함수 호출 규약

SWING 32기 이유진

서론

지난 한 주간 처음으로 해킹 분야에 입문하게 되었다. 그 중 리버싱을 배우며 IDA를 활용한 실습도 진행했지만, 글을 읽는 도중에도 이해되지 않는 부분이 많았다. 특히 의문이 든 부분은 '함수 호출 규약' 부분이었다. 전에 리버싱을 배운 적이 없어서 그런지, 처음 듣는 단어에 상당히 혼란스러웠었다. 그래서 나는 여기서 제대로 리버싱을 잡고 공부하지 않으면 이후 리버싱 스터디을 진행함에 있어 큰 걸림돌이 될 거라는 생각이 들어 이에 대해 보다 심화된 이해가 필요함을 느꼈다.

그렇기에 이번 SSR에서는 리버싱을 호출 규약을 중점으로 그와 관련된 개념들을 집중적으로 다뤄보고자 한다.

본론

[함수 호출 규약]

리버싱 실습 문제를 풀면서 가장 의문이 들었던 부분이 있다. 바로 동적 실습 부분 중, rsp+0x20에 대한 부분이었다. 왜 rsp +0x20인지부터가 잘 이해되지 않아 어떻게 이해해야할지 이 강좌의 QnA를 살펴보던 중, 다음과 같은 질문을 발견하였다.

`rsp+0×20에 저장된 값을 ecx에 옮깁니다. 이는 함수의 첫 번째 인자를 설정하는 것입니다.
Sleep함수를 호출합니다. ecx가 0x3e8이므로, Sleep(1000)이 실행되어 1초간 실행이 멈춥니다.

.text:00000001400010EC mov ecx, [rsp+38h+dwMilliseconds] ; dwMilliseconds
.text:00000001400010F0 call cs:Sleep
`

함수 인자가 rdi→rsi→rdx→rcx(ecx)→r8.... 순서로 들어가는 것으로 알고있는데
Sleep() 의 인자로 왜 edi도 아니고 ecx가 들어간건가요?

내가 이해하고 있던 것은 단지 mov, call 이정도였기에 함수 인자가 애초에 왜 저 순서대로 들어간다는 건지 잘 몰랐다. 아래 답변들을 보니 "리눅스와 윈도우의 호출규약이 다르다"라는 말을 하고 있기에 호출 규약이 뭐지? 라는 의문이 들어 이를 제일 먼저 공부해보기로 정했다.

일단 함수 호출 규약이란 뭘까?

간단히 생각해 '함수가 호출될 때 적용되는 규약'이라고 정의하면 될 것 같다. 이때 함수가 호출되면서 정해야 할 부분은 스택을 어떻게 쌓고, 이를 레지스터로 활용할지 같은 게 있다.

일단 어떤 비트이든지 상관없이, 함수 호출 규약은 다음과 같은 순서를 꼭 따라준다고 한다.

1. 모든 인자들은 4바이트(혹은 8바이트)로 확장되어 메모리에 삽입된다.

-> 이 위치들은 보통 스택 상 메모리이지만 레지스터로 활용될 수도 있다. 이는 호출 규약에 따른다. (뒤 shadow space와 관련된 내용)

2. 프로그램이 실행되면 호출된 함수의 주소로 점프한다.

3. 함수 프롤로그: 함수 안에서 함수 보존 레지스터들이 스택에 저장된다. (컴파일러가 알아서 작성함) -> 보존 레지스터가 정확히 무엇인지는 아직 모르지만, 내 추측으로는 리버싱 어셈블리 배울 때 나온 rsp, rbp같은 것들이 아닐까 한다.

4. 함수 코드 실행: 함수 코드대로 실행되고 return 값은 eax에 저장된다.

5. 함수 에필로그: 3번에서 스택에 저장한 보존 레지스터들을 다시 꺼내어 복구함(얘도 컴파일러

가 함)

6. 스택 비우기: 스택에서 함수 인자들이 제거되는 과정. (3번 과정에서 스택에 넣어졌다 보면 될 것 같다)

-> 피호 출자 함수 내에서 실행되거나 호출자에 의해 실행된다. 이는 함수 호출 규약에 따른다.

그럼 이제 함수 호출 규약에는 정확히 무엇이 있는지를 알아보자.

함수 호출 규약=Cailling Convention

크게 함수 호출 규약은 word(=한 번에 처리가능한 비트 크기)에 따라 다르다고 한다.

Segment word size	Calling Convention	Parameters in registers	Parameter order on stack	Stack cleanup by
32bits	__cdecl		right to left	Caller
	__stdcall			Function
	__fastcall	ecx, edx		Function
	__thiscall	ecx		Function
64bits	-	rcx, rdx, r8, r9 (xmm0, xmm1, xmm2, xmm3)		Caller

1. 32bits

우선 32bits부터 살펴보기로 했다. 32bits의 경우 호출 규약이 __cdecl, __stdcall, __fastcall, __thiscall 이렇게 4가지가 있다고 한다. (처음에는 32bits가 규약이고 이 4가지는 함수인 줄 알았다..)

규약	뜻	설명
`__cdecl`	**C Declaration**	C 언어 기본 호출 방식
`__stdcall`	**Standard Call**	WinAPI 등에서 쓰는 표준 호출 방식
`__fastcall`	**Fast Call**	레지스터를 활용해서 빠르게 호출
`__thiscall`	**This Call**	C++ 클래스의 멤버 함수 호출용 (`this` 사용)

1) __cdecl

- 인자(=파라미터)들은 오른쪽에서 왼쪽으로 스택에 쌓임. (레지스터일 경우는 제외)

ex) sum(a, b) 라 쓰면 b가 먼저 스택에 쌓임 (맨 위가 가장 첫번째 파라미터가 된다.)

- 스택이 호출자에 의해 정리됨

* 호출자 = Caller = 함수를 부르는 쪽의 코드

: 함수 내부는 Callee라고 불리며, 얘는 안에 가변인자(ex: printf)가 있으면 자기 인자가 몇 개인지 파악하지 못한다. 그때 Caller를 사용해 인자 개수를 파악하면 스택을 정리할 수 있다.

ex: add esp, 8

2) __stdcall

- 인자는 오른쪽에서 왼쪽으로 스택에 쌓임.

- 스택이 피호출자에 의해 정리됨

* 피호출자 = Callee (즉, 함수 내부) = 호출된 함수 그 자체

: 인자 개수가 항상 정해져있을 때 사용한다. 이를테면 WinAPI같이 항상 매개변수가 고정되어 있는 경우. 피호출자로 스택을 정리할 수 있다.

* WinAPI란?

WindowsApplicationProgrammingInterface의 약자. 윈도우에서 제공해주는 함수의 집약체라고 보면 된다. 메시지 띄우기, 메모리 할당, 프로세스 종료 같은 것들도 WinAPI의 부분 함수이다.

ex: ret 8 = 스택에서 8바이트 제거 후 리턴

* 만약 ret 만 쓰면 아무것도 제거 안 한다는 의미이니 호출자가 직접 add 같은 문구를 써서 따로 정리해줘야한다. 즉, 우리는 어셈블리 ret을 통해 이게 어떤 호출 규약을 사용하는지 구분할 수 있다.

3) __fastcall

- 인자가 오른쪽 왼쪽으로 스택에 쌓이는 방식x

- 인자들이 4바이트 보다 작을 시, 스택이 아닌 레지스터에 저장된다.

인자 저장 레지스터에는 ecx, edx가 있다.

*다음 레지스터들은 x86(=32bits)의 레지스터들이다.

eax: 함수 반환값, 연산 결과 저장 레지스터

ecx(counter): fastcall의 첫번째 인자

edx(data): fastcall의 두번째 인자

```
//어셈블리로 보면
mov ecx, a
mov edx, b
push c          //3번째 인자만 스택에 넣음
call sum
```

즉 sum(2, 3, 4) 이 있다면 fastcall의 경우 2가 ecx, 3은 edx에 저장된다. 인자에 대한 레지스터는 2개 밖에 없기 때문에 스택은 어쩔 수 없이 4에 들어가게 된다. 즉, 4바이트 이하인 하나으 인자는 무조건 한 레지스터 공간을 차지하고, 그렇기 때문에 3번째 인자부터는 스택에 저장된다는 소리인 듯 하다

- 레지스터는 CPU내부에 있기 때문에, 더 효율적인 저장 방법이다. = 함수 호출 비용을 줄인다.

- 인자 2개의 경우처럼 스택에 쌓인 게 없으면 스택 정리를 하지 않고 필요하면 피호출자로 정리

4) __thiscall

- this 포인터는 c++에서 나온 개념이다. 즉 이 호출규약은 c++을 위해 존재함. 코드에 나오는 this 포인터는 ecx에, 나머지 인자들은 오른쪽에서 왼쪽 순서대로 스택에 저장됨.

2. 64 bits

이번에는 64비트에서는 호출 규약이 어떻게 적용되는지 알아보자. 32 bits일 때와 다르게 64 bits에서는 함수 호출 규약이 한가지이다. Microsoft x64 calling convention라고 한다. (윈도우 기준)

또한 64 bits 는 전부 다 호출자가 스택을 정리한다.

매개 변수 유형	다섯 번째 이상	네 번째	세 번째	second	가장 왼쪽
부동 소수점	스택	XMM3	XMM2	XMM1	XMM0
integer	스택	R9	R8	RDX	RCX
집계(8, 16, 32 또는 64비트) 및 `__m64`	스택	R9	R8	RDX	RCX
기타 집계, 포인터로	스택	R9	R8	RDX	RCX
`__m128` , 포인터로	스택	R9	R8	RDX	RCX

위 그림을 보면 알 수 있듯이, 함수 인자들은 왼쪽에서 부터 순서대로 RCX, RDX, R8, R9까지 각 레지스터에 저장되고 그 후부터는 스택에 쌓임을 확인할 수 있다.

여기서 잠깐 앞 드림핵에서의 질문을 떠올렸다. 분명 거기있던 질문은, 자기가 아는 호출 규약대로라면 64비트일 경우 함수 인자가 RDI부터 시작해 들어가지 않냐는 질문이었다. 이제 질문이 무엇을 의미하는지는 알았지만, 왜 방식이 다른지는 따로 알아봐야했다. 그에 대한 답변을 살펴보면 os가 윈도우인 것과 리눅스인 것이 이유라고 한다. 그래서 윈도우와 리눅스(64bits)의 호출 규약 비교 자료를 찾아봤다.

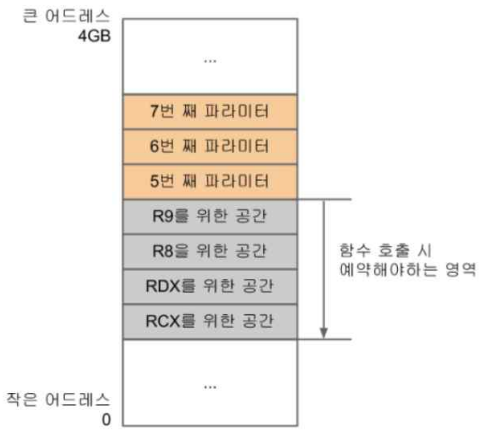
리눅스는 64비트 모드에서 인자를 전달할 때 윈도우보다 레지스터를 더 많이 사용한다. RDI, RSI, RDX, RCX, R8, R9의 순서로 총 6개의 레지스터를 사용한 후 나머지는 스택에 저장하는 방식을 채택했다. 다만 실수 타입의 경우, 인자들은 XMM0 ~ XMM7까지 8개를 순서대로 사용하고 그 이상이면 스택으로 전달한다.

* RCX와 ECX 차이 정리

- 앞 32 bits에서는 분명 ECX, EDX가 나왔는데, 64 bits에서는 갑자기 RCX, RDX로 바뀌어 있어 찾아보았다. 요약하자면 각 비트별 사용하는 레지스터가 다르다. 단순히 이름뿐만아니라 크기도 ECX, EDX가 4바이트라면 RCX, RDX는 8바이트인 점을 주의해야 할 것 같다.

즉 드림핵에서는 리눅스가 아닌 윈도우의 64 bits에서 학습하고 있기 때문에 4개 레지스터 방식으로 보는 게 맞다. 즉, 질문에 대한 답변은 이렇게 정리할 수 있을 것 같다. 본 강좌는 윈도우를 중심으로 하고 있다. 그렇기 때문에 레지스터 저장 순서는 rdi->rsi->rdx->rcx(ecx)->r8 같은 방식이 아니라, RCX, RDX, R8, R9 이다.

추가로 둘은 단순히 레지스터 개수만 다른 게 아니라, 레지스터를 채우는 방식 자체가 다르다고 한다. 이를 shadow space 혹은 home space라고 부른다.



지금 당장 나는 어셈블리어로 코딩을 할 게 아니라 필요없을 수도 있지만, 그래도 이해에 도움이 될 것 같아 정리해보고자 한다.

[shadow space]

윈도우에서는 인자를 레지스터에 저장을 해두는 동시에, 스택에도 레지스터를 위한 공간을 할당해준다.

윈도우의 경우 -> 스택에서 채우면서 미리 4개의 레지스터를 위한 공간을 할당(예약) 해 둠. 이 공간이 바로 shadow space이다. 여긴 64 bits이니 각 인자 1개당 8바이트로 잡아 8*4 = 32 bytes 크기의 레지스터 여유 공간이 필요하다. 거기에 8 bytes 크기의 스택 정렬을 위한 공간까지 필요해 총 40 bytes 가 필요하다 생각하면 된다.

ex: 내가 리눅스 과제에서 한 어셈블리 코드를 살펴보자. 위를 보면 sub rsp, 30h라고 적혀있다.

```
; Attributes: bp-based frame

; int __fastcall main(int argc, const char **argv, const char **envp)
public main
main proc near

dwMilliseconds= dword ptr -4

push    rbp
mov     rbp, rsp
sub     rsp, 30h
call    __main
mov     [rbp+dwMilliseconds], 1F4h
mov     eax, [rbp+dwMilliseconds]
mov     ecx, eax        ; dwMilliseconds
mov     rax, cs:__imp_Sleep
call    rax ; __imp_Sleep
call    hello
call    cal
mov     eax, 0
add     rsp, 30h
pop     rbp
retn
main endp
```

방금 배웠던 것을 적용해서, 그럼 이게 바로 shadow space? 라는 궁금증이 생겨 찾아봤다. 일단 30h는 10진수가 아니라 16진수이니 변환해보면 48바이트이다. 앞에서는 분명 32바이트+8바이트 러 기억하고 있다. 그럼 나머지 8바이트는 과연 무엇일까? 이는 dwMilliseconds와 같은 지역 변수를 위해 따로 할당해준 공간이라고 한다. 즉, 이건 shadow space에 정렬과 지역변수를 위한 공간까지 합쳐 할당한 것이라고 추측할 수 있다고 한다. 찾아보니 보통 20h(=32 bytes), 28h(=40 bytes)가 대표적인 shadow space 부여 크기라고 한다.

리눅스의 경우 -> 스택을 여유공간 없이 채워나감 = 인자 하나를 넣고 4바이트 까지 채우려 하지말고 끊기면 거기서 공간 멈춤. 스택에 레지스터를 위한 여유 공간 만들지 않음.

그렇다면 왜 굳이 윈도우에서는 레지스터를 위한 스택 공간을 따로 만들어두는 걸까?

만약 저 공간을 할당해두지 않으면 파라미터가 잘못 전달되어 함수 호출 자체가 불가 하다고 한다. 또한 비록 파라미터가 4개 미만이더라도 함수 호출 시 저 영역은 무조건 할당해야 한다. 이렇게 윈도우는 항상 스택에서 4개 레지스터를 위한 공간을 가진다는 것을 알기때문에, 이후 컴파일 할 때 좀 더 효율적으로 디버깅 할 수 있다는 게 이유였다.

아래는 윈도우와 리눅스의 호출 규약을 간단히 비교 정리한 표이다.

구분	Windows x64	Linux x64 (System V)
레지스터	RCX, RDX, R8, R9	RDI, RSI, RDX, RCX, R8, R9
this 포인터	RCX	RDI
스택 정리	호출자	호출자
인자 순서	오른쪽 → 왼쪽 (논리상)	왼쪽 → 오른쪽

결론

- 리눅스를 처음으로 공부하는 동안 어셈블리나 레지스터 등 낯선 개념들이 많이 등장해 상당히 어려움을 겪었었다. 그런데 이번 SSR과제를 통해 내가 정확히 무엇을 모르고 있었는지를 정확히 알 수 있었다. 어셈블리를 읽을 때 과연 어떤 포인트를 중심으로 보면 되고, 이 규칙들과 코드가 왜 이렇게 쓰이고 있는지 정확한 구조를 알 게 된 것 같았다.

- 함수 호출 규약에 대해 공부하면서 레지스터라는 단어가 참 많이 나온 것 같다. 스택과 레지스터의 주소 부분이 조금 이해가 부족한 것 같아 메모리와 레지스터의 역할 자체가 조금 더 공부가 필요할 것 같다는 생각이 들었다.

또한 X64라는 개념 자체도 아직 나에게 낯선 내용인 듯 하다. 이와 함께 X86 stack frame도 이 SSR 자료 조사를 하면서 종종 등장했던 부분인데, 시간관계상 완전히 이해를 하지 못해 이에 대해서도 추가 공부가 필요할 것 같다고 생각한다.

- 처음하는 SSR이라 주제를 무엇을 선정하고 어떻게 구글링해 자료를 찾을 수 있는지에 대해 많이 삽질 하였지만 동시에 많이 배울 수 있는 경험이었던 것 같다. 정보보안을 학습함에 있어서는 바로 질문하지말고 여러 블로그를 탐방하며 꼭 삽질해봐야지 많이 성장할 수 있다는 것을 깨달을 수 있었다.

[함수 호출 규약]

1. 나라킷. (2020, February 3). 함수 호출 규약 (Calling Convention) 정리. Tistory. https://narakit.tistory.com/145

2. Johnson, A. (n.d.). Understanding cdecl and fastcall. Substack. https://andrewjohnson4.substack.com/p/understanding-cdecl-and-fastcall

3. sundg0162. (2022, July 6). [C/C++] WINAPI란 무엇인가?. Tistory. https://sundg0162.tistory.com/62

4. 까마귀. (2020, October 20). 리눅스 vs 윈도우 운영체제의 함수 호출 방식 차이. Tistory. https://kkamagui.tistory.com/811

5. Stack Overflow. (2015, May 12). What is the "shadow space" in x64 assembly?. https://stackoverflow.com/questions/30190132/what-is-the-shadow-space-in-x64-assembly

6. gpt 표 만들기

OpenAI. (2025, April 2). ChatGPT conversation: Function calling conventions, fastcall, shadow space, and stack alignment[Large language model]. ChatGPT. https://chatgpt.com/c/67e894df-2088-8011-8ee3-1807427f3f7b