TU Wien
Institut für Information Systems Engineering
Forschungsgruppe Industrial Software
Arbeitsgruppe ESSE
`https://security.inso.tuwien.ac.at/`

Report Lab1

Advanced Security for Systems Engineering

183.645 – WS 2021

10.12.2021

Team 03

Name	Matrnr.
Daniel Suchan	11810787
Paul Florian Sattlegger	11810278
Othmar Lechner	11841833

Contents

1 Heap Overflow in Sudo - CVE-2021-3156

Category: C - Heap Overflow

Difficulty: Advanced

Real-World Vulnerability: CVE-2021-3156

1.1 Documentation Phase 1

The vulnerability we found was discovered in the Sudo program by the IT security company Qualys. We first read about the vulnerability on their blog post, which already explains the vulnerability quite well and also contains the vulnerable code snippets. The source code of Sudo can be found on GitHub. The vulnerability was introduced in the commit 8255ed69 in July 2011 and was published in the National Vulnerability Database (NVD) and the Common Vulnerabilities and Exposures Database in January 2021 with a CVSS 3 score of 7.8. According to the bug description on the Sudo website itself the affected versions of Sudo are 1.7.7 through 1.7.10p9, 1.8.2 through 1.8.31p2, and 1.9.0 through 1.9.5p1. The vulnerability was patched in version 1.9.5p2. The Debian security advisory lists these corresponding revisions:

- Revision 9b97f1787804
- Revision a97dc92eae6b
- Revision 049ad90590be (condition check patch)
- Revision 09f98816fc89
- Revision c125fbe68783

Another short bug description and several advisories are linked in the corresponding NVD entry which also lists three proof of concept exploits:

- POC Exploit 1
- POC Exploit 2

- POC Exploit 3

However, for developing our vulnerable program and the corresponding exploit, we relied primarily on the initial advisory by Qualys. Additionally, to get a better understanding of the vulnerability, we also watched a comprehensive video covering the Sudo vulnerability on YouTube.

1.1.1 Vulnerability analysis

The Sudo program prior version 1.9.5p2 contains an off-by-one error resulting in a heap-based buffer overflow that ultimately allows privilege escalation to root. The vulnerability is called *Baron Samedit* and can be exploited by running `sudoedit -s` together with a command-line parameter ending with a single backslash character `\`.

In the following we go into a bit more detail and analyse the vulnerable code based on the initial advisory from Qualys:

First of all, one has to know that Sudo can be executed in different modes that can be set with different parameters. One of them is the `MODE_SHELL` which can be activated through the `-s` parameter. If this mode is activated, Sudo concatenates all given command-line parameters and escapes all meta characters with an additional backslash in the `parse_args()` function of `parse_args.c`.

The normal execution flow then continues to execute the `set_cmnd()` function of `sudoers.c` which can also be found in listing 1. One can see, that the command-line parameters that were escaped before are now stored in a heap-based buffer in line 864-871 with the meta-characters getting unescaped in line 866-867.

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

Listing 1: Vulnerable part of the `set_cmnd()` function of `sudoers.c` in commit 8255ed69.

However, when a command-line parameter ends with a single backslash the code in listing 1 leads to a heap-based buffer overflow. Given a parameter ends with a single backslash the `from[0]` in line 866 is the backslash character and `from[1]` is the null terminator of the string. In line 867 the `from` is incremented and points to the null terminator which is then copied to the `user_args` buffer in line 868. At the same time `from` gets incremented again and is now pointing to the first character after the null terminator which allows out-of-bounds characters to be copied to the `user_args` buffer through the **`while`** loop (line 865-869) leading to a buffer overflow.

Anyway, we saw that all meta-characters including backslashes are escaped beforehand in the `parse_args()` function. Unfortunately, the conditions surrounding the escaping code and the conditions to reach the vulnerable code are not equal as stated in listing 2. If one is able to somehow set `MODE_EDIT` and `MODE_SHELL` but not `MODE_RUN`, the vulnerable code is reached without the prior escaping code.

It turns out, that the command `sudoedit -s` does exactly that and we are able to exploit the vulnerability simply by calling `sudoedit -s` with a single backslash character as argument leading to a heap-based buffer overflow.

This vulnerability was patched by adding the `MODE_RUN` to the condition of the vulnerable code so that the code can only be reached if also the prior escaping code was run.

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

vs.

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

Listing 2: Unequal condition check of escaping code in `parse_args()` function (line 571) and vulnerable code (lines 819 and 858).

1.1.2 Other real-world vulnerabilities considered

During our search we came across the following other real-world vulnerabilities:

CVE-2021-37981 Discarded since it is more difficult to abstract the relevant code of CVE-2021-37981 and involves C++ code which our team has little experience with.

CVE-2021-43267 Although this vulnerability is interesting since it can be exploited locally and remotely, we discarded it since it is difficult to *mock* the vulnerable code in this exercise accordingly. Additionally in CVE-2021-43267 there are no POC exploits listed however, in the original blog post a bit more information is available.

CVE-2021-21148 This vulnerability is interesting since it was (and probably is exploited) in the wild according to the corresponding blog post but we discarded it since there are only few details available as well as no POC exploits are listed in the NVD entry.

1.2 Documentation Phase 2

The vulnerable program is found in `vuln_heapcorruption-advanced.c`. It consists of several functions that we copy-pasted from the original source code to recreate the real vulnerability as closely as possible. However, we have abstracted away the logic of Sudo and the Name Server Switch, which is superfluous for demonstrating the heap-based buffer overflow vulnerability.

Let's take a closer look at the main function: The `main` function consists of the vulnerable part of the previously mentioned `set_cmnd()` function (see listing 1). Here we have only removed the **`if`** statements that belong to internal logic of Sudo itself and are therefore not of interest for our vulnerable program. Further, we added the allocation of a `service_user` struct. This is not present in the original code, but most POC exploits rely on that specific struct to be allocated on the heap right after the overflowable `user_args` buffer. Here, it is worth mentioning that a considerable part of exploiting the vulnerability in real-world is to force that specific struct to be allocated exactly there. In one of the POC exploits this is done by modifying the size of environment variables and brute force (heap feng shui). However, to showcase this heap-based buffer overflow and emulate the vulnerable heap layout we simply allocate the `service_user` directly after the `user_args` buffer. Finally, we added a call to the function `nss_load_library()`, which uses values of the `service_user` struct to load a dynamic library. Internally, `dlopen()` is then used to load a shared library, using the value of the `name` field from the `service_user` struct. Since we can control the values of the `service_user` fields through the buffer overflow, this allows loading an arbitrary library resulting in arbitrary code execution. This is consistent with the original implementation of Sudo and we have not made any changes here.

All other snippets are essentially *dependencies* of the `nss_load_library()` function and were merely copied from the corresponding source code and header files of the Name Server Switch. We have only made one relevant change here and that is to remove the first **`if`** statement of the `nss_load_library()` function, as this would require internal logic from the Name Server Switch.

1.3 Documentation Phase 3

In order to create our exploit we, first carefully studied the advisory of Qualys and additionally watched a comprehensive video on YouTube. Subsequently, we went through the aforementioned POC exploits, as well as one that we discovered recently. As soon as we understood their underlying concept, we started to develop our own exploit based on our abstracted code. Also, we initially disabled ASLR using the `aslr=False` argument in the `start()` function of `exploit_heapcorruption-advanced.py`.

As described above, we assume a fixed heap layout in which the `service_user` struct is located right after the overflowable `user_args` buffer. Therefore, we only need to construct a proper overflow payload in the exploit to ultimately load a attacker-controlled *evil* shared library.

To better understand the exploit below, it is worth studying the function `nss_load_library()`, which can be found in listing 3. First of all, the `nss_load_library()` function checks if `ni->library->lib_handle == NULL` holds. Therefore in our exploit, we need `ni->library` to point to a location where `ni->library->lib_handle` is a `NULL` pointer. Later on, in line 13 the `name` of the `service_user` struct is used to craft the name of the external library that is loaded. In our exploit we therefore overwrite the `ni->name` of the `service_user` with the string `X/X`[1]. In lines 10-15 this name is used to craft the name of the external library and we are able to load our evil library `libnss_X/X.so.2` instead of e.g., the `libnss_systemd.so.2` library. In line 16 our evil library is then loaded and the attackers code from `libnss_X/X.so.2` is executed.

```
   // https://code.woboq.org/userspace/glibc/nss/nsswitch.c.html
 2 static int
   nss_load_library(service_user *ni)
 4 {
     if (ni->library->lib_handle == NULL)
 6   {
       /* Load the shared library.  */
 8     size_t shlen = (7 + strlen(ni->name) + 3 + strlen(__nss_shlib_revision) + 1);
       int saved_errno = errno;
10     char shlib_name[shlen];
       /* Construct shared object name.  */
12     __stpcpy(__stpcpy(__stpcpy(__stpcpy(shlib_name,
                                           "libnss_"),
14                                   ni->name),
                            ".so"),
16          __nss_shlib_revision);
       ni->library->lib_handle = __libc_dlopen(shlib_name);
18     if (ni->library->lib_handle == NULL)
       {
20       /* Failed to load the library.  */
         ni->library->lib_handle = (void *)-1l;
22       __set_errno(saved_errno);
       }
24   }
   }
```

Listing 3: `nss_load_library()` function from `nsswitch.c`.

The code of the evil library is found in `libnss_X/X.c` and additionally given in in listing 4. The code is pretty straightforward and the only thing worth mentioning here is the function attribute `__attribute__((constructor))`. The functions exported with it are executed immediately after calling `dlopen()`, which means that no explicit call to our malicious function is required.

[1]Here we use a `/` to link to a library located below the current working directory.

Next, we constructed our payload, by first analysing the heap layout and all offsets between the variables, and then overflowing the buffer with the corresponding padding and data. As already mentioned, if an argument ends with \, then it writes \x00, the next argument and a trailing space, before the loop continues with the next argument. For instance the arguments `'aaa\' 'bbb'` will be written into `user_args` as `'aaa\x00bbb bbb '`. Therefore we can duplicate strings to get a large overflow and use \ to write zeros.

On the heap the following variables are allocated:

1. `user_args`: the size of it depends on the size of the arguments passed

2. `ni`: a `service_user` instance

3. `ni->library`: a `service_library` instance

The payload first adds a padding, which uses a \ to duplicate the argument after it, and then some random characters to fill the space up to the next chunk on the heap. Then we overwrite the `ni` chunk. When overwriting the beginning of a chunk, we overwrite the chunk size metadata with the original value[2], to ensure we don't encounter a `SEGFAULT` on a subsequent `malloc`. Here, it is worth mentioning that `dlopen()` uses `malloc` internally, so not messing up the heap is mandatory. Then we add four single backslashes to produce a `NULL` pointer, which we can use for `ni->library->lib_handle`[3]. After some more padding, we overwrite the `ni->library` pointer with the address of the `NULL` pointer minus 4, so `ni->library->lib_handle` is `NULL`. At the end, we set `ni->name` to `X/X` and overwrite the next chunk size with its original value.

Since our payload and exploit already works without ALSR, the only thing missing is to make it compatible with ASLR. We do this by looping our exploit that works without ASLR until the mapping of the heap addresses with ASLR match the ones in GDB without ASLR. Since we are on 32-bit this only takes a few seconds and our exploit terminates successfully.

[2] `0x00000031` for `ni` and `0x00000011` for `ni->library`

[3] Four backslashes produce a lot of garbage after them, because each of them copies all arguments after it until it reaches an argument that does not end with \. Therefore we add the `NULL` pointer at this location, where we have a lot of space we don't need for the rest of the exploit.

```c
#include <errno.h>
#include <fcntl.h>
#include <stdio.h>

// https://gcc.gnu.org/onlinedocs/gcc/Function-Attributes.html
static void constructor() __attribute__((constructor));

// https://man7.org/linux/man-pages/man3/dlopen.3.html
void constructor()
{
    printf("Shared library loaded\n");
    char *filepath = "/home/privileged/0xBADCODED";
    printf("Create file %s: ", filepath);
    if (open(filepath, O_CREAT | O_EXCL) == -1)
        printf("%m\n");
    else
        printf("Done\n");
}
```

Listing 4: Attacker-controlled evil library `libnss_X/X.c`.