SVF를 활용한 스택 내에서만 사용되는 메모리 할당자 호출 지점 분석

하선[1], 박찬영[2], 곽영준[3], 문현곤[4]
[1]울산과학기술원 컴퓨터공학과 박사과정
[2]울산과학기술원 컴퓨터공학과 석박통합과정
[3]울산과학기술원 컴퓨터공학과 석사과정
[4]울산과학기술원 컴퓨터공학과 교수
seonha@unist.ac.kr, chanyoung@unist.ac.kr, kyj05137@unist.ac.kr,
hyungon@unist.ac.kr

Analysis of Memory Allocator Call sites Used Only Within The Stack Using SVF

Seon Ha[1], Chanyoung Park[2], Yeongjun Kwak[3], Hyungon Moon[4]
[1]Dept. of Computer Science Engineering, UNIST
[2]Dept. of Computer Science Engineering, UNIST
[3]Dept. of Computer Science Engineering, UNIST
[4]Dept. of Computer Science Engineering, UNIST

요 약

해제 후 재사용 (Use-After-Free, UAF)는 오랜 시간 동안 소프트웨어 보안에서 중요한 문제로 인식되어 왔다. 이 문제를 해결하기 위해 다양한 완화 방법과 방어 연구가 활발히 진행되고 있다. 이러한 연구들은 대부분 기존 벤치마크 성능과 비교했을 때 낮은 성능을 보인다. 이는 메타 데이터와 코드 계측 정보가 증가하여 포인터를 많이 사용하는 벤치마크의 메모리 사용량이 증가하기 때문이다. 이 연구는 SVF를 활용하여 스택에서만 메모리 할당자 호출 지점을 분석한다. 추 후 이 분석 정보를 여러 UAF 연구에 적용하여 런타임 오버헤드를 줄이는 것을 목표로 한다.

1. 서론

Use-After-Free (UaF) 취약점은 최근 C, C++ 프로그램에서 중요한 문제로 여겨진다. 이 취약점은 프로그램의 메모리 무결성을 침해하고 악의적인 공격에 사용될 수 있다. 또한 잘못된 메모리 참조는 예기치 않은 동작을 유발할 수 있다.

UaF의 원인은 dangling pointer이다. 포인터는 메모리를 할당하고 사용한 후에는 해당 메모리를 해제하여 다시 사용할 수 있는 공간으로 반환한다. 포인터가 이미 해제된 메모리를 가리킬 때 이를 dangling pointer라고 한다. 이는 프로그램이 실행하는 동안 동적 메모리의 잘못된 참조를 초래한다. 프로그램이 예상하지 못한 방식으로 동작하거나 비정상적으로 종료되는 문제점도 발생할 수 있다.

최신 연구들 대부분은 메모리 해제 후 사용 버그를 방지하거나 감지하기 위해 동적 완화 기법을 사용한다. 정적 분석을 통해 버그를 방지하거나 감지하는 것은 어렵다. 왜냐하면 메모리 할당 및 참조 조건을 해결하기 어렵기 때문이다. 동적 완화 방법론에는 여러 방법이 있다. 첫 번째로 소개할 방법은 모든 메모리 청크에 레이블을 지정하여 메모리 접근 시 할당되었는지 해제되었는지 확인한다 [1][2]. 두 번째는 객체의 메모리가 해제될 때 dangling pointer를 무효화하는 방법이다 [3][4]. 동적 완화 방법론은 실용적이라는 장점이 있다. 단점으로는 실행하는 동안 해당하는 정보를 계측해야하기 때문에 실행 시간 및 메모리 오버헤드가 발생한다.

이 연구에서는 정적 분석을 통해 스택 내에서만 존재하는 memory 청크의 포인터를 찾는다. 함수가 종료되면 스택에서만 사용되는 포인터에는 더 이상 접근 할 수 없다. 이러한 포인터가 dangling pointer인지 분석하는 것은 비교적 쉽다. 추후 연구에서 동적 완화 방법론의 실행 시간 및 메모리 오버헤드를 줄이

	안전한 메모리 할당 함수	안전하지 않은 메모리 할당 함수	안전한 메모리 할당 함수 비율
400.perlbench	4	2	66.67%
401.bzip2	2	7	22.22%
403.gcc	5	3	62.50%
429.mcf	0	3	0.00%
433.milc	45	53	45.92%
445.gobmk	11	39	22.00%
456.hmmer	16	53	23.19%
458.sjeng	2	8	20.00%
462.libquantum	5	9	35.71%
464.h264ref	18	252	6.67%
470.lbm	0	1	0.00%
482.sphinx3	10	23	30.30%

표 1. 정적 분석 후 분류된 메모리 할당 함수 개수

	안전한 메모리 할당 함수	안전하지 않은 메모리 할당 함수	안전한 메모리 할당 함수 비율
400.perlbench	0	350,033,594	0.00%
401.bzip2	12	312	3.70%
403.gcc	30,645,557	13,658	99.96%
429.mcf	0	35	0.00%
433.milc	6,460	128	98.06%
445.gobmk	0	607,039	0.00%
456.hmmer	500,000	2,040,875	19.68%
458.sjeng	0	30	0.00%
462.libquantum	84	118	41.58%
464.h264ref	852	347,088	0.24%
470.lbm	0	31	0.00%
482.sphinx3	2,512,369	23,096,393	9.81%

표 2. 프로그램을 실행한 후 얻은 메모리 할당 함수 호출 횟수

기 위해 분석한 결과를 활용할 예정이다.

2. 관련 연구

Use-after-Free (UaF): 메모리 관련 취약점 중 하나로, 프로그램이 이미 해제된 메모리를 계속 사용하려고 할 때 발생한다. 일반적으로, 동적 메모리 할당을 통해 생성된 객체나 메모리 블록을 해제한 후에도 해당 메모리를 사용하는 경우를 말한다.

UaF는 주로 C/C++와 같은 저수준 언어에서 발생하며, 동적 메모리 할당 및 해제를 사용하는 프로그램에서 더 자주 발생한다. 이러한 오류는 보안 취약점으로 간주되며, 악의적인 공격자들이 이를 악용하여 프로그램의 동작을 손상시키거나 시스템에 악의적인 코드를 삽입할 수 있다. 예를 들어, UaF 취약점을 이용하여 악의적인 코드를 실행하거나 시스템의 제어를 탈취할 수 있다.

Static Value-Flow analysis (SVF) [5]: 기존 C 및 C++ 프로그램에 대한 확장 가능하고 정확한 분석으로, LLVM 기반 언어를 위한 도구이다. 이 분석 도구는 값-흐름 구축 및 포인터 분석을 수행함으로써 포인터 별칭 분석, 메모리 SSA 형식 구축, 값-흐름 추적 및 메모리 오류 확인을 제공한다.

3. 스택 내에서만 사용되는 memory allocator call site의 특징

스택에만 존재하는 포인터들은 해당 변수가 선언된 함수가 종료될 때 더 이상 사용되지 않는다. 이 경우에 해당 포인터의 선언과 해당 함수의 반환만 확인하면 된다. 스택에서만 사용되는 포인터는 포인터 수명이 짧다. 함수가 종료되면 스택 프레임이 해제되어서 더 이상 해당 포인터에 대한 접근이 불가능하기 때문이다.

따라서 우리는 메모리 할당 함수에 의해 할당된 메모리 청크를 가리키는 포인터들을 분석한다. 이 포인터들이 모두 스택에만 존재할 경우, 이 memory allocator call site는 안전하다. 메모리 할당 함수의 예시로는 C언어의 malloc, calloc, realloc 등이 있다. 현재 이 논문에서 다루는 분석은 malloc/calloc 함수만 분석한다.

4. 분석 방법

스택에서만 사용되는 memory allocator call site를 찾기 위해 포인터 변수의 값 흐름을 추적 할 수 있는 SVF를 사용한다. 먼저 Anderson-style 포인터 분석을 사용하여 포인터 간의 관계를 그래프로 생성한다. 이 포인터 정보는 상호 프로시저 값 흐름을 나타낸다.

SVF 그래프의 노드를 순회하여 memory allocator call site를 포함한 노드를 찾는다. memory allocator call site는 SVF 상의 Addr 노드로 저장된다. 특정 노드가 Addr 노드이면 이 노드가 포함하고 있는 LLVM 명령어의 CallInst이 malloc, calloc인지 확인한다. malloc, calloc 함수를 포함하고 있는 노드인 경우 이 노드로부터 연결된 모든 노드를 순회하여 아래와 같이 두 가지 조건을 만족해야한다.

첫 번째, memory allocator call site를 포함한 노드와 연결된 모든 노드는 스택에만 있어야 한다. 만약 연결된 노드가 StoreSVF 노드와 연결되어 있다면, 해당 노드의 LLVM IR 명령어를 확인한다. 그 명령어가 store이고 값이 저장될 메모리 위치를 가리키는 포인터는 Alloca type이어야만 한다. Alloca type은 LLVM IR에서 스택에 할당된 메모리를 나타낸다.

두 번째, 메모리 관련 함수로 인해 해당 포인터의

복사가 발생하지 않아야 한다. UaF 발생시키는 dangling pointer의 원인은 포인터를 사용 한 후 초기화되지 않은 포인터가 스택, 힙, 데이터/BSS로 복사 될 때 발생한다. memory allocator call site와 연결된 SVF 노드의 type이 AddrSVF 노드인 경우, 이 노드의 LLVM IR 명령어는 memory 관련 함수를 포함하고 있지 않아야 한다. 또한 현재 구현에서는 기존 메모리 블록을 재할당하고 메모리를 복사하는 realloc 함수는 고려하고 있지 않기 때문에 realloc 함수를 포함한 노드일 경우 안전하지 않다고 여긴다.

위 두 조건을 모두 만족하면 memory allocator call site는 안전하다고 간주된다. 분석된 결과를 얻기 위해 해당하는 안전한 메모리 할당 함수에 prefix를 추가하여 함수 이름을 변경하였다.

5. 실험

운영체제는 Ubuntu 20.04를 사용하였다. Intel Xeon E-2288G 및 64GB DRAM, CPU 32KB 명령 및 데이터 캐시, 256KB L2 캐시 및 16MB 공유 L3 캐시를 가지고 있는 서버에서 실험하였다. 구현된 분석을 확인하기 위해서 SPEC CPU2006 벤치마크의 C언어 프로그램을 대상으로 평가한다. SPEC CPU2006은 UaF 취약점을 평가하는데 널리 쓰이는 벤치마크이다.

6. 결과

SVF를 사용해 Spec2006 C언어 프로그램을 분석한다. 총 분석 시간은 372.583초 소모된다. 먼저 정적으로 코드를 분석하였다. 스택에만 존재하는 memory allocator call는 전체 memory allocator call 중 약 30%를 차지한다. (표 1)

해당 벤치마크를 실행시켜 스택에만 존재하는 memory allocator call의 호출 횟수를 조사하였다. 전체 벤치마크 메모리 할당 함수 호출 중에 22.75%가 스택에서만 존재하는 memory allocator call site를 사용한다. (표 2) 전체 벤치마크 실행 시간은 2648.44초 소모된다.

7. 결론

SVF를 사용해 스택에서만 존재하는 memory allocator call site 분석을 수행하였다. 스택에서만 사용되는 포인터는 함수가 종료되면 더 이상 접근이 불가능하다. 이 포인터의 dangling pointer를 분석하기 비교적 용이하다. 추후에 이 분석 결과를 활용하여 기존에 개발된 여러 동적 메모리 보호 방법에 적용하여 성능 향상을 기대 할 수 있다.

Acknowledgement

본 연구는 정부(과학기술정보통신부)의 재원으로 한국연구재단의 지원을 받아 수행된 연구임 (NRF-2022R1F1A1076100). 또한 이 논문은 2024년도 정부(과학기술정보통신부)의 재원으로 정보통신기획평가원의 지원을 받아 수행된 연구임 (No. 2021-0-00724, 임베디드 시스템 악성코드 탐지·복원을 위한 RISC-V 기반 보안 CPU 아키텍처 핵심기술 개발)

참고문헌

[1] Roger S. Pressman "Software Engineering A Practltiners' Approach" 3rd Ed. McGraw Hill

[2] S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic, "Cets: compiler enforced temporal safety for c." in ISMM, J. Vitek and D. Lea, Eds. ACM, 2010.

[2] H. Cho, J. Park, A. Oest, T. Bao, R. Wang, Y. Shoshi- taishvili, A. Doupé, and G.-J. Ahn, "Vik: practical mitigation of temporal memory safety violations through object id inspection," in Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, 2022, pp. 271-284.

[3] B. Lee, C. Song, Y. Jang, T. Wang, T. Kim, L. Lu, and W. Lee, "Preventing use-after-free with dangling pointers nullification," in Proceedings of the 2015 Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA, Feb. 2015.

[4] E. van der Kouwe, V. Nigade, and C. Giuffrida, "Dangsan: Scalable use-after-free detection," in Proceedings of the Twelfth European Conference on Computer Systems, ser. EuroSys '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 405-419.

[5] Yulei Sui and Jingling Xue. 2016. SVF: interprocedural static value-flow analysis in LLVM. In Proceedings of the 25th International Conference on Compiler Construction (CC 2016). Association for Computing Machinery, New York, NY, USA, 265 - 266. https://doi.org/10.1145/2892208.2892235