

Hotel-California

문제 설명

You know the rules: you can check out any time you like but you can never leave!
(flag in /FLAG.txt)

- hotelcaliforniaquals2019.ooverflow.io 7777

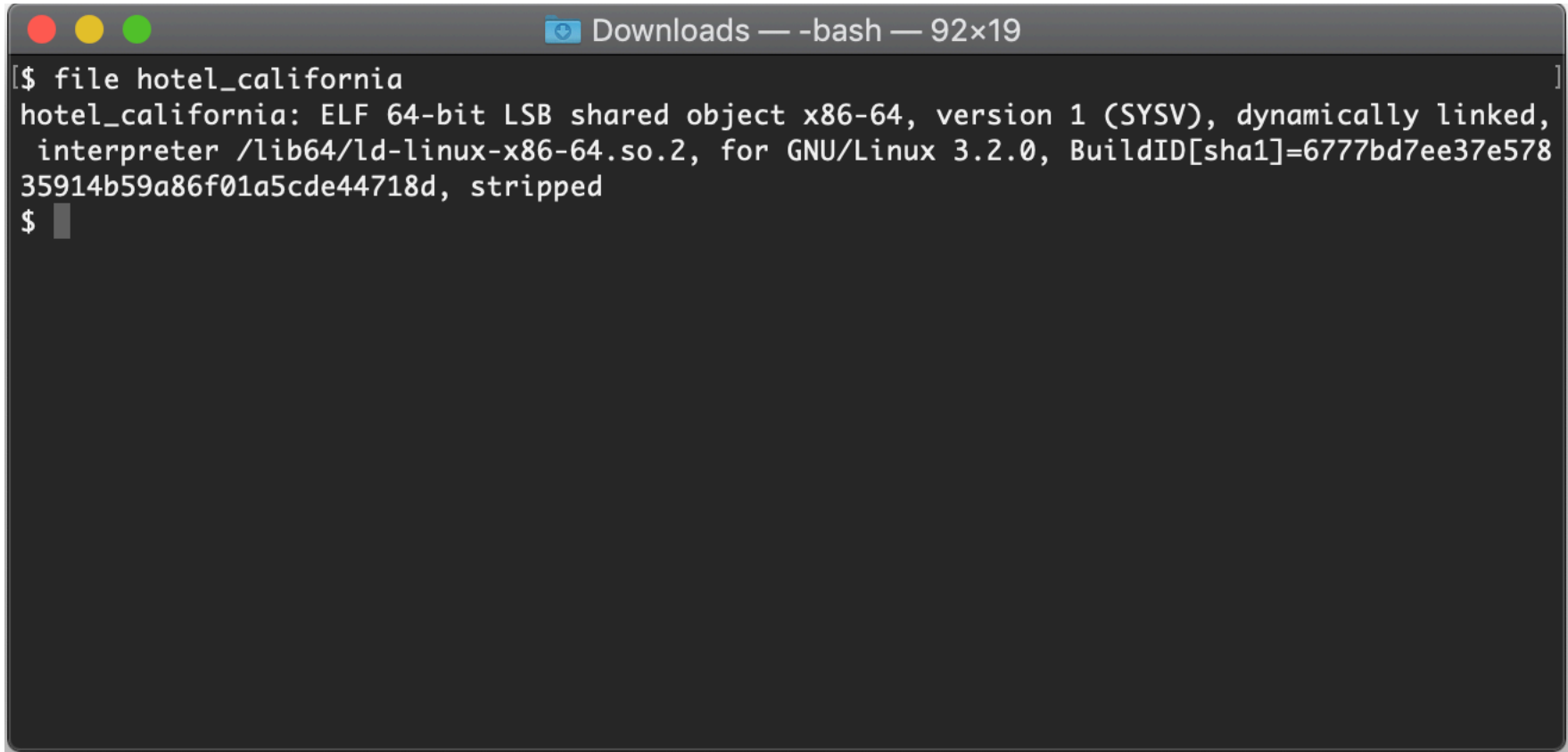
Files:

- hotel_california

- x86-64 shellcode, Intel TSX

주어진 것

ELF 실행 파일

A terminal window with a dark background and light gray text. The window title bar shows 'Downloads — -bash — 92x19'. The terminal content shows a command prompt '\$' followed by 'file hotel_california'. The output is a single line: 'hotel_california: ELF 64-bit LSB shared object x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 3.2.0, BuildID[sha1]=6777bd7ee37e57835914b59a86f01a5cde44718d, stripped'. The prompt '\$' is followed by a cursor.

```
Downloads — -bash — 92x19
[$ file hotel_california]
hotel_california: ELF 64-bit LSB shared object x86-64, version 1 (SYSV), dynamically linked,
  interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 3.2.0, BuildID[sha1]=6777bd7ee37e578
35914b59a86f01a5cde44718d, stripped
$
```

셸코드 (shell-code)

- 취약점을 공격하기 위해 서버에 보내 실행하는 코드
- 어셈블리어로 작성

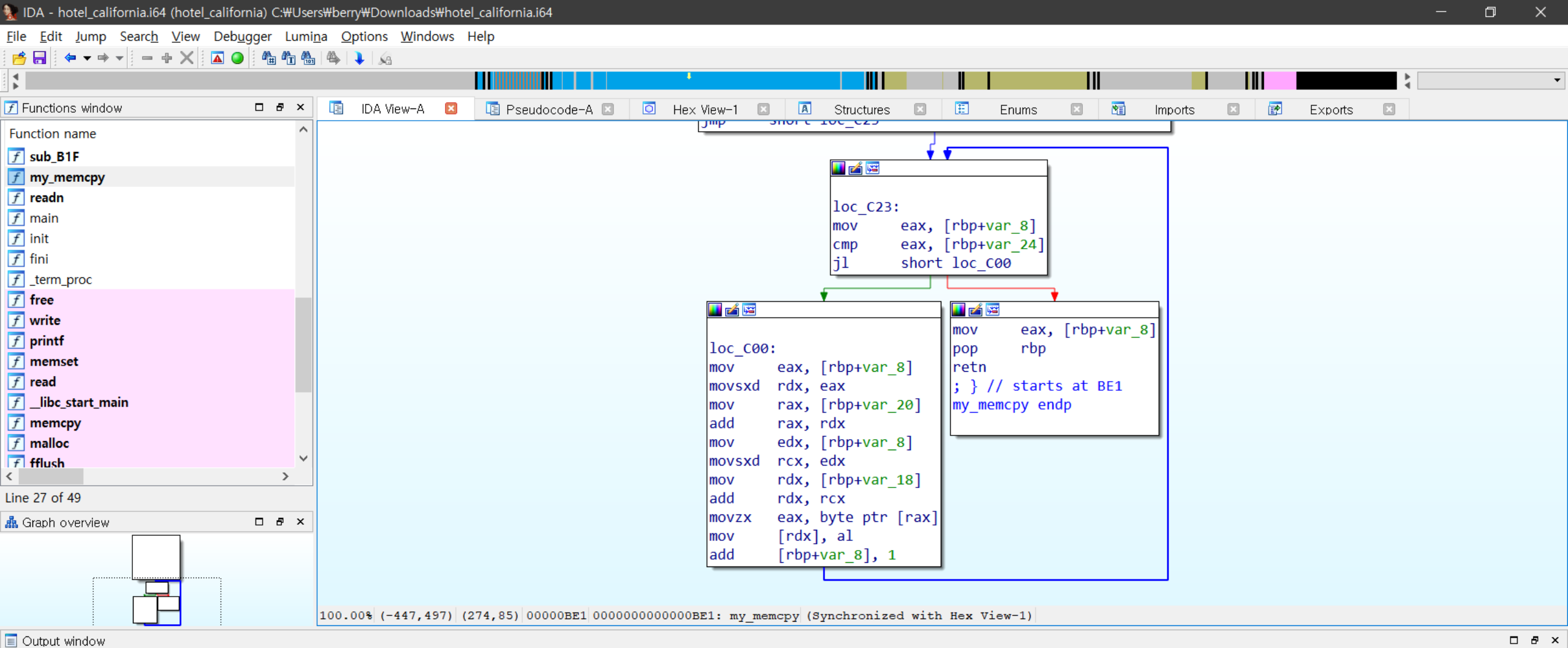
```
\x31\xc0_\x50_\x68\x2f\x2f\x73\x68_\x68...
```

```
xor eax,eax
```

```
push eax
```

```
push '//sh' (0x68732f2f)
```

```
push ...
```



프로그램 분석

문제의 흐름

1. 웰코드를 입력받고
2. 레지스터들을 0으로 세팅한 뒤
3. 실행 (반복)

```
1 void main()
2 {
3     int r1 = 0, r2 = 0; // [rsp+10h] [rbp-450h]
4     char input[1024]; // [rsp+30h] [rbp-430h]
5     int fd = open("/dev/urandom", O_RDONLY);
6
7     setvbuf(stdout, 0, _IOLBF, 0x2000);
8     write(1, "Welcome to the Hotel California.\n", 0x21);
9
10
11     while (1)
12     {
13         // Read shellcode & random integers
14         long nbytes = read(0, input, 1024);
15         read(fd, &r1, 4);
16         read(fd, &r2, 4);
17
18         void* ptr = allocate_code(nbytes, 0, r1);
19         printf("\n(received %d bytes)\n", nbytes);
20         if (nbytes)
21             my_memcpy(&ptr[sizeof(stub_code) + 4], input, nbytes);
22
23         r1 = 0;
24         __asm { vzeroall, mov {r10, r12, r13, rsi, rdi}, 0 }
25         sleep(1);
26         r2 = 0;
27
28         // Execute & free shellcode
29         (funcptr_t *)ptr(rbx := r2);
30         free(ptr);
31         sleep(2);
32     }
33 }
34
```

문제의 흐름

1. 셸코드를 입력받고
2. 레지스터들을 0으로 세팅한 뒤
3. 실행 (반복)

```
1 void main()
2 {
3     int r1 = 0, r2 = 0; // [rsp+10h] [rbp-450h]
4     char input[1024]; // [rsp+30h] [rbp-430h]
5     int fd = open("/dev/urandom", O_RDONLY);
6
7     setvbuf(stdout, 0, _IOLBF, 0x2000);
8     write(1, "Welcome to the Hotel California.\n", 0x21);
9
10    while (1)
11    {
12        // Read shellcode & random integers
13        long nbytes = read(0, input, 1024);
14        read(fd, &r1, 4);
15        read(fd, &r2, 4);
16
17        void* ptr = allocate_code(nbytes, 0, r1);
18        printf("\n(received %d bytes)\n", nbytes);
19        if (nbytes)
20            my_memcpy(&ptr[sizeof(stub_code) + 4], input, nbytes);
21
22        r1 = 0;
23        __asm { vzeroall, mov {r10, r12, r13, rsi, rdi}, 0 }
24        sleep(1);
25        r2 = 0;
26
27        // Execute & free shellcode
28        (funcptr_t *)ptr(rbx := r2);
29        free(ptr);
30        sleep(2);
31    }
32 }
33
34
```

문제의 흐름

1. 셸코드를 입력받고
2. 레지스터들을 0으로 세팅한 뒤
3. 실행 (반복)

```
1 void main()
2 {
3     int r1 = 0, r2 = 0; // [rsp+10h] [rbp-450h]
4     char input[1024]; // [rsp+30h] [rbp-430h]
5     int fd = open("/dev/urandom", O_RDONLY);
6
7     setvbuf(stdout, 0, _IOLBF, 0x2000);
8     write(1, "Welcome to the Hotel California.\n", 0x21);
9
10    while (1)
11    {
12        // Read shellcode & random integers
13        long nbytes = read(0, input, 1024);
14        read(fd, &r1, 4);
15        read(fd, &r2, 4);
16
17        void* ptr = allocate_code(nbytes, 0, r1);
18        printf("\n(received %d bytes)\n", nbytes);
19        if (nbytes)
20            my_memcpy(&ptr[sizeof(stub_code) + 4], input, nbytes);
21
22        r1 = 0;
23        __asm { vzeroall, mov {r10, r12, r13, rsi, rdi}, 0 }
24        sleep(1);
25        r2 = 0;
26
27        // Execute & free shellcode
28        (funcptr_t *)ptr(rbx := r2);
29        free(ptr);
30        sleep(2);
31    }
32 }
33
34
```


문제의 흐름

1. 웰코드를 입력받고

- 난수 r1, r2를 생성
- 코드를 메모리 상에 배치

```
1 void main()
2 {
3     int r1 = 0, r2 = 0; // [rsp+10h] [rbp-450h]
4     char input[1024]; // [rsp+30h] [rbp-430h]
5     int fd = open("/dev/urandom", O_RDONLY);
6
7     setvbuf(stdout, 0, _IOLBF, 0x2000);
8     write(1, "Welcome to the Hotel California.\n", 0x21);
9
10    while (1)
11    {
12        // Read shellcode & random integers
13        long nbytes = read(0, input, 1024);
14        read(fd, &r1, 4);
15        read(fd, &r2, 4);
16
17        void* ptr = allocate_code(nbytes, 0, r1);
18        printf("\n(received %d bytes)\n", nbytes);
19        if (nbytes)
20            my_memcpy(&ptr[sizeof(stub_code) + 4], input, nbytes);
21
22        r1 = 0;
23    }
24
25    free(ptr);
26    sleep(2);
27 }
```



↑
여기서부터 실행

stub code

- 입력한 쉘코드가 실행되기 전, 고정된 코드를 실행

```

.rodata:0000000000000F70 loc_F70:                                ; DATA XREF: .rodata:loc_F70↓o
.rodata:0000000000000F70      lea      rdi, loc_F70
.rodata:0000000000000F77      sub      rdi, 14h
.rodata:0000000000000F7B      mov      eax, [rdi]
.rodata:0000000000000F7D      mov      [rdi], eax
.rodata:0000000000000F7F      xor      rax, rax
.rodata:0000000000000F82      xor      rcx, rcx
.rodata:0000000000000F85      xor      rdx, rdx
.rodata:0000000000000F88      xor      rsi, rsi
.rodata:0000000000000F8B      xacquire lock xor [rdi], ebx
.rodata:0000000000000F8F      xtest
.rodata:0000000000000F92      jnz      short loc_F95
.rodata:0000000000000F94      retn
.rodata:0000000000000F95 ; -----
.rodata:0000000000000F95
.rodata:0000000000000F95 loc_F95:                                ; CODE XREF: .rodata:0000000000000F92↑j
.rodata:0000000000000F95      xor      rbp, rbp
.rodata:0000000000000F98      xor      rsp, rsp
.rodata:0000000000000F9B      xor      rdi, rdi
.rodata:0000000000000F9E      xor      rbx, rbx
.rodata:0000000000000F9E ; -----
.rodata:0000000000000FA1 user_code_here db 0

```

```
0000000000000F70 loc_F70:                                ; DATA XREF: .rodata:loc_F70↓o
0000000000000F70      lea     rdi, loc_F70
0000000000000F77      sub     rdi, 14h
0000000000000F7B      mov     eax, [rdi]
0000000000000F7D      mov     [rdi], eax
0000000000000F7F      xor     rax, rax
0000000000000F82      xor     rcx, rcx
0000000000000F85      xor     rdx, rdx
0000000000000F88      xor     rsi, rsi
0000000000000F8B      xacquire lock xor [rdi], ebx
0000000000000F8F      xtest
0000000000000F92      jnz     short loc_F95
0000000000000F94      retn
0000000000000F95 ; -----
0000000000000F95
0000000000000F95 loc_F95:                                ; CODE XREF: .rodata:0000000000000F92↑j
0000000000000F95      xor     rbp, rbp
0000000000000F98      xor     rsp, rsp
0000000000000F9B      xor     rdi, rdi
0000000000000F9E      xor     rbx, rbx
0000000000000F9E ; -----
0000000000000FA1 user_code_here db 0
```

1. rdi := &rand
rax, rcx, rdx, rsi := 0



```

00000000000000F70 loc_F70:                                ; DATA XREF: .rodata:loc_F70↓o
00000000000000F70      lea      rdi, loc_F70
00000000000000F77      sub      rdi, 14h
00000000000000F7B      mov      eax, [rdi]
00000000000000F7D      mov      [rdi], eax
00000000000000F7F      xor      rax, rax
00000000000000F82      xor      rcx, rcx
00000000000000F85      xor      rdx, rdx
00000000000000F88      xor      rsi, rsi
00000000000000F8B      xacquire lock xor [rdi], ebx
00000000000000F8F      xtest
00000000000000F92      jnz      short loc_F95
00000000000000F94      retn
00000000000000F95 ; -----
00000000000000F95
00000000000000F95 loc_F95:                                ; CODE XREF: .rodata:00000000000000F92↑j
00000000000000F95      xor      rbp, rbp
00000000000000F98      xor      rsp, rsp
00000000000000F9B      xor      rdi, rdi
00000000000000F9E      xor      rbx, rbx
00000000000000F9E ; -----
00000000000000FA1 user_code_here db 0

```

2. start HLE (xacquire) after
[rand] := r1 ⊕ r2(ebx)
3. if in HLE (xtest)
run my code
else (aborted during HLE)
return immediately

Intel TSX

- Transactional Synchronization Extensions
- Intel CPU의 새로운 명령어 집합
 - xacquire - xrelease / xbegin - xend / xtest
- Transactional memory의 하드웨어 레벨 구현
(중략)

Intel TSX

- 기본적인 컨셉 (DB와 비슷함)
 - "트랜잭션"을 시작 후
"커밋"을 하기 전까지의 모든 연산을 CPU에서 기록
 - 중간에 "취소"를 하면 해당 연산들이 버려지고, 결과는 반영이 안 됨
 - 인터럽트, 예외 등에 의해서도 트랜잭션이 취소됨
 - RTM, HLE 방식이 있음 (패턴의 차이)

Intel TSX – Restricted Transactional Memory

```
1
2 xbegin L0
3     # xabort/exception rolls back everything
4     add [rdi], rax
5     ...
6
7     # end transaction
8     xend
9     < transaction success! >
10
11 L0:
12     # eax: error status
13
```

← 트랜잭션 시작
(취소 시 L0으로 이동)

← 트랜잭션 종료

Intel TSX – Hardware Lock Elision

```
1
2  lea rdi, [rip + lock_variable]
3
4  # HLE start
5  xacquire lock xchg [rdi], eax
6  ... <other thread runs their code, too>
7
8  # HLE end
9  xrelease lock xchg [rdi], eax
10
11 lock_variable: .long 0, 0
12
```

←트랜잭션 시작

←트랜잭션 검증&커밋

Intel TSX – Hardware Lock Elision

```
1
2  lea rdi, [rip + lock_variable]
3
4  # HLE start
5  xacquire lock xchg [rdi], eax
6  ... <other thread runs their code, too>
7
```

1. 트랜잭션 시작

- 특정 메모리 주소에 임의 값을 넣으면서 시작 (R이라 칭함)
- 명령어는 xacquire lock 접두어가 붙은 add, sub, xchg 등 몇 가지로 제한됨

Intel TSX – Hardware Lock Elision

```
7
8  # HLE end
9  xrelease lock xchg [rdi], eax
10
11 lock_variable: .long 0, 0
12
```

2. 트랜잭션 검증&커밋

- 시작 시 지정된 메모리 주소에 값 R이 들어가게 하는 명령어를 xrelease 접두어로 지정
- 해당 명령어에 지정된 주소, 값, 접근 크기 불일치 시 롤백

Intel TSX – Hardware Lock Elision

```
1
2  lea rdi, [rip + lock_variable]
3
4  # HLE start
5  xacquire lock xchg [rdi], eax
6  ... <other thread runs their code, too>
7
```

3. 트랜잭션 롤백

- 시작 명령어(xacquire)가 xacquire 접두어가 빠진 채로 다시 실행됨
- 이번엔 (xacquire이 빠졌으므로) 트랜잭션 안에서 실행되지 않으며, xtest로 체크 가능

```

00000000000000F70 loc_F70:                                ; DATA XREF: .rodata:loc_F70↓o
00000000000000F70      lea      rdi, loc_F70
00000000000000F77      sub      rdi, 14h
00000000000000F7B      mov      eax, [rdi]
00000000000000F7D      mov      [rdi], eax
00000000000000F7F      xor      rax, rax
00000000000000F82      xor      rcx, rcx
00000000000000F85      xor      rdx, rdx
00000000000000F88      xor      rsi, rsi
00000000000000F8B      xacquire lock xor [rdi], ebx
00000000000000F8F      xtest
00000000000000F92      jnz      short loc_F95
00000000000000F94      retn
00000000000000F95 ; -----
00000000000000F95
00000000000000F95 loc_F95:                                ; CODE XREF: .rodata:00000000000000F92↑j
00000000000000F95      xor      rbp, rbp
00000000000000F98      xor      rsp, rsp
00000000000000F9B      xor      rdi, rdi
00000000000000F9E      xor      rbx, rbx
00000000000000F9E ; -----
00000000000000FA1 user_code_here db 0

```

2. start HLE (xacquire) after
[rand] := r1 ⊕ r2(ebx)
3. if in HLE (xtest)
run my code
else (aborted during HLE)
return immediately

```

0000000000000F70 loc_F70:
0000000000000F70      lea     rdi, loc_F70
0000000000000F77      sub     rdi, 14h
0000000000000F7B      mov     eax, [rdi]
0000000000000F7D      mov     [rdi], eax
0000000000000F7F      xor     rax, rax
0000000000000F82      xor     rcx, rcx
0000000000000F85      xor     rdx, rdx
0000000000000F88      xor     rsi, rsi
0000000000000F8B      xacquire lock xor [rdi], ebx
0000000000000F8F      xtest
0000000000000F92      jnz     short loc_F95
0000000000000F94      retn
0000000000000F95 ; -----
0000000000000F95 loc_F95:
0000000000000F95      xor     rbp, rbp
0000000000000F98      xor     rsp, rsp
0000000000000F9B      xor     rdi, rdi
0000000000000F9E      xor     rbx, rbx
0000000000000F9E ; -----
0000000000000FA1 user_code_here db 0

```

```
; DATA XREF: .rodata:loc_F70↓o
```

2. start HLE (xacquire) after
[rand] := r1 ⊕ r2(ebx)
3. if in HLE (xtest)
run my code
else (aborted during HLE)
return immediately

← xrelease가 코드에 없음

또한 트랜잭션 안에서 시스템 콜 등을
실행 시 바로 트랜잭션 취소 및 롤백

```

0000000000000F70 loc_F70:
0000000000000F70      lea     rdi, loc_F70
0000000000000F77      sub     rdi, 14h
0000000000000F7B      mov     eax, [rdi]
0000000000000F7D      mov     [rdi], eax
0000000000000F7F      xor     rax, rax
0000000000000F82      xor     rcx, rcx
0000000000000F85      xor     rdx, rdx
0000000000000F88      xor     rsi, rsi
0000000000000F8B      xacquire lock xor [rdi], ebx
0000000000000F8F      xtest
0000000000000F92      jnz     short loc_F95
0000000000000F94      retn
0000000000000F95 ; -----
0000000000000F95
0000000000000F95 loc_F95:
0000000000000F95      xor     rbp, rbp
0000000000000F98      xor     rsp, rsp
0000000000000F9B      xor     rdi, rdi
0000000000000F9E      xor     rbx, rbx
0000000000000F9E ; -----
0000000000000FA1 user_code_here db 0

```

```
; DATA XREF: .rodata:loc_F70↓o
```

2. start HLE (xacquire) after
[rand] := r1 ⊕ r2(ebx)
3. if in HLE (xtest)
run my code
else (aborted during HLE)
return immediately

← xrelease가 코드에 없음

또한 트랜잭션 안에서 시스템 콜 등을
실행 시 바로 트랜잭션 취소 및 롤백

코드에서 트랜잭션을 빠져 나와야 됨

xrelease mov [rand], r1 로 가능하지만,
r1 또는 r2의 값은 코드 주변에 없음

“목표: r1, r2 중 하나 찾기”

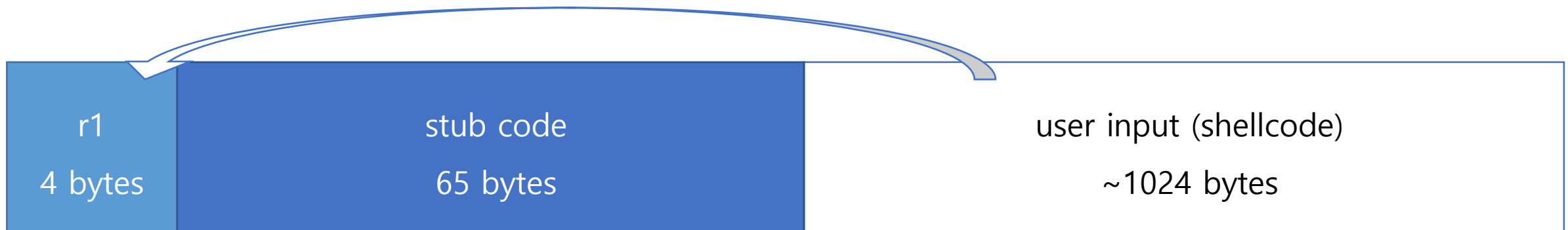
- xacquire 직전, 코드 바로 앞 주소(&rand)에는 r1의 값이 있었음
 - 현재는 $r1 \oplus r2$ 가 있음
- 스택, 바이너리, 라이브러리 중 어떤 주소도 모름
 - $rsp = 0, rbx = 0, rdi = 0$

코드에서 트랜잭션을 빠져 나와야 됨
xrelease mov [rand], r1 로 가능하지만,
r1 또는 r2의 값은 코드 주변에 없음

“목표: r1, r2 중 하나 찾기”

```
00000000000000F70 loc_F70: ; DATA XREF: .rodata:loc_F70↓o
00000000000000F70          lea    rdi, loc_F70
00000000000000F77          sub    rdi, 14h
```

- Q. rdi(&rand)의 원래 값은? A. RIP-relative addressing
 - `lea rdi, [rip - 0x...]` (현재 명령어 근처의 주소를 얻어옴)
 - 64비트 기능 (32비트는 `fstenv`로 가능)



“목표: $r1$, $r2$ 중 하나 찾기”

- Q. $r1$, $r2$ 의 값은 어디에?
 - 레지스터 (정수, 실수 등등), 코드 주변에는 없음
 - 스택에는?

main에서의 r1, r2

```
13 // Read shellcode & random integers
14 long nbytes = read(0, input, 1024);
15 read(fd, &r1, 4);
16 read(fd, &r2, 4);
17
18 void* ptr = allocate_code(nbytes, 0, r1);
19 printf("\n(received %d bytes)\n", nbytes);
20 if (nbytes)
21     my_memcpy(&ptr[sizeof(stub_code) + 4], input, nbytes);
22
23 r1 = 0;
24 __asm { vzeroall, mov {r10, r12, r13, rsi, rdi}, 0 }
25 sleep(1);
26 r2 = 0;
27
28 // Execute & free shellcode
29 (funcptr_t *)ptr(rbx := r2);
30 free(ptr);
```

allocate_code:

```
1
2 char *allocate_code(int bytes, int a2, int r1)
3 {
4     char *buffer; // [rsp+10h] [rbp-10h]
5     long pagesize; // [rsp+18h] [rbp-8h]
6
7     buffer = malloc(sizeof(stub) + bytes + 4);
8     memset(buffer, 0x90, sizeof(stub) + bytes + 4);
9     mprotect(-sysconf(_SC_PAGESIZE) & buffer, 0xFA0, 7);
10    *(int *)&buffer[a2] = r1;
11    memcpy(&buffer[a2 + 4], &stub, sizeof(stub));
12    return buffer;
13 }
14
```

- 코드 할당, stub 배치

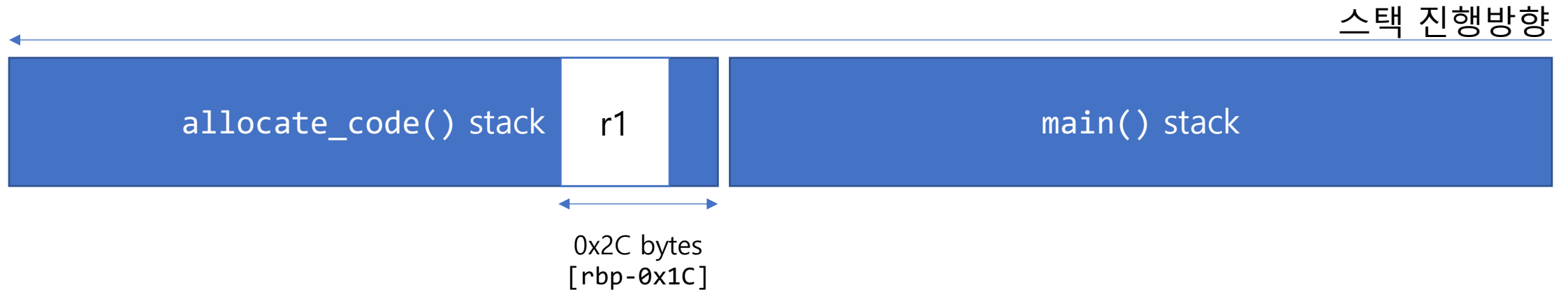
r1 4 bytes	stub code 65 bytes	user input (shellcode) ~1024 bytes
---------------	-----------------------	---------------------------------------

allocate_code (assembly):

```
.text:00000000000000B1F ; char *__fastcall allocate_code(int bytes, int a2, int r1)
.text:00000000000000B1F allocate_code    proc near                                ; CODE XREF: main+116↓p
.text:00000000000000B1F
.text:00000000000000B1F var_1C          = dword ptr -1Ch
.text:00000000000000B1F var_18          = dword ptr -18h
.text:00000000000000B1F var_14          = dword ptr -14h
.text:00000000000000B1F buffer          = qword ptr -10h
.text:00000000000000B1F var_8           = qword ptr -8
.text:00000000000000B1F
.text:00000000000000B1F ; __unwind {
.text:00000000000000B1F         push    rbp
.text:00000000000000B20         mov     rbp, rsp
.text:00000000000000B23         sub     rsp, 20h
.text:00000000000000B27         mov     [rbp+var_14], edi
.text:00000000000000B2A         mov     [rbp+var_18], esi
.text:00000000000000B2D         mov     [rbp+var_1C], edx ; r1
.text:00000000000000B30         mov     eax, cs:Stub_1en
.text:00000000000000B36         mov     eax, [rbp+var_14]
.text:00000000000000B39         add     eax, edx
.text:00000000000000B3B         add     eax, 4
```

찾았다, r1!

- 하지만... 함수가 끝나도 이 값이 남아있을까?



```
Breakpoint *0x55555554000+0xdb9
pwndbg> x/xw $rsp-16-0x1c
0x7fffffffde14: 0x4b8bc263
pwndbg> █
```

- 있다! 하지만, `sleep`, `printf`, `my_memcpy` 후에 남아있을까?

The long road to xacquire ...

```
18 void* ptr = allocate_code(nbytes, 0, r1);
19 printf("\n(received %d bytes)\n", nbytes);
20 if (nbytes)
21     my_memcpy(&ptr[sizeof(stub_code) + 4], input, nbytes);
22
23 r1 = 0;
24 __asm { vzeroall, mov {r10, r12, r13, rsi, rdi}, 0 }
25 sleep(1);
26 r2 = 0;
27
28 // Execute & free shellcode
29 (funcptr_t *)ptr(rbx := r2);
```

printf → my_memcpy → sleep → 실행

The long road to xacquire ...

```
18 void* ptr = allocate_code(nbytes, 0, r1);
19 printf("\n(received %d bytes)\n", nbytes);
20 if (nbytes)
21     my_memcpy(&ptr[sizeof(stub_code) + 4], input, nbytes);
22
23 r1 = 0;
24 __asm { vzeroall, mov {r10, r12, r13, rsi, rdi}, 0 }
25 sleep(1);
26 r2 = 0;
27
28 // Execute & free shellcode
29 (funcptr_t *)ptr(rbx := r2);
```

printf → my_memcpy → sleep → 실행

○ X ○

nbytes = 0이면 건너뛰기 가능!

소켓을 shutdown 또는 close 하면 됨

거의 다 왔다!

<code>r1=? rsp=?</code>	<code>...</code>	<code>&libc.so.6</code>
<code>__environ</code> → <code>rsp</code>	<code>shutdown(fd)</code> → <code>skip memcpy</code>	<code>allocate_code</code> → <code>rsp-0x2C: r1</code>

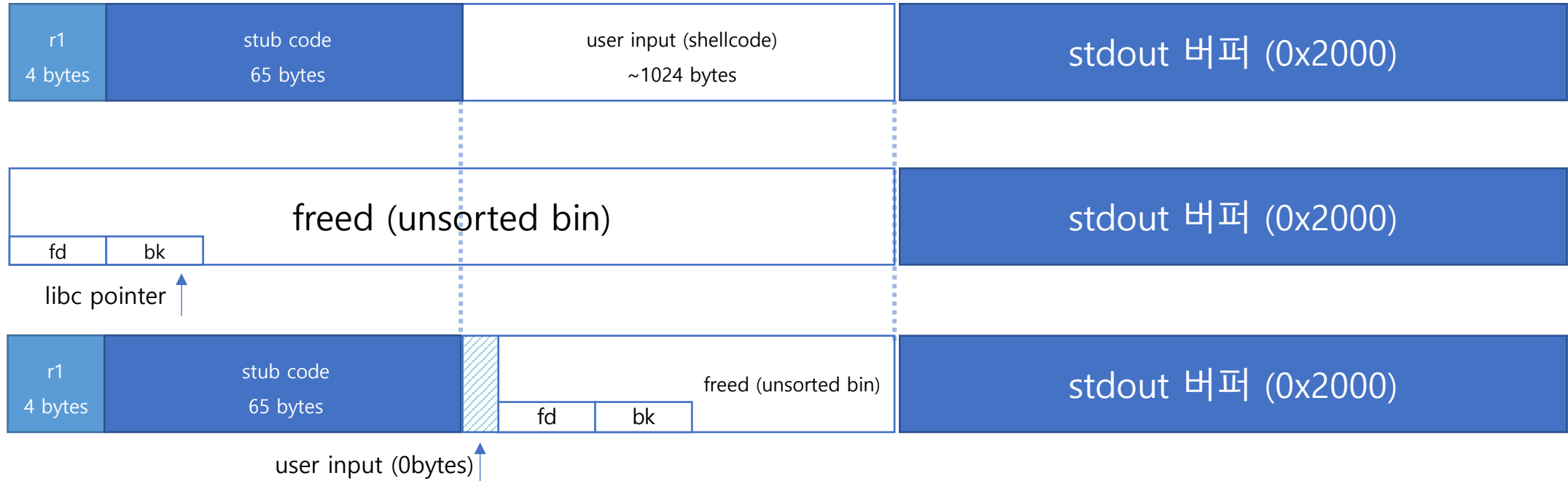
- 마지막 연결 고리: libc의 주소
 - libc의 `argv`, `environ` 변수가 `main` 뒤의 스택 주소를 가리키고 있음

libc 근처 포인터 얻기

```
8      setvbuf(stdout, 0, _IOLBF, 0x2000);
9
10     while (1)
11     {
12         // 1. malloc(0x45 + nbytes)
13         void* ptr = allocate_code(nbytes, 0, r1);
14         // 2. malloc(0x2000) - buffer of stdout
15         printf("\n(received %d bytes)\n", nbytes);
16         ...
17         free(ptr);
18     }
```

main = [malloc x 2 + free x 1] x n → unsorted bin → &libc.so

어째서 free 후 libc 포인터가 남는가?



- stdout에서 line buffering을 수행, 셸코드 free 시 top chunk와 병합되지 않음 (setvbuf w/ _IOLBF)
- 1024 byte 이상의 힙 청크를 free할 경우 tcache가 적용되지 않고 unsorted bin 생성
- unsorted bin 생성 시 해당 힙 청크 주소에는 두 개의 libc 포인터(&main_arena.top)가 저장됨

풀이

아래의 쉘코드를 1024-0x41-4바이트 이상의 크기로 패딩하여 보낸 후, 소켓 shutdown

- 코드 주위의 fd/bk 포인터(main_arena)로 environ 포인터(스택)를 얻고,
 - `mov rax, [rip + ...]`
- 스택에 있는 r1 값을 얻어와 값을 복구하는 명령어를,
 - `mov rax, [rax]; sub rax, ...; mov eax, [rax]`
- xrelease prefix를 붙여서 실행, 트랜잭션 밖으로 나오기
 - `xrelease mov [rip - ...], eax; <바깥에서 실행할 쉘코드>`
- 그 뒤의 코드에서는 /FLAG.txt를 읽어서 출력

Reference

Intel TSX - HLE

- [Intel TSX-NI \(Wikipedia\)](#)
- [HLE instructions - xacquire / xrelease](#)
- [Intel® 64 and IA-32 Architectures Software Developer's Manual](#) – Chapter 16

glibc heap (malloc, free)

- [\[glibc\] 동적 메모리 관리](#)
- [malloc diagram](#)