임베디드 시스템에서 Address Sanitizer 의 한계

박재열 [1], 박성환 [2], 권동현 [3]
[1]부산대학교 정보컴퓨터공학부 학부생
[2]부산대학교 정보융합공학과 박사과정
[3]부산대학교 정보컴퓨터공학부 조교수(교신저자)

woduf0628@pusan.ac.kr, starjara@pusan.ac.kr, kwondh@pusan.ac.kr

Challenges of Address Sanitizer in Embedded Systems

Jaeyeol Park[1], Seonghwan Park [2], Donghyun Kwon[3]
[1]School. of Computer Science and Engineering, Pusan National University
[2]Dept. of Information Convergence Engineering, Pusan National University
[3]School. of Computer Science and Engineering, Pusan National University

요 약

메모리 손상으로 인해 발생하는 컴퓨터 시스템의 버그는 아주 오랫동안 지속적으로 발견된 컴퓨터 보안 이슈 중 하나이다. 이에 대한 보호 기법이 많이 제안되었으며 Address Sanitizer(ASAN) 또한 buffer overflow, use-after-free 와 같은 메모리 손상 버그를 보호하기 위한 기술 중 하나이다. 그러나 해당 기술은 소프트웨어적으로만 구현되었고, 충분한 컴퓨팅 자원이 있을 때만 그 유효성과 실용성이 검증되었고 컴퓨팅 자원이 제한된 임베디드 시스템에서의 적용에 대한 연구나 실효성 검증이 부족하다. 이에 본 논문에서는 임베디드 시스템에 ASAN 를 적용하기 위한 코드를 작성하고 성능을 측정하고 분석하였다.

1. 서론

메모리 손상으로 발생하는 각종 오류는 가장 오랫동안 꾸준히 제기된 컴퓨터 보안 이슈 중 하나이다. 이를 해결하기 위한 효과적인 방안에 대해 지속적인 연구가 이뤄졌고, Address Sanitizer (ASAN)도 이러한 메모리 손상 버그를 보호하기 위해 개발되었다.[1]

ASAN 은 x86 기반의 일반적인 컴퓨터 시스템을 대상으로 많이 사용되며 충분히 그 유효성과 실용성이 검증되었다.[2] 그러나 ASAN 을 임베디드 시스템과 같이 자원이 제한적인 상황에서 사용하기에는 무리가 있다.

임베디드 시스템은 가상메모리를 지원하지 않고 물리 메모리가 용도별로 분리되어 있어서, 높은 런타임 및 메모리 부하를 가진 ASAN 을 적용하기 힘들다.

이에 본 논문에서는 임베디드 시스템에서 ASAN 을 구현한 예제와 ASAN 의 한계 및 임베디드 시스템에서 메모리 손상 버그를 보호하기 위한 향후 연구 방향을 제안한다.

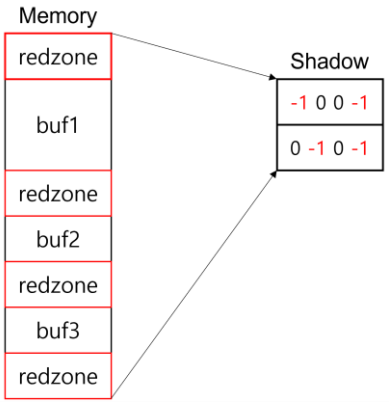
2. Address Sanitizer (ASAN)

ASAN 은 계측 모듈과 런타임 라이브러리 크게 두 부분으로 구성되어 있다.

계측 모듈은 코드에서 메모리 연산에 대한 코드를 찾아 ASAN 코드를 삽입하는 것으로, 해당 메모리 주소에 대한 shadow 메모리 주소를 계산하고 shadow 메모리 주소에 있는 값이 유효한지 비교하는 코드를 삽입한다.

Shadow 메모리란 런타임 검사를 용이하게 하기 위해 도입된 개념으로, <그림 1>과 같이 ASAN 에서는 기본적으로 메모리 공간의 1/8 을 할당하여 각 바이트가 응용 프로그램의 8 바이트 주소 공간이 할당 가능한 상태인지 아닌지를 표시한다.

런타임 라이브러리는 실제 메모리 주소와 shadow 메모리 주소가 항상 매핑 된 상태를 유지할 수 있도록 실행 중에 포인터가 새로 할당되거나 해제될 때 해당 포인터에 대한 shadow 메모리 주소를 관리한다.[1]



<그림 1> Address Sanitizer 메모리 매핑

3. Address Sanitizer Code snippet 구현

컴파일러에 의해 포인터 연산마다 계측 코드가 삽입되기 때문에 타겟 시스템 환경에 관계없이 빌드 시 계측 코드가 삽입되는 부하를 고려할 필요는 없다.

즉 ASAN 이 메모리 버그를 탐지하는 런타임 모듈의 부하만 측정하면 되며, 본 논문에서는 ASAN 을 임베디드 시스템에서 사용할 때 발생하는 최소한의 부하를 측정하기 위해 구현에 필요한 기능만으로 <그림 2>의 코드를 작성 후 ASAN 코드의 성능을 측정했다.

```
1  cmp.w r9, #0x20000000
2  bcc.n ExitProcess
3  movw r2, #0xFFFF
4  movt r2, 0x2000
5  cmp r9, r2
6  bhi.n ExitProcess
7  add.w r2, r9, 0xe0000000
8  movw r3, #0x270
9  lsrs r2, r2, #2
10 movt r3, 0x2000
11 ldrb r2, [r3, r2]
12 cmp r2, #-1
13 beq.n ReportASANError
```

<그림 2> ASAN 메모리 버그 탐지 코드 조각

<그림 2>의 코드가 버그를 탐지할 메모리 주소를 인자로 받게되면, 해당 주소에 매핑 되는 shadow 메모리 주소를 계산하여 shadow 메모리에 저장된 바이트 값이 -1 인지 비교한다. 만약 반환된 값이 -1 이라면 메모리 버그를 탐지한 것으로 판단해 에러를 반환한다.

4. Address Sanitizer 실험 및 한계

ASAN 코드 조각이 임베디드 보드에서 실행될 때 CPU 사이클이 총 몇 사이클 필요한 지를 측정하면 코드의 CPU 자원 소모를 알 수 있으며, 해당 실험을 위해 STM32L552 칩셋이 탑재된 NUCLEO-L552ZE-Q 보드에서 사이클을 측정했다.

<표 1>의 결과와 같이 ASAN 코드 조각은 총 49 사이클을 소모하는 것을 알 수 있다. 실시간으로 빠르게 명령을 처리하는 것이 중요한 임베디드 시스템에서 포인터 유효성 검사에만 49 사이클을 소모하는 것은 부담이 있다. 심지어 이는 포인터가 SRAM 영역에 있음을 가정하고 해당 영역에 대한 유효성 검사만 있는 ASAN 코드에서 측정한 것이다. 만약 SRAM 뿐 아니라 DRAM, Peripheral 영역에 대한 포인터 검사도 적용된다면 포인터 유효성 검사에서 오는 성능 부하가 더 커질 것이다.

	사이클 수
메모리 영역 체크	29
Shadow 메모리 주소 계산	13
Shadow 값 비교	7
총	49

<표 1> ASAN 코드 조각 CPU 사이클

ASAN 코드를 실행하면 CPU 뿐 아니라 메모리 부담 또한 존재하는데, 32 비트로 운영되는 임베디드 시스템에서는 메모리 공간 4 바이트당 shadow 메모리 1 바이트를 할당해야 한다. 즉 가용 메모리 공간과 shadow 메모리를 4:1 로 매핑하여 사용해야 하기 때문에 그만큼 메모리 손해를 봐야 하는 부담이 존재하므로 ASAN 을 임베디드 시스템에 적용하기에는 어려움이 있다.

5. 결론

메모리 손상 버그를 막기 위해 ASAN 을 임베디드 시스템에 적용하는 사례가 없어 이를 위한 코드를 작성 후 분석했다. 그 결과 앞서 살펴본 바와 같이 임베디드 시스템이라는 자원이 한정적인 상황에서는 ASAN 을 적용하는 것에 부담이 크다.

본 논문에서는 임베디드 시스템에서 발생하는 메모리 손상 버그를 막기 위해, 임베디드 시스템 상에 ASAN 을 간략히 구현했다. 실험을 통해 해당 시스템의 성능을 측정했지만, ASAN 의 구현 특성으로 인해 많은 부하가 발생하여 실제로 사용하기에는 한계가 있었다.

그러므로 새로운 하드웨어 기능을 임베디드 환경에 맞게 개발[3,4]하거나 확장[6]하는 것처럼 하드웨어를 활용하여 임베디드 시스템에 새로운 보안 기능이 추가되었을 때, 소프트웨어로 발생하는 부하를 줄일 수 있는 새로운 솔루션이 필요하다.

본 연구는 과학기술정보통신부 및 정보통신기획평가원의 대학 ICT 연구센터육성지원사업의 연구결과로 수행되었음 (IITP-2023-2020-0-01797)

본 연구는 과학기술정보통신부 및 정보통신기획평가원의 융합보안핵심인재양성사업의 연구 결과로 수행되었음 (IITP-2023-2022-0-01201)

참고문헌

[1] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, “Addresssanitizer: A fast address sanity checker,” in the 2012 USENIX Annual Technical Conference, 2012, pp. 309–318.

[2] Zhang, Y., Pang, C., Portokalidis, G., Triandopoulos, N., & Xu, J. “Debloating Address Sanitizer,” in 31st USENIX Security Symp. (USENIX Security 22). USENIX Association, Aug. 2022, pp. 4345-4363.

[3] C. Song, H. Moon, M. Alam, I. Yun, B. Lee, T. Kim, W. Lee, and Y. Paek, “HDFI: Hardware-assisted data-flow isolation,” in 2016 IEEE Symposium on Security and Privacy (SP). IEEE, 2016, pp. 1–17.

[4] L. Delshadtehrani et al., “PHMon: A programmable hardware monitor and its security use cases,” in 29th USENIX Security Symp. (USENIX Security 20). USENIX Association, Aug. 2020, pp. 807–824.

[5] ARM. “Hardware-assisted AddressSanitizer Design Documentation”. Clang 17.0.0git documentation https://clang.llvm.org/docs/HardwareAssistedAddressSanitizerDesign.html