2016년 정보보호학술논문발표회

메모리 접근 분리기법의 활용 비교
A comparison of memory access split schemes

홍 진 아*, 김 광 조**

요 약

코드 주입 공격(Code Injection Attack)은 데이터가 실행가능 하고, 리턴 기반의 프로그래밍 공격(Return-Oriented Programming)은 코드가 읽기 가능하기 때문에 발생한 공격으로 근본적인 원인은 데이터와 코드 사이의 분리가 명확하지 않은 것이다. 이 문제를 포함한 다양한 문제를 풀기 위해 최신 논문에서 데이터와 코드를 분리하려는 노력을 하였다. 그러나 일반화된 용어나 방식은 존재하지 않았다. 본 논문에서는 데이터와 코드가 서로 다른 메모리 영역을 가지도록 관리하는 방식을 메모리 접근 분리기법으로 정의하고, 최신 논문들에서 메모리 접근 분리기법를 활용한 방식을 코드 숨기기 목적과 접근 관리 목적 두 가지로 나누고, 여러 가지 문제를 어떻게 해결하였는지를 비교 분석하였다. 마지막으로 메모리 접근 분리기법 사용한 향후 연구 방향을 제안한다.

Ⅰ. 서 론

버퍼 오버 플로우(Buffer Overflow) 취약점을 사용하여 빈번하게 발생하는 코드 주입 공격(Code Injection Attack)은 프로세서가 공격자에 의해 주입된 데이터를 코드로 실행할 수 있기 때문에 가능한 공격이다. 또한 리턴 기반의 프로그래밍(Return - Oriented Programming)은 공격자가 라이브러리나 프로그램의 코드를 읽어 특정 명령어의 주소를 알아낼 수 있기 때문에 가능한 공격이다. 이런 문제의 근본적인 원인은 데이터가 '실행' 가능하고 코드가 '읽기' 가능한 것처럼 데이터와 코드 사이의 분리가 명확하지 않기 때문이다.

본 논문에서는 메모리 접근 분리기법으로 정의하고, 이 기술을 활용한 최신 논문들을 비교 분석하였다. 메모리 분리기법은 데이터와 코드가 서로 다른 메모리 영역을 가지도록 관리하는 방식이다. 우리는 크게 두 가지 용도로 이 기법의 활용 방안을 구분하였다. 첫 번째 방법은 외부 프로그램으로부터 자신의 프로그램 코드를 숨기기 위한 방법이다. 이는 공격자로부터 코드를 보호할 수도 있으며, [3,4,5] 방어프로그램으로부터 공격코드를 보호하는 목적으로도 사용될 수 있다. [1,6] 두 번째는 접근 관리 목적으로 데이터와 코드를 분류하여 코드는 실행만 가능하고, 데이터는 읽기, 쓰기만 가능하도록 규정하는 방법이다. [2,6,7] 이 방식은 공격의 근본적인 원천을 차단하기 위한 보호 방법으로 많이 사용된다. 우리는 이 논문을 통해 다양한 이름으로 흩어져서 활용되어 왔던 기술을 메모리 접근 분리기법으로 정의하고 이를 일반화 하였다. 또한 이 기술을 활용한 논문들을 취합 정리하였다.

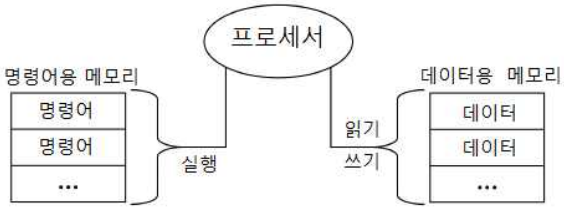
이후 논문의 구성은 메모리 접근 분리기법에 대한 설명과 이 기술을 활용한 최신 논문을 비교 분석하였다. 마지막으로 결론 및 향후 연구로 정리한다.

Ⅱ. 메모리 분리 기법

메모리 접근 분리기법은 데이터와 코드가 서로 다른 메모리 영역을 가지도록 관리하는 방식이다. 그러나 데이터와 코드를 분리하는 것은 쉬운 작업이 아니다. 파일 헤더에서 코드와 데이터영역을 제공하지만 실행 중인 프로세스에겐 유용한 정보가 못된다. 이를 해결하기

This research was supported by the KUSTAR-KAIST Institute, under the R&D program supervised by the Korea Advanced Institute of Science and Technology (KAIST), South Korea.

* 한국과학기술대학원 정보보호대학원 (jina3453@kaist.ac.kr)
** 한국과학기술대학원 전산학부 (kkj@kaist.ac.kr)



[그림 1] 메모리 분리 기법[1]

위해 프로세서가 메모리에 접근하는 읽기, 쓰기, 실행(read, write, execute) 행위로 데이터와 코드를 분리할 수 있다.

정상적인 경우, 코드는 실행하기 위한 목적이기 때문에 프로세서가 메모리를 '실행'으로 접근하면 해당 영역은 코드로 간주한다. 데이터는 읽기와 쓰기가 주목적이다. 그러므로 프로세서가 메모리를 '읽기', '쓰기'로 접근하면 해당 영역은 코드로 간주한다. 코드와 데이터를 논리적으로 분리하였다면 그 다음으로는 〔그림1〕과 같이 실질적으로 분리되어 있는 메모리 공간이 필요하다. 하나의 메모리는 데이터용 메모리로 읽기, 쓰기가 발생하였을 때 접근하는 메모리 영역이고, 다른 메모리는 명령어용 메모리로 실행이 발생하였을 때 사용하는 메모리이다.

Ⅲ. 메모리 접근 분리기법의 활용 비교

메모리 분리기법은 크게 두 가지 용도로 사용되었다. 첫 번째로 외부 프로그램으로부터 자신의 프로그램 코드를 숨기기 위한 방법이다. 두 번째는 접근 관리 목적으로 데이터와 코드를 분류하여 코드는 실행만 가능하고, 데이터는 읽기/쓰기만 가능하도록 규정하는 방법이다. 각 논문의 위협 모델과 이를 해결하기 위한 방법은 〔표 1〕에 정리되어 있다.

3.1. 숨기기 위한 목적

프로세서가 코드 영역에 '읽기'로 접근한다면 이는 정상적이지 않은 것으로 판단하여 코드를 숨길 수 있다.

Sherri Sparks 등[1]은 루트킷을 숨기기 위하여 메모리 분리 기법을 처음으로 제안하였다. 루트킷 탐지 프로그램은 커널코드를 읽고 루트킷 존재 여부를 식별한다. Sherri Sparks 등은 이를 우회하기 위해 프로세서가 '읽기'로 접근하면 쓰레기 코드를 보여주고 '실행'으로 접근할 경우에는 정상적인 루트킷 코드가 실행된다. 그러므로 읽기로만 커널에 접근할 수 있는 루트킷 탐지 프로그램은 어떠한 이상 징후를 발견하지 못하게 된다.

〔표 1〕 제안 방식별 비교 분석

참조 문헌	위협 모델	보호 목표	공격 목표
〔1〕	커널 무결성 검사	-	루트킷 코드 숨김
〔3〕	커널 루트킷	안전한 커널 코드 숨김	-
〔4〕	커널 루트킷	안전한 커널 코드 숨김	-
〔5〕	안티-디버깅기술	INT3 명령어 숨김	-
〔2〕	코드 주입 공격	코드와 데이터 분리	-
〔6〕	-	코드 무결성 검사	루트킷 코드 숨김
〔7〕	리턴 기반의 프로그래밍	코드 읽기 불가	-

이후 메모리 분리 기법은 방어프로그램에 많이 적용되었다. Riley등[3]과 Michael Grace 등[4]은 루트킷으로부터 커널을 보호하기 위해 일반적인 커널코드와 안전한 커널 코드를 별도로 관리한다. 프로세서가 '읽기'로 접근할 때 일반적인 커널코드가 존재하고, '실행'으로 접근할 때는 안전한 커널 코드를 실행한다. 루트킷은 커널을 '읽기'로만 접근하기 때문에, 일반적인 커널을 변조하게 된다. 하지만 이 변조된 일반적인 커널은 프로세서에 의해 절대 실행되지는 않는다. 왜냐하면 프로세서가 '실행'으로 커널 코드에 접근할 때에는 다른 메모리 영역에 존재하는 안전한 커널 코드를 실행하기 때문이다.

이 기술은 디버거에도 활용될 수 있다. 최신 악성코드는 안티-디버깅 기술을 가지고 있다. 한 가지 예로 브레이크 포인트에 해당하는 INT3명령어가 악성코드 내에 존재한다면 이는 디버거가 존재한다고 판단하여 악성코드 실행을 중지해 버린다. 이러한 안티-디버깅 기술을 우회하기 위해 Zhui Deng 등[5]은 보이지 않는 브레이크 포인트를 제안하였다. 안티 디버깅 기술은 코드를 읽기로 접근하는 점을 활용하여 프로세서가 메모리를 '읽기'로 접근할 경우에는 INT3 명령어가 존재하지 않는 일반적인 코드를 보여준다. 프로세서가 '실행'으로 접근하는 경우는 실제 디버거가 실행하는 것이기 때문에 INT3명령어가 보이도록 하여 정상적으로 브레이크 포인트를 수행할 수 있다.

3.2. 코드, 데이터 분리 목적

Riley 등 [2] 은 코드 주입 공격을 막기 위해 코드와 데이터가 분리되어 있는 하버드 아키텍쳐를 가상화하였다. 하나의 메모리에서 데이터와 코드를 모두 처리하는 폰노이만 아키텍쳐와는 다르게 하버드 아키텍쳐는 코드와 데이터가 별도의 메모리영역을 가진다. 폰 노이만 아키텍쳐 위에 이러한 특징을 가상화하기 위해 페이지단위에서 프로세서의 접근을 읽기, 쓰기, 실행을 구분하여, '읽기'는 데이터용 메모리만 접근가능하고, '실행'은 명령어용 메모리만 접근하도록 하였다.

Jacob Torrey [6] 는 실행프로그램에서 코드영역을 분리하여 코드 무결성 체크를 제안하였다. 이때 운영체제나 부트로더는 코드와 데이터가 섞여있기 때문에 코드영역만 분리하기가 힘들다. 그렇기 때문에 프로세서가 '읽기'로 메모리에 접근하는 경우는 코드로 간주하여 이 메모리에 대해 무결성을 검증한다.

Michael Backes 등 [7] 은 ROP공격을 막기 위해 코드는 오직 실행만 가능하고 읽지 못하도록 하는 방식을 제안하였다. 이를 구현하기 위해 첫 번째로 '읽기'는 데이터로, '실행'은 코드로 접근 방식을 나누었다. 기존의 접근 방식과는 다르게 한번 더 절차를 거친다. 파일의 헤더정보를 활용하여 읽기가 수행된 메모리 영역이 코드영역인지 데이터 영역인지를 확인한다. 만약 코드영역이라면 이는 ROP를 수행하기 위한 사전작업이라고 판단하여 해당 프로그램을 종료한다.

Ⅳ. 결론 및 향후 연구

본 논문에서는 코드와 데이터를 분리하기 위한 메모리 접근 분리 기법이 무엇인지와 이 기술을 활용한 논문들을 소개하였다. 우리는 메모리 분리 기법의 활용 방법을 숨기기 위한 목적과 코드, 데이터 분리 목적으로 나누어서 살펴보았다. 사용자는 원하는 목적에 맞게 이 기술을 사용할 수 있다. 단, 메모리 접근 분리 기법은 단지 기술이기 때문에 공격도구 혹은 방어도구로도 사용될 수 있다.

이 기술을 사용하여 공격 도구를 만들지 못하도록, 숨겨진 코드를 찾아내는 방법이 연구되어야 할 것이다. 그러므로 방어 도구로 이 기술을 사용하기 위해서는 코드를 숨기는 목적보다는 코드와 데이터를 분리하는 목적으로 사용 되여야 한다. 이는 방어방법이 아닌 사전에 공격을 막을 수 있는 보호방법이기 때문에 더 효과적으로 공격을 차단할 수 있을 것이다.

참 고 문 헌

〔1〕 Sparks, Sherri, and Jamie Butler. "Shadow Walker": Raising the bar for rootkit detection", *Black Hat Japan*, 2005

〔2〕 Riley, Ryan, Xuxian Jiang, and Dongyan Xu. "An architectural approach to preventing code injection attacks", *Dependable and Secure Computing, IEEE Transactions* pp. 351-365, 2010

〔3〕 Riley, Ryan, Xuxian Jiang, and Dongyan Xu. "Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing", *Recent Advances in Intrusion Detection*. Springer Berlin Heidelberg, pp. 1-20, 2008.

〔4〕 M. C. Grace, Z. Wang, D. Srinivasan, J. Li, X. Jiang, Z. Liang, and S. Liakh, "Transparent protection of commodity os kernels using hardware virtualization". *Conference on Security and Privacy in Communication Networks*, Springer Berlin Heidelberg, pp. 62-180, 2010

〔5〕 Deng, Zhui, Xiangyu Zhang, and Dongyan Xu. "Spider: Stealthy binary program instrumentation and debugging via hardware virtualization." *Proceedings of the 29th Annual Computer Security Applications Conference.* ACM, pp. 289-298, 2013.

〔6〕 Torrey, J. More shadow walker: Tlb-splitting on modern x86, 2014

〔7〕 Backes, M., Holz, T., Kollenda, B., Koppe, P., Nürnberger, S., & Pewny, J. You can run but you can't read: Preventing disclosure exploits in executable code. *In Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security,* ACM, pp. 1342-1353. 2014