Journal of The Korea Institute of Information Security & Cryptology ISSN 1598-3986(Print)
VOL.24, NO.1, Feb. 2014 ISSN 2288-2715(Online)
http://dx.doi.org/10.13089/JKIISC.2014.24.1.215

정의되지 않은 행동에 의한 안전성 검사 무효화 탐지 기법*

이 종 협†‡
한국교통대학교

Detection of invalidated sanity checks caused by undefined behaviors*

JongHyup Lee†‡
Korea National University of Transportation

요 약

C언어에는 프로그래머의 의도와는 다르게 동작하는 정의되지 않은 행동(undefined behavior)들이 존재한다. 정상적인 데이터를 확인하기 위한 안전성 검사에서 정의되지 못한 행동에 해당되는 경우를 인지하지 못하고 사용하여 안전성 검사가 컴파일러에 의해 무효화 되는 문제점이 발생한다. 본 논문에서는 이러한 문제점들을 해결하기 위해 소스코드에서 안전성 검사를 표기하고 실행파일에서 유효성을 확인하는 자동화된 시스템을 제안한다.

ABSTRACT

C programming language has undefined behaviors, which cause unintended execution of a program. When a programmer adds sanity checks without caring about undefined behaviors, compilers may misunderstand and invalidate the sanity checks. Thus, we propose an automated system to detect invalidated sanity checks by marking sanity checks in source code and checking the marks in binary code.

Keywords: Software Security, Undefined behavior

I. 서 론

C언어는 assembly보다 조금 더 높은 수준의 프로그래밍 기능과 높은 성능을 제공하기 위해 개발되었다. 하지만 성능을 극대화하기 위하여 이상상황에 대한 안전성이 희생되었다. 흔히 '정의되지 않은 동작(undefined behavior)'이라고 불리는 명시되지 않은 경우들이 C언어에 남아있어 프로그램의 버그나 보안의 취약점으로 연결되기도 한다. 설정되어 있는 메모리 영역을 벗어난 영역에 쓰기 명령을 수행하는 버퍼 오버플로우(buffer overflow)나 할당해제(free)된 메모리 공간을 쓰는 use after free등의 보안 취약점들이 정의되지 않은 동작에서 발생하는 대표적인 보안 취약점 들이라고 할 수 있다.

정의되지 않은 동작들은 직접적인 보안 문제들뿐만 아니라 프로그래머의 잘못된 이해나 실수를 유발하게 하여 또 다른 보안의 문제점들을 발생시킨다. 예를 들어 자료형 크기 이상의 쉬프트(shift) 연산이 일어나는 크기 초과 쉬프트(oversized shift) 경우도 정의되지 않은 동작으로 x86 시스템과 powerpc 시스템에 따라 다른 결과를 가진다. 하지만 이러한 상황을 인지하지 못하고, 특정 환경에서만 얻을 수 있는 정의되지 않은 동작의 결과를 일반적인 상황에 적용하면서 버그들이 발생한다. 특히 프로그램내의 정수 오버플로우(integer overflow)와 같이 정상적이지 않은 데이터를 검사하는 안전성 검사 코드에서 정의되지 않은 동

접수일(2014년 1월 9일), 수정일(2014년 1월 21일), 게재확정일(2014년 1월 21일)
* 이 논문은 2012년도 정부(미래창조과학부)의 재원으로 한국연구재단의 지원을 받아 수행된 기초연구사업임(No. 2012R1A1A1044693)
† 주저자, jhlee@ut.ac.kr
‡ 교신저자, jhlee@ut.ac.kr (Corresponding author)

작을 잘못 적용한 경우들이 자주 발생하게 된다. 잘못 적용된 안전성 검사는 의도했던 문제를 검출하지 못하게 되거나, 컴파일러 최적화 과정에서 의미없는 코드로 분류 후 삭제되어 안전성 검사가 무효화되는 경우가 발생한다. 무효화된 안전성 검사는 프로그램을 다시 보안의 위험에 노출된다는 점에서 큰 문제가 된다.

본 논문에서는 정의되지 않은 동작 때문에 발생하는, 소스코드에 존재하던 안전성 검사가 실제 실행파일에서 무효화되어 버리는 문제를 해결하기 위하여 소스코드의 안전성 검사가 실행파일에 적용되어 있음을 확인하는 기법을 제안한다. 제안하는 시스템에서는 소스코드를 분석하여 안전성 검사를 위해 사용한 조건문 위치에 실행파일에도 남아있을 표기를 주입한다. 주입된 소스코드의 컴파일 후, 해당 표기가 실행파일에 남아있음을 확인하여 무효화된 안전성 검사를 통해 발생할 수 있는 보안 문제점을 예방한다.

II. 정의되지 않은 동작에 의해 발생하는 문제점

C 언어에서 정의되지 않은 행동에 의해서 발생되는 문제점들은 다양하게 나타나지만[1,2,3], 실제 프로그램에서 발생하여 보안의 문제점으로 연결되어 잘 알려진 다음과 같은 경우들을 고려해볼 수 있다.

2.1 부호 있는 정수형 데이터의 오버플로우

부호가 있는 정수형(signed int)의 타입을 가지는 데이터에 대한 연산 과정 중에, 최대 또는 최소값을 넘어서는 오버플로우가 발생하는 경우의 결과는 정의되어 있지 않다. 흔히 양의 최대값을 넘어서는 경우에는 음의 값으로 음의 최소값보다 작아지는 경우에는 양의 값으로 돌아갈(wrapping around) 것으로 예상하고 프로그램을 작성하지만, 이는 정의되어 있지 않은 행동이기 때문에 gcc와 clang과 같은 컴파일러들에서는 부호 있는 정수형 데이터는 오버플로우가 일어나지 않는다고 가정해 버린다. 따라서 부호 있는 정수의 연산과정에서 오버플로우 발생을 확인하기 위한 안전성 검사를 다음과 같이 수행하는 경우,

```
if ( a + 10000 < a ) {
        ... 정수 오버플로우 처리 루틴 ...
}
```

컴파일러는 `(a + 10000 < a)`라는 조건은 부호 있는 정수의 오버플로우에 해당되어 발생할 수 없으므로 항상 거짓일 수밖에 없다고 가정하고 최적화의 과정에서 위의 안전성 검사 조건문 자체를 삭제해 버린다. 하지만 이 안전성 검사는 x86 플랫폼에서는 유효함에도 불구하고 컴파일러에서 삭제되어버리기 때문에 실제 정수 오버플로우가 일어난 경우에 이를 방지하지 못하는 취약점 문제로 이어지기도 한다[4].

2.2 크기 초과의 쉬프트 연산

C언어에서 자료형 크기 이상의 쉬프트 연산에 대한 결과는 정의되어 있지 않다. 즉, 32-bit 머신 환경에서 `x << 32` 와 같은 연산의 결과가 정의 되어 있지 않다. 실제 이 결과는 x86과 powerpc이 서로 다른 결과를 가지기도 하지만(x86에서는 1, powerpc에서는 0), clang과 같은 컴파일러는 해당 연산이 정의되어 있지 않기 때문에 항상 자료형 크기 이상의 쉬프트 연산은 존재하지 않는다고 가정한다. 따라서 32-bit 머신 환경에서 변수 a의 값이 32보다 큰지를 확인하려는 다음 안전성 검사 코드에 대하여,

```
if ( ! (1 << a) ) { ... }
```

clang 컴파일러는 `(1 << a)`의 연산에서 a는 32보다 클 수 없다고 가정하고 `(1 << a)`의 조건이 무조건 참이라고 생각하여 해당 안전성 검사 조건문을 최적화 과정에서 삭제한다. 이 또한 프로그래머의 의도와는 다르게 보안 취약점의 문제로 연결된다[5].

2.3 포인터 연산에서의 오버플로우

부호 있는 정수형 데이터의 오버플로우가 정의되어 있지 않은 동작인 것처럼 포인터에 대한 연산의 오버플로우도 정의 되어 있지 않다. 예를 들어 `ptr`이라는 포인터에 대하여 `(ptr + size < ptr)`와 같이 포인터 오버플로우를 통하여 `size` 변수가 너무 큰 값을 가지고 있는지 확인하려는 경우에도 컴파일러가 포인터 연산이 오버플로우 되지 않는다고 가정하고 해당 안전성 검사를 무효화해 버리는 문제가 발생한다[6].

III. 안전성 검사 유효성 확인 시스템

정의되지 않은 동작들에 의해 안전성 검사가 무효화 되는 경우가 발생하지만, 정의되지 않은 동작들은

종류가 다양하고 복합적인 문제로 나타나기 때문에 개발과정에서 모두 고려하기란 쉽지 않다. 따라서 본 논문에서는 안전성 검사가 무효화 되었는지의 유무를 자동적으로 확인할 수 있는 시스템을 제안한다. 프로그래머의 의도대로 안전성 검사가 실행파일에도 남아있는지 확인하기 위하여, 제안하는 시스템은 소스코드의 안전성 검사마다 고유한 값을 가지면서 실행파일 파일에서도 고유값을 확인 할 수 있는 표기를 주입한다. 주입된 표기는 컴파일러의 최적화 과정에서 안전성 검사가 의도와 달리 삭제되는 경우 함께 삭제되어, 해당 안전성 검사가 실행파일에 적용되었는지의 유무를 실행파일에서 바로 검사할 수 있다.

Fig. 1은 제안하는 시스템의 전체 과정을 보여준다. 코드 작성 시 안전성 검사에 해당되는 조건문을 추가할 때 프로그래머가 직접 코드에 표시하면, 제안된 시스템에서 자동으로 표시된 안전성 검사 조건문들을 찾아 해당 안전성 검사마다 컴파일 후 실행파일에서 고유번호로 확인할 수 있는 새로운 표기를 조건문에 주입한다. 주입된 표기는 별도로 저장되어 코드가 컴파일 된 이후에 실행파일에서도 주입된 표기들이 모두 남아있는지를 확인하여 안전성 검사가 유효한지 확인한다.

3.1 안전성 검사 표시

프로그램 작성 과정에서 안전성 검사를 위하여 사용하는 조건에 대해서는 프로그래머가 간단하게 표시한다. 소스코드에 영향을 주지 않으면서 특정 조건을 표시하기 위하여 "CHECK"란 keyword를 다음과 같이 정의하여 사용한다.

```
#define CHECK
    (void *__attribute((sec_check))) 1 &&
```

전처리 과정에서 CHECK 키워드는 "sec_check"란 attribute를 부여하는 코드로 바뀐다. 따라서 프로그래머는 안전성 검사 조건에 다음과 같이 표시한다.

```
if  (CHECK  a + 10000 < a ) { ... }
```

3.2 고유 표기 주입

표시된 안전성 검사들은 특정한 attribute를 가지고 있기 때문에 시스템에서 자동적으로 찾을 수 있다. 제안된 시스템은 검색된 안전성 검사에 실행파일에서 찾을 수 있는 표기를 주입한다. 주입하는 표기는 다음의 두 가지 조건을 만족해야 한다.

1) 주입한 표기를 실행파일에서도 쉽게 찾을 수 있어야 한다.
2) 안전성 검사가 컴파일러에 의해 무효화(삭제)되는 경우 주입된 표기도 함께 삭제되어야 한다.

이러한 조건을 만족시키기 위하여, 제안하는 시스템에서는 표시된 안전성 검사 조건문의 then block의 시작 위치에 다음과 같은 표기를 주입한다.

```
asm("prefetchnta 0x1a2a3a4a");
```

prefetchnta는 prefetch를 위한 명령어로 side effect가 없기 때문에 Control Flow Integrity(CFI)에서도 label로 사용되는 명령어이다. 뒤의 0x1a2a3a4a은 해당 안전성 검사를 구별하기 위해 자동 생성된 고유값을 의미한다. 즉, 3.1절에서 CHECK로 표기된 안전성 검사는 다음과 같이 변형된다.

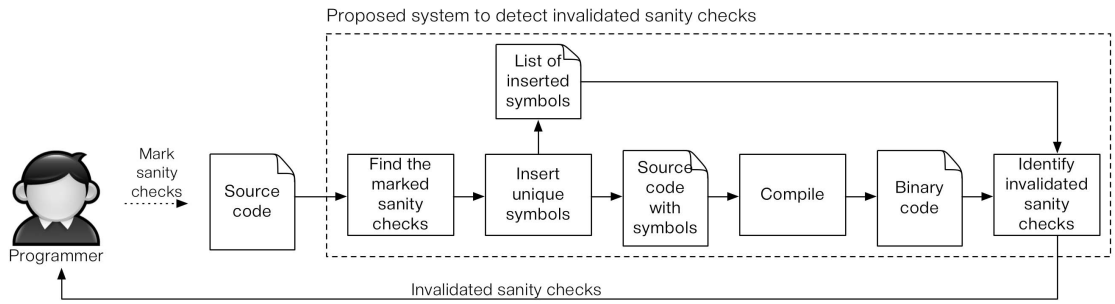

Fig. 1. Process of the proposed system

Table 1. Invalidated sanity checks depending on compiler optimization options and the result of the detection by the proposed system. (The results of the proposed system are showed in ())

	gcc		clang	
	O0	O2	O0	O2
Signed integer overflow	X (X)	O (O)	X (X)	O (O)
Oversized shift	X (X)	X (X)	X (X)	O (O)
Pointer arithmetic overflow	X (X)	O (O)	X (X)	O (O)

```
if  (a + 10000 < a ) {
    asm("prefetchnta 0x1a2a3a4a");
    ... }
```

안전성 검사에 표기가 주입되면서 생성된 고유값은 주입된 위치와 함께 별도로 저장된다.

3.3 실행파일에서 안전성 검사 유효성 확인

표기가 주입된 소스코드를 컴파일 후 실행파일에서 주입된 표기들이 남아있는 지를 확인한다. Little endian의 x86 환경에서 주입된 표기는 실행파일에서 (`prefechnta`의 고정된 opcode, `0f 18 05`) + (고유값의 역순)의 형태로 나타난다. 즉 3.2절에서 주입된 표기는 `0f 18 05 4a 3a 2a 1a`와 같이 나타난다. 제안된 시스템에서는 저장되어있는 고유값마다 해당되는 표기를 실행파일에서 찾고, 찾지 못하는 경우 무효화되었다고 간주하여 프로그래머에게 무효화된 안전성 검사를 통보하고 반영할 수 있도록 한다.

IV. 구현 및 검증

제안한 시스템은 CIL[7]을 기반으로 구현하였다. CIL의 visitor를 이용하여 3.1절의 "`sec_check`"란 attribute가 있는 조건문을 찾고, 해당 조건문의 then block에 고유값을 집어넣는 rewrite 작업을 수행하며 해당 고유값을 주입된 위치와 함께 별도의 파일로 저장한다. 컴파일 후 binutils를 활용하여 실행파일의 text 섹션에서 주입되었던 고유값들을 찾는다.

구현된 시스템을 정의되지 않은 동작들의 test case에 적용하여, gcc 4.6.3와 clang 3.0의 두 컴파일러의 최적화 옵션에 따라 무효화된 안전성 검사를 찾아낼 수 있는지 확인하였다. Table 1은 검증 결과를 보여준다. 최적화 옵션에 따라 안전성 검사가 무효화되는 경우가 달라지지만, 제안하는 시스템이 이를 모두 정확하게 탐지할 수 있었음을 알 수 있다.

V. 결 론

본 논문은 자동화된 방법을 통하여 안전성 검사의 유효성을 확인할 수 있는 시스템을 제안하였다. 정상적이지 않은 데이터를 확인하기 위한 안전성 검사 조건은 정의되지 않은 행동을 유발하여 본래의 목적을 달성하지 못하고 무효화될 수 있다. 인지하기 어려운 정의되지 않은 행동의 특수성을 고려할 때, 실행파일에서 안전성 검사의 유효성을 확인하는 접근 방법이 편의성과 안전성에서 유리한 해결책이라 할 수 있다.

References

[1] X. Wang, N. Zeldovich, M. F. Kaashoek, and A. Solar-Lezama, "Towards optimization-safe systems: analyzing the impact of undefined behavior," Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP), pp. 260-275, Nov. 2013

[2] X. Wang, H. Chen, A. Cheung, Z. Jia, N. Zeldovich, and M. F. Kaashoek, "Undefined behavior: what happened to my code?" Proceedings of the 3rd ACM SIGOPS Asia-Pacific conference on Systems, Jul. 2012

[3] C. Lattner. "What every C programmer should know about undefined behavior," http://blog.llvm.org/2011/05/what-every-c-programmer-should-know.html, May 2011.

[4] GCC Bug 30475, "assert(int+100 〉 int) optimized away," Jan. 2007.

[5] Linux Bug 14287, "ext4: fixpoint divide exception at ext4_fill_super," Oct. 2009.

[6] "C compilers may silently discard some wraparound checks," Vulnerability Note VU#162289, US-CERT, Apr. 2008.

[7] G. Necula, S. McPeak, and S. Rahul, "CIL:

Intermediate language and tools for analysis and transformation of C programs," Compiler Construction, pp. 213-218, Jan. 2002.

〈저자소개〉

이 종 협 (JongHyup Lee) 종신회원
2002년 2월: 연세대학교 기계전자공학부 졸업
2004년 2월: 연세대학교 컴퓨터과학과 석사
2009년 8월: 연세대학교 컴퓨터과학과 박사
2009년~2012년: Carnegie Mellon University, CyLab, Postdoc. 연구원
2012년 3월~현재: 한국교통대학교 소프트웨어학과 조교수
〈관심분야〉 소프트웨어 보안, 네트워크 보안