힙 공격으로부터 방어를 위한 데이터 포인터 인코딩

김경태○ 표창우
홍익대학교 컴퓨터 공학과
kt0755@gmail.com , pyo@hongik.ac.kr

Data Pointer Encoding for Defense against Heap Attack

Kyungtae Kim○ Changwoo Pyo
Department of Computer Engineering, Hongik University

요 약

버퍼 오버플로우의 공격은 스택의 영역뿐만 아니라 데이터 세그먼트나 힙 영역에서도 다양한 형태가 가능하다. 이 논문은 힙 영역에 대한 동적 메모리 할당 함수의 취약점 공격을 방지하는 방안을 제시한다. 제안된 방법은 데이터 포인터의 값을 암호화 하여 저장하고, 참조할 때 복호화 한다. 힙 공격은 원하는 주소에 원하는 값을 기록할 수 있게 하기 때문에 데이터 변수 또는 포인터 공격에 활용될 수 있다. 데이터 포인터 암호화는 아직 알려지지 않은 데이터 포인터와 변수에 대한 공격까지 방어할 수 있을 것으로 예상된다.

1. 서 론

버퍼 오버플로우 공격은 1988년에 처음 등장한 이후 지속적으로 공격에 이용되어 왔고 앞으로도 계속 될 전망이다. 시스템 개발 언어인 C는 배열의 한계 검사를 요구하지 않기 때문에 공격자는 버퍼에 기록할 때 한계를 넘어 공격 대상이 되는 메모리 공간을 변조시킬 수 있다. 스택 버퍼 오버플로우에서 함수 호출시 스택에 쌓이게 되는 반환주소가 가장 대표적인 변조 대상이다. 변조된 반환주소는 함수가 복귀할 때 프로그램 카운터에 적재되어 공격자가 원하는 주소로 제어 흐름을 이동 시킬 수 있게 한다.

버퍼 오버플로우 공격은 스택의 영역뿐만 아니라 데이터 세그먼트나 힙 영역에서도 다양한 형태가 가능하다. bind8(DNS 네임서버) 공격[1]은 힙 오버플로우를 통해 루트권한을 얻는 예를 보여준다. 힙 오버플로우의 대표적인 공격 형태는 할당 해제된 메모리공간을 이중 연결 리스트 형태로 유지하기 위한 두 개의 데이터 포인터를 변조시키는 방식이다. 공격자는 이 데이터 포인터들을 변조하여 임의의 메모리 주소의 저장된 값을 변경시킬 수 있다. 지금까지는 코드 포인터들을 조작하여 제어 흐름을 가로채는 공격이 주류를 이루고 있지만 데이터 메모리 값 변조를 이용한 공격도 증가하고 있고[2], 힙 공격이 그 가능성을 높이고 있다.

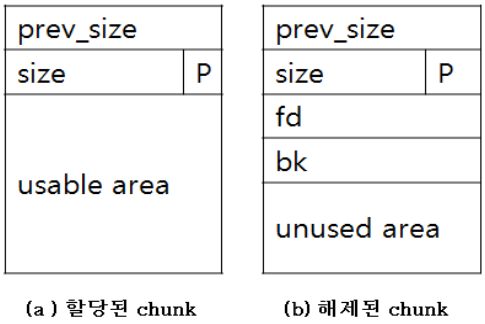
본 논문은 힙 오버플로우를 이용하여 동적 메모리 할당 함수의 취약점 공격을 방지하는 방안을 제시한다. 프로그램 카운터 인코딩[3]을 데이터 포인터에 적용 하는 방식으로서 데이터 포인터 값을 암호화 하여 저장하고, 참조할 때 복호화 한다.

논문의 구성은 다음과 같다. 2절에서는 힙 공간 관리에 대하여 살펴보고 3절에서는 데이터 포인터의 취약성과 그 공격 방법을 소개한다. 4절에서 데이터 포인터의 보호 기법을 제시하고 5절에서 마무리 한다.

- 이 논문은 2008학년도 홍익대학교 학술연구진흥비에 의하여 지원되었음
- 본 연구는 지식경제부 및 한국산업기술평가관리원의 IT산업원천기술개발사업의 일환으로 수행하였음 [2010-KI002090, 신뢰성 컴퓨팅 (Trustworthy Computing) 기반 기술 개발]

2. 힙 공간 관리

GNU C 라이브러리와 레드햇, 데비안을 비롯한 대부분의 리눅스는 기본적으로 더그 리(Doug Lea)의 dlmalloc[4]을 기반으로 힙 메모리를 관리하고 있다. dlmalloc 에서는 힙 메모리는 조각(chunk) 단위로 구성되고 조각은 할당된 조각과 할당이 해제된 자유조각(free chunk)으로 구분한다. [그림 1]은 조각의 구조를 나타낸다. prev_size 는 앞 조각의 크기를 나타내고 size 는 현재 조각의 크기를 의미한다. prev_size 를 통해 앞 조각의 접근이 가능하고 size 를 통해 다음 조각의 접근이 가능하다. 조각이 사용자에게 할당이 되면 오직 size 필드만 의미 있는 값을 가진다. 해당 조각이 해제되었을 때 size외의 다른 부분들에 역시 의미있는 정보가 기록된다. P(PREV_INUSE) 플래그로 앞의 조각이 할당되었는지 해제되었는지를 확인 할 수 다. 이 값이 '0'일 경우 앞의 조각이 해제되었다는 의미이고 '1'일 경우 할당되었다는 의미이다. P플래그가 '1'일 때 prev_size 가 유효하지 않고 '0'일 때만 유효하다. 이로 인하여 할당된 조각에서 사용자가 실제 사용가능한 공간은 size 의 아래쪽 영역부터 시작하여 다음 조각의 prev_size 를 포함한 영역이다. 자유조각들은 fd(forward pointer)와 bk(back pointer)라는 2개의 데이터 포인터를 통해 이중



[그림 1] 힙 메모리 관리를 위한 구조체

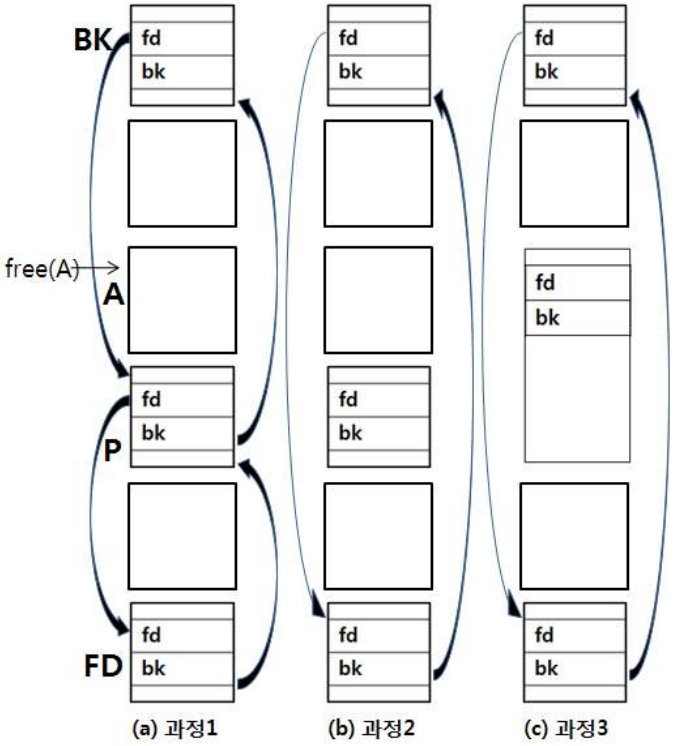
연결 리스트(bins)로 관리된다. 자유 조각 리스트는 그 크기에 따라 여러 종류로 분류된다. large bin, small bin 이 대표적이고 캐시의 역할로 쓰이는 unsorted bin 등이 있다.

메모리 조각은 free()연산 도중 합쳐질 수 있다. 할당 해제 시킬 조각의 물리적으로 인접해 있는 조각들이 자유 조각이라면 서로 합쳐서 하나의 자유조각이 되도록 만든다. 이를 통하여 자유 조각의 관리를 편하게 하고 단편화도 최소화 시킬 수 있다. 예를 들면 조각 A를 free(A) 호출로 할당해제 시킬 때 조각 A의 앞 조각이나 뒷 조각이 자유 조각이라면 이들을 합쳐서 하나의 자유 조각만 존재하도록 한다(물리적으로 인접한 자유 조각이 없도록 한다). 하나의 자유 조각으로 합치기 위해서 합쳐져야 할 자유 조각을 자신이 소속된 자유 조각 리스트로부터 제거해야 한다. 이 작업은 unlink 매크로가 담당한다. [그림 2]와 같이 unlink 매크로는 데이터 포인터들을 조작하여 자유 조각 P를 자유 조각 리스트에서 제거한다.

```
#define unlink (P, BK, FD) {
        FD = P->fd;
        BK = P->bk;
        FD->bk = BK;
        BK->fd = FD;
}
```

[그림 2] unlink 매크로

[그림 3]에서는 free(A) 함수 호출로 조각 A가 할당 해제되는 과정의 일부를 보여준다. [그림 3](a)와 같이 할당해제 할 조각 A의 뒷 조각은 자유조각이다. 따라서 자유조각이 연속되는 것을 방지하기 위해 조각 A의 할당 해제 시 조각 P와 합쳐져야 하고 먼저 조각 P를 자신이 속한 자유조각 리스트에서 제거해야 한다. [그림 3](b)는 조각 P를 unlink 시킨 결과를 보여준다. 이 후 [그림 3](c) 처럼 두 개의 조각이 하나의 자유조각으로 통합된다. 통합된 자유조각은 캐시역할로 사용하는 자유조각 리스트인 unsorted bin 에 연결된다.



[그림 3] unlink를 통한 자유조각의 합병

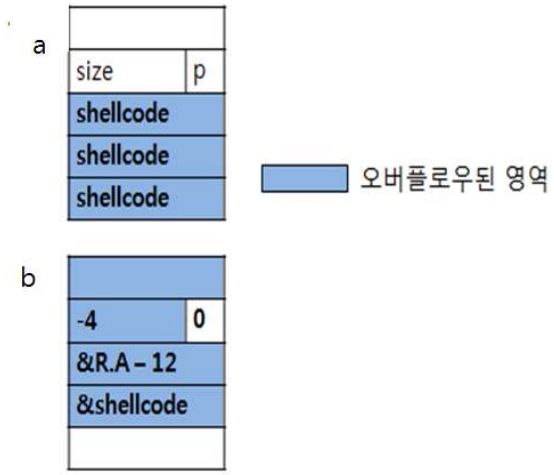
3. 데이터 포인터 취약성과 공격

```
.....
a = malloc(40);
b = malloc(10);
c = malloc(10);
strcpy(a, argv[1]);
free(a);
free(b);
free(c);
.....
```

[그림 4] 취약 프로그램

[그림 4]의 취약한 코드를 통해 힙 오버플로우의 공격 패턴을 확인할 수 있다. 입력크기 제한 없는 strcpy()에서 a 의 크기보다 더 큰 값을 입력받게 되면 버퍼 오버플로우가 일어난다. 즉 조각 b의 변조가 가능하다. [그림 5] 는 표준입력을 통해 오버플로우를 발생시킨 후의 각 조각 상태를 나타낸다. 조각 b의 fd는 R.A (return address) 의 주소에 12를 뺀 값으로 변조되었고 bk는 쉘코드의 주소로 변조되었다.

free(a) 함수 호출이 되면 조각 a 의 뒷 조각이 해제되었는지 확인하기 위해 조각 c 의 P 비트를 조사할 것이다. 이 때 조각 b 의 size 필드를 이용하는데 이 값은 이미 버퍼 오버플로우로 변조가 되어 -4 의 값을 가지고 있고 그 결과로 조각 c 를 참조하기 위해 조각 b 의 위치에서 -4만큼 연산이 이루어져 이는 마치 조각 b 가 조각 c 인 것처럼 위장하게 된다. 조각 b 의 P 비트는 '0'이고 결과적으로 조각 c 의 P 비트가 '0'인 것처럼 보



[그림 5] 오버플로우 된 조각

여 조각 b 는 자유 조각으로 가정되고 조각 b 에서 unlink 과정이 일어난다. unlink 과정에 따라 조각 b 의 변조된 fd 와 bk 를 조작하여 결국엔 반환주소 의 값에 쉘 코드 주소가 삽입된다. 본 예제에서는 반환주소를 변조하였으나 반환주소 뿐만 아니라 GOT나 함수포인터 같은 다른 코드포인터들도 변조될 수 있다. 결국 변조된 코드 포인터들이 참조될 때 쉘 코드가 실행된다.

4. 데이터 포인터 암호화

이 같은 데이터 포인터의 취약성을 해결하기 위해 논문에서는 데이터 포인터를 암호화하는 기법을 제안한다. 이는 프로그램 카운터 인코딩[3]의 개념을 데이터 포인터에 적용하는 방식이다. 암호화와 복호화는 키와 데이터 포인터의 XOR 연산을 통해 수행된다. 안전한 소스로부터 키를 받아서 암호화 시점에 fd/bk 데이터 포인터를 각각 키와 XOR 시키고 마찬가지로 복호화 시점에 fd/bk 각각을 암호화 시켰던 키로 XOR 시킨다. 암호화 기법을 도입하기 위하여 몇 가지 고려해야 할 사항은 다음과 같다.

1. 암호화시점
2. 복호화시점
3. 키의 보안성

1. 암호화 과정은 데이터 포인터가 유효해진 직후의 시점에 적용한다. malloc() 호출시 사용자에게 메모리 공간을 할당하기 위해 각 bin 들을 탐색하는데 자유 조각의 크기와 사용자가 요구한 조각의 크기가 일치(best fit)하지 않으면 리스트내의 다음 조각을 탐색하게 된다. unsorted bin 내의 자유 조각들은 탐색 후에 크기가 일치하지 않으면 unsorted bin 에서 제거되어 자신의 크기에 맞는 자유조각 리스트로 재연결된다. 이때 탐색된 조각의 데이터 포인터들이 재설정되고 암호화가 필요하다.
2. 복호화 시점은 실제 데이터 포인터가 참조되기 직전이다. 즉 unlink 매크로가 수행되는 시점이 실제 fd/bk 가 참조되는 시점이고 [그림 6]과 같은 수정이 필요하다.
3. 키가 노출이 된다면 암호화의 의미는 사라지게 된다. 따라서 암호화를 어떤 키 값으로 할 것인지는 민감한 사항이다. 하지만 키 값에 관련된 논의는 논문 주제에서 벗어나기 때문에 본 논문에서는 키 값은 노출 되지 않는 보안성을 갖는다고 가정한다.

```
FD = ( P->fd ) ^ key;
BK = ( P->bk ) ^ key;
FD->bk = P->bk;
BK->fd = P->fd;
```

[그림 6] 수정된 unlink 매크로

데이터 포인터 암호화 기법을 도입함으로써 공격자는 암호화 된 데이터 포인터를 변조하게 되고 결국 복호화 과정에서 공격자가 의도한 값이 바뀌기 때문에 공격자는 자신이 원하는 주소를 데이터 포인터에 쓸 수 없게 된다. 3절의 예제 같이 위장 조각을 이용하게 되더라도 암호화는 거치지 않을 수 있지만 unlink 에서 복호화 과정을 거치게 되므로 공격자가 의도한 공격을 차단할 수 있다.

현재의 동적 메모리 관리 코드를 포함하는 glibc 최신 버전은 데이터 포인터들의 변조여부를 확인(detecting)하는 코드들이 추가되었다. 하지만 이 같은 확인 코드들은 코드크기에 의한 성능저하를 유발한다. 또한 확인의 방법은 데이터 포인터 자체를 보호 해주는 것이 아니기 때문에 또 다른 보호 대상을 만들 수 있고 공격자가 더 고차원적인 우회공격을 사용한다면 또 다른 추가적인 방어 코드가 필요하다. 마치 스택의 반환주소를 보호하기 위해 스택가드[5]의 방법이 소개되었지만 이를 우회할 수 있는 포맷문자열과 같은 공격법이 제시되어 이에 대한 추가적인 방어가 필요한 것과 같다. 결국은 어떠한 우회공격도 궁극적으로는 데이터 포인터를 변조시켜야 하기 때문에 공격자가 모든 확인 과정을 우회한다는 최악의 상황을 고려한다고 하더라도 데이터 포인터 스스로 방어할 수 있는 무엇인가가 있다면 데이터 포인터의 변조는 허용할 수 있지만 공격자가 원하는 위치에는 쓸 수 없게 막을 수 있다. 논문에서 제안한 암호화/복호화 기법은 데이터 포인터 자체를 보호하기 때문에 확인과정에서 확인하지 못한 부분을 가장 마지막 차원에서 보호 할 수 있다.

5. 결론

힙 공격은 원하는 주소에 원하는 값을 기록할 수 있게 하기 때문에 데이터 변수 또는 포인터 공격에 활용될 수 있다. 논문에서 제안한 데이터 포인터 암호화는 아직 알려지지 않은 데이터 포인터와 변수에 대한 공격까지 방어할 수 있을 것으로 예상된다.

6. 참고문헌

[1]http://www.openpkg.com/security/advisories/OpenPKG-SA-2002.011.html, 2002

[2] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, R. K. Iyer, Non-control-data attacks are realistic threats, *14th UNSENIXSecuritySymposium*, pp.177-191, 2005

[3] C. Pyo and G. Lee. Encoding function pointers and memory arrangement checking against buffer overflowattack. In *In Proceedings of the 4th International Conference In Information and Communications Security*, 2002

[4] Doug lea, malloc.c version 2.8.4, http://gee.cs.oswego.edu/pub/misc/malloc.c, 2009

[5] C. Cowan, C. Pu, D Maier, J. Walphole, P Bakke, S. Beattie, A. Grier, P Wagle, Q. Zhang, and H. Hinton, "StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks", *In Proc. 7th USENIX Security Symposium*, Jan 1998