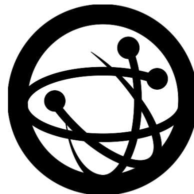
Source Code Audit on libcap
for Open Source Technology Improvement Fund (OSTIF)

Final Report and Management Summary

2023-05-10

X41 D-SEC GmbH
Krefelder Str. 123
D-52070 Aachen
Amtsgericht Aachen: HRB19989

https://x41-dsec.de/
info@x41-dsec.de



Organized by the Open Source Technology Improvement Fund

Revision	*Date*	*Change*	*Author(s)*
1	2023-04-14	Final Report Draft	MSc. H. Moesl, Dipl.-Ing. D. Gstir, R. Weinberger, and M. Vervier
2	2023-05-10	Report Marked For Publication	M. Vervier

Contents

1 Executive Summary **4**
1.1 Findings Overview 6

2 Introduction **7**
2.1 Scope 7
2.2 Coverage 8
2.3 Recommended Further Tests 10

3 Rating Methodology for Security Vulnerabilities **11**
3.1 Common Weakness Enumeration 12

4 Results **13**
4.1 Findings 14
4.2 Informational Notes 18

5 About X41 D-Sec GmbH **21**

A Fuzzing Harnesses **23**
A.1 CLI Fuzzing of Tools capsh 23
A.2 CLI Fuzzing of Tools getcap, setcap, getpcaps 24
A.3 Fuzzing Harness for cap_copy_int, cap_from_text, cap_to_text, cap_from_name . . 24
A.4 Other Test Harnesses 27

Dashboard

Target	
Customer	Open Source Technology Improvement Fund (OSTIF)
Name	libcap
Type	Source Code
Version	commit (5496a0e3854dba9374823e9b561ee8c5fd9c59f4)

Engagement	
Type	Source Code Audit
Consultants	3: MSc. H. Moesl, Dipl.-Ing. D. Gstir, and R. Weinberger
Engagement Effort	15 person-days, 2023-03-17 to 2023-04-12

Total issues found	2

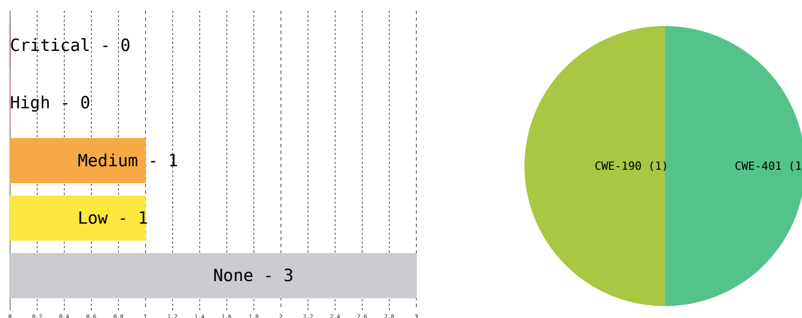


Figure 1: Issue Overview (l: Severity, r: CWE Distribution)

1 Executive Summary

In March and April 2023, X41 D-Sec GmbH performed a source code audit against libcap to identify security vulnerabilities. The test was made possible by the Open Source Technology Improvement Fund[1].

A total of two vulnerabilities were discovered during the audit by X41. None were rated as having a critical or high severity, one as medium, and one as low. Additionally, three issues without a direct security impact were identified.

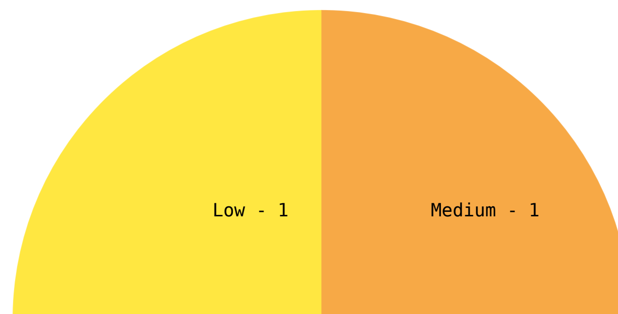


Figure 1.1: Issues and Severity

libcap is a library for getting and setting POSIX.1e (formerly POSIX 6) draft 15 capabilities, with natively supported languages of C/C++ and Go. As it is common for compiled languages, a particular focus has been placed on the identification of typical memory safety issues such as buffer overflows, memory disclosure, or use-after-free. Furthermore, the testing team put focus and effort into the identification of logic vulnerabilities such as disagreement between kernel and libcap

[1] https://ostif.org

shadow state.

In a source code audit, the testers receive all available information about the target. The test was performed by three experienced security experts between 2023-03-17 and 2023-04-12.

Despite multiple auditors independently reviewing the same section of the code for better coverage, only two vulnerabilities have been identified during the review process. Therefore, addressing these vulnerabilities can add an additional layer of defense and help reduce the potential attack surface of the library.

To further improve the security posture, it is encouraged to implement additional security controls, which have been listed as part of the Informational Notes list. These describe potential improvements with regards to code safety which make exploiting potential bugs harder.

It is worth emphasizing that the libcap library was put through rigorous testing by X41's team and, overall, it was found to be highly robust and secure. The testing process revealed that the libcap is designed and implemented with great care and attention to detail, and it demonstrated a strong ability to withstand scrutiny. As a result, the overall impression and outcome of this security assessment is very positive.

Again, it must be reiterated that this assessment provided valuable insights into the security posture at the time of testing, but it is important to note that any penetration test is unable to guarantee that the software complex is free of additional bugs.

In terms of dynamic testing, the testing team developed several fuzz testing harnesses. Fuzz testing is, in general, essential for the overall security of the libcap project, especially since it is implemented in C, which is often prone to memory corruption vulnerabilities. For the purpose of this test, code coverage driven fuzzing using AFL++ in combination with address space sanitizers (such as ASAN) has been performed. Even though no new issues were identified using the developed harnesses, it is highly recommended to incorporate the developed harnesses into the libcap project and to make fuzzing a fixed part of the libcap, either using AFL++ or libFuzzer, resulting in better testing coverage.

1.1 Findings Overview

DESCRIPTION	SEVERITY	ID	REF
Memory Leak on func pthread_create() Error	LOW	LCAP-CR-23-01	4.1.1
Integer Overflow in func _libcap_strdup()	MEDIUM	LCAP-CR-23-02	4.1.2
Missing Check of func pam_set_data() Return Code	NONE	LCAP-CR-23-100	4.2.1
Missing Permission Check of User Capability File	NONE	LCAP-CR-23-101	4.2.2
Problematic Usage of exttt pthread_kill()	NONE	LCAP-CR-23-102	4.2.3

Table 1.1: Security-Relevant Findings

2 Introduction

The assessment comprised a security review of the libcap library, utilizing static source code analysis as well as dynamic testing using dedicated fuzz testing harnesses. The branch in scope for this inspection was the ***main*** branch with the commit id ***5496a0e3854dba9374823e9b561ee8c5fd9c59f4***.

At the beginning of the project, an initial kick-off meeting was set up between X41, OSTIF and the maintainer of the library in order to to align on the scope of this engagement. The meeting helped to clarify the expectations of this assessment and also narrowed down the key focus areas.

Since the testing team had a clear understanding of the scope and goals, X41 did not need to interact extensively with OSTIF or the maintainer. However, the communication was excellent, and help was provided whenever requested. Generally speaking, OSTIF as well as the maintainer of the library deserve a lot of praise for their overall support and assistance. It was a pleasure for the testing team working with them.

From a programming style and software design perspective the code and design is clean and very well written, with security in mind.

2.1 Scope

During the kickoff call, the audit team, in collaboration with OSTIF and the maintainer, defined and narrowed down the scope of the testing efforts. It was mutually agreed that the primary focus for libcap would be on conducting in-depth static code analysis and incorporating fuzzing support for various functions of interest.

In addition to that, the assessment focused a general examination for typical memory corruption vulnerabilities.

2.2 Coverage

A security assessment attempts to find the most important or sometimes as many of the existing problems as possible, though it is practically never possible to rule out the possibility of additional weaknesses being found in the future.

A manual approach for code review is used by X41. This process was combined with fuzzing given the nature of libcap being exposed to parsing of potentially untrustworthy data.

The time allocated to X41 for this code review was sufficient to yield a reasonable coverage of the given scope.

2.2.1 Fuzzing

While conducting a source audit of the libcap library it was observed that the project did not include any fuzz tests. As a result, our team decided to focus on fuzzing as a critical aspect of our testing efforts.

For the fuzzing efforts, AFL++[1] was used in two ways / modes:

1. *argv* (command-line) fuzzing of the libcap tools utilizing AFL++ persistent mode
2. Fuzzing of selected interesting looking functions of libcap utilizing AFL++ persistent mode

Persistent mode fuzzing is a feature in the AFL++ fuzzer that keeps the target program running in the background and continuously feeds it with new test cases. This is in contrast to the default "one-shot" mode in which the fuzzer launches the target program with each new test case. By utilizing this approach, it was possible to achieve execution speed improvements of 10 to 20 times. Moreover, AFL++ has recently incorporated support for command-line interface (CLI) fuzzing in persistent mode through the ***AFL_INIT_ARGV_PERSISTENT*** macro, rendering it an ideal choice for the CLI fuzzing of libcap.

2.2.1.1 Fuzzing Hardware

The fuzzing process was carried out on a system equipped with an AMD Ryzen Threadripper Processor, which boasts 64 cores and 128GB of RAM.

[1] `https://github.com/AFLplusplus/AFLplusplus/`

2.2.1.2 CLI Fuzzing

Considering that libcap comprises various tools, such as *capsh*, *getcap*, *setcap* and *getpcaps*, as a component of its code base, X41, decided to conduct command-line interface (CLI) fuzzing against these tools to detect any bugs associated with the parsing of *argv* parameters.

To conduct CLI fuzzing on each of the aforementioned tools, X41 generated a valid set of CLI parameters and utilized them as input test cases for the fuzzer. Additionally, X41 built the code base with address sanitization enabled (*-fsanitize=address*) to detect any memory management errors. To facilitate the fuzzer in quickly finding valid CLI parameters, X41 configured the AFL++ compiler to create a dictionary using the *AFL_LLVM_DICT2FILE* flag based on the compiled C code. As a final measure, it was necessary to perform fuzzing within a containerized environment. This was important to prevent any changes to capabilities resulting from the fuzzing process that could leave the host machine in an unknown or compromised state.

The fuzzer executed each of the aforementioned tools for a total of approximately 5 billion times. Despite the extensive number of executions, X41 was unable to identify any immediate crashes. Given that it can be concluded that the code base for these tools and libcap is well tested and written with security in mind, at least under the conditions and parameters we used for the fuzzing process.

However, it is worth noting that the absence of immediate crashes does not necessarily imply that the code is free from bugs or vulnerabilities.

2.2.1.3 Fuzzing Selected Functions

During this project, X41 discovered additional noteworthy functions that were not included in the fuzzing efforts of OSS-Fuzz and created dedicated fuzzing harnesses:

- Functions working on external capability representation:
  - ***cap_copy_int***
  - ***cap_copy_int_check***
- Functions doing string manipulation:
  - ***cap_to_text***
  - ***cap_from_text***
  - ***cap_from_name***
- Other interesting functions:

  - ***read_capabilities_for_user***

The folders *tests/fuzzinput* and *tests/fuzznames* already contained input test cases provided by OSS-Fuzz, which were utilized as a starting point for generating test cases using the *radamsa* tool[2]. X41 compiled the source code base with address sanitization enabled (-*fsanitize=address*).

The fuzzing harnesses executed each of the aforementioned functions approximately 3 billion times, but no memory memory corruptions or crashes were detected during this process.

Fuzzing the function *are_se_sortlist* resulted in multiple crashes.

2.3 Recommended Further Tests

X41 recommends to subject all newly developed code to regular source code audits.

Due to the widespread usage of libcap, it is encouraged to perform recurring security audits as it has already been performed in the past, because new vulnerabilities may be introduced as more features are added and also changes within one part of the system may have unintentional security impact to other parts.

[2] `https://gitlab.com/akihe/radamsa`

3 Rating Methodology for Security Vulnerabilities

Security vulnerabilities are given a purely technical rating by the testers as they are discovered during the test. Business factors and financial risks for Open Source Technology Improvement Fund (OSTIF) are beyond the scope of a penetration test which focuses entirely on technical factors. Yet technical results from a penetration test may be an integral part of a general risk assessment. A penetration test is based on a limited time frame and only covers vulnerabilities and security issues which have been found in the given time, there is no claim for full coverage.

In total, five different ratings exist, which are as follows:

Severity Rating
None
Low
Medium
High
Critical

A low rating indicates that the vulnerability is either very hard for an attacker to exploit due to special circumstances, or that the impact of exploitation is limited, whereas findings with a medium rating are more likely to be exploited or have a higher impact. High and critical ratings are assigned when the testers deem the prerequisites realistic or trivial and the impact significant or very significant.

Findings with the rating 'none' are called side findings and are related to security hardening, affect functionality, or other topics that are not directly related to security. X41 recommends to mitigate these issues as well, because they often become exploitable in the future. Doing so will strengthen the security of the system and is recommended for defense in depth.

3.1 Common Weakness Enumeration

The CWE[1] is a set of software weaknesses that allows the categorization of vulnerabilities and weaknesses in software. If applicable, X41 provides the CWE-ID for each vulnerability that is discovered during a test.

CWE is a very powerful method to categorize a vulnerability and to give general descriptions and solution advice on recurring vulnerability types. CWE is developed by *MITRE*[2]. More information can be found on the CWE website at `https://cwe.mitre.org/`.

[1] Common Weakness Enumeration
[2] `https://www.mitre.org`

4 Results

This chapter describes the results of this test. The security-relevant findings are documented in Section 4.1. Additionally, findings without a direct security impact are documented in Section 4.2.

4.1 Findings

The following subsections describe findings with a direct security impact that were discovered during the test.

4.1.1 LCAP-CR-23-01: Memory Leak on *pthread_create()* Error

Severity:	LOW
CWE:	401 – Improper Release of Memory Before Removing Last Reference ('Memory Leak')
Affected Component:	libcap/psx/psx.c:__wrap_pthread_create()

4.1.1.1 Description

X41 found that the error handling in ***__wrap_pthread_create()*** function is wrong and will leak memory in case of an error.

Function ***libpsx*** hooks the ***pthread_create()*** function and replaces it with ***__wrap_pthread_create()***. This wrapping function will then register the required signal handler and call the actual ***pthread_create()*** (***__real_pthread_create()***). Here, the error handling for ***__real_pthread_create()*** is faulty as it checks for a negative return value which cannot happen. Instead, ***pthread_create()*** will return a value ***> 0*** in case of an error[1]. Thus, for every error in ***__real_pthread_create()*** where the tread routine (***_psx_start_fn***) is not called, the buffer ***starter*** will not be freed and thus this memory will be leaked once ***__wrap_pthread_create()*** returns.

A malicious actor who is in the position to cause ***__real_pthread_create()*** to return an error, can potentially abuse this to exhaust the process memory. As ***libpsx*** hooks all ***pthread_create()*** calls of a process, this affects every thread.

```
1  *
2   * __wrap_pthread_create is the wrapped destination of all regular
3   * pthread_create calls.
4   */
5  int __wrap_pthread_create(pthread_t *thread, const pthread_attr_t *attr,
6                void *(*start_routine) (void *), void *arg) {
7      psx_starter_t *starter = calloc(1, sizeof(psx_starter_t));
8
9      // [...]
```

[1] https://man7.org/linux/man-pages/man3/pthread_create.3.html

```
10
11      int ret = __real_pthread_create(thread, attr, _psx_start_fn, starter);
12      if (ret == -1) {
13          psx_new_state(_PSX_CREATE, _PSX_IDLE);
14          memset(starter, 0, sizeof(*starter));
15          free(starter);
16      } /* else unlock happens in _psx_start_fn */
17
18      /* the parent can once again receive psx interrupt signals */
19      pthread_sigmask(SIG_SETMASK, &orig_sigbits, NULL);
20
21      return ret;
22  }
```

Listing 4.1: Code Snippet Showing the Affected Part of ***__wrap_pthread_create()***

4.1.1.2 Solution Advice

While not critical, X41 advises fixing the error handling code to prevent any abuse from being possible.

4.1.2 LCAP-CR-23-02: Integer Overflow in *_libcap_strdup()*

Severity:	MEDIUM
CWE:	190 – Integer Overflow or Wraparound
Affected Component:	libcap/cap_alloc.c:_libcap_strdup()

4.1.2.1 Description

X41 found that in 32 bits execution mode, where ***sizeof(size_t)*** equals 4, the ***_libcap_strdup()*** function can suffer from an integer overflow of the input string is close to a length of 4GiB. In this case ***len = strlen(old) + 1 + 2*sizeof(__u32);*** will overflow and ***results*** into a value much smaller than 4GiB.

As consequence the overflow check *len & 0xffffffff) != len* will have no effect and the ***strcpy()*** function at the end of the function will overwrite the heap.

```
__attribute__((visibility ("hidden"))) char *_libcap_strdup(const char *old)
{
    struct _cap_alloc_s *header;
    char *raw_data;
    size_t len;

[...]

    len = strlen(old) + 1 + 2*sizeof(__u32);
    if (len < sizeof(struct _cap_alloc_s)) {
        len = sizeof(struct _cap_alloc_s);
    }
    if ((len & 0xffffffff) != len) {
        _cap_debug("len is too long for libcap to manage");
        errno = EINVAL;
        return NULL;
    }

    raw_data = calloc(1, len);

[...]

    strcpy(raw_data, old);
    return raw_data;
}
```

Listing 4.2: Code Snippet Showing the Affected Part of ***_libcap_strdup()***

4.1.2.2 Solution Advice

While the overflow is impossible to exploit on a pure 32 bits system because no user space application can use the whole 32 bits address space it might be possible on a 64 bits kernel in 32 bits compat mode. In this mode user space is allowed to use the full 32 bits address space. X41 advises checking whether ***strlen()*** returns a sufficient large number to overflow the addition.

4.2 Informational Notes

The following observations do not have a direct security impact, but are related to security hardening, affect functionality, or other topics that are not directly related to security. X41 recommends to mitigate these issues as well, because they often become exploitable in the future. Doing so will strengthen the security of the system and is recommended for defense in depth.

4.2.1 LCAP-CR-23-100: Missing Check of *pam_set_data()* Return Code

Affected Component:	pam_cap

4.2.1.1 Description

While reviewing the pam_cap.so source code X41 noticed that the return code of ***pam_set_data()*** is not checked. This function can fail and not detecting this failure can lead to undefined behavior.

```
static int set_capabilities(struct pam_cap_s *cs)
{
[...]
    if (cs->defer) {
        D(("configured to delay applying IAB"));
        pam_set_data(cs->pamh, "pam_cap_iab", iab, iab_apply);
        iab = NULL;
    } else if (!cap_iab_set_proc(iab)) {
        D(("able to set the IAB [%s] value", conf_caps));
        ok = 1;
    }
[...]
}
```

Listing 4.3: Code Snippet Showing the Affected Part of ***set_capabilities()***

4.2.1.2 Solution Advice

X41 recommends to always check the returned value for being ***PAM_SUCCESS*** and abort otherwise.

4.2.2 LCAP-CR-23-101: Missing Permission Check of User Capability File

Affected Component:	pam_cap

4.2.2.1 Description

While reviewing the pam_cap.so source code X41 noticed that the access permissions of the user capability file are not checked. Relaxed permission of that file could allow an attacker overwriting it and gaining capabilities at login time.

```
static char *read_capabilities_for_user(const char *user, const char *source)
{
[...]
    cap_file = fopen(source, "r");
    if (cap_file == NULL) {
        D(("failed to open capability file"));
        goto defer;
    }
[...]
}
```

Listing 4.4: Code Snippet Showing the Affected Part of read_capabilities_for_user()

4.2.2.2 Solution Advice

While not critical, X41 advises to check the DAC permissions of the file to make sure it is not world writable and owned by root.

4.2.3 LCAP-CR-23-102: Problematic Usage of `pthread_kill()`

Affected Component:

4.2.3.1 Description

While reviewing the psx source code X41 noticed that the code seems to assume that `pthread_kill()` returns an error code if a thread is no longer existent. Depending on the program logic this can result into undefined behavior since it is not guaranteed that `pthread_kill()` will work on terminated threads.

4.2.3.2 Solution Advice

While not critical, X41 advises to not assume that `pthread_kill()` will exhibit an error code if the targeted thread is gone.

5 About X41 D-Sec GmbH

X41 D-Sec GmbH is an expert provider for application security and penetration testing services. Having extensive industry experience and expertise in the area of information security, a strong core security team of world-class security experts enables X41 D-Sec GmbH to perform premium security services.

X41 has the following references that show their experience in the field:

- Review of the Mozilla Firefox updater[1]
- X41 Browser Security White Paper[2]
- Review of Cryptographic Protocols (Wire)[3]
- Identification of flaws in Fax Machines[4,5]
- Smartcard Stack Fuzzing[6]

The testers at X41 have extensive experience with penetration testing and red teaming exercises in complex environments. This includes enterprise environments with thousands of users and vendor infrastructures such as the Mozilla Firefox Updater (Balrog).

Fields of expertise in the area of application security encompass security-centered code reviews, binary reverse-engineering and vulnerability-discovery. Custom research and IT security consulting, as well as support services, are the core competencies of X41. The team has a strong technical background and performs security reviews of complex and high-profile applications such as Google Chrome and Microsoft Edge web browsers.

X41 D-Sec GmbH can be reached via `https://x41-dsec.de` or `mailto:info@x41-dsec.de`.

[1] https://blog.mozilla.org/security/2018/10/09/trusting-the-delivery-of-firefox-updates/
[2] https://browser-security.x41-dsec.de/X41-Browser-Security-White-Paper.pdf
[3] https://www.x41-dsec.de/reports/Kudelski-X41-Wire-Report-phase1-20170208.pdf
[4] https://www.x41-dsec.de/lab/blog/fax/
[5] https://2018.zeronights.ru/en/reports/zero-fax-given/
[6] https://www.x41-dsec.de/lab/blog/smartcards/

Acronyms

CWE Common Weakness Enumeration 12

A Fuzzing Harnesses

For additional coverage, this section provides all test harnesses used during the fuzzing campaign of the ***libcap*** library.

A.1 CLI Fuzzing of Tools capsh

```
...

#ifndef SHELL
#define SHELL "/bin/bash"
#endif /* ndef SHELL */

#include "./capshdoc.h"

#include <limits.h>
#include "/AFLplusplus/utils/argv_fuzzing/argv-fuzz-inl.h"

...

ssize_t fuzz_len;
unsigned char fuzz_buf[1024000];

#ifndef __AFL_FUZZ_TESTCASE_LEN
    #define __AFL_FUZZ_TESTCASE_LEN fuzz_len
    #define __AFL_FUZZ_TESTCASE_BUF fuzz_buf
    #define __AFL_FUZZ_INIT() void sync(void);
    #define __AFL_LOOP(x) \
    ((fuzz_len = read(0, fuzz_buf, sizeof(fuzz_buf))) > 0 ? 1 : 0)
    #define __AFL_INIT() sync()
#endif

__AFL_FUZZ_INIT();

int main(int argc, char *argv[], char *envp[])
{
    pid_t child = 0;
    unsigned i;
    int strict = 0, quiet_start = 0, dont_set_env = 0;
```

```
33          const char *shell = SHELL;
34
35          __AFL_INIT();
36
37          unsigned char *fuzzbuf2 = __AFL_FUZZ_TESTCASE_BUF;
38          while (__AFL_LOOP(UINT_MAX))
39          {
40                  AFL_INIT_ARGV_PERSISTENT(fuzzbuf2);
41
42                  ... // remaining main function of acountry
43          }
44      }
```

Listing A.1: Fuzzer capsh Tools

A.2 CLI Fuzzing of Tools getcap, setcap, getpcaps

The test harnesses are very similar to the one presented for the tool capsh in A.1.

A.3 Fuzzing Harness for cap_copy_int, cap_from_text, cap_to_text, cap_from_name

```
1   //#include "cap_extint.c"
2
3   #include <stdio.h>
4   #include <stdlib.h>
5   #include <limits.h>
6   #include <unistd.h>
7
8   #include "libcap.h"
9
10  #define CAP_EXT_MAGIC "\220\302\001\121"
11  #define CAP_EXT_MAGIC_SIZE 4
12  const static __u8 external_magic[CAP_EXT_MAGIC_SIZE+1] = CAP_EXT_MAGIC;
13
14  struct cap_ext_struct {
15      __u8 magic[CAP_EXT_MAGIC_SIZE];
16      __u8 length_of_capset;
17      /*
18       * note, we arrange these so the caps are stacked with byte-size
19       * resolution
20       */
21      __u8 bytes[CAP_SET_SIZE][NUMBER_OF_CAP_SETS];
22  };
23
```

```
24     ssize_t fuzz_len;
25     unsigned char fuzz_buf[1024000];
26
27     #ifndef __AFL_FUZZ_TESTCASE_LEN
28
29         #define __AFL_FUZZ_TESTCASE_LEN fuzz_len
30         #define __AFL_FUZZ_TESTCASE_BUF fuzz_buf
31         #define __AFL_FUZZ_INIT() void sync(void);
32         #define __AFL_LOOP(x) \
33         ((fuzz_len = read(0, fuzz_buf, sizeof(fuzz_buf))) > 0 ? 1 : 0)
34         #define __AFL_INIT() sync()
35
36     #endif
37
38     __AFL_FUZZ_INIT();
39
40     //#define FUZZ_GENERATE_TESTCASE 1
41     //#define FUZZ_TEST_TESTCASE 1
42
43     #define CAP_COPY_INT 1
44     //#define CAP_FROM_TEXT 1
45     //#define CAP_FROM_NAME 1
46
47     void generate_testcase(const char *filename)
48     {
49     #ifdef CAP_COPY_INT
50         cap_t caps = cap_get_pid(1);
51
52         ssize_t size = cap_size(caps);
53         void *buffer = malloc (size);
54
55         ssize_t copy_size = cap_copy_ext(buffer, caps, size);
56
57         FILE *fp = fopen(filename, "wb");
58         if (fp)
59         {
60             fwrite(buffer + CAP_EXT_MAGIC_SIZE, 1, size - CAP_EXT_MAGIC_SIZE, fp);
61             fclose(fp);
62         }
63
64         free(buffer);
65     #elif CAP_FROM_TEXT
66         const char *txt = ⌋
       ↪  "cap_chown,cap_dac_override,cap_fowner,cap_fsetid,cap_kill,cap_setgid,cap_setuid,cap_setpcap,cap_net_bind_se
       ↪  ;
67
68         FILE *fp = fopen(filename, "wb");
69         if (fp)
70         {
71             fwrite(txt, 1, strlen(txt) + 1, fp);
72             fclose(fp);
73         }
```

```
74  #endif
75  }
76
77  void fuzz_test(unsigned char *buf, ssize_t len)
78  {
79  #ifdef CAP_COPY_INT
80      void *cap_ext = malloc(ssizeof(struct cap_ext_struct));
81
82      memcpy(cap_ext, external_magic, CAP_EXT_MAGIC_SIZE);
83      memcpy(cap_ext + CAP_EXT_MAGIC_SIZE, buf, len);
84
85      cap_t cap = cap_copy_int(cap_ext);
86      if (cap)
87      {
88          ssize_t textlen = 0;
89
90          char * text = cap_to_text(cap, &textlen);
91          printf("%s\n", text);
92
93          cap_free(text);
94          cap_free(cap);
95      }
96
97      free(cap_ext);
98  #elif CAP_FROM_TEXT
99      cap_t cap = cap_from_text(buf);
100     if (cap)
101     {
102         cap_free(cap);
103     }
104 #elif CAP_FROM_NAME
105     cap_value_t value = 0;
106     cap_from_name(buf, &value);
107 #endif
108 }
109
110 int main(int argc, char **argv)
111 {
112
113 #ifdef FUZZ_GENERATE_TESTCASE
114     generate_testcase("testcase");
115     return 0;
116 #endif
117
118 #ifdef FUZZ_TEST_TESTCASE
119     FILE *fp = fopen(argv[1], "rb");
120     if (fp)
121     {
122         ssize_t len = fread(fuzz_buf, 1, sizeof(fuzz_buf), fp);
123         fclose(fp);
124
125         printf("%ld bytes read\n", len);
```

```
126
127             fuzz_test(fuzz_buf, len);
128         }
129         return 0;
130     #endif
131
132     #ifdef __AFL_HAVE_MANUAL_CONTROL
133       __AFL_INIT();
134     #endif
135
136         unsigned char *fuzz_buf2 = __AFL_FUZZ_TESTCASE_BUF;
137
138         while (__AFL_LOOP(UINT_MAX))
139         {
140             ssize_t cur_fuzz_len = __AFL_FUZZ_TESTCASE_LEN;
141             ssize_t remainLen = ssizeof(struct cap_ext_struct) - CAP_EXT_MAGIC_SIZE;
142
143             if (cur_fuzz_len > remainLen) continue;
144
145             fuzz_test(fuzz_buf2, cur_fuzz_len);
146         }
147
148         return 0;
149     }
```

Listing A.2: Fuzzer ***getcap, setcap, getpcaps***

A.4 Other Test Harnesses

```
1     #include <stdio.h>
2     #include <stdlib.h>
3     #include <limits.h>
4     #include <unistd.h>
5      #include <fcntl.h>
6
7     extern char *read_capabilities_for_user(const char *user, const char *source);
8
9     int main(int argc, char **argv)
10    {
11        __AFL_INIT();
12
13        char *ret = read_capabilities_for_user("root", argv[1]);
14        if (ret)
15        {
16            free(ret);
17        }
18
```

```
19          return 0;
20      }
```

Listing A.3: Fuzzer read_capabilities_for_user()