보안성 및 호환성을 위한 Backward-edge Control-flow Integrity 전용 예외 처리기 설계 및 구현*

진홍주*, 김선권**, 이동훈*

*고려대학교 정보보호대학원 정보보호학과

**고려대학교 일반대학원 정보보안학과

Design and Implementation of Exception Handler Dedicated to Backward-edge Control-flow Integrity for Security and Compatibility

Hongjoo Jin*, Seon Kwon Kim**, Dong Hoon Lee*

*Graduate School of Information Security, Korea University.

**Graduate School of Cyber Security, Korea University.

요 약

애플리케이션의 제어 흐름을 가로채기 위해서 주로 사용되는 코드 포인터를 조작 공격은 공격자가 시스템에 악영향을 주는 코드를 실행할 수 있게 한다. 특히, ROP 공격을 활용한 실제 시스템 해킹에는 Backward-edge에 대한 익스플로잇이 빈번하게 사용된다. Backward-edge 보호 기법들은 주로 함수 블록의 프롤로그와 에필로그에 Code Instrumentation 방식으로 구현된다. Instrumented Code는 해당 함수의 리턴 주소를 암호화/복호화 하거나 별도의 메모리 공간에 리턴 주소를 저장하는 명령어로 구성된다. 프로그램의 예외 처리를 위해 사용되는 라이브러리 함수가 호출될 때 함수 프롤로그와 에필로그의 불일치 현상이 발생하며, Instrumented Code는 이러한 불일치를 조정하지 못해 프로그램 충돌을 일으킨다. 본 논문에서는 Backward-edge CFI 기법이 적용된 프로그램의 예외 처리 과정에서 보안성, 호환성 및 효율성을 모두 고려한 향상된 예외 처리기를 제안한다. 또한, Backward-edge CFI와 함께 사용 가능한 실용적이고 효율적인 C/C++의 예외 처리기의 프로토타입을 구현했다. SPEC CPU2017 벤치마크 프로그램에 제안 프로토타입을 적용하여 호환성과 성능을 측정하였으며, 호환성 문제가 없이 1% 미만의 성능 오버헤드를 부여함을 확인했다.

I. Introduction

소프트웨어 버그를 악용하여 코드 포인터(Code Pointer)를 조작하는 공격은 공격자가 애플리케이션의 제어 흐름을 가로챌 수 있으며, 프로그램의 제어권을 획득한 공격자는 시스템에 악영향을 줄 수 있는 코드를 실행할 수 있다[1]. 코드 포인터는 Function Pointer, Virtual Table Pointer와 같은 Forward-edge와 Return Address의 Backward-edge로 나눌 수 있다. 특히, Backward-edge에 대한 공격은 Return-oriented Programming (ROP)[2, 3]와 같은 실제 시스템 해킹을 위한 익스플로잇(Exploit)에 빈번하게 사용된다.

리턴 주소(Return Address)의 무결성을 보장하는 보호 기법을 Backward-edge Control-flow Integrity (Backward-edge CFI)라 하며 보안성 및 효율성 측면에서 다양한 기법들이 제안되었다. Stack Canary[4], Shadow Stack[1, 5], Stack Isolation[6, 7], Dynamic Remote Attestation[8] 와 같은 기법들은 스택 메모리에 저장되는 리턴 주소를 보호하기 위해 기존 소스 코드에 보호 기법 코드를 Instrumentation 한다. Instrumented Code는 별도의 공간에 리턴 주소를 저장하는 명령어 및 저장 공간을 관리하는 포인터를 조정하는 명령어 등으로 구성된다. 주로 프로그램 함수의 프롤로그(Prologue) 및 에필로그(Epilogue)의 시작 부분에 Instrumented Code가 삽입되며, 함수 프롤로그와 에필로그의 쌍이 맞지 않으면 호환성 문제를 유발한다. 즉, Instrumented Code의 프롤로그가 실행되었으면 반드시 에필로그도 실행해야 프로그램의 기능 및 보안상 문제가 없다. 하지만, 함수 간 분기를 지원하는

* 이 성과는 2021년도 과학기술정보통신부의 재원으로 한국연구재단의 지원을 받아 수행된 연구임 (No.NRF-2021R1A2C2014428).

setjmp() 및 *longjmp()* 라이브러리 함수 호출과 C++의 예외 처리기로 사용되는 *throw()* 및 *catch()*의 사용은 함수 프롤로그와 에필로그 쌍을 파괴하는 특수한 형태의 제어 흐름을 실행한다. 라이브러리 함수 사용의 경우 기본적으로 제공하는 Stack Unrolling 기능을 활용하여 함수 프롤로그와 에필로그의 비대칭을 풀어준다. 하지만, Instrumented Code의 경우 스택 메모리의 특수한 움직임과 관련된 예외 처리기가 고려되지 않으면 프로그램 충돌 등의 호환성 문제가 발생한다. 이 경우 두 가지 방법으로 해결이 가능하며, (i) *setjmp()*, *longjmp()* 등의 예외 처리기가 있는 함수 블록에만 Code Instrumentation을 적용하지 않는 방법, (ii) Instrumented Code를 위한 예외 처리기를 함께 작성하는 방법이다. 전자의 경우 성능상의 이점이 있지만, 보호 기법이 적용되지 않는 코드 블록이 생기기 때문에 미탐(False Negative)이 발생하며, 후자의 경우 처리 방법에 따라 심각한 성능 오버헤드가 발생한다.

본 논문에서는 미탐을 발생시키지 않으면서 효율적인 Backward-edge 전용 예외 처리기를 제안한다. Code Instrumentation 과정에서 모든 스택 비대칭 유발 함수(e.g., *setjmp()*, *throw()*)를 찾아 제안 기법을 삽입하며, 기법은 효율적인 해시 테이블 구조 설계 및 *backtrace()* 시스템 콜 활용 등으로 효율적으로 발생 예외를 처리한다. 구현한 프로토타입을 Backward-edge Control-flow Integrity가 Instrumentation 된 SPEC CPU2017 벤치마크 프로그램에 적용하여 호환성 및 성능을 평가하였으며, 프로그램 충돌 없이 평균 1% 미만의 성능 오버헤드가 추가됨을 확인하였다.

II. Background

2.1 Backward-edge CFI

리턴 주소를 보호하기 위한 Backward-edge CFI 기법은 다양한 형태로 개발 및 구현된다. 리턴 주소가 스택 메모리에 저장될 때 암호화하고 함수 리턴을 실행할 때 복호화하는 암호학적 기법이 있으며, 별도의 안전한 공간에 리턴 주소를 저장하여 보호하는 Shadow Stack 기반 기법이 있다. 이들 기법 모두 함수 블록의 프롤로그에 암호화 또는 안전한 영역에 리턴 주소를 저장하는 명령어들이 필요하고, 에필로그에 복호화 또는 함수 복귀를 위한 리턴 주소를 로드하는 명령어가 필요하다. 따라서, 보호 기법 적용을 위해서 함수 블록의 프롤로그와 에필로그에 Code Instrumentation이 필요하며, 필요한 명령어를 추가하여 보안 기능을 수행한다(Fig. 1).

Prologue

```
0   # BB#0:
1   movq    %gs:0, %r10
2   movq    (%rsp), %r11
3   movq    %r11, (%r10)
4   addq    $8, %r10
5   movq    %r10, %gs:0

6   pushq   %rbp
7   pushq   %r14
8   subq    $16, %rsp

9   movq    %rbx, %rdi
10  callq   printf

11  addq    $16, %rsp
12  popq    %r14
13  popq    %rbp

14  movq    %gs:0, %r10
15  subq    $8, %r10
16  movq    (%r10), %r11
17  movq    %r10, %gs:0
18  movq    %r11, (%rsp)
19  retq
```

Epilogue

Fig. 1. 실제 함수 블록의 프롤로그 및 에필로그. line 1-5, line 14-18은 Instrumented Code.

2.2 예외 처리 라이브러리 함수

예외 처리 라이브러리 함수는 프로그램 실행 중 코드 작성자가 설정한 예외에 도달하면 해당 지점에서의 실행을 멈추고 정상적인 실행을 이어가기 위해 사용한다. 타겟 라이브러리 함수는 *setjmp()/longjmp()* 및 *throw()/catch()* 이며, 이들 함수는 함수 블록에 진입하여 예외 발생 시 함수 에필로그를 수행하지 않고 스택 프레임을 정리한다(Fig. 2). 이러한 동작은 Instrumented Code의 스택 불일치를 유발하며, 이 불일치가 제대로 처리되지 않는 경우 프로그램 충돌을 유발한다.

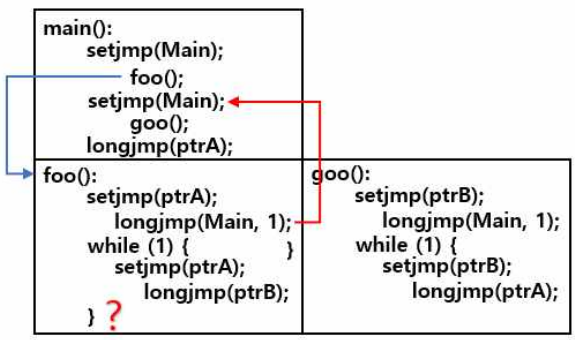

Fig. 2. *setjmp()/longjmp()* 실행 예. *foo()* 함수에 진입한 실행이 *longjmp()*로 인해 *main()*함수로 분기하며, *foo()* 함수의 에필로그 불일치를 유발함.

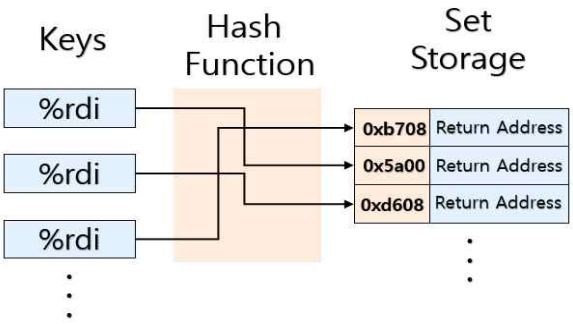

Fig. 3. setjmp()/longjmp() 스택 메모리 예외 처리를 위한 해시 테이블 구조. 해시 함수는 %rdi 값을 입력으로 받아 hash += ((%rdi >> 4) & 0xff) 식을 수행하여 계산한다.

III. Exception Handler for Backward-edge CFI

Backward-edge CFI 전용 예외처리기는 *setjmp()/longjmp()*, *throw()/catch()*에 대한 스택 메모리 추적을 제공한다. *setjmp()/longjmp()*, *throw()/catch()* 각각에 대한 효율적인 추적 기법을 설계하여 Backward-edge CFI 기법에서 발생할 수 있는 미탐을 방지하고, 호환성 문제를 해결한다.

3.1 setjmp() and longjmp()

SPEC CPU2017 벤치마크 프로그램의 x86-64 어셈블리 코드를 분석하여 *setjmp()*와 *longjmp()*가 기계어 코드에서 활용하는 레지스터 및 메모리 저장 명령어를 확인했다. 분석 결과 *setjmp()*를 수행하기 전 %rdi 레지스터에 해당 *setjmp()*에 대응되는 *longjmp()*와의 링크 정보가 저장되는 것을 확인했다. 이를 이용하여 *setjmp()*가 수행되는 함수 블록에서 *%rdi* 값과 해당 블록의 리턴 주소를 해시 테이블의 형태로 작성하도록 기법을 설계했다. Fig. 3의 예로 모든 *setjmp()* 호출 이전의 *%rdi* 값을 해시테이블의 key로 사용하고, 해당 함수 블록의 리턴 주소를 해시 함수를 통과한 해시값과 함께 Set Storage에 저장한다. 이 테이블을 이용해서 함수 블록에 *longjmp()*가 있는 모든 지점에서 *%rdi* 값을 얻어 해시 테이블 서칭을 통해 *longjmp()*가 수행되어 돌아갈 함수 블록의 리턴 주소로 Instrumented Code가 가진 리턴 주소를 변경해준다. 해시 테이블 구현된 프로토타입은 *setjmp()*와 *longjmp()* 사이의 링크 정보와 리턴 주소는 프로그램 실행 및 성능에 영향을 거의 주지 않는 효율적인 테이블 관리 및 검색을 수행한다(Fig. 4).

```
srand((unsigned int)time(NULL));

set_base = rand()%0x7ffb;
set_base = (set_base << 32) | rand() | 0x100000000;
set_ptr = mmap(set_base, setlong_Size, PROT_READ | PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);

__asm__ __volatile__(
"movq %0, %%gs:800₩n₩t"
:
:"r"(set_ptr)
);
```

```
__asm__ __volatile__(
"movq %%rdi, %0₩n₩t"
:"=r"(jmp_Buffer)
:
);
__asm__ __volatile__(
"movq %%gs:0, %0₩n₩t"
:"=r"(ptr_LEAF)
:
);
__asm__ __volatile__(
"movq %%gs:800, %0₩n₩t"
:"=r"(set_ptr)
:
);
set_ptr += ((jmp_Buffer >> 4)  & 0xff);
*set_ptr = ptr_LEAF;
```

```
__asm__ __volatile__(
"movq %%rdi, %0₩n₩t"
:"=r"(jmp_Buffer)
:
);
__asm__ __volatile__(
"movq %%gs:800, %0₩n₩t"
:"=r"(set_ptr)
:
);
set_ptr += ((jmp_Buffer >> 4)  & 0xff);
ptr_LEAF = *set_ptr;
__asm__ __volatile__(   //set_ptr_LEAF
"movq %0, %%gs:0₩n₩t"
:
:"r"(ptr_LEAF)
);
```

Fig. 4. 프로세스 메모리 임의의 위치에 해시 테이블을 생성하는 코드(맨 위)로 큰 메모리 공간에 해시테이블의 위치를 숨김. 또한, setjmp() 호출 전에 %rdi 값을 얻어 해시 테이블의 key로 사용하고, 리턴 주소를 저장함(왼쪽 아래). 마지막으로 longjmp() 호출 시 해시 테이블을 검색하고 저장된 리턴 주소를 Instrumented Code에 전송함(오른쪽 아래).

```
volatile register unsigned long bt_size = 0;
volatile register void* frame_addr[4096];
int i = 0;
size_t backtrace_size;

backtrace_size = backtrace(frame_addr, 4096);
bt_size = (backtrace_size - 4) * 8;
```

Fig. 5. C++ 예외 처리 함수인 throw()/catch()를 처리하는 구현 부분. backtrace() 시스템 콜을 활용 및 가공하여 call depth의 차이를 얻는다.

3.2 throw() and catch()

C++의 예외 처리 라이브러리 함수인 *throw()/catch()*는 *%rdi*와 같은 레지스터에 링크 정보를 저장하지 않는다. 따라서, Instrumented Code가 해시 테이블을 이용하는 기법을 사용할 수 없다. 이를 해결하기 위해 *backtrace()* 시스템 콜을 활용하여 *throw()*가 호출되는 call depth와 *catch()*가 호출되는 call depth의 차이를 구한다. Call depth의 차이 값을 이용해서 *catch()* 함수에 도달했을 때 현재 리턴 주소 값을 구한다. 이후, Instrumented Code가 관리하는 리턴 주소 대신 현재 리턴 주소를 대체하여 호환성 문제를 해결한다.

3.3 Evaluation

Backward-edge CFI 전용 예외 처리기의 프로토타입을 SPEC CPU2017 벤치마크 프로그램으로 평가했다. 벤치마크 프로그램 중 C/C++ 예외 처리 함수를 사용하는 6개 프로그램(500.perlbench_r, 510.parest_r, 511.povray_r 520.omnetpp_r, 523.xalancbmk_r, 526.blender_r)에 간단한 Backward-edge CFI 기법 코드를 Instrumentation 해서 실험했다. 각각의 프로그램에 대한 8번의 실행 결과 호환성 문제는 발생하지 않았고, 평균 1% 미만의 낮은 성능 오버헤드를 보였다.

IV. Conclusion

본 논문에서는 Backward-edge CFI 기법의 호환성 및 효율성 문제를 해결하기 위한 전용 예외 처리기를 제안한다. 해시 테이블 및 *backtrace()* 시스템 콜을 활용한 효율적인 예외 처리 기법을 설계하고, 실제 프로세스 메모리에서 작동 가능한 프로토타입을 구현했다. 기법의 프로토타입을 SPEC CPU2017 벤치마크 프로그램에 적용하여 실제 프로그램에서 호환성 문제를 해결하고, 프로그램 성능에 거의 영향을 미치지 않는 효율적인 기법임을 보였다.

[참고문헌]

[1] Burow, N., Zhang, X., & Payer, M. (2019, May). SoK: Shining light on shadow stacks. In 2019 IEEE Symposium on Security and Privacy (SP) (pp. 985-999). IEEE.

[2] Prandini, M., & Ramilli, M. (2012). Return-oriented programming. IEEE Security & Privacy, 10(6), 84-87.

[3] Checkoway, S., Davi, L., Dmitrienko, A., Sadeghi, A. R., Shacham, H., & Winandy, M. (2010, October). Return-oriented programming without returns. In Proceedings of the 17th ACM conference on Computer and communications security (pp. 559-572).

[4] Cowan, C., Pu, C., Maier, D., Walpole, J., Bakke, P., Beattie, S., ... & Hinton, H. (1998, January). Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks. In USENIX security symposium (Vol. 98, pp. 63-78).

[5] Dang, T. H., Maniatis, P., & Wagner, D. (2015, April). The performance cost of shadow stacks and stack canaries. In Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security (pp. 555-566).

[6] Kuznetzov, V., Szekeres, L., Payer, M., Candea, G., Sekar, R., & Song, D. (2018). Code-pointer integrity. In The Continuing Arms Race: Code-Reuse Attacks and Defenses (pp. 81-116).

[7] Zieris, P., & Horsch, J. (2018, May). A leak-resilient dual stack scheme for backward-edge control-flow integrity. In Proceedings of the 2018 on Asia Conference on Computer and Communications Security (pp. 369-380).

[8] Kil, C., Sezer, E. C., Azab, A. M., Ning, P., & Zhang, X. (2009, June). Remote attestation to dynamic system properties: Towards providing complete system integrity evidence. In 2009 IEEE/IFIP International Conference on Dependable Systems & Networks (pp. 115-124). IEEE.