Analysis on Heap based buffer overflow (CVE 2021- 3156)

Senarathna K.M.G.S.B – IT19118932

Cyber security – Information systems engineering department

Sri Lanka Institute of Information Technology (Malabe, Sri Lanka)

it19118932@my.sliit.lk

Abstract

In almost all Linux/Unix systems "Sudo" utility is used to grant non-privileged users the maximum level of permissions as "root" privileged users. CVE 2021-3156 is a vulnerability where Sudo versions before 1.9.5p2 contains a vulnerability that leads to heap-based buffer overflow which will eventually allow a non-privileged user to privilege escalation to root user level. As it is required for the secure software systems assignment guidelines, this report contains an analysis on the above-mentioned vulnerability. [1]

The initial part of this research document contains an introduction to the technical details and information about the vulnerability, and a discussion on how heap-based buffer overflows occur. In the introduction part, what is buffer overflow, what is heap-based buffer overflow and how this Sudo vulnerability is connected with the heap-based buffer overflow are discussed.

In the middle part of the document, it is described in technical details how this vulnerability allows non-privileged users to elevate their privileges to root level. In addition, impacts of this CVE 2021-3156 vulnerability and how it affects systems are described.

In the latter part of this report, a description on already suggested mitigations for this vulnerability and a discussion about the future influences of the vulnerability is included.

Keywords: Heap; Overflow; Vulnerability; Privilege escalation; root; exploit.

I. Introduction

This vulnerability has been implemented in July 2011 (commit 8255ed69) and it was identified initially in 2021. Therefore, this has been hidden for almost 10 years without anyone noticing. CVE 2021-3156 which is Heap-based buffer overflow, is also identified as "Baron Samedit" was initially discovered by a research team at Qualys in January 2021 and reported it to Sudo developers.

Sudo is a utility that is used in almost all Linux/Unix systems to grant permissions for users that does not possess "root" privileges in the system. This vulnerability was discovered in Sudo versions released before1.9.5.5p2. And it affects all legacy versions from 1.8.2 to 1.8.31p2 in their default configuration, as well as all stable

versions from 1.9.0 to 1.9.5p1. Since many Linux distros including Debian, Fedora, Gentoo, Ubuntu, and SUSE that was released before 2021 uses Sudo legacy versions mentioned above, this was considered as a severe vulnerability that affects data confidentiality and integrity of systems and as well as system availability.

An analysis on CVSS (Common Vulnerability Scoring System) that was published in redhat.com is included for the reference.

	Red Hat	**NVD**
CVSS v3 Base Score	7.8	7.8
Attack Vector	Local	Local
Attack Complexity	Low	Low
Privileges Required	Low	Low
User Interaction	None	None
Scope	Unchanged	Unchanged
Confidentiality	High	High
Integrity Impact	High	High
Availability Impact	High	High
CVSS v3 Base Score	7.8	7.8

[2]

According to the CVSS (Common Vulnerability Scoring System) analysis, it is discovered that this heap-based buffer overflow vulnerability affects all three components of CIA (confidentiality, Integrity, Availability) in an equal manner. And the risk is indicated high for all the three components.

II. Memory segments/ Memory Layout

Every compiled program's memory is divided into five memory segments. Those five memory segments are,

- text
- data
- bss (Uninitialized data)
- heap
- stack

These each segment is a segment of memory which has been allocated for a specific purpose. Fig.1 below shows an explanation on memory segments.

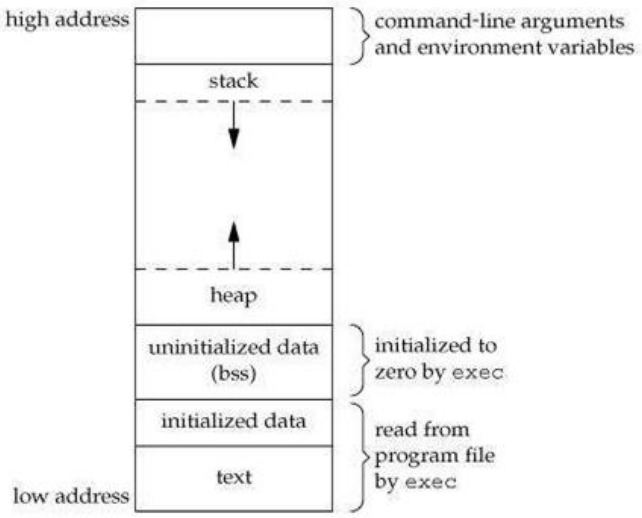

Fig.1 Memory segments

In regarding to the Heap based buffer overflow that is discussed in this research

paper, most important memory layout components are heap and stack.

Stack – Stack segment, which is variable sized, is used to store different local function variables and context during function calls. Since **EIP** is changing when a function is called, stack is a frequently used valuable memory segment to remember all the variables and the location where EIP should return after executing the called function.

Heap – Heap is also a variable memory segment which is not automatically managed similar to stack. It can be allocated and used by the programmer for whatever data that programmer needs to store or used in a program. In addition, heap only allows accessing stored variables globally. Heap grows towards higher memory addresses which is the opposite direction of the stack growth.

III. Buffer overflow

Buffer – A buffer is a memory segment which is used as a temporary memory location that stores data/information while a function or a program is in execution.

A buffer overflow vulnerability is when a buffer is given an excessive amount of data inputs, it overwrites the adjacent data locations in memory passed the boundary of the buffer, which will corrupt the original data. This can lead the program to give an error or to execute in an unpredictable way than it was originally programmed.

Both **C** and **C++** programming languages are vulnerable for buffer overflow attacks since they lack in safeguards regarding to overwriting. Therefore, **Linux Operating systems, Windows** and **Mac OSX** are vulnerable for these attacks since most of them are written using C and C++ programming languages.

Buffer overflow vulnerability mainly has two types of buffer overflow methods which are,

- Stack based buffer overflow attacks
- Heap based buffer overflow attacks

IV. Stack based buffer overflow

As it was described earlier stack is a very frequently used memory segment to store variable and to remember the location that EIP register should return after execution of a different stack frame.

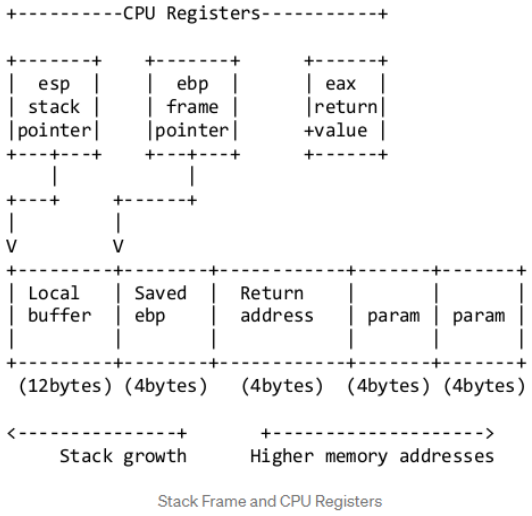

Fig.2 Stack frame and CPU registers

In stack-based buffer overflow an overwriting of these return memory address locations are happened. Because of that overwritten malicious addresses, the

programs EIP could be directed to a different location that contains malicious instructions to execute.

Since this research paper is focused on Heap-based buffer overflow, stack-based buffer overflow is not discussed in a detailed manner.

V. Heap-based buffer overflow and CVE 2021-3156 (Sudo Baron Samedit)

As discussed earlier, heap stores data allocated dynamically by a programmer or by a program at runtime of the program and heap contains program data related to current execution. In addition, it is considered that all global variables are stored inside the heap. Heap memory area is not considerably managed automatically by the CPU (*Central Processing Unit*) unlike the stack.

Dynamic memory allocation of heap is done through few means.

malloc() function – "malloc()" is known as memory allocation. By using this function, a block of memory is reserved inside the heap for the mentioned size of bytes inside the syntax. [3] [4]

Syntax:

```
ptr = (castType*) malloc(size);
```

Ex:

```
ptr = (int*) malloc (200* sizeof(int));
```

(This allocates 200 elements inside memory block which is 4 bits (size of int). Therefore 800 bits are allocated.)

calloc() function – "calloc" is known as contiguous allocation. By using this function, a programmer can allocate memory inside heap and keep the bits of memory blocks initialized with the value zero. This is mainly used to avoid initialization of memory block bits with garbage values by malloc() function.

Syntax:

```
ptr = (castType*) calloc(n, size);
```

Ex:

```
ptr = (int*) calloc(25, sizeof(int));
```

(This allocates 25 elements of contiguous memory blocks which contains of 4 bits (size of int). Therefore, 100 bits are allocated)

realloc() function – initial dynamically allocated could be insufficient. In such incidents, realloc() is used to chance the initially allocated memory block sizes.

Syntax:

```
ptr = realloc(ptr, newSize);
```

(This will reallocate the pointer ptr with the newly mentioned size in the syntax.)

Since heap does not automatically de-allocate the initially allocated memory blocks similar to stack, the relevant pointers of allocated memory blocks need to be de-allocated properly in a manual manner.

For this de-allocation, **free()** function is used.

Syntax:

```
free(ptr);
```

Data allocated using above-mentioned methods and structure pointers reside in memory blocks at heap area. A heap overflow is performed using changing or overwriting these original program data inside the heap, which will lead the program to crash or for an abnormal behavior.

In addition, with heap overflow, it can be used to overwrite pointer addresses which will open and run structures and functions that the original program behavior does not intend to. These structures could be attacker's own scripts or encapsulated functions that need special privileges to execute within the program.

```
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <stdio.h>
#include <sys/types.h>

struct data {
  char name[64];
};

struct fp {
  int (*fp)();
};

void winner()
{
  printf("level passed\n");
}

void nowinner()
{
  printf("level has not been passed\n");
}

int main(int argc, char **argv)
{
  struct data *d;
  struct fp *f;

  d = malloc(sizeof(struct data));
  f = malloc(sizeof(struct fp));
  f->fp = nowinner;

  printf("data is at %p, fp is at %p\n", d, f);

  strcpy(d->name, argv[1]);

  f->fp();
}
```

Fig.3 Vulnerable C code [5] [6]

In Fig.3 there is a vulnerable C source code for a program that could be exploited using Heap based buffer overflow. In this protostar exercise [5], what is expected to be achieved from the exploitation is to execute the function "winner" by overflowing the heap memory blocks allocated during the runtime of the program.

There will be two contiguous memory blocks allocated within the heap for the,

```
d = malloc(sizeof(struct data));
f = malloc(sizeof(struct fp));
```

code lines. Since "winner()" function is not called inside the main program, to execute the function memory block allocated for nowinner() function should be overflowed. To overflow the nowinner() function,

```
strcpy(d->name, argv[1]);
```

command could be used. Commands "strcpy", "strcat" and "strcmp" are considered as vulnerable commands in C language because they do not check for buffer lengths which could lead to overwrite memory blocks. [7] To analyze the change of memory with the number of inputs, gdb debugger can be used. In gdb, address of the starting point and endpoint of heap memory is located with `info proc map`. Then, the heap memory blocks can be examined with

`x/200x 0x804c000` (*0x804c000 is an example memory address*) inside gdb to read the changes when the out-of-bound characters are input. While running the program, examine process can be done by increasing the number of characters inputted. At the point of overflow, it will display the error, segmentation fault. In this particular C code (in Fig.3) that was taken as an example,

the number of characters that needed to overflow the heap was 80. In addition with a piped `grep` command in an `objdump` command the exact memory address of "nowinner()" could be retrieved.

```
$ objdump -t heap0 | grep winner
08048464 g     F .test  00000014              winner
08048478 g     F .test  00000014              nowinner
```

Finally by a python script, the address `0x0804878` (nowinner) will be passed concatenated with 72 other characters to overflow the heap memory block allocated to winner() function.

```
$ ./heap0 `python -c 'print "A"*72+"\x64\x84\x04\x08"'`
data is at 0x804a008, fp is at 0x804a050
level passed
```

With this script, EIP will be overwritten to the address of winner() function and it will execute the function. Similarly, an actor with malicious intent can overflow the heap memory area of a running program and execute malicious functions that cause abnormal behavior of the program.

In CVE 2021-3156 vulnerability which is known as sudo baron samedit, there is a defect in the shell code of the source code of sudo utility releases from 1.8.2 to 1.8.31p2 and from 1.9.0 through 1.9.5p1. This new code was introduced in July 2011. Therefore, this vulnerability has remained undiscovered for 10 years until it was discovered the Qualys research team in January 2021. [8] CVE 2021-3156 requires initial access to a local user in the vulnerable system for exploitation. [9] In addition, Linux distributions such as Ubuntu 20 (Sudo 1.8.31),), Fedora 33 (Sudo 1.9.2), Debian 10 (Sudo 1.8.27) and in random occasions, Mac os are discovered to be vulnerable. [9]

Initially to identify or to exploit the vulnerability in sudo's source code, it must run in its "shell" mode through `-s` option. Substantially shell mode's primary purpose is to effectively communicate with interactive subprocesses. [10] In `main()` function, there is `parse_args()` (*lines 590-591*)which escapes all meta characters using "\\" characters. (Fig.4)

```
for (src = *av; *src != '\0'; src++) {
    /* quote potential meta characters */
    if (!isalnum((unsigned char)*src) && *src != '_' && *src !
        *dst++ = '\\';
    *dst++ = *src;
}
```

Fig.4 escaping meta characters

But later for "sudoers matching and logging purposes" in `sudoers_policy_main()` and `set_cmnd()` concatenates arguments to `usr_args`(*lines 864-871*)which is a heap based buffer. If the passed command line argument ends with a "\", then it leads to copying of out-of-bounds characters to the `usr_args` buffer making the `set_cmnd()` vulnerable for heap-based buffer overflow.

As Qualys research team mentions, there were additional conditions that surrounding the vulnerable set_cmnd() to bypass in order to overflow. [1]

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

        if (ISSET(sudo_mode, MODE_SHELL|MODE_LOGIN_SHELL)) {
```

Fig.5 additional conditions (*lines 819,958*)

In order to reach the vulnerable code, it should be executed in `MODE_SHELL` with `MODE_EDIT` or `MODE_CHECK`. The suggestion of Qualys research was to run the sudo as `sudoedit` instead of `sudo` which automatically sets the mode to MODE_EDIT (*line 270*). Therefore, the command that should execute is

```
sudoedit -s
```

to enable both `MODE_EDIT` and `MODE_SHELL`. In addition, to bypass the escape of meta characters, arguments end with a single backslash ("\") should be passed. [1]

```
sudoedit -s '\' `perl -e 'print "A" x 65536'`
malloc(): corrupted top size
Aborted (core dumped)
```

Python script:

```
sudoedit -s '\' $(python3 -c 'print("A"*1000)')
```

With this overflow, attackers can manage the size of `usr_args` buffer the way they intend and the content of the overflow. Suggested exploits represents that using the defect of heap overflow in this code, attackers can escalate the privileges of a local user to a level of root user.

Ex: By LiveOverflow at Youtube.com suggests few exploitation methods for the vulnerability CVE 2021-3156. (How SUDO on Linux was HACKED! CVE-2021-3156)(https://www.youtube.com/watch?v=TLa2VqcGGEQ) [11]

A. Detection and mitigation

To identify if a system/operating system is vulnerable for CVE 2021-3156,

```
sudoedit -s /
```

command can be used. If the system is vulnerable, it displays and error starting with `sudoedit`. `[9]` In addition for detailed detection with high technical aspects, AFL-fuzzing (Americal Fuzzy Lop) can be used. [11]

In sudo versions, from 1.9.5p2 sudoe vendors have patched the vulnerability as a mitigation for the CVE 2021-3156.

VI. Conclusion

Despite the fact that heap-based buffer overflows are uncommon when comparing against stack-based buffer overflows, the potential harm that can cause by a heap overflow is not negligible. In this brief review it is described in a detailed and technical manner how this heap overflows occurs and how to exploit the vulnerability.

Additionally, a technical analysis on CVE 2021-3156, a specific sudo-based vulnerability was discussed. In the analysis, it was described how the heap-based buffer overflowing can affect a critical utility such as sudo that leads for escalation of privileges of a local user to the root level. Therefore, by referring this review on heap-based buffer overflow and CVE 2021-3156, a reader can obtain an idea about the vulnerability's behavior and the exploitation methods fulfilling this review paper's objectives.

VII. Acknowledgement

I am grateful for Dr. Lakmal Rupasinghe and Ms. Chethana Liyanapathirana for granting me the opportunity to engage in this analysis and for continuous guidance for the research paper.

I am indebted for all the skilled researchers of Qualys research team for discovering the CVE 2021-3156 vulnerability which has been remained undiscovered for almost 10 years and for the clear documentations provided.

Additionally, I am grateful for all the original authors of research publications, GitHub repositories, blogs, YouTube videos and other electronic sources that has been used in this review as resources for creating resourceful content.

References

[1] A. Jain, "qualys.com," 26 January 2021. [Online]. Available: https://blog.qualys.com/vulnerabilities-research/2021/01/26/cve-2021-3156-heap-based-buffer-overflow-in-sudo-baron-samedit. [Accessed 12 May 2021].

[2] Redhat.com, "Redhat.com," 21 March 2021. [Online]. Available: https://access.redhat.com/security/cve/cve-2021-3156. [Accessed 11 May 2021].

[3] ""The Malloc Maleficarum"," packetstormsecurity.com, 2005.

[4] blackngel, ""MALLOC DES-MALEFICARUM"," phrack.org, 2009.

[5] z3tta, "github.com," 29 Novemebr 2014. [Online]. Available: https://github.com/z3tta/Exploit-Exercises-Protostar/blob/master/13-Heap0.md. [Accessed 01 June 2021].

[6] P. Kacherginsky, "exploit-exercises-protostar-heap/##heap-0, https://iphelix.medium.com/#heap-0," Medium.

[7] "Common vulnerabilities guide for C programmers," CERN Computer Security, 2021.

[8] "https://www.cirt.gov.bd/cve-2021-3156-heap-based-buffer-overflow-in-sudo/," Bangladesh Computer Council, Bangladesh.

[9] S. Chierici, "How to detect sudo's CVE-2021-3156 using Falco," sysdig, January 28, 2021.

[10] swiat, "Preventing the exploitation of user mode heap corruption vulnerabilities," msrc-blog.microsoft.com, August 4, 2009.