Date: Tue, 3 Oct 2023 17:50:36 +0000
From: Qualys Security Advisory <qsa@...lys.com>
To: "oss-security@...ts.openwall.com" <oss-security@...ts.openwall.com>
Subject: CVE-2023-4911: Local Privilege Escalation in the glibc's ld.so

Qualys Security Advisory

Looney Tunables: Local Privilege Escalation in the glibc's ld.so
(CVE-2023-4911)

==========
Contents
==========

==========
Summary
==========

The GNU C Library's dynamic loader "find[s] and load[s] the shared
objects (shared libraries) needed by a program, prepare[s] the program
to run, and then run[s] it" (man ld.so). The dynamic loader is extremely
security sensitive, because its code runs with elevated privileges when
a local user executes a set-user-ID program, a set-group-ID program, or
a program with capabilities. Historically, the processing of environment
variables such as LD_PRELOAD, LD_AUDIT, and LD_LIBRARY_PATH has been a
fertile source of vulnerabilities in the dynamic loader.

Recently, we discovered a vulnerability (a buffer overflow) in the
dynamic loader's processing of the GLIBC_TUNABLES environment variable
(https://www.gnu.org/software/libc/manual/html_node/Tunables.html). This
vulnerability was introduced in April 2021 (glibc 2.34) by commit 2ed18c
("Fix SXID_ERASE behavior in setuid programs (BZ #27471)").

We successfully exploited this vulnerability and obtained full root
privileges on the default installations of Fedora 37 and 38, Ubuntu
22.04 and 23.04, Debian 12 and 13; other distributions are probably also
vulnerable and exploitable (one notable exception is Alpine Linux, which
uses musl libc, not the glibc). We will not publish our exploit for now;
however, this buffer overflow is easily exploitable (by transforming it
into a data-only attack), and other researchers might publish working
exploits shortly after this coordinated disclosure.

==========
Analysis
==========

At the very beginning of its execution, ld.so calls __tunables_init() to
walk through the environment (at line 279), searching for GLIBC_TUNABLES
variables (at line 282); for each GLIBC_TUNABLES that it finds, it makes

a copy of this variable (at line 284), calls parse_tunables() to process
and sanitize this copy (at line 286), and finally replaces the original
GLIBC_TUNABLES with this sanitized copy (at line 288):

```
---------------------------------------------------------------------
269 void
270 __tunables_init (char **envp)
271 {
272   char *envname = NULL;
273   char *envval = NULL;
274   size_t len = 0;
275   char **prev_envp = envp;
...
279   while ((envp = get_next_env (envp, &envname, &len, &envval,
280                                &prev_envp)) != NULL)
281     {
282       if (tunable_is_name ("GLIBC_TUNABLES", envname))
283         {
284           char *new_env = tunables_strdup (envname);
285           if (new_env != NULL)
286             parse_tunables (new_env + len + 1, envval);
287           /* Put in the updated envval.  */
288           *prev_envp = new_env;
289           continue;
290         }
---------------------------------------------------------------------
```

The first argument of parse_tunables() (tunestr) points to the
soon-to-be-sanitized copy of GLIBC_TUNABLES, while the second argument
(valstring) points to the original GLIBC_TUNABLES environment variable
(in the stack). To sanitize the copy of GLIBC_TUNABLES (which should be
of the form "tunable1=aaa:tunable2=bbb"), parse_tunables() removes all
dangerous tunables (the SXID_ERASE tunables) from tunestr, but keeps
SXID_IGNORE and NONE tunables (at lines 221-235):

```
---------------------------------------------------------------------
162 static void
163 parse_tunables (char *tunestr, char *valstring)
164 {
...
168   char *p = tunestr;
169   size_t off = 0;
170
171   while (true)
172     {
173       char *name = p;
174       size_t len = 0;
175
176       /* First, find where the name ends.  */
177       while (p[len] != '=' && p[len] != ':' && p[len] != '\0')
178         len++;
179
180       /* If we reach the end of the string before getting a valid name-value
181          pair, bail out.  */
182       if (p[len] == '\0')
183         {
184           if (__libc_enable_secure)
185             tunestr[off] = '\0';
186           return;
187         }
188
189       /* We did not find a valid name-value pair before encountering the
190          colon.  */
191       if (p[len]== ':')
192         {
193           p += len + 1;
194           continue;
195         }
196
197       p += len + 1;
```

```
198
199       /* Take the value from the valstring since we need to NULL terminate it.  */
200       char *value = &valstring[p - tunestr];
201       len = 0;
202
203       while (p[len] != ':' && p[len] != '\0')
204         len++;
205
206       /* Add the tunable if it exists.  */
207       for (size_t i = 0; i < sizeof (tunable_list) / sizeof (tunable_t); i++)
208         {
209           tunable_t *cur = &tunable_list[i];
210
211           if (tunable_is_name (cur->name, name))
212             {
...
219               if (__libc_enable_secure)
220                 {
221                   if (cur->security_level != TUNABLE_SECLEVEL_SXID_ERASE)
222                     {
223                       if (off > 0)
224                         tunestr[off++] = ':';
225
226                       const char *n = cur->name;
227
228                       while (*n != '\0')
229                         tunestr[off++] = *n++;
230
231                       tunestr[off++] = '=';
232
233                       for (size_t j = 0; j < len; j++)
234                         tunestr[off++] = value[j];
235                     }
236
237                   if (cur->security_level != TUNABLE_SECLEVEL_NONE)
238                     break;
239                 }
240
241               value[len] = '\0';
242               tunable_initialize (cur, value);
243               break;
244             }
245         }
246
247       if (p[len] != '\0')
248         p += len + 1;
249     }
250 }
```

Unfortunately, if a GLIBC_TUNABLES environment variable is of the form "tunable1=tunable2=AAA" (where "tunable1" and "tunable2" are SXID_IGNORE tunables, for example "glibc.malloc.mxfast"), then:

- during the first iteration of the "while (true)" in parse_tunables(), the entire "tunable1=tunable2=AAA" is copied in-place to tunestr (at lines 221-235), thus filling up tunestr;

- at lines 247-248, p is not incremented (p[len] is '\0' because no ':' was found at lines 203-204) and therefore p still points to the value of "tunable1", i.e. "tunable2=AAA";

- during the second iteration of the "while (true)" in parse_tunables(), "tunable2=AAA" is appended (as if it were a second tunable) to tunestr (which is already full), thus overflowing tunestr.

A note on fuzzing: although we discovered this buffer overflow manually, we later tried to fuzz the vulnerable function, parse_tunables(); both AFL++ and libFuzzer re-discovered this overflow in less than a second, when provided with a dictionary of tunables (which can be compiled by

running "ld.so --list-tunables").

```
================================================================================
Proof of concept
================================================================================

$ env -i "GLIBC_TUNABLES=glibc.malloc.mxfast=glibc.malloc.mxfast=A" "Z=`printf '%08192x'
1`" /usr/bin/su --help
Segmentation fault (core dumped)
```

```
================================================================================
Exploitation
================================================================================
```

This vulnerability is a straightforward buffer overflow, but what should we overwrite to achieve arbitrary code execution? The buffer we overflow is allocated at line 284 by tunables_strdup(), a re-implementation of strdup() that uses ld.so's __minimal_malloc() instead of the glibc's malloc() (indeed, the glibc's malloc() has not been initialized yet). This __minimal_malloc() implementation simply calls mmap() to obtain more memory from the kernel.

The question, then, is: what writable pages can we overwrite in the mmap region? To the best of our knowledge, we have only two options (because this buffer overflow takes place at the very beginning of ld.so's execution):

1/ The read-write ELF segment of ld.so itself (the first pages of this read-write segment are actually ld.so's RELRO segment, but they have not been mprotect()ed read-only yet):

```
--------------------------------------------------------------------------
7f209f367000-7f209f369000 r--p 00000000 fd:00 10943                      /usr/lib/x86_64-
linux-gnu/ld-linux-x86-64.so.2
7f209f369000-7f209f393000 r-xp 00002000 fd:00 10943                      /usr/lib/x86_64-
linux-gnu/ld-linux-x86-64.so.2
7f209f393000-7f209f39e000 r--p 0002c000 fd:00 10943                      /usr/lib/x86_64-
linux-gnu/ld-linux-x86-64.so.2
7f209f39f000-7f209f3a3000 rw-p 00037000 fd:00 10943                      /usr/lib/x86_64-
linux-gnu/ld-linux-x86-64.so.2
--------------------------------------------------------------------------
```

However, on all the Linux distributions that we checked, the unmapped hole immediately below ld.so's read-write segment is at most one page, but ld.so's __minimal_malloc() always allocates at least two pages ("one extra page to reduce number of mmap calls"). In other words, the buffer we overflow cannot be allocated immediately below ld.so's read-write segment, and therefore cannot overwrite this segment.

2/ Our only option, then, is to overwrite mmap()ed pages that were allocated by tunables_strdup() itself: because __tunables_init() can process multiple GLIBC_TUNABLES environment variables, and because the Linux kernel's mmap() is a top-down allocator, we can mmap() a first GLIBC_TUNABLES (without overflowing it), mmap() a second GLIBC_TUNABLES (immediately below the first one) and overflow it, thus overwriting the first GLIBC_TUNABLES. As a result, we can:

- either replace this first GLIBC_TUNABLES with a completely different environment variable, for example LD_PRELOAD or LD_LIBRARY_PATH -- but these dangerous variables are later removed from the environment by ld.so (in process_envvars()), and such a replacement would therefore be useless;

- or replace the first GLIBC_TUNABLES with a GLIBC_TUNABLES that contains dangerous (SXID_ERASE) tunables, which were previously removed by parse_tunables() -- although this seems promising at first, exploiting such a replacement would require a SUID-root program that setuid(0)s and execve()s another program with a preserved environment

  (to process the dangerous GLIBC_TUNABLES as root, but without
  __libc_enable_secure).

  Alas, we do not know of such a SUID-root program on Linux (on OpenBSD,
  /usr/bin/chpass setuid(0)s and execv()s /usr/sbin/pwd_mkdb, and was
  exploited in CVE-2019-19726); if you, dear reader, know of such a
  SUID-root program on Linux, please let us know!

At that point, the situation looked quite hopeless, but a comment in
ld.so's _dl_new_object() (which is called long after __tunables_init())
caught our attention (at line 105):

```
 56 struct link_map *
 57 _dl_new_object (char *realname, const char *libname, int type,
 58                  struct link_map *loader, int mode, Lmid_t nsid)
 59 {
 ..
 84   struct link_map *new;
 85   struct libname_list *newname;
 ..
 92   new = (struct link_map *) calloc (sizeof (*new) + audit_space
 93                                     + sizeof (struct link_map *)
 94                                     + sizeof (*newname) + libname_len, 1);
 95   if (new == NULL)
 96     return NULL;
 97
 98   new->l_real = new;
 99   new->l_symbolic_searchlist.r_list = (struct link_map **) ((char *) (new + 1)
100                                                             + audit_space);
101
102   new->l_libname = newname
103     = (struct libname_list *) (new->l_symbolic_searchlist.r_list + 1);
104   newname->name = (char *) memcpy (newname + 1, libname, libname_len);
105   /* newname->next = NULL;      We use calloc therefore not necessary.  */
```

ld.so allocates the memory for this link_map structure with calloc(),
and therefore does not explicitly initialize various of its members to
zero; this is a reasonable optimization. As mentioned earlier, calloc()
here is not the glibc's calloc() but ld.so's __minimal_calloc(), which
calls __minimal_malloc() *without* explicitly initializing the memory it
returns to zero; this is also a reasonable optimization, because for all
intents and purposes __minimal_malloc() always returns a clean chunk of
mmap()ed memory, which is guaranteed to be initialized to zero by the
kernel.

Unfortunately, the buffer overflow in parse_tunables() allows us to
overwrite clean mmap()ed memory with non-zero bytes, thereby overwriting
pointers of the soon-to-be-allocated link_map structure with non-NULL
values. This allows us to completely break the logic of ld.so, which
assumes that these pointers are NULL.

We first tried to exploit this buffer overflow by overwriting the
link_map structure's l_next and l_prev pointers (a doubly linked list of
link_map structures), but we failed because of two assert()ion failures
in setup_vdso(), which immediately abort() ld.so (all the distributions
that we checked compile their glibc, and hence ld.so, with assert()ions
enabled):

```
 96       assert (l->l_next == NULL);
 97       assert (l->l_prev == main_map);
```

We then realized that many more pointers in the link_map structure are
not explicitly initialized to NULL; in particular, the pointers to
Elf64_Dyn structures in the l_info[] array of pointers. Among these,
l_info[DT_RPATH], the "Library search path", immediately stood out: if
we overwrite this pointer and control where and what it points to, then

we can force ld.so to trust a directory that we own, and therefore to
load our own libc.so.6 or LD_PRELOAD library from this directory, and
execute arbitrary code (as root, if we run ld.so through a SUID-root
program).

Where should the overwritten l_info[DT_RPATH] point to? The easy answer
to this question is: the stack; more precisely, our environment strings
in the stack. On Linux, the stack is randomized in a 16GB region, and
our environment strings can occupy up to 6MB (_STK_LIM / 4 * 3, in the
kernel's bprm_stack_limits()): after 16GB / 6MB = 2730 tries we have a
good chance of guessing the address of our environment strings (in our
exploit, we always overwrite l_info[DT_RPATH] with 0x7ffdfffff010, the
center of the randomized stack region). In our tests, this brute force
takes ~30s on Debian, and ~5m on Ubuntu and Fedora (because of their
automatic crash handlers, Apport and ABRT; we have not tried to work
around this slowdown).

What should the overwritten l_info[DT_RPATH] point to? In other words,
what should we store in our 6MB of environment strings? l_info[DT_RPATH]
is a pointer to a small (16B) Elf64_Dyn structure:

- an int64_t d_tag, which should be DT_RPATH (15), but this value is
  never actually checked anywhere, so we can store anything there;

- a uint64_t d_val, which is an offset into the ELF string table of the
  SUID-root program that is being executed (this offset references a
  string that is the "Library search path" itself).

In our exploit, we simply fill our 6MB of environment strings with
0xfffffffffffffff8 (-8), because at an offset of -8B below the string
table of most SUID-root programs, the string "\x08" appears: this forces
ld.so to trust a relative directory named "\x08" (in our current working
directory), and therefore allows us to load and execute our own
libc.so.6 or LD_PRELOAD library from this directory, as root.

One major problem remains unsolved, however: to avoid the kind of
assert()ion failures mentioned earlier (when we tried to overwrite the
l_next and l_prev pointers of the link_map structure), we must overwrite
the soon-to-be-allocated link_map structure with NULL pointers only
(except l_info[DT_RPATH], of course); but intuitively, the ability to
overflow a buffer with a large number of null bytes while parsing a
null-terminated C string sounds quite unusual.

Luckily for us attackers, the bytes that are written out-of-bounds by
parse_tunables() are also read out-of-bounds (at line 234), but not from
the mmap()ed copy of our GLIBC_TUNABLES environment variable (tunestr),
but from our original GLIBC_TUNABLES environment variable in the stack
(valstring, at line 200). Consequently, if we store a large number of
empty strings (null bytes) immediately after our GLIBC_TUNABLES in the
stack, followed by the string "\x10\xf0\xff\xff\xfd\x7f", followed by
more empty strings (null bytes), then we safely overwrite the link_map
structure with null bytes (NULL pointers), except for l_info[DT_RPATH]
(which we overwrite with 0x7ffdfffff010, which points to our own
Elf64_Dyn structures in the stack with a probability of 1/2730).

Final note: the exploitation method described in this advisory works
against almost all of the SUID-root programs that are installed by
default on Linux; a few exceptions are:

- sudo on all distributions, because it specifies its own ELF RUNPATH
  (/usr/libexec/sudo), which overrides our l_info[DT_RPATH];

- chage and passwd on Fedora, because they are protected by special
  SELinux rules;

```
- snap-confine on Ubuntu, because it is protected by special AppArmor
  rules.

Last-minute note: although glibc 2.34 is vulnerable to this buffer
overflow, its tunables_strdup() uses __sbrk(), not __minimal_malloc()
(which was introduced in glibc 2.35 by commit b05fae, "elf: Use the
minimal malloc on tunables_strdup"); we have not yet investigated
whether glibc 2.34 is exploitable or not.


========================================================================
Acknowledgments
========================================================================

We thank Red Hat Product Security, Siddhesh Poyarekar, the members of
linux-distros@...nwall, Salvatore Bonaccorso, and Solar Designer.


========================================================================
Timeline
========================================================================

2023-09-04: Advisory and exploit sent to secalert@...hat.

2023-09-19: Advisory and patch sent to linux-distros@...nwall.

2023-10-03: Coordinated Release Date (17:00 UTC).
```