# FRRmalloc:일회성 할당 및 리매핑 기반의 효율적인 Use-After-Free 방지

김정훈[1],조영필[2]
[1]한양대학교 컴퓨터소프트웨어학과 (미래자동차-SW융합전공) 석박통합과정
[2]한양대학교 컴퓨터소프트웨어학과 교수
rlawjdgns527@hanyang.ac.kr , ypcho@hanyang.ac.kr

# FRRmalloc : Efficient Use-After-Free prevention based on One-time-allocation and batch remapping

Jeong-Hoon Kim[1], Yeong-Pil Cho[2]
[1]Department of Computer and Software (Automotive-Computer Convergence), Han-Yang University
[2]Dept. of Computer Science, Han-Yang University

요 약

UAF(Use-After-Free)는 heap 영역에서 메모리 오염을 발생시킬 수 있는 취약점이다. UAF를 방지하기 위해 다양한 방법으로 관련 연구가 활발히 이루어지고 있지만, 아직까지 여러 오버헤드 측면에서 모두 좋은 성능을 발휘한 결과는 나오지 않고 있다. 할당자 수준에서의 수정을 통하여, UAF 취약점 방어를 보장하는 동시에 높은 성능과 낮은 오버헤드를 발생시킬 수 있는 방법을 제시한다. 본 논문에서는 UAF 취약점 및 관련 연구를 소개하고, 이를 기반으로 UAF 취약점에 대처할 수 있는 방법을 제시한다.

## 1. 서론

UAF(Use-After-Free)은 프로그램이 힙 공간 내 메모리를 해제(free)한 뒤 그 메모리 영역을 다시 사용하려 할 때 발생하는 메모리 취약점이다. 메모리 해제 이후 해당 포인터를 null로 설정하지 않거나, 다른 방법으로 해제된 메모리에 대한 접근을 명시적으로 차단하지 않는 경우, 프로그램은 여전히 해제된 메모리 영역을 참조할 수 있다. 이때 이미 해제된 메모리 영역을 가리키고 있는 포인터를 dangling pointer라고 하며, 이 포인터는 유효한 메모리 주소를 참조하는 것처럼 보이지만, 실제로는 시스템에 의해 다른 용도로 재할당되거나 접근이 금지된 메모리를 가리키게 된다. Dangling pointer를 사용하여 악의적인 공격자가 해당 메모리를 이용하여 임의의 코드를 삽입/실행하거나 시스템에 대한 권한을 획득할 수 있다. 이는 무결성과 기밀성,시스템 가용성을 심각하게 위협할 수 있고, CWE Top 25(2023)[1]에서도 4위에 등재될 만큼 영향력과 파급력이 큰 취약점 중 하나이다. UAF 취약점을 해결하기 위해 지금까지 많은 연구들이 진행되었는데, 대표적으로 다음과 같은 3가지 방식으로 진행되었다.

첫 번째는 Dangling pointer 사용 감지 및 접근 검증 방법이다. 실행 시간(runtime)에 해제된 메모리에 대한 접근을 감지하고, 해당 접근을 차단하거나 경고를 발생시킨다. 접근 검증을 통해, 시스템은 포인터가 유효한 메모리 영역을 가리키고 있는지 확인하고, 그렇지 않은 경우 오류를 발생시켜 UAF 문제를 예방할 수 있다[2][3][4]. 두 번째는 Dangling Pointer 생성을 방지하는 방법이다. 메모리가 해제될 때 모든 관련 포인터를 null과 같은 안전한 값으로 초기화하여, dangling pointer가 생성되는 것을 방지하는 방법이다. 이를 통해, 프로그램이 해제된 메모리 영역에 접근하려고 할 때, 오류를 발생시켜 프로그램의 안전한 종료나 오류 처리 루틴을 실행할 수 있다[5][6]. 마지막은 안전하지 않은 메모리를 재사용하지 않는 방법이다. 이는 기본적으로 메모리 해제 후에 메모리 영역을 짧은 시간 내 다시 사용하지 않는 방법을 의미한다. 메모리가 해제되면, 해당 메모리 영역을 즉시 다시 사용할 수 없도록 함으로써, 해제된 메모리에 대한 재사용 유효성이 보장될 때까지 재사용을 하지 않는 방법이다. 해제된 메모리에 대한 재사용 유효성은 대표적으로 mark-and-sweep 과정을 통하여 이루어진다. Mark-and-sweep 알고리즘은 가비지 컬렉션의 일종으로,MarkUS[7]에서는 mark와 sweep 단계를 통하여 메모리를 스캔 및 회수하여 UAF을 방지하고 있다. 안전하지 않은 메모리를 재사용하지 않는 다른 방법으로는, 매 할당시 새로운(높은) 가상 주소를 사용하여 할당하는 방식이다. 항상 새로운 가상 주소를 사용하여 가상 주소의 중복된 사용이 일어나지 않기 때문에 UAF를 방지하는 간편하지만 강력한 방법으로 사용될 수 있다.이러한 할당 방식을

One-Time-Allocation(OTA)라고 하며 FFmalloc[8]은 할당 요청이 들어올 시, 항상 이전 할당보다 상위 주소를 리턴하는 OTA방식으로 메모리를 할당한다.

앞서 기술한 UAF방지를 위한 방식들은 false negative 발생, 성능 오버헤드, 메모리 오버헤드 등과 같은 문제점을 야기한다. 따라서 아직까지 UAF 방지를 위한 보편적인 해결책은 파악하기가 힘든 상황이다. 이 논문에서는 여러 UAF 방지법을 결합 및 개선하여 기존의 장점을 잃지 않고, 단점을 보완할 수 있는 새로운 메모리 할당자 FRRmalloc(Forward-Remap-reclaim memory allocator)을 제시한다.

## 2. 연구 동기

### 2.1. One-Time-Allocation

앞서 소개한 방법중 OTA(One-Time-Allocation)은 성능 오버헤드가 매우 훌륭하여 이를 충분히 고려할만한 동기를 제공한다. OTA는 할당 시 이미 사용한 가상 주소 공간을 재사용하지 않기 때문에, 할당 방식 자체만으로도 UAF가 방지되는 효과가 있다. 따라서 UAF 방지를 위한 추가 작업이 runtime시 수행될 필요가 없기 때문에, 단순하지만 성능 오버헤드가 발생하지 않으면서 UAF를 방지할 수 있는 강력한 방법이다. 하지만 이 방식의 경우 특정 할당 집약적인 프로그램에 대해서는, 메모리 오버헤드가 상당하다는 단점이 있다. 따라서 성능 오버헤드의 장점은 그대로 유지하면서, 메모리 오버헤드를 완화시킬 필요가 있다.

```
void* OTAmalloc(size_t allocsize){
....
void* allocation;
void* lastalloc;
lastalloc=get_lastalloc();
allocation=lastalloc+allocsize;
return allocation;
}
```

(그림1) OTAmalloc 예시

### 2.2. Memory remapping

sOTA의 메모리 오버헤드를 완화시키기 위해서는, 해제된 메모리 공간에 대한 사용이 필연적이다. FRRmalloc은 remap을 통한 가상 주소 재사용으로 이러한 문제를 해결한다. 이 방식은 기본적으로 두 가지 유형의 페이지를 사용한다. 첫 번째는 canonical 페이지이며, 이는 메모리의 실제 위치를 나타낸다. 두번째는 shadow 페이지로, 이는 canonical 페이지의 remap을 통해 canonical 페이지와 같은 내용을 가지지만(동일한 physical frame을 가리키지만) 다른 가상 주소를 가지게 된다. Oscar[9]에서 이러한 방식을 채택하여 가상 주소를 재사용하고 있다. 하지만 Oscar는 객체가 생성될 때 마다, 1회의 페이지 단위 remap을 하게 되어 system call이 매우 잦아져서 성능 오버헤드가 높아지게 된다. FFRmalloc은 객체 단위가 아닌 페이지 단위로 remap을 진행하고, 페이지 또한 batch remapping을 사용하여 system call을 줄인다.

## 3. 디자인

### 3.1. small size bin 분할

크기가 작은 할당 요청이 들어오는 경우, 할당 사이즈 혹은 할당 타입에 따라 bin을 분할하는것은 이전부터 많이 사용되었던 방법이다. 같은 컨텍스트를 가지는 객체들이 모일 수 있기 때문에, 메모리 오버헤드를 발생시키는 segmentation을 완화할 수 있다는 장점이 있어 많이 사용되고 있다. 하지만 기존의 size 및 할당 타입에 따른 bin의 분할은, 할당 집약적인 프로그램에서는 큰 효과를 발휘하지 못한다는 것을 확인하였다. 따라서 FRRmalloc은 기본적인 size를 통한 할당에, 새로운 컨텍스트를 추가하여 bin을 더욱 세분화하여 분할한다. bin을 세분화할 수 있는 여러 컨텍스트(Stack Pointer, Stack depth, 할당 주소 등) 중, Stack Pointer(SP)와 할당 주소(call site)가 유의미한 컨텍스트를 가진다는 것을 확인하여, size+SP+callsite를 결합하여bin을세분화한다.이때 callsite를 얻는 방법으로는 인라인 함수 __builtin_return_address(depth)를 사용하였다.

```
struct bin_t bins[BIN_COUNT]; // Split bins into 45 sizes
```

(그림2) Size별 bin 분할

```
callsite0 = __builtin_return_address(0);
callsite1 = __builtin_return_address(1);
callsitexor = (uint64_t)callsite0 ^ (uint64_t)callsite1;
callsitexor = (uint64_t)callsitexor ^ (uint64_t)sp;
```

(그림3) callsite +SP 컨텍스트 반영

### 3.2. Page remapping

가상 주소의 재사용을 위해, FFRmalloc은 batch 페이지 단위로 remap을 진행한다. batch 페이지 단위란, remap을 진행할때 얼마나 많은 페이지를 한번에 remap하는지를 의미한다. 이 batch 크기가 크다면 system call이 줄어 성능 오버헤드 이점이 있지만, remap 이후 실질적인 사용 가능한 메모리의 비율이 시간에 따라 다소 떨어질 수 있다는 단점이 있다. 반대의 경우에도, batch 사이즈가 작다면 메모리 재사용 비율은 높아질 수 있지만, 잦은 system call 호출로 인하여 성능 오버헤드가 높을 수 있다는 단점이 있다. 따라서 FFR의 초기 모델은 두 사이를 절충하여 remap batch 사이즈를 64로 선택하였다.

```
mremap(remappages->start,0,BATCHREMAP*4096,
       MREMAP_MAYMOVE|MREMAP_FIXED,
       remapblock->remappages[0]->start);
```

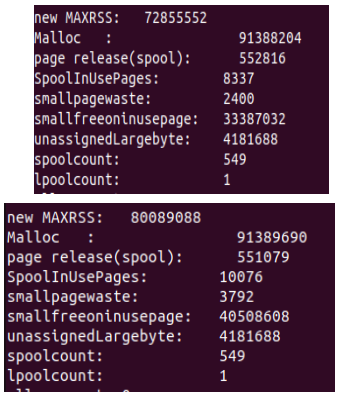
(그림4) batch remap 사용 예시

## 4. 평가

논문에서 다루었던 FFmalloc과의 비교를 통하여 FRRmalloc에 대한 평가를 진행하였다. 메모리 오버헤드 측면에서의 비교를 위해, 할당 집약적인 벤치마크인 Cfrac을 사용하여 두 할당자의 메모리 오버헤드를 비교하였다. 200000개 이상의 malloc을 호출하는 충분히 큰 인자로 벤치마크를 테스트하였고, 전체 프로그램 런타임의 25% 시점에 remap을 진행하였다.

### 4.1. bin 분할

기존 size로만 이루어졌던 bin 분할에서, FFRmalloc은 SP와 callsite라는 컨텍스트를 추가하였다. 이에 따라 할당 시 bin에 같은 컨텍스트를 가지는 객체들이 더 잘 모아지는 효과를 기대할 수 있다. 4.1평가에서는, 추가된 컨텍스트가 0(없음),3,5,10으로 나누어서 메모리 사용량을 조사하였고, 그 결과 0~10까지의 빈 분할은 분할수에 비례하여 메모리 사용량이 감소하는 것을 확인하였다.




(그림4,5) callsite +SP 컨텍스트 반영 후 최대 메모리 사용량
(좌)추가 분할 10 메모리 최대 사용량 :72855552byte
(우)추가 분할 0 메모리 최대 사용량 :80089088byte

### 4.2.Remap 메모리 오버헤드

| | Total malloc | Maxrss malloc | OTA page alloc | Remappage alloc | Page unmap | Remap start | Rss page (overhead) |
|---|---|---|---|---|---|---|---|
| No remap | 20313301 | 20115466 | 103425 | 0 | 98398 | | 5027 |
| Remap | | 20098981 | 75997 | 38748 | 74687 | 5115466 | 1310 |
| No remap | 44384719 | 44376260 | 266241 | 0 | 259014 | | 7227 |
| Remap | | 44383364 | 150251 | 128036 | 144358 | 11096179 | 5893 |

[표1]Remap에 따른 메모리 오버헤드

[표1]은 두 경우의 충분히 큰 할당(20313301회,44384719회)에서, remapping의 활용 유무에 따른 결과를 정리한 것이다. 두 경우 모두 maxrss의 최종 갱신은 프로그램이 종료되기 직전 시점에 maxrss가 갱신된 것을 확인할 수 있다. 또한 메모리 오버헤드의 경우, 두 경우 각각 73,18%가 감소된 것을 확인하였다. 하지만 이는 remap을 위해 추가된 별도의 metadata의 메모리 오버헤드를 고려하고 있지 않기 때문에, 추가된 metadata로 인한 메모리 오버헤드 프로파일링이 필요하다. 또한 이러한 metadata의 효율적 활용 및 관리방법에 대한 연구가 필요하다.

### 5. 결론

UAF 방지를 위한 방법은 지금까지도 새로운 방식을 찾는 연구가 활발히 이루어지고 있고, 이미 존재하던 방식에 대한 개선 또한 활발히 이루어지고 있다.이에 대한 해결책으로 FFRmalloc에서는 기존의 One-Time-Allocation과 page remapping 방식을 기반으로 기존 방식이 가진 장점을 유지하고 단점을 완화시키는 새로운 할당자를 구현하였다. 기존의 방식 이외에도 새로운 컨텍스트 사용, batch system call(remap) 등을 추가하여 기존의 문제점을 개선하고 UAF를 효율적으로 방지할 수 있는 메모리 할당자를 제시한다.

이 논문은 과학기술정보통신부의 재원으로 정보통신기획평가원(No.2020-0-01840, 스마트폰의 내부데이터 접근 및 보호 기술 분석)과 한국연구재단(No. NRF-2022R1A4A1032361, Processing-in-Memory 보안 기술 개발)의 지원을 받아 수행된 연구임

### 참고문헌

[1] CWE Top 25(2023)
https://cwe.mitre.org/data/definitions/1425.html

[2] Farkhani, Reza Mirzazade, Mansour Ahmadi, and Long Lu. "{PTAuth}: Temporal memory safety via robust points-to authentication." 30th USENIX Security Symposium (USENIX Security 21). 2021.

[3] Nagarakatte, Santosh, et al. "CETS: compiler enforced temporal safety for C." Proceedings of the 2010 international symposium on Memory management. 2010.

[4] Simpson, Matthew S., and Rajeev K. Barua. "MemSafe: ensuring the spatial and temporal memory safety of C at runtime." Software: Practice and Experience 43.1 (2013): 93-128.

[5] Van Der Kouwe, Erik, Vinod Nigade, and Cristiano Giuffrida. "Dangsan: Scalable use-after-free detection." Proceedings of the Twelfth European Conference on Computer Systems. 2017.

[6] Younan, Yves. "FreeSentry: protecting against use-after-free vulnerabilities due to dangling pointers." NDSS. 2015.

[7] Sam Ainsworth and Timothy M Jones. 2020. MarkUs: Drop-in use-after-free prevention for low-level languages. In Proceedings of the 41st IEEE Symposium on Security and Privacy (Oakland). San Francisco, CA.

[8] Wickman, Brian, et al. "Preventing {Use-After-Free} Attacks with Fast Forward Allocation." 30th USENIX Security Symposium (USENIX Security 21). 2021.

[9] Li, Xiujun, et al. "Oscar: Object-semantics aligned pre-training for vision-language tasks." Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part XXX 16. Springer International Publishing, 2020.