 223, Apr il 2011.

〈저자소개〉

강 상 용 (Sangyong Kang) 학생회원
2014년 8월: 전남대학교 컴퓨터공학 공학사
2014년 9월~2016년 8월: 전남대학교 정보보안협동과정 이학석사
2016년 9월~현재: 전남대학교 정보보안협동과정 박사과정
〈관심분야〉 소프트웨어 취약점 분석 및 탐지, 시스템 보안

박 성 현 (Sunghyun Park) 학생회원
2016년 8월: 전남대학교 컴퓨터정보통신공학 공학사
2016년 9월~2018년 2월: 전남대학교 정보보안협동과정 이학석사
2018년 2월~현재: 전남대학교 정보보안협동과정 박사과정
〈관심분야〉 취약점 분석, 악성코드 탐지, 시스템 보안

노 봉 남 (Bongnam Noh) 종신회원
1978년: 전남대학교 수학교육과 학사
1982년: KAIST 대학원 전산학과 석사
1994년: 전북대학교 대학원 전산과 박사
1983년~현재: 전남대학교 전자컴퓨터공학부 교수
2000년~현재: 전남대학교 시스템보안연구센터 소장
〈관심분야〉 정보보안, 시스템 및 네트워크 보안