시스템콜 시퀀스 기반의 컨테이너 보안 기법 효율성 분석

송소민○ 김유양, 탁병철
경북대학교 컴퓨터학부
{sominsong, youyangkim, bctak}@knu.ac.kr

Effectiveness Analysis of System call Sequence-based Container Security Mechanism

Somin Song○, Youyang Kim and Byungchul Tak
School of Computer Science and Engineering,
Kyungpook National University

요 약

컨테이너 탈출(Container Escape)은 호스트 커널을 공유하는 컨테이너 환경에서 가장 치명적인 위협 중 하나이다. 공격자는 일련의 조작된 시스템콜들을 통해 커널 취약점을 악용하고, 컨테이너 탈출로 이어질 수 있는 권한 상승을 달성한다. Seccomp는 컨테이너에서 널리 사용되는 보안 메커니즘으로 정책에 따라 악의적인 시스템콜 호출을 필터링함으로써 컨테이너의 격리 수준을 강화한다. 하지만 시스템콜을 개별적으로 차단하는 Seccomp의 필터링 메커니즘은 정책에 의해 허용된 시스템콜 집합만을 사용해서 여전히 공격이 가능하다는 근본적인 한계를 지닌다. 따라서 본 논문에서는 기존의 개별 시스템콜 기반 필터링 메커니즘이 가지는 한계를 극복하기 위하여, 시스템콜 시퀀스(Sequnece) 기반 필터링 메커니즘을 제안한다. 이를 위해 본 논문에서는 악성코드에서 호출되는 시스템콜 시퀀스를 분석하여 위험한 시스템콜 호출을 개별적으로 차단하는 기존 Seccomp 필터링 메커니즘의 한계에 대해 밝히고, 제안된 시퀀스 기반 필터링 메커니즘이 이러한 한계를 극복할 수 있는지에 대해 평가한다.

1. 서 론

컨테이너 기술은 유연성, 확장성 등 다양한 이점으로 인해 클라우드 환경의 가상화를 [1]지원하는 핵심 구성 요소로 자리 잡았다. 그러나 하이퍼바이저를 통해 독립된 게스트 커널을 제공하는 VM(Virtual Machine) 환경과 달리, 컨테이너는 호스트 커널을 공유한다. 따라서 컨테이너 내의 공격자는 일련의 조작된 시스템콜 호출을 통해 호스트 커널에 존재하는 보안 취약점을 악용하여 격리된 컨테이너 환경을 탈출할 수 있을 뿐만 아니라 호스트 및 호스트 위의 다른 컨테이너에까지 위협을 가할 수 있다[1]. 대표적인 예로는 리눅스 커널의 Dirty Pipe(CVE-2022-0847) 및 Di-rty COW(CVE-2016-5195) 취약점을 이용한 컨테이너 탈출(Container Escape)이 있다.

Seccomp[2]는 정책에 따라 프로세스가 호출하는 시스템콜을 제한함으로써 프로세스를 샌드박싱(Sandboxing)하는 커널의 기능이다. 컨테이너 런타임은 Seccomp를 활용하여 컨테이너화된 어플리케이션이 시스템콜을 통해 호스트 커널과 직접 상호 작용하는 것을 제한함으로써 컨테이너에 노출되는 호스트의 공격 표면을 감소시킨다. 기존 연구에서는 어플리케이션이 사용하는 시스템콜들을 조사하여 최소 권한을 부여하기 위한 자동화된 Seccomp 정책 설정 방법들이 주로 연구되었다[3,4]. 하지만 어플리케이션 동작에 필요한 시스템콜들만을 허용하는 완전한 정책을 설정하더라도 Seccomp의 기존 필터링 메커니즘은 시스템콜을 개별적으로 차단하기 때문에, 여전히 공격자가 허용된 시스템콜 집합 내의 시스템콜만을 호출하여 취약점을 악용할 수 있다는 한계가 존재한다. 이것은 개별 시스템콜 수준에서 작동하는 필터링 메커니즘의 근본적인 한계이다.

따라서 본 논문에서는 기존의 개별 시스템콜 기반 필터링 메커니즘의 한계를 극복하기 위하여 시스템콜 시퀀스 기반 필터링 메커니즘을 제안한다. 본 논문에서는 악성코드 분석을 통해 악성코드에서 호출되는 시스템콜을 조사하여 개별 시스템콜 기반 필터링 메커니즘의 한계에 대해 밝히고, 제안된 시스템콜 시퀀스 기반 필터링 메커니즘의 보안성을 평가한다.

* 이 논문은 2021년도 정부(과학기술정보통신부)의 재원으로 한국연구재단의 지원을 받아 수행된 연구임(No. 2021R1A5A1021944)

2. 연구 동기 및 개별 시스템콜 기반 필터링의 한계

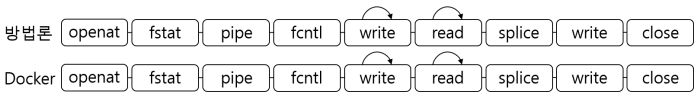

그림 1 Dirty Pipe 취약점(CVE-2022-0847) 악성코드에서 생성된 시스템콜 시퀀스를 통한 공격 차단 예시. 반복 화살표는 1개 이상 반복을 의미.

도커가 제공하는 기본 Seccomp 정책[5]은 51개의 시스템콜들에 대한 컨테이너의 접근을 차단한다. 본 논문에서는 3절에서 설명하는 방법론의 '(a) 악성코드 분석' 과정을 통해 리눅스 커널의 취약점을 대상으로 하는 106개의 악성코드 중 58개(약 55%)가 도커의 기본 Seccomp 정책을 통해 방어될 수 없음을 밝혔다. 도커의 기본 Seccomp 정책에 시스템콜을 추가함으로써 더 많은 공격을 방어할 수 있지만, 이는 어플리케이션의 정상 동작까지도 실패하게 만들 수 있다. 도커의 기본 Seccomp 정책에 의해 방어할 수 없는 악성코드에서 사용된 108개의 시스템콜에는 어플리케이션이 사용하는 write, close 및 clock_nanosleep과 같은 시스템콜들이 다수 포함되어 있기 때문이다.

따라서 본 논문에서는 어플리케이션의 정상적인 동작을 방해하지 않으면서도 개별 시스템콜 필터링으로는 차단할 수 없는 공격을 방어하기 위한 접근 방법으로 시스템콜 시퀀스 기반 필터링을 제안한다. 그림 1에서는 본 논문에서 제안하는 방법론을 통해 도커의 기본 Seccomp로 인해 방어되지 않는 악성코드 중 하나인 Dirty Pipe 취약점(CVE-2022- 0847) 악성코드를 분석하여 생성한 시스템콜 시퀀스 정책과 악성코드를 도커에서 실행했을 때 실제로 호스트에 관측되는 시스템콜 시퀀스를 비교하여 보여준다. 두 시퀀스가 완전히 일치함에 따라, 시퀀스 기반 필터링 메커니즘이 개별 시스템콜 기반 필터링 메커니즘이 방어하지 못한 Dirty Pipe 취약점 공격의 실행을 차단할 수 있음을 보여준다.

3. 시스템콜 시퀀스 조사 방법

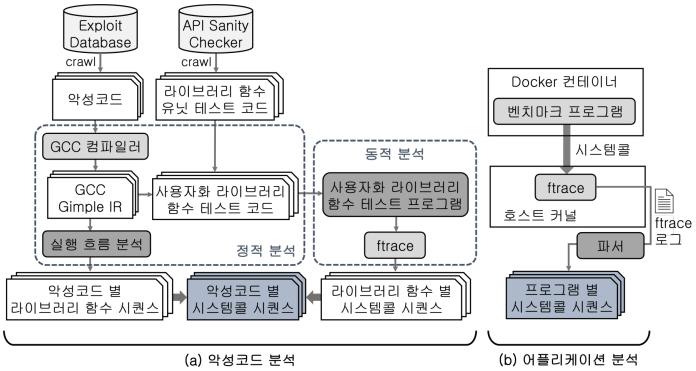

그림 2 시스템콜 시퀀스 추출 방법 구조도

본 논문에서는 악성코드 및 어플리케이션이 호출하는 시스템콜 시퀀스를 조사하여, 개별 시퀀스 기반 필터링 메커니즘의 한계를 분석하고 시스템콜 시퀀스 기반 필터링 메커니즘의 보안성을 평가한다. 그림 2는 분석을 위해 사용된 시스템콜 시퀀스 조사 방법에 대한 구조도를 보여준다. 방법론은 크게 악성코드에서 시스템콜 시퀀스를 추출하는 (a) 과정과 어플리케이션의 정상 동작에서 시스템콜 시퀀스를 추출하는 (b) 과정으로 나뉜다. 그림 2의 '(a) 악성코드 분석' 과정에서는 악성코드를 정적 및 동적 분석하여 코드를 직접 실행시키지 않고 악성코드가 호출하는 시스템콜 시퀀스를 조사한다. 그림 2의 '(b) 어플리케이션 분석' 과정에서는 다양한 어플리케이션을 직접 실행시키며 호스트 커널에 도달하는 시스템콜을 추적함으로써 어플리케이션이 정상 동작할 때 호출하는 시스템콜 시퀀스를 조사한다.

3.1 악성코드가 호출하는 시스템콜 시퀀스 조사

그림 2의 '(a) 악성코드 분석' 부분은 악성코드를 실행시키지 않고 악성코드로부터 호출되는 시스템콜 시퀀스를 조사하기 위해 정적 분석과 동적 분석을 결합하여 사용하는 방법을 보여준다. 정적 분석 방법을 통해 악성코드에서 사용된 라이브러리 함수의 시퀀스를 조사하고 동적 분석 방법을 통해 라이브러리 함수가 호출하는 시스템콜 시퀀스를 조사함으로써, 두 분석 방법의 결과를 결합하여 악성코드가 호출하는 시스템콜 시퀀스 정보를 얻을 수 있다.

먼저, Exploit Database[8]에서 공개적으로 사용 가능한 악성코드를 크롤링한다. 본 논문에서는 컨테이너 탈출을 초래할 수 있는 커널 취약점 대상 악성코드 중 시스템콜 조작과 가장 밀접한 관련이 있는 권한 상승 악성코드를 분석한다. Exploit Database 내에 존재하는 369개의 전체 리눅스 커널 악성코드 중 가장 일반적인 언어인 C언어로 작성된 106개의 권한 상승 악성코드를 수집하였다.

정적 분석 과정에서는 GCC 컴파일러를 이용한 실행 흐름 분석을 통해 악성코드가 호출하는 라이브러리 함수의 시퀀스 정보를 추출한다. GCC 컴파일러를 통해 악성코드를 컴파일하여 GCC Gimple IR(Intermediate Representation) 형식으로 출력하고, IR 형식 출력 파일에서 코드의 실행 흐름을 분석하여 악성코드별 라이브러리 함수 시퀀스를 추출한다. 이후, 동적 분석에서 사용할 라이브러리 함수의 테스트 코드를 생성하기 위해 악성코드에서 사용된 218개 라이브러리 함수의 유닛 테스트 코드를 API Sanity Checker[7]에서 수집한다. 기존 라이브러리 함수 유닛 테스트 코드의 인자를 악성코드에 사용된 인자로 수정하여 사용자화 라이브러리 함수 테스트 코드를 생성한다.

동적 분석에서는 정적 분석 과정에서 생성한 사용자화 라이브러리 함수 테스트 코드를 직접 실행시키고 커널의 ftrace 도구를 통해 사용자화 라이브러리 함수 테스트 프로그램이 호출하는 시스템콜 시퀀스를 추적함으로써 라이브러리 함수별 시스템콜 시퀀스를 조사한다. 이 두 결과를 결합하여 악성코드별 시스템콜 시퀀스 정보를 생성한다.

3.2 어플리케이션이 호출하는 시스템콜 시퀀스 조사

그림 2의 '(b) 어플리케이션 분석' 부분은 직접 어플리케이션을 실행시켜 어플리케이션이 정상적으로 동작할 때 호출하는 시스템콜 시퀀스를 조사하는 방법을 나타낸다.

조사에 사용된 어플리케이션은 8개이며, Docker Hub에서 인기 있는 서버 어플리케이션 4개(nginx, httpd, tomcat, node)와 데이터베이스 어플리케이션 4개(mongodb, mysql, mariadb, redis)를 사용하였다. 서버 어플리케이션은 AB(Apache Benchmark)[8] 벤치마크 프로그램, 데이터베이스 어플리케이션은 YCSB[9] 벤치마크 프로그램을 통해 다양한 워크로드를 테스트한다. 벤치마크 프로그램이 동작하는 동안 ftrace 도구를 이용해 어플리케이션이 호출하는 시스템콜 시퀀스를 조사하고, ftrace의 로그 파일을 파싱(Parsing)하여 어플리케이션별 시스템콜 시퀀스 정보를 생성한다.

4. 분석 및 평가

개별 시스템콜 기반 필터링 메커니즘과 시스템콜 시퀀스 기반 필터링 메커니즘의 보안성을 평가하기 위해 정상 동작 어플리케이션의 실행을 방해하지 않으면서 악성코드를 차단할 수 있는 정책을 각각 생성해야 한다. 이를 위해 그림 2의 '(a) 악성코드 분석' 부분에서 조사된 시스템콜 시퀀스와 '(b) 어플리케이션 분석' 부분에서 조사된 시스템콜 시퀀스를 이용해 정책을 생성한다. 개별 시스템콜 기반 필터링 메커니즘 정책의 경우, 그림 2의 (a)에서 조사된 시스템콜 시퀀스의 시스템콜 집합에서 그림 2의 (b)에서 조사된 시스템콜 시퀀스의 시스템콜 집합을 제외하여 생성한다. 시스템콜 시퀀스 기반 필터링 메커니즘 정책의 경우, 그림 2의 (a)에서 조사된 시스템콜 시퀀스를 다양한 길이로 나누어 N의 길이를 가지는 하위 시스템콜 시퀀스(N-gram[12])를 생성한다. 즉, (a)에서 조사된 시스템콜 시퀀스의 N-gram에서 (b)에서 조사된 시스템콜 시퀀스를 제외하여 시스템콜 시퀀스 정책을 생성한다. 본 장에서는 생성된 각 정책을 기반으로 방어할 수 있는 악성코드를 비교하여 제시함으로써 두 메커니즘의 보안성을 평가한다.

4.1 개별 시스템콜 기반 필터링 메커니즘 평가

본 절에서는 3절의 시스템콜 시퀀스 조사 방법을 이용하여 개별 시스템콜 기반 필터링 메커니즘을 위한 정책을 생성하고 이를 통해 방어할 수 있는 악성코드를 조사한다.

수집된 106개의 악성코드에서 그림 2의 (a) 과정을 통해 추출된 시스템콜 시퀀스는 140개의 개별 시스템콜로 구성되며, 8개의 어플리케이션에서 그림 2의 (b) 과정을 통해 추출된 시스템콜 시퀀스는 125개의 개별 시스템콜로 구성된다. 악성코드에서 추출한 시스템콜 집합에서 어플리케이션에서 추출한 시스템콜 집합을 제외하여 개별 시스템콜 기반 필터링 메커니즘을 위한 시스템콜 정책을 생성한 결과, 42개의 시스템콜을 차단하는 정책이 생성된다.

42개의 시스템콜을 차단하는 정책을 사용하는 개별 시스템콜 기반 필터링 메커니즘의 경우, 전체 106개의 악성코드 중 67개(약 63%)의 악성코드를 방어할 수 있었다.

4.2 시스템콜 시퀀스 기반 필터링 메커니즘 평가

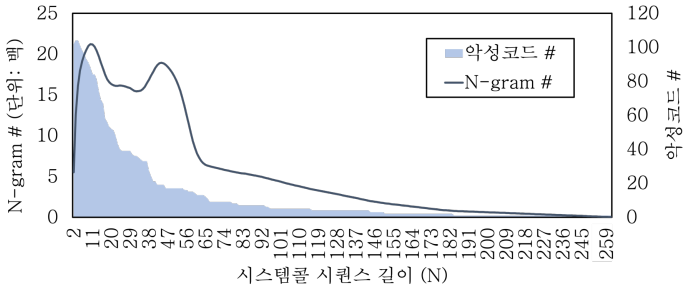

그림 3 길이별 N-gram 시스템콜 시퀀스의 개수 및 관련 악성코드의 개수

본 논문에서는 시스템콜 시퀀스 기반 필터링 메커니즘을 위한 정책을 생성하기 위해 그림 2의 (a)와 (b)에서 조사된 시스템콜 시퀀스를 다양한 길이의 하위 시스템콜 시퀀스(N-gram[12])로 나눈다. 그림 2의 (a) 과정을 통해 추출된 시스템콜 시퀀스를 N만큼의 길이를 가진 N-gram으로 나눌 경우, 최소 길이 2부터 259까지 142,665개의 시스템콜 시퀀스가 생성된다.

그림 2의 (b) 과정을 통해 추출된 시스템콜 시퀀스는 어플리케이션이 정상 동작할 경우 호출하게 되는 시스템콜 시퀀스이다. 해당 시퀀스에 필터링 정책이 포함될 경우, 어플리케이션이 동작을 멈추게 된다. 따라서 그림 2의 (a) 과정을 통해 악성코드로부터 추출된 시스템콜 시퀀스의 N-gram 집합에서 어플리케이션이 정상 동작할 경우 호출하는 시스템콜 시퀀스와 동일한 N-gram 976개를 제외하면, 최소 길이 2부터 259까지 141,689개의 시스템콜 시퀀스가 남는다.

그림 3은 시퀀스 길이별로 생성된 N-gram 시스템콜 시퀀스의 개수와 해당 N-gram 시스템콜 시퀀스와 일치하는 악성코드의 분포를 보여준다. 예를 들어, 길이가 2인 2-gram 시스템콜 시퀀스는 552개가 존재하며, 102개의 악성코드와 일치한다. 그림 3에 따르면, 시스템콜 시퀀스의 길이가 짧을수록 더 많은 악성코드와 일치하는 것을 확인할 수 있다, 즉, 짧은 길이의 시스템콜 시퀀스가 정책으로 사용될 경우 더 많은 악성코드를 차단할 수 있다.

본 논문에서는 개별 N-gram 시스템콜 시퀀스가 얼마나 많은 악성코드를 차단할 수 있는지에 대해 조사하였으며, 그림 3에 나타난 결과를 토대로 짧은 길이의 시스템콜 시퀀스부터 차례로 정책에 추가할 경우 정책과 일치하여 차단되는 악성코드에 대해 조사하였다. 효율적인 정책 설정을 위해, 같은 길이 내에서는 더 많은 악성코드와 일치하는 N-gram 시스템콜 시퀀스를 먼저 정책에 추가하였으며, 이미 추가된 정책에 의해 차단 되는 악성코드와 일치하는 N-gram 시스템콜 시퀀스의 경우 정책에 중복으로 추가하지 않았다. 그림 4는 시퀀스 길이 별로 정책에 추가된 시스템콜 시퀀스와 시스템콜 시퀀스에 의해 차단된 악성코드의 개수를 보여준다. 이를 통해, 전체 141,689개의 시스템콜 시퀀스 중 2-gram 시스템콜 시퀀스 31개와 3-gram 시스템콜 시퀀스 2개로 이루어진 33개의 시스템콜 시퀀스 정책을 사용할 경우, 시퀀스 기반 필터링 메커니즘은 전체 106개의 악성코드 중 104개(약 98%)의 악성코드를 방어할 수 있다는 것을 알 수 있다.

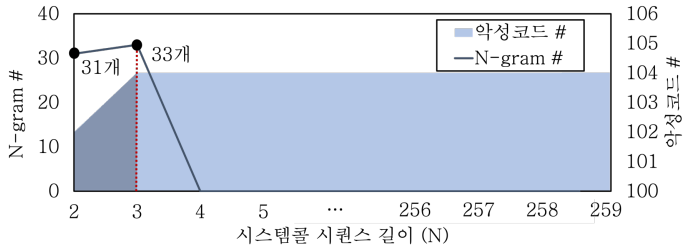

그림 4 짧은 길이의 N-gram 시스템콜 시퀀스부터 누적하여 정책에 추가할 경우 방어 가능한 악성코드 개수

4.3 개별 시스템콜과 시스템콜 시퀀스 기반 필터링 메커니즘의 보안성 비교

표 2 보안성 비교

필터링 메커니즘	정책 개수	방어 가능한 악성코드 개수 (비율)
개별 시스템콜 기반	42	67 (63%)
시스템콜 시퀀스 기반	33	104 (98%)

본 절에서는 표 1을 통해 개별 시스템콜 기반 필터링 메커니즘과 시스템콜 시퀀스 기반 필터링 메커니즘의 보안성을 비교하여 보여준다. 개별 시스템콜 시퀀스 기반 필터링 메커니즘에 비해, 시스템콜 시퀀스 기반 필터링 메커니즘은 전체 106개의 악성코드 중 37개(35%p) 많은 104개(98%)의 악성코드를 9개 더 적은 33개의 정책만으로 방어할 수 있었다.

5. 결론

본 논문에서는 시스템콜 필터링 메커니즘인 Seccomp의 개별 시스템콜 차단 방식이 가지는 한계점에 대해 밝히고, 이러한 한계점을 극복할 수 있는 대안으로 시스템콜 시퀀스 기반 필터링 메커니즘을 제안한다. 106개의 악성코드를 통해 분석한 결과, 개별 시스템콜 기반 필터링 메커니즘은 42개의 시스템콜 정책을 사용해 최대 63%의 악성코드를 방어하는데 비해, 시스템콜 시퀀스 기반 필터링 메커니즘은 그보다 9개 적은 33개의 시스템콜 시퀀스 정책을 사용해 최대 98%의 악성코드를 방어할 수 있었다.

참 고 문 헌

[1] Xin Lin, Lingguang Lei, Yuewu Wang, Jiwu Jing, Kun Sun, and Quan Zhou. A measurement study on linux container security: Attacks and countermeasures. In Proceedings of the 34th Annual Computer Security Applications Conference, pp.418-429,2018.
[2] Seccomp, "https://www.kernel.org/doc/Documentation/prctl/seccomp_filter.txt.
[3] S. Ghavamnia, T. Palit, S. Mishra, and M. Polychronakis, "Temporal system call specialization for attack surface reduction – usenix," ser. Proceedings of the 29th USENIX Security Symposium, Aug 2020.
[4] X. Wang, Q. Shen, W. Luo, and P. Wu, "Rsds: Getting system call whitelist for container through dynamic and static analysis," in 2020 IEEE 13th International Conference on Cloud Computing, pp.600-608, 2020.
[5] Seccomp profiles for Docker, https://docs.docker.com/engine/security/ seccomp/.
[6] Exploit Database, https://www.exploit-db.com/.
[7] API Sanity Checker, https://lvc.github.io/api-sanity-checker/.
[8] AB,https://www.tutorialspoint.com/apache_bench/index.html.
[9] YCSB, https://github.com/brianfrankcooper/YCSB.
[10] Pei, J. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and M.-C. Hsu, "Prefixspan,:mining sequential patterns efficiently by prefixprojected pattern growth," in Proceedings 17th International Conference on Data Engineering, 2001.