코드 재사용 공격과 방어의 발전에 관한 연구

안선우, 이영한, 방인영, 백윤흥*
*서울대학교 전기정보공학부

A Study on Development of Code Reuse Attacks and Defenses

Sunwoo Ahn, Younghan Lee, Inyoung Bang, Yunheung Paek*
*Department of Electrical and Computer Engineering, Seoul National University.

요 약

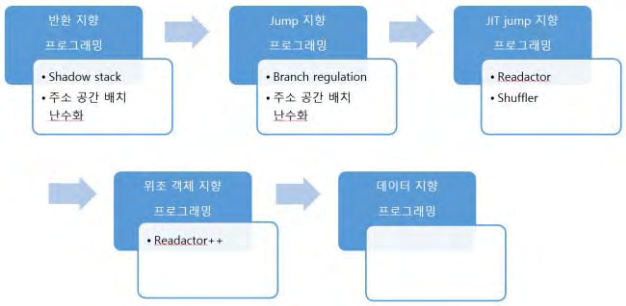
과거의 가장 흔한 공격이었던 코드 삽입 공격은 방어 기법이 발전함에 따라 점점 어려워지고 있다. 공격자는 코드를 삽입하지 않고도 공격할 수 있는 방법을 찾기 시작하였고, 공격 대상에 존재하는 코드를 연결하여 원하는 동작을 실행하게 만드는 코드 재사용 공격을 하기 시작했다. 코드 재사용 공격을 막는 방어 기법 역시 제안되었지만, 다시 이를 우회하는 발전된 코드 재사용 공격들도 나오면서 공격과 방어를 거듭하고 있다. 본 논문에서는 코드 재사용 공격의 전신인 return into libc 부터 Data Oriented Programming (DOP)까지의 공격과 방어를 정리하고, 코드 재사용 공격이 발전되는 과정을 살펴보는 것을 목표로 한다.

1. 서론

코드 삽입 공격은 공격자가 원하는 코드를 공격 대상에 삽입하여 원하는 행위를 하게 만드는 강력한 공격이다. 따라서 코드 삽입 공격은 공격자가 가장 먼저 생각하는 흔한 공격이었다. 하지만 이 공격은 W^X (Write XOR Execute)에 의해 방어되게 된다. W^X는 쓰기 권한과 실행 권한을 동시에 주지 않음으로써, 즉, 데이터 영역에는 쓰기 권한만을, 코드 영역에는 실행 권한만을 주어 코드 삽입 공격을 방어해낸다. W^X 이 도입됨으로써 코드 삽입 공격을 하는 공격자는 데이터 영역에 코드를 삽입하면 실행 권한이 없기 때문에, 코드 영역에는 쓰기 권한이 없기 때문에 애초에 코드를 삽입 하지도 못하게 된다.

이러한 상황에서 공격자들은 코드를 삽입하지 않고 원하는 행위를 하는 방법을 생각하게 되었다. 코드를 삽입하지 않는다면 이미 존재하는 코드를 이용하여 공격을 해야 하는데, 코드 영역에 존재하는 작은 코드 조각들을 연결하여 하는 공격 방법을 코드 재사용 공격이라고 한다. 이 때 문제는 '원하는 코드 조각(가젯)을 어떻게 연결할 것인가?' 이다. 원하는 가젯을 연결하는 것은 분기 명령어를 사용하거나, jump 명령어를 사용하는 등의 다양한 방법으로 할 수 있다. 이 방식에 변화를 주어가며 코드 재사용 공격은 발전해 왔고, 이에 따라 방어 기법도 역시 발전해 왔다.

본 논문에서는 코드 재사용 공격이 발전해온 궤적을 따라가 보려 한다. 또한 발전의 방향성은 방어 기법에 필연적으로 영향을 받기 때문에 방어 기법 역시 같이 언급하려 한다. 본 논문의 범위는 코드 재사용 공격의 전신인 return into libc 공격부터 시작해서 최신 공격인 데이터 지향 프로그래밍까지이다.



(그림 1) 코드 재사용 공격과 방어의 발전

2. 코드 재사용 공격과 방어

코드 재사용 공격은 적은 수(4~5)개의 명령어로 이루어진 코드 조각인 가젯을 연결하여 악의적인 코드를 수행하도록 하는 공격이다. 이 공격이 성립하기 위해서는 몇가지 조건들이 필요하다.

- 프로그램의 코드에 대한 reverse-engineering 혹은 정적 분석
- 제어 흐름을 하이재킹하기 위한 버퍼 오버플로우 (즉, 공격자는 데이터 영역을 오염시킬 수 있어야 한다.)
- 가젯들과 그 가젯들의 주소

위의 조건들이 만족되는 코드에서 코드 재사용 공격은 다음과 같은 과정을 거쳐 수행되게 된다.

1) 존재하는 코드에서 가젯들을 찾는다.
2) 가젯들의 순서를 정하고 제어 흐름을 바꾸는 명령어를 이용해 엮도록 디자인 한다.
3) 오버 플로우 공격을 통해 스택 또는 힙에 가젯들의 주소를 2)에서 정한 순서대로 저장한다.
4) 코드를 실행시킨다.

위의 과정을 보면 가젯들이 어떻게 연결되어 실행

되는지가 명확하지 않다. 이는 뒤에 나올 각 공격들의 설명에서 언급하도록 하겠다.

2.1. Return Into Libc

Return into libc 공격은 코드 재사용 공격의 전신이라 할 수 있다.[1] 서론에서 말한 코드를 삽입하는 대신 이미 존재하는 코드를 사용하는 아이디어를 이용하였다. 다만, 코드 재사용 공격과 다른 점은 코드 재사용 공격은 작은 조각들인 가젯을 엮어서 원하는 행위를 하도록 만든다면, return into libc 공격은 C 라이브러리인 libc 를 사용하는 차이가 있다.

Libc 는 거의 모든 유닉스 프로그램에 링크 되어 있고, system 이나 execve 와 같은 공격자가 사용하기 좋은 함수들이 내장되어 있다. 따라서 존재하는 코드에서 공격에 유용한 함수를 찾아 사용한다고 했을 때, libc 는 대부분의 코드에 존재하고 공격에 유용한 함수를 다수 포함하기 때문에, 공격자의 요구를 만족시키는 대상이다. 공격자는 이에 착안하여 실행 흐름을 libc 의 내장 함수로 바꾸어 공격을 하였고, 원하는 행위를 할 수 있게 되었다.

하지만 return into libc 공격에는 한계가 있었다. 우선, 공격 자체가 libc 에 강하게 의존하는 점을 꼽을 수 있다. Libc 디자이너가 return into libc 공격의 존재를 알고 있기 때문에, system 과 같이 강력한 함수를 제거하거나 필요한 경우에만 포함시키도록 바꿀 수 있다. 또 다른 문제점은 공격자가 오직 하나의 함수만을 실행할 수 있다는 점이다. 공격자가 원하는 다양한 행위를 하기 위해서는 분기가 가능해야 하는데, return into libc 공격은 이것이 가능하지 않다.

2.2. 반환 지향 프로그래밍

Return into libc 공격의 한계를 인식하고 공격자는 다른 방법으로 이미 존재하는 코드를 사용하는 방법을 생각했다. 그렇게 나온 것이 앞서 언급한 코드 재사용 공격이다. 코드 재사용 공격의 초기에는 가젯을 연결할 때 분기 명령어 중 return 을 사용하였고, return 으로 원하는 행위를 프로그래밍한다고 하여 반환 지향 프로그래밍이라는 이름이 붙었다.[2]

좀 더 자세히 반환 지향 프로그래밍의 동작 방식에 대해 설명하면 다음과 같다. 먼저 공격자는 원하는 행위를 하는 코드를 생각해 두고, 이에 해당하는 return 으로 끝나는 가젯들을 찾는다. 그 다음 메모리 오염을 이용하여 스택에 원하는 가젯들의 주소를 차례로 쌓아 둔다. 그리고 스택 포인터를 첫 가젯의 주소를 가진 곳으로 두어 return 이 일어나면, 호출 규약에 의해 스택 포인터는 스택의 다음 영역을 가리키게 되고 해당 가젯이 실행된다. 이 가젯의 마지막은 또 다시 return 이고 현재 스택 포인터는 공격자가 저장한 다음 주소를 가리키고 있으므로 두번째 가젯으로 return 이 일어나게 된다. 이러한 과정이 반복되면서 공격자가 찾아 둔 가젯이 차례로 실행되어 공격자가 원하는 행위를 하게 된다.

반환 지향 프로그래밍을 막는 기법으로 대표적인 예시가 shadow stack 이다. Shadow stack 은 return 이 프로그래머가 의도한 대로 return 이 되는지 감시하는 방법이다. 호출이 일어날 때 반환 주소가 스택에 저장되게 되는데, 이 때 같은 값을 shadow stack 에 저장시킨다. 후에 return 이 일어나 반환 주소를 참조할 때 shadow stack 에 있는 값과 비교하여 원래 저장되었던 반환 주소로 return 이 되는지 확인한다. 만약 공격자가 반환 주소를 변조하였다면 shadow stack 에 저장된 값과 다른 값으로 return 되기 때문에 이는 shadow stack 의 규칙을 위반한다.[3]

2.3. Jump 지향 프로그래밍

Return 명령어를 이용한 가젯의 연결이 어렵게 되자, 공격자는 앞서 질문한 ‘원하는 가젯을 어떻게 연결할 것인가?’로 돌아가서 새로운 공격을 만든다. 가젯의 연결을 return 명령어 대신에 jump 명령어를 사용하고, 반환 지향 프로그래밍과 같은 방법으로 공격을 시도하였다. 이 공격을 반환 지향 프로그래밍과 같이 이름을 붙여, jump 지향 프로그래밍이라고 한다.[4]

이 때 발생하는 새로운 문제는 반환 지향 프로그래밍에서는 연속된 위치에 있는 주소들은 return 이 스택 포인터를 바꾸는 특성을 이용하여 자연스럽게 읽도록 만들 수 있었는데, jump 명령어는 그렇지 않다는 것이다. 이에 따라 새로운 공격에서는 반환 지향 프로그래밍에서 스택 포인터를 옮기는 역할을 하는 가젯이 필요하게 되고, 이 가젯을 디스패처라고 한다. 디스패처는 명령어 집합에 따라 다르게 만들어진다. 먼저 Intel x86 에서는 pop %eax; jmp *%eax 와 같은 가젯을 찾아서 레지스터에 있는 값으로 jump 를 하게 만든다. 반면 ARM 에서는 pop-jump 시퀀스가 존재하지 않기 때문에, Update-Load-Branch(ULB) 시퀀스를 이용하여, 레지스터의 값을 update 하고, 이를 branch 되는 레지스터에 load 한 뒤, load 된 값으로 branch 하는 방식을 사용하게 된다.

Jump 지향 프로그래밍의 동작 방식을 더 자세히 설명하면 다음과 같다. 먼저 디스패처 역할을 할 수 있는 가젯을 찾은 뒤, 원하는 행위를 하고 디스패처로 jump 하는 명령어로 끝나는 가젯들을 찾는다. 이제 메모리 오염을 이용하여 가젯들의 주소를 순차적으로 저장시키고 디스패처에서 update 에 이용하는 레지스터를 첫번째 가젯의 주소를 가진 곳을 가리키게 만든다. 이제 가젯을 실행시키면, 디스패처로 jump 되고, 디스패처에서는 레지스터 값을 4 증가시킨 뒤(update), 이 값을 다른 레지스터에 load 하고, load 된 레지스터의 값으로 jump 하게 되는데, 이는 다음 가젯의 주소로 jump 하는 것과 같다. 이 행위를 반복하면 원하는 가젯 시퀀스를 실행시켜서 공격자의 목적을 달성하게 된다.

Jump 지향 프로그래밍을 막는 기법으로는 branch regulation 이 있다. Branch regulation 은 분기 명령어들에 대해서 적합한 분기 명령어만을 허용하는 방법이다. Jump 지향 프로그래밍에서 활용되는 jump 명령어는 적합하지 않은 분기 명령어로 분류되어 branch regulation 에 의해 막히게 된다. 여기서 적합한 분기 명령어는 세가지로 나눌 수 있다.

1) 같은 function 으로 뛰는 분기 명령어
2) 다른 function 의 entry point 로 뛰는 분기 명령어
3) 앞 두가지를 만족하는 call 로 생성된 return address 로 뛰는 분기 명령어

이 세가지를 검사하기 위해서 반환 주소에 대해서는 저장된 반환 주소로 뛰는지 확인하고, indirect jump 에 대해서는 먼저 같은 함수로 뛰는지 확인한 뒤, 다른 함수라면 뛰는 곳이 함수의 엔트리 포인트인지 확인한다. 마지막으로 호출에 대해서는 목적 주소가 함수의 엔트리 포인트인지와 저장된 반환 주소와 일치하는지를 확인한다. Direct jump 는 코드 영역을 공격자가 수정할 수 없기 때문에 따로 검사하지 않는다.[5]

2.4. Just In Time (JIT) 반환 지향 프로그래밍

Branch regulation 가 jump 지향 프로그래밍을 억제하였지만 두가지 문제가 있었다. 이진 파일을 다시 써야한다는 점은 이전의 이진 파일에 대한 적용을 어렵게 한다. 더 큰 문제점은 함수가 충분히 크다면 그 함수 안에서 가젯들을 찾아서 코드 재사용 공격을 할 수 있다는 점이다. 이는 branch regulation 이 완전한 보안상의 안전을 보장하지 못함을 의미한다.[6]

주소 공간 배치 난수화는 코드 재사용 공격을 막기 위해 다른 갈래로 고안된 방어 기법이다. 주소 공간 배치 난수화는 코드, 스택, 힙 등의 메모리 각 영역의 시작 주소에 난수의 오프셋을 더하여 공격자가 원하는 메모리의 주소를 알 수 없게 만든다. 이전까지의 공격에서는 공격자가 정확한 주소를 알아야 했기 때문에 이는 간단하면서도 효과적인 방어법이다. 주소 공간 배치 난수화가 많이 쓰임에 따라, Jump 지향 프로그래밍 이후의 공격과 방어는 주소 공간 배치 난수화를 중심으로 발전하게 된다.[7]

이제 주소 공간 배치 난수화에 의해, 공격자는 주소를 정확히 알 수 없게 되어 메모리 오염을 일으킬 수 있는 능력이 있더라도, 가젯의 주소를 알 수 없게 된다.[8] 특히, 정교한 주소 공간 배치 난수화는 라이브러리의 시작 주소 뿐만 아니라 명령어 순서까지 난수화한다. 공격자는 주소 공간 배치 난수화는 프로세스가 시작될 때만 주소를 섞는다는 것을 알고 런타임에 가젯을 찾아내는 방법을 생각한다. 가젯을 찾아내기 위해서는 결국 코드 영역을 봐야하는데, 코드 영역을 볼 방법은 코드를 직간접적으로 가리키는 포인터를 찾아내는 것이다. 이것이 가능한 이유는 페이지 내의 한 주소만 유출되어도 그 페이지 안의 모든 주소를 유추해 낼 수 있기 때문이다. 이렇게 런타임에 찾아낸 가젯들을 모아 반환 지향 프로그래밍을 하는 것이 JIT 반환 지향 프로그래밍이다.[9]

JIT 반환 지향 프로그래밍을 방어하는 방법에는 Readactor 와 Shuffler 가 있다. Readactor 는 확장 페이지 테이블을 이용하여 각 페이지에 독립적으로 다른 권한을 부여하여, 코드 영역을 읽을 수 없게 만드는 방법이다. 가상 메모리 상에 있는 코드 페이지들은 페이지 테이블에서 읽기가 항상 포함된 권한으로 게스트 물리 메모리로 가고, 게스트 물리 메모리에서 호스트 물리 메모리로 갈 때는 확장 페이지 테이블을 거치면서 실행 권한만을 가지게 변경되어 공격자가 코드 영역을 읽지 못하게 만든다. 데이터 페이지에 있는 포인터가 코드 영역을 가리키는 경우에는 포인터가 직접 코드 영역을 가리키지 않고 사이에 실행 권한만을 가지는 코드 페이지인 트램폴린을 두어 공격자가 간접적으로도 코드 영역을 읽지 못하도록 만든다.[10]

Shuffler 는 주소 공간 배치 난수화가 프로세스가 시작될 때만 난수화를 해서 공격 받았기 때문에, 런타임에도 지속적으로 난수화를 하는 기법이다. 이 때 중요한 것은 난수화 때문에 코드의 실행 속도가 많이 느려지면 안된다는 것이다. Shuffler 에서는 연산과 재난수화 과정을 병렬적으로 진행하여 최종적으로 배치를 바꾸는 짧은 시간 동안에만 프로그램을 지연시켜 이 문제를 해결한다.[11]

2.5. 위조 객체 지향 프로그래밍

이제 주소를 알아내기 점점 더 어려워짐에 따라, 공격자들은 주소가 필요 없는 간접 호출, 간접 jump 명령어를 이용하려 한다. 위조 객체 지향 프로그래밍의 저자는 많은 방어들이 C++의 객체 지향성을 간과하고 있는 점을 지적한다. 위조 객체 지향 프로그래밍은 이러한 점에 주목하여, 가상 함수 포인터를 이용하여 코드 재사용 공격을 한다. 가상 함수 포인터를 이용하면 방어자는 C++의 고급 의미가 필요하게 된다. 앞서 언급한 Readactor 역시 가상 함수에 대한 부분을 간과하고 있어서 위조 객체 지향 프로그래밍을 막지 못한다.[12]

코드 재사용 공격에서의 가젯 대신 가상 함수 가젯을 정의하여 사용한다. 가상 함수 가젯은 가상 함수 자체를 가젯으로 활용하여 가상 함수들을 모으면 원하는 행위를 하도록 찾는다. 이때 원하지 않는 코드들은 원하는 행위에 영향을 미치지 않도록 하는 것이 중요하다.

위조 객체 지향 프로그래밍은 다음과 같은 과정을 거쳐 이루어진다.

1) 객체를 가리키는 포인터를 순회하여 가상 함수를 호출하는 가상 함수를 찾는다.
2) 가상 함수 가젯들을 찾는다.
3) 메모리 오염을 이용해 1 의 포인터가 가리키는 객체를 2 에서 찾은 가상 함수를 가지는 객체로 덮어쓴다.
4) 코드를 실행시킨다.

코드를 실행시키면 1)에서 찾은 가상 함수에서 원하는 가상 함수 가젯이 차례로 실행되어 공격자가 원하는 행위를 하게 된다. 주의해야 할 점은 3)에서 객체를 덮어쓸 때 함수의 매개 변수가 원하는 변수가 되도록 잘 덮어써야 한다는 점이다.

위조 객체 지향 프로그래밍은 공격 상의 특징들 덕분에 다음과 같은 장점들을 가진다.

- 코드 포인터를 삽입하지 않기 때문에 코드 포인터에 대한 방어를 우회한다.
- 코드의 저급 의미에 의존하기 때문에 코드 다시 쓰기나 섞기를 우회한다.

● 스택 포인터를 중심으로 작동하기 때문에 일반적인 제어 흐름 무결성이나 shadow stack 을 우회한다.

방어자는 위조 객체 지향 프로그래밍을 막기 위해 Readactor 를 발전시킨 Readactor++을 만든다. Readactor++는 readactor 와 비슷하게 가상 메소드 테이블, 스택, 힙에 저장된 코드 포인터에서 코드 배치를 유추하는 것을 막는다. Readactor++에서는 가상 메소드 테이블을 읽기 권한만을 가진 테이블(rvtable)과 실행 권한만을 가진 테이블(xvtable)로 나누어 저장한다. Readactor 에서의 트램폴린 역할을 rvtable 이 하게 되어 위조 객체 지향 프로그래밍을 막는다.[13]

2.6. 데이터 지향 프로그래밍

이전의 코드 재사용 공격들은 모두 분기 명령어를 이용하여 제어 흐름을 변조하는 방식으로 이루어졌다. 이에 반해 데이터 지향 프로그래밍은 분기 명령어를 이용하지 않고, 비제어 데이터만을 변조하여 합법적인 제어 흐름 그래프안에서 제어 흐름을 변조하여 공격을 해낸다.[14] 예를 들면 반복 문이 10 번을 실행되어야 하는데 5 번만 실행시키게 되면, 제어 흐름 그래프 내에서는 문제가 없지만 제어 흐름은 변조되었다.

데이터 지향 프로그래밍은 반복문 안의 분기문을 선택하여 가젯들을 엮는다. 비제어 데이터를 변조하여 원하는 분기문이 실행되게 하여 원하는 가젯이 원하는 순서대로 실행되도록 만든다.

3. 결론

본 논문에서는 코드 재사용 공격과 방어 방법의 발전 과정에 대해 알아보았다. 많은 공격들과 방어들이 있었지만, 여전히 방어에는 어려움이 있다. 성능 저하 문제 때문에 실제 시스템에는 안 쓰이는 경우가 많기 때문이다. 또한 데이터 지향 프로그래밍은 아직 state-of-the-art 의 방어법이 나오지 않은 상태이다.

방어자가 앞으로 연구해야 될 부분은 주소 공간 배치 난수화와 같이 실제 시스템에서 사용할 수 있고 범용적인 방어법이다. 데이터 지향 프로그래밍에 대한 방어법 또한 해결해야 할 과제이다.

4. ACKNOWLEDGEMENT

이 논문은 2017 년도 두뇌한국 21 플러스사업에 의한 지원과 2017 년도 정부(미래창조과학부)의 재원으로 한국연구재단의 지원, 2017 년도 정부(미래창조과학부)의 재원으로 정보통신기술진흥센터의 지원(No.2016-0-00078, 맞춤형 보안서비스 제공을 위한 클라우드 기반 지능형 보안 기술 개발) 및 2017 년도 정부(과학기술정보통신부)의 재원으로 정보통신기술진흥센터의 지원을 받아 수행된 연구이며 (2015-0-00573, 시스템 침입 탐지를 위한 프로세서 모니터링 기술 및 주요 HW/SW 모듈 개발) 과학기술정보통신부 및 정보통신기술진흥센터의 대학 ICT 연구센터육성지원사업의 연구결과로 수행되었음 (IITP-2017-2015-0-00403).

참고문헌

[1] Solar Designer. Getting around non-executable stack (and fix). http://seclists.org/bugtraq/1997/Aug/63

[2] S. Krahmer. x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique, Sept. 2005. http://www.suse.de/~krahmer/no-nx.pdf.

[3] Vendicator. Stack Shield: A "stack smashing" technique protection tool for Linux. http://www.angelfire.com/sk/stackshield.

[4] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. Return-oriented programming without returns. In 17th ACM CCS, 2010.

[5] M. Kayaalp, M. Ozsoy, N. Abu-Ghazaleh, and D. Ponomarev. Branch regulation: Low overhead mitigation of code reuse attacks. In Proceedings of ISCA, 2012.

[6] M. Kayaalp, T. Schmitt, J. Nomani, D. Ponomarev, and N. AbuGhazaleh, “Scrap: Architecture for signature-based protection from code reuse attacks,” 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA), vol. 0, pp. 258–269, 2013.

[7] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In the 34th IEEE Symposium on Security and Privacy, 2013.

[8] PaX Team. Address Space Layout Randomization (ASLR). http://pax.grsecurity.net/docs/aslr.txt.

[9] Snow, Kevin Z., et al. "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization." Security and Privacy (SP), 2013 IEEE Symposium on. IEEE, 2013.

[10] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A.-R. Sadeghi, S. Brunthaler, and M. Franz. Readactor: Practical code randomization resilient to memory disclosure. In IEEE Symposium on Security and Privacy, 2015.

[11] D. Williams-King, G. Gobieski, K. Williams-King, J. P. Blake, X. Yuan, P. Colp, M. Zheng, V. P. Kemerlis, J. Yang, and W. Aiello, “Shuffler: Fast and deployable continuous code re-randomization,” in OSDI, 2016.

[12] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications. In 36th IEEE Symposium on Security and Privacy, S&P, 2015.

[13] S. Crane, S. Volckaert, F. Schuster, C. Liebchen, P. Larsen, L. Davi, A.-R. Sadeghi, T. Holz, B. D. Sutter, and M. Franz. It’s a TRaP: Table randomization and protection against function reuse attacks. In ACM SIGSAC Conference on Computer and Communications Security, CCS, 2015.

[14] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang. Data-oriented programming: On the effectiveness of non-control data attacks. In IEEE S&P, 2016.