# 실용적인 버퍼 취약점 정적 검출기의 구현

전진성° 김건우 한환수 한태숙
KAIST 전자전산학과 전산학전공
jinseong.jeon@arcs.kaist.ac.kr°, reshout@pllab.kaist.ac.kr, {hshan, han}@cs.kaist.ac.kr

# Practical Buffer Overrun Vulnerabilities Detection using Static Analysis

Jinseong Jeon°, Gunwoo Kim, Hwansoo Han, Taisook Han
Division of Computer Science, KAIST

**요 약**

버퍼 오버런과 같은 소프트웨어의 보안 취약점이 알려진 이후로 이를 해결하기 위한 분석 도구 개발이 다양한 연구그룹에 의해 수행되었다. 하지만 범용 소프트웨어를 분석할 수 있는 실용적인 도구는 많지 않다. 본 논문은 모든 버그를 빠트림 없이 찾는 정적 분석에서 한발 물러나 조금 부정확하지만 빠른 시간 안에 보안 취약점을 검출할 수 있는 방법을 소개하고, 버그가 알려진 소프트웨어에 대한 실험 결과를 통해 제안하는 검출기의 실용성을 보인다.

## 1. 서론

버퍼 오버런(overrun) 또는 버퍼 오버플로우(overflow)는 할당된 버퍼의 경계를 지나 허용되지 않은 영역에 접근하는 것을 말한다. 버퍼 오버런을 통해 허용되지 않은 영역에 값을 기록하면 프로그램 내에서 사용되는 데이터에 영향을 줄 수 있으며, 프로그램의 수행 흐름을 바꿀 수도 있다. 이는 찾기 어려운 버그의 원인이 되거나 문제가 발견되지 않더라도 소프트웨어의 보안 취약점으로 남게 된다[1].

버퍼 오버런이 알려진 이후로 이를 해결하기 위한 연구가 다양한 방법으로 시도되었다. 이들 중에서, 프로그램을 실제로 돌려보지 않고 개발 단계에서 버그를 찾을 수 있다는 장점 때문에 본 연구자들은 정적 분석을 통한 취약점 검출을 연구하게 되었다. 하지만 정적 분석은 프로그램이 취할 수 있는 모든 가능성을 예측해야하기 때문에 실제 생활에서 사용 가능한 속도와 정확성을 동시에 얻기 어렵다.

소프트웨어의 크기가 점차 커져감에 따라 개발자와 사용자에게 보다 실용적인 도구가 되기 위해서는 속도 측면이 더 중요하다고 판단하여, 안정성(soundness)은 조금 포기하더라도 확장성(scalability)있는 분석을 기반으로 속도와 정확성(precision)을 개선시키는 방향으로 연구하였다.

## 2. 선행연구 및 제약식의 정의

D. Wagner 등[2]은 버퍼 취약점을 찾는 문제를 프로그램 도중에 사용된 버퍼의 크기가 할당된 크기를 넘지 않았는가를 확인하는 것으로 생각하였다. 스트링 버퍼에 대해 사용된 크기와 할당된 크기를 제약식(constraint)을 통해 구한 뒤, **len(buf) ≤ alloc(buf)** 조건을 검사하여 버퍼 오버런 가능성을 검출한다.

V. Ganapathy 등[3]은 [2]의 부정확성을 개선하기 위해 노력하였다. 버퍼의 사용된 크기와 할당된 크기를 [2]에서는 정수들의 집합을 최소값과 최대값의 쌍으로 나타내는 인터벌(interval)을 사용하는데 반해, [3]에서는 최소값과 최대값을 위한 변수를 따로 만들고 각각에 대한 방정식을 linear programming 기법을 통해 해결한다. 변수를 나누었기 때문에 정확도 개선을 위한 확장이 용이하고, 실제로 함수 호출 시점 구별[1]에 따른 정확도 개선을 실험을 통해 보이고 있다.

본 논문은 [2]의 제약식 기반의 정적 분석을 토대로 하였다. [2]에서는 스트링 관련 library 함수들에 대한 모델을 주로 다루지만, C 언어에서 스트링 버퍼가 사용되는 경우가 인덱스를 통한 접근과 포인터를 통한 접근도 있기 때문에, 인덱스로 사용될 수 있는 정수형 변수에 대한 값 분석과 포인터 분석을 추가하였다.

$$
\begin{aligned}
constraint &:= s \supseteq e \\
s &:= Len(v) \mid Alloc(v) \mid Range(v) \\
e &:= s \mid i \mid e+e \mid e-e \mid e \times e \mid min(e,\ldots,e) \\
&where\ i \in \mathbb{Z}^{\infty} = \mathbb{Z} \cup \{-\infty, \infty\}
\end{aligned}
$$

[그림 1] 제약식의 정의

이에 따라 프로그램으로부터 알아야 하는 정보들은 스트링 버퍼의 사용된 크기, 할당된 크기, 정수형 변수가 가질 수 있는 값이고, 이를 위한 제약식을 [그림 1]과 같이 디자인하였다. [그림 1]의 s가 가질 수 있는 값의 영역은 [2]와 마찬가지로 인터벌을 사용하였다.

## 3. 제약식의 생성

제약식은 기본적으로 정보의 포함관계를 나타낸다. 따라서 제약식의 생성은 프로그램을 살펴보면서 스트링 버퍼의 사용 혹은 할당과 관련된 부분, 정수형 변수의 값이 달라질 수 있는 부분처럼 정보의 전달이 일어나는 곳에서 이루어진다.

정수형 변수의 입장에서 정보의 전달은 대입(assign)을 통해서 발생한다. 예를 들어, `v = e` 라는 대입문에 대해서 `Range(v)` ⊇ `e` 와 같은 제약식이 생성된다. 단, 우변의 e는 [그림 1]의 제약식으로 표현할 수 있는 연산으로만 이루어졌을 때 가능하다.

1) context-sensitive

| C code | constraint |
|---|---|
| char str[n]; | Alloc(s) ⊇ n |
| p = malloc(n); | Alloc(p) ⊇ n |
| i = strlen(s); | Range(i) ⊇ Len(s) - 1 |
| strcpy(d,s); | Len(d) ⊇ Len(s) |
| strncpy(d,s,n); | Len(d) ⊇ min(Len(src),n) |
| strcat(b,suf); | Len(b) ⊇ Len(b)+Len(suf)-1 |
| gets(buf); | Len(buf) ⊇ 1, Len(buf) ⊇ ∞ |
| p[i] = '\0'; | Len(p) ⊇min(Len(p),Range(i)+1) |

[표 1] 스트링 관련 연산에 대한 제약식 생성의 예[2]

스트링 버퍼의 입장에서도 역시 대입을 통해 정보의 전달이 일어난다. 정수형 변수와 다른 점은 버퍼가 사용되는 정보와 할당된 정보가 같이 전달되어야 한다는 것이다. 예를 들어, p = q 에 대해, Len(p) ⊇ Len(q) 와 Alloc(p) ⊇ Alloc(q) 라는 제약식이 생성된다.

[표 1]은 [2]에서 제안한 스트링 버퍼를 다루는 연산에 대한 모델을 본 논문에서 설명한 [그림 1]의 제약식에 맞게 수정한 예이다. [2]와 달라진 점은, p[i] 와 같이 인덱스를 통한 접근 시에 정수형 변수에 대한 정보를 활용하여 더 정확한 분석을 할 수 있다는 것이다.

분석의 확장성을 위해 전술한 제약식의 생성은 프로그램의 수행 흐름을 고려하지 않고 동작한다.[2] strcat과 같은 경우를 제외하고는 대부분의 스트링 관련 함수는 같은 인자에 대해 동일한 동작(idempotent)을 하기 때문에 수행 흐름을 고려하지 않음으로 인한 정확성의 감소는, 얻을 수 있는 성능향상에 비해 그리 심각하지 않다[2]. 또한 함수 단위 분석을 할 때, 함수 호출 시점에 대한 구별도 하지 않는다.[3] 대신, 함수에 전달되는 인자(actual parameter)와 전달받는 인자(formal parameter) 사이에 대입문이 있는 것으로 여기고, 함수 내부의 결과(return)가 함수를 호출하는 곳으로 전달되도록 하였다.

## 4. 제약식의 해

제약식을 푸는 과정은 생성된 제약식을 모두 만족하는 해를 찾는 과정이다. 조건을 만족하는 해는 여러 가지가 있을 수 있으나, 본 연구에서는 구할 수 있는 해 중에서 가장 정확한 해를 빠른 속도로 구하는 방법을 제안한다.

일반적으로 상호 재귀적(mutual recursive)인 방정식들은 단조성(monotone)을 가졌다면 해의 변화가 없을 때까지 이전 과정에서 구한 값을 사용해서 다시 푸는 방법(chaotic iteration[4])으로 모든 방정식을 만족하는 가장 작은 해(least fix-point solution)를 찾을 수 있다. 하지만 해가 속한 영역(domain)이 무한하거나, 유한하되 그 높이가 긴 경우에는 해를 찾는 과정 또한 끝나지 않거나, 많은 시간을 소모한다. 이때에는 넓히기(widening)를 정의해야 하며, 방정식 간의 관계 중 순환(cycle)이 있는 경우 그 중 한 방정식을 택해 넓히기를 적용하면 분석의 종료를 보장하면서 효율적인 해를 구할 수 있다[4]. 본 연구의 제약식은 포함관계를 나타내므로 단조 증가함수

```
Constraint-Solver
  graph initializaion.
  work_list := all the nodes of graph.
  call Fix-Point

Fix-Point
  If work_list is not empty, do
    node := pop work_list.
    If node.visited is not black, do
      If node.visited is gray, call Awake-DFS(node).
      cycle := empty and call Visit(node).
    call Fix-Point.

Visit(node)
  node.visited := gray.
  revisit_list := all the successors of node.
  While the range of node isn't changed,
    call ReVisit(node,revisit_list).
  node.visited := black.

Revisit(node,revisit_list)
  For each element(alias succ) of revisit_list, do
    If corresponding constraint make improvement, do
      remove succ from revisit_list.
      update the range of node.
      For each case of succ.visited,
        white: push cycle, call Visit(succ), pop cycle.
        gray : call Handle-Cycle.
        black: succ.visited := gray and push work_list.
    If not, add succ to revisit_list again.
```

[그림 2] 제약식을 푸는 효율적인 알고리즘

이고, 제약식을 푼 해는 무한한 영역인 인터벌이기 때문에 전술한 과정이 필요하다.

순환을 찾기 위해서 제약식이 내포하고 있는 정보들 사이의 관계를 그래프로 나타낸 뒤 활용한다. 그래프의 노드(node)는 [그림 1]에 나타난 제약 언어의 s에 해당하는 정보이고, 노드 사이의 연결(edge)은 정보 사이의 포함관계로 나타낼 수 있다. 깊이 우선순위(Depth-First)로 노드들을 방문하고, 방문 전, 방문 중, 그리고 방문이 끝났음을 색깔로 표시하면서 순환을 찾는다.

만들어진 그래프의 연결이 해당 방정식의 변화가 어떤 노드들에게 영향을 주는 지를 나타낸다는 사실과 방문 상태를 나타내는 색깔을 활용하면 방정식을 푸는 반복적인 과정을 할일만 하는 방식(worklist)으로 개선할 수 있다. 그래프의 연결 관계를 할일만 하는 방식에서는 다음 할일이 무엇인지 지정하는 역할로 사용할 수 있고, 순환을 찾는 도중 방문이 끝났음을 나타내는 검은색과 방문 중임을 나타내는 회색을, 할일만 하는 방식에서는 각각 할일이 다 끝났다는 것과 다시 살펴봐야 한다는 것을 나타내도록 활용할 수 있다.

[그림 2]는 [2]에서 설명하는 반복적인 방식을 개선하여, 할일만 하는 방식으로 제약식을 효율적으로 푸는 알고리즘을 기술하고 있다. 깊이 우선순위로 노드를 방문하는 것을 Visit과 ReVisit으로 분리시킨 이유는 후임자(successor) 방문 순서 때문에 살펴봐야 할 제약식을 놓치는 경우를 방지하기 위해서이다. [그림 2]에서 설명하지 않은 Awake-DFS 함수는 해당 노드의 결과가 영향을 주는 노드들을 방문하면서 다시 할일이 있음을 알려주는

2) flow-insensitive
3) context-insensitive

| Software | wu-ftpd-2.5.0 |
|---|---|
| SLOC[7] | 16,373 lines |
| Known BUG | CA-1999-13[8] |

| Analysis Time | CFG | PTA | GEN | SOL | Total |
|---|---|---|---|---|---|
| | 0.05s | 0.44s | 0.26s | 0.08s | 0.83s |

| Sound | Over | Under | Both | Inacu | Total |
|---|---|---|---|---|---|
| 41 | 7 | 3 | 21 | 60 | 91 |

[표 2] 실험 결과 (wu-ftpd-2.5.0)

데 쓰이고, Handle-Cycle 함수는 방문 순서를 기록하고 있는 cycle 리스트를 살펴보면서 순환 내의 정보의 증감 여부를 확인한 후 변동이 있을 때에만 넓히기를 적용하는 역할을 한다.

## 5. 구현 및 실험 결과

논문에서 설명한 버퍼 취약점 검출을 위한 분석은 CIL 프레임워크[5] 위에서 구현하였다. CIL은 컴파일러의 전처리 단계 역할을 해주고, 구문 트리 수준에서 프로그램을 다룰 수 있게 해준다. CIL에는 한 단계까지 정확하게 분석하는 효율적인 포인터 분석[6]이 구현되어 있고, 이는 중첩된(alias) 스트링 버퍼에 대한 제약식을 안전하게 만들 때와 함수 포인터를 통한 함수 호출 시에 어떤 함수들이 호출 가능한지 알아볼 때 활용된다.

분석의 실용성을 확인하기 위해 실제 버그가 있는 것으로 밝혀진 wu-ftpd 파일 전송 서버의 2.5.0 버전에 대해 실험하였고, 선행 연구[3]와의 비교를 위해 2.6.2 버전에 대해서도 실험하였다. 실험은 인텔 제온 3.2GHz 듀얼 프로세서와 4GB 메모리를 갖춘 우분투 리눅스 시스템에서 이루어졌다.

wu-ftpd 2.5.0 버전에 대한 실험 결과는 [표 2]에 있다. CFG는 프로그램으로부터 Control-Flow Graph를 만드는 시간이고, PTA는 포인터 분석을 하는 시간이다. GEN은 본 논문에서 설명한 분석방법으로 제약식을 생성하는 시간이고, SOL은 만들어진 제약식을 푸는 시간이다. Sound는 안전하게 사용하는 스트링 버퍼의 수를 나타내고, Over와 Under는 각각 오버런과 언더런 가능성이 있는 버퍼의 수를 나타낸다. Both는 오버런과 언더런 모두 가능한 버퍼의 수를 나타내고 Inacu는 부정확한 분석으로 인해 할당된 크기가 0이거나 무한대로 나온 버퍼의 수를 나타낸다. 이렇게 버퍼 취약점을 구분하는 것은 경고를 살펴볼 사용자의 수고를 덜기 위함이다. 실험 결과 본 논문에서 제시한 버퍼 취약점 검출기는 1초 이내의 빠른 속도로 실제 알려진 버퍼 오버런 취약점[8]을 검출하였다.

wu-ftpd 2.6.2 버전에 대한 실험 결과 비교는 [표 3]에 있다. CIL을 사용하기 위해서는 프로그램 파일을 하나로 모으는 과정이 필요하기 때문에, 코드의 전처리 과정에 대한 비교는 생략하였다. GEN과 SOL은 각각의 분석기가 프로그램으로부터 방정식을 만들고 푸는 시간을 의미하고, 경고 개수는 안전하지 않은 버퍼에 대한 것을 모두 포함하여 비교하였다. [3]의 실행환경이 듀얼 프로세서가 아니란 점만 다르나, 이로 인한 차이는 크지 않을 것으로 생각된다. 실험 결과 linear programming 기법 보다 제약식 기반의 분석이 빠르다는 것을 확인할 수 있었다. 경고마저 적게 나온 것은 본 연구의 제약식 생성이 안전하지 않기 때문에 빠트리는 경우가 있을 것으로 생각된다. [2]에는 분석 시간에 대한 언급이 없어 비교할 수 없었다.

| Software | wu-ftpd-2.6.2 |
|---|---|
| SLOC[7] | 18,361 lines |

| | LP approach [3] | Our Analysis |
|---|---|---|
| GEN | 74.88s | 0.30s |
| SOL | 23.21s | 0.10s |
| Warnings | 178 | 101 |

[표 3] 실험 결과 비교 (wu-ftpd-2.6.2)

## 6. 결론 및 향후 연구

본 연구는 버퍼 취약점 검출을 실제 사용되는 소프트웨어에 적용할 수 있을 만큼 빠르게 만드는 것에 주안점을 두었다. 모든 취약점을 찾는다는 것을 보장할 수는 없지만, 비교적 큰 소프트웨어의 실제 알려진 버그를 선행 연구[3]보다 빠른 시간 안에 찾는 도구를 만들 수 있었다.

현재 구조는 버퍼의 사용 자체에 관심을 두기 때문에 프로그램 어디에서 취약점이 있는지 정확하게 지적하지 못한다. 이는 프로그램의 수행 흐름을 고려하지 않기 때문에 발생하는 문제점이고, 프로그램을 SSA 형태[9]로 바꾸면 해결할 수 있을 것이다. 또한 C 언어가 제공하는 만큼 수식을 나타낼 수 있도록 제약식을 확장하고 대응되는 제약식 생성 과정을 추가한다면 좀 더 정확한 도구가 될 것으로 기대된다.

## 참고문헌

[1] 김유일, 전진성, 한환수. C프로그램을 위한 효율적인 버퍼 오버런 분석기의 개발. 한국정보과학회 프로그래밍언어논문지, 제19권, 제2호, 1-9쪽, 2005년 11월.

[2] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities. In Network and Distributed System Security(NDSS), 2000.

[3] V. Ganapathy, S. Jha, D. Chandler, D. Melski, and D. Vitek. Buffer Overrun Detection using Linear Programming and Static Analysis. In Computer and Communication Security(CCS), 2003.

[4] Francois Bourdoncle. Efficient Chaotic Iteration Strategies with Widenings. In Formal Method in Programming and their Applications(FMPA), 1993.

[5] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In Compiler Construction(CC), 2002.

[6] Manuvir Das. Unification-based Pointer Analysis with Directional Assignments. In Programming Language Design and Implementation(PLDI), 2000.

[7] SLOCCount www.dwheeler.com/sloccount/

[8] www.cert.org/advisories/CA-1999-13.html

[9] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, An Efficient Method of Computing Static Single Assignment Form, In Principles of Programming Languages(POPL), 1989