Use-After-Free 버그 탐지 및 예방 연구 동향 분석

김진환[1], 조영필[2]
[1]한양대학교 컴퓨터소프트웨어학과 (미래자동차-SW 융합전공) 석박통합과정
[2]한양대학교 컴퓨터소프트웨어학과 교수
adsll156@hanyang.ac.kr., ypcho@hanyang.ac.kr

Analyze trends in Use-After-Free bug detection and blocking research

Jin-Hwan Kim[1], Yeong-Pil Cho[2]
[1]Department of Computer and Software (Automotive-Computer Convergence), Han-Yang University
[2]Dept. of Computer Science, Han-Yang University

요 약

전통적 프로그래밍 언어인 C/C++는 시스템 프로그래밍 언어로 널리 사용되고 있으며, 이는 저수준 메모리 제어와 하드웨어 상호작용 등의 특성 때문이다. 하지만 C/C++가 가지고 있는 특성중 하나인 저수준 메모리 제어는 프로그래머가 직접 메모리를 관리해야한다. 다양한 메모리 버그들중에서 특히 Use-after-free버그는 오래전부터 현재까지 해결되지 않은 버그로써 존재하고 있으며, 이는 프로그래머가 수동으로 메모리를 관리함으로써 발생한다. 이 버그를 예방 및 감지하기 위한 연구가 현재까지도 활발하게 진행되고 있다. 이 버그를 차단 및 감지하는 연구들의 동향을 분석하여 앞으로의 관련 연구의 지속적인 필요성을 제시한다.

1. 서론

전통적인 프로그래밍 언어인 C와 C++의 사용성은 해당 언어가 가지고 있는 특성으로 인해 여전히 인기 있는 언어이다[1]. 이는 고성능 컴퓨팅, 임베디드 시스템, 운영체제등 다양한 분야나 저수준 코딩에서 C/C++가 여전히 중요한 역할을 담당하기 때문이다. 특히 저수준 언어의 특징인 하드웨어와의 밀접한 상호작용, 메모리 제어, 실행 속도 등이 핵심 요소로 작용하며, C/C++는 이러한 요구사항을 효과적으로 충족시킬 수 있는 언어로 인식된다.

하지만 C/C++언어의 특징 중 하나인 메모리 제어 부분은 다른 언어들과 달리 프로그래머가 직접 메모리를 수동으로 관리해야 한다. 프로그래머가 직접 메모리를 수동으로 관리해야 한다는 특징은 강력한 제어 능력을 제공하는 동시에 메모리 관련 버그의 발생 가능성을 높이는 요인으로 작용한다. 특히 Use-After-Free버그는 이러한 메모리 수동 관리와 관련이 있다.

이러한 메모리 수동 관리 특성으로 인해 Use-After-Free는 여전히 다양한 프로그램 버그중 해결되지 않은 난제로 남아 있으며, 이 버그는 임의 코드 실행 등으로 인해 권한 상승, 정보 유출, 시스템 탈취등의 문제로 이어질 수 있다. 이로 인해 이 Use-After-Free 버그는 해당 버그가 발생하지 않도록 제어하거나 혹은 발생 자체를 예방하는 방식을 선택해 차단하는 등 메모리 관리 기법 및 프로그램 분석을 비롯하여 다양한 방법들이 현재 연구되고 있다.

이에 본 논문에서는 이러한 Use-After-Free버그를 감지 및 차단하는 방법들을 살펴보고, 향후 이러한 연구의 지속적인 필요성을 제시한다

2. 감지 및 예방

2.1 감지

Use-After-Free버그 감지 방식은 소프트웨어 코드 내에서 버그를 찾거나, 바이너리 내에서 버그가 일어날 가능성은 있지만 해당 버그가 발생하는 경우 그 즉시 감지 하여 해당 버그가 발생하지 않도록 하는 방법이다. 이는 대체적으로 이미 작성된 소스코드, 배포된 바이너리 파일, 혹은 사용중인 라이브러리에 대해 Use-After-Free버그를 감지한다. 주로 소프트웨어의 안전성 및 신뢰성이 중요한 경우에 사용된다.

2.2 예방

Use-After-Free버그 예방은 메모리 할당자를 바꾸거나 수정하여 해당 버그가 자체적으로 일어나지 않게하거나, Garbage Collector를 사용하는 언어를 사용하거나 스마트 포인터 등을 사용해 이 버그 자체가 발생하지 않도록 하는 경우를 말한다. Garbage Collector를 사용하는 언어는 구글의 Go, Java 등이 있다. Garbage Collector외에 스마트 포인터를 사용하는 C++나 최근 모질라 재단에서 만든 RUST 언어도 역시 Use-After-Free를 차단하는 방식 중 하나이다.

3. Use-After-Free

3.1 Use-After-Free 감지

소스코드를 분석해 Use-After-Free를 찾거나 버그 악용을 방지하기 위한 방식이다. 감지 방식에 대한 최근 연구로는 Palfrey[2], PUMM[3]등이 있다.

Palfrey의 경우 Use-After-Free가 발생하는 부분은 소스 코드내에서 상당 부분이 특정 규칙을 따른다고 언급하고 있다. 이러한 규칙을 총 7개를 찾아냈으며 이를 기반으로 소스 코드를 검사할 수 있는 정적 검사기를 개발했다. 75개의 버그중 40개의 버그를 찾는데 성공하였으며, GCC 및 Clang에는 없는 새로운 버그 패턴으로 인해 더 많은 버그를 발견할 수 있다고 발표 했다.

PUMM의 경우는 지연해제와 OTA방식을 결합한 방법이다. 이 방식은 어떤 메모리 할당에 대해 반복 작업(파일 입력 등)이 있는 경우, 해당 작업이 이전 반복에 대한 데이터 종속성이 없다는 것을 기반으로 하여, 종속성이 없다고 판단된 반복 할당 작업에 대해서는 재사용을 허가하게 해준다. 소스코드가 없는 바이너리 파일로부터 실행 흐름을 복구 하고 그것을 기반으로 격리 정책을 만든다. 따라서 동적으로 생성된 코드의 경우는 격리 정책을 해제할 수 없어 이런 경우는 OTA방식을 사용하여 재사용 자체를 할 수 없게 한다.

3.2 Use-After-Free 예방

이 방식은 메모리 할당자를 바꾸거나 메모리 관리언어를 사용해 재 작성하는 등의 방식이다. 예방 방식에 대한 최근 연구 및 언어로는 FFMalloc[4], Dangzero[5], RUST[6] 등이 있다.

FFMalloc은 Fast Forward Allocation의 약자로 빠른 일회성 할당 방식을 사용한다. 이 방식은 모든 할당에 대해 해당 주소를 한번만 사용하고 그 메모리가 해제 되더라도 다른 메모리 요청에 재사용하지 않는 방식이다. 이 경우 문제점이 3가지 발생한다 : 1. 한 페이지의 고갈 및 낭비, 2. 가상주소 공간 고갈. 3.시스템 콜 명령으로 인한 오버헤드. FFMalloc은 크기가 작은 할당의 경우는 객체를 그룹화 하여 할당하는 방식으로 1번과 2번문제를 해결하였으며, 3번의 경우는 메모리 매핑과 해제를 일괄적으로 처리하여 해결하였다. FFMalloc은 이 방식으로 버그가 있는 9개 프로그램의 방어를 성공했다고 보고하고 있다. 하지만 이 방식은 페이지 내에 객체가 한 개라도 남아있다면 해당 페이지의 해제가 불가능하므로 적용했을때의 성능 오버헤드는 낮지만, 메모리 오버헤드가 매우 높은 것으로 평가된다.

DangZero의 OSCAR[7]를 모태로 삼는다. 메모리 할당 시 객체에 대해 새로운 가상 페이지를 생성하되, 이를 동일한 물리적 페이지에 매핑하는 방식을 선택한다. 기존의 방식으로는 페이지 엘리어싱을 할 수 없어 MAP_SHARED를 사용해 메모리를 얻도록 수정했다. 페이지단위로 메모리를 할당하게끔 수정하여 커널에 의존함과 동시에, 이러한 페이지 테이블 메모리 회수가 불가능 하여 높은 오버헤드가 발생했었다. 이에 Dangzero는 메모리 할당자에게 커널 영역에서만 접근 할 수 있는 페이지 테이블을 메모리 할당자가 접근 및 수정할 수 있도록 접근권한을 부여하여 메모리 할당자가 직접 페이지 테이블에 엑세스하여 수정하는 방식으로 설계 했다. 또한 Dangzero는 OSCAR에서 한계점으로 언급한 페이지 회수를 구현하여 엘리어싱 페이지를 회수할 수 있도록 별도의 회수기를 구현해 해당 가상 페이지에 대해 재사용 역시 보장하고 있어 메모리 오버헤드 역시 낮추었다.

RUST의 경우는 위 두 개의 방식과 달리 새로운 언어이다. C/C++과 비슷하게 시스템 언어이지만 C/C++와 다른점은 메모리 안전을 컴파일러 레벨에서 보장해준다. RUST의 특이한점은 C++의 RAII(Resource Acquisition Is Initialization)이 적용되어 있다. 이는 객체의 소멸이 해당 스코프를 벗어나게 되면 자동으로 객체를 해제하게끔 되어있다. 또한 RUST는 안전한 영역에서 생성된 객체를 포인터로 참조하게 되는 경우 해당 포인터가 해당 객체의 범위를 넘어가면 자동으로 오류를 호출하게끔 되어있다. 하지만 이와 별개로 RUST에서는 Unsafe라는 영역을 제공하는데, 이는 C/C++과 비슷하게 작

동한다. 이 영역에서 RUST는 Raw포인터라고 불리는 C/C++과 동일한 포인터를 사용할 수 있으며, 이 영역에서 사용되는 Raw포인터는 기존의 C/C++과 동일하게 다른 객체를 참조할 수 있고, 할당되지 않은 다른 메모리 영역 역시 참조할 수 있다. 이 포인터는 safe영역에서도 선언할 수 있지만 값을 참조하거나 사용하려면 Unsafe영역 내에서만 사용할 수 있다. Unsafe 영역으로 인해 Use-After-Free버그를 완전하게 방어할 수는 없다. 하지만 RUST의 다양한 메모리 보호 개념이 컴파일 타임에 적용되므로 앞서 C/C++보다 메모리 안전 측면에서는 더 안전하다고 할 수 있을 것이다.

앞서 언급한 방식들 외에도 Garbage Collector와 유사한 MarkUs[8]나 Minesweeper[9]가 있으며, 포인터에 태그를 붙이는 방식[10]등을 통해 다양한 방법으로 Use-After-Free를 감지 및 예방하고 있다.

4. 결론

본 논문에서는 Use-After-Free버그를 예방하거나 감지하기 위한 방법들을 조사 및 분석하였다.

감지 방식은 기존의 소스코드와 통합 하거나 별도의 수정 없이 사용할 수 있다는 장점이 있지만 높은 오버헤드와 소스코드의 크기가 큰 경우로 인해 그 분석이 정확하지 않을 수 있으며, 소스코드가 큰 경우에는 그 경로를 모두 탐색하지 못할 가능성이 있고, 그 분석 결과에 대해 어느정도 오탐이 있는 것으로 확인되었다.

예방 방식 역시 높은 오버헤드를 갖거나, 그 성능적인 부분에서 낮은 오버헤드를 갖더라도 높은 메모리 오버헤드를 갖는 경우가 있었다. 또한 기존의 코드와 통합이 어렵거나, 메모리관리 언어로 재작성, 혹은 메모리 할당기를 수정하는 등의 작업이 필요했다. 이를 보아 여전히 메모리 수동 관리 및 Use-After-Free를 방지하는 방식은 여전히 어렵다는 측면이 있다. Use-After-Free뿐 아니라 기타 유사한 다른 메모리 버그들 역시 지속적인 연구를 통해 효율적인 관리 방법을 도출해야 할 것이며 동시에 안전하면서도 오버헤드는 적은 메모리 버그 방지를 위해 해당 문제에 대해서 지속적인 연구가 필요하다고 생각되는 바이다.

이 논문은 과학기술정보통신부의 재원으로 정보통신기획평가원(No. 2020-0-01840, 스마트폰의 내부데이터 접근 및 보호 기술 분석)과 한국연구재단(No. NRF-2022R1A4A1032361, Processing-in-Memory 보안 기술 개발)의 지원을 받아 수행된 연구임

참고문헌

[1] TIOBE Index for March 2024, https://www.tiobe.com/tiobe-index/

[2] Chen, Z., Liu, D., Xiao, J., & Wang, H. “ All Use-After-Free Vulnerabilities Are Not Created Equal: An Empirical Study on Their Characteristics and Detectability” RAID 2023: The 26th International Symposium on Research in Attacks, Intrusions and Defenses. Hong Kong China, 2023 p. 623-638.

[3] YAGEMANN, Carter, et al. {PUMM}: Preventing {Use-After-Free} Using Execution Unit Partitioning. In: 32nd USENIX Security Symposium (USENIX Security 23). CA, USA 2023. p. 823-840.

[4] WICKMAN, Brian, et al. Preventing {Use-After-Free} Attacks with Fast Forward Allocation. In: 30th USENIX Security Symposium (USENIX Security 21). Vancouver, Canada 2021. p. 2453-2470.

[5] GORTER, Floris, et al. Dangzero: Efficient Use-After-Free detection via direct page table access. In: Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security. Copenhagen, Denmark 2022. p. 1307-1322.

[6] MATSAKIS, Nicholas D.; KLOCK, Felix S. The rust language. ACM SIGAda Ada Letters, 2014, 34.3: 103-104.

[7] DANG, Thurston HY; MANIATIS, Petros; WAGNER, David. Oscar: A practical {Page-Permissions-Based} scheme for thwarting dangling pointers. In: 26th USENIX security symposium (USENIX security 17). 2017. p. 815-832.

[8] AINSWORTH, Sam; JONES, Timothy M. MarkUs: Drop-in Use-After-Free prevention for low-level languages. In: 2020 IEEE Symposium on Security and Privacy (SP). IEEE, San Francisco, USA 2020. p. 578-591.

[9] ERDŐS, Márton; AINSWORTH, Sam; JONES, Timothy M. MineSweeper: a “clean sweep” for drop-in Use-After-Free prevention. In: Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems. San Diego, USA 2022. p. 212-225.
[10] BERNHARD, Lukas, et al. xTag: Mitigating Use-After-Free Vulnerabilities via Software-Based Pointer Tagging on Intel x86-64. In: 2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P). IEEE, Genoa, Italy 2022. p. 502-519.