ITP 30002-02 Operating System, 2023 Spring

Homework 2

21800030 구현우 <21800030@handong.ac.kr> , 22100600 이현서 <hslee@handong.ac.kr>

1. Overview of the program design

(1) Cimin

CIMIN, short for Crashing Input Minimizer, is a program that minimizes inputs that cause crashes in executable files and returns the minimized string that does not cause crashes. The program provides a function that removes some substrings from the input string to minimize crashes.

CIMIN can run not only with executable files but also with other programs. This is done by processing the input string and checking the output of the executable program, which is a typical process.

Through this process, CIMIN identifies the parts of the input string that cause crashes and minimizes them to help identify and fix security vulnerabilities. It can also be used to improve program stability.

(2) User interface

The command ./cimin -I crash -m “SECV” -o reduced ./a.out runs the cimin program, which takes the following values as arguments:

-I crash : Receive crash, an input file that can cause an error.

-m “SECV” : means that messages containing SECV as error messages should be avoided.

-o reduced : Use reduced as a place to store the minimized string.

./a.out : type crash input into the executable file a.out to see if an error occurs or not.

After executing the above command, the cimin program converts the crash file into a minimized string. At this time, cimin inputs the crash file into a.out to check whether an error occurs. If an error occurs, the minimized string is further shortened; if no error occurs, the minimized string is determined. Save this minimized string to a reduced file.

(3) Delta Debugging Algorithm

The algorithm consists of two main functions: **Reduce(t_m) and Minimize(t).**

The Reduce(t_m) function takes a program p and an input t_m, and aims to minimize t_m. To do this, we need to figure out what input t_m causes an error when we run program p. And for this, t_m is first divided into small sub-pieces, and then the process of finding non-error-causing parts is repeatedly performed. This returns the minimized input t_m.

The Minimize(t) function is a function that calls Reduce(t), with the goal of transforming a given input t into a minimized input.

This algorithm aims to minimize a program if it is error-prone on large inputs by breaking the input into smaller pieces and finding the non-error-prone ones.

(4) pipe(), fork(), and execl()

This program provides the ability to check whether a given input causes a crash within an executable file, while the child and parent processes communicate using pipe().

Using the fork() function, a parent process creates a child process, and the pipe() function establishes a communication pipe between the two processes. The child process then runs the executable file using the execl() function, and passes the result of the execution to the parent process via pipe(). The resulting value passed can be analyzed by the parent process to determine whether the input value causes a crash within the executable file.

2. Description of the used system functions

This code is a program called "Crash Input Minimizer", which is used to minimize input that causes the program to quit unexpectedly. This program takes an executable file and an option (-i) that accepts input that causes that file to exit unexpectedly. The program then tries to minimize this input.

The program uses the getopt function to process command-line arguments. Several structures and functions are also used to allow programs to read, write, and execute files and install signal handlers.

The variable type “Test” (which is basically a structure) stores several parameters used in the program. crashingInput is the program's input file, crashMessage is the error message from the program, and outputPath is the path to the file where the minimized output will be saved. minimizedOutput stores the minimized output, program is

the path to the executable file, and programOptions is any options needed to run the executable file. execOptions is an array of separated options for the executable file.

```
typedef struct {
	char* crashingInput;
	char* crashMessage;
	char* outputPath;
	char* minimizedOutput;
	char* program;
	char* programOptions;
	char** execOptions;
	int isFirstRun;
} Test;
```

Structure declaration

The help function explains how to use the program. The reader function reads a string from a file. The writer function writes a string to a file. The sigintHandler function handles the Ctrl-C keystroke. The execute function provides input to the program and returns results. The minimize function implements the main function of minimizing the input. The reduce function implements an auxiliary function that reduces the minimized input.

The program first processes the command line arguments, then calls the minimize function. The minimize function does the work of reducing the input. This function calls the reduce function to reduce the input, and calls the execute function to check if the reduced input still causes an unexpected exit. When the executable file exits, the function saves the reduced input and calls the reduce function again to reduce the input. Repeat this until the minimized input no longer shrinks. Finally, write the minimized input to a file using the writer function.

3. Demonstration

(1) Demonstration of program compiling and argument parsing (Using programs compiled with make debug)

```
$ make debug
gcc cimin.c -Wall -Wextra -DDEBUG -o cimin
$ ./cimin -i example/jsmn/testcases/crash.json -m "AddressSanitizer: heap-buffer-overflow" -o example/output/jsmn.json ./example/jsmn/jsondump
[CIMIN] DEBUG MODE
argc: 8
argv[0]: ./cimin
argv[1]: -i
argv[2]: example/jsmn/testcases/crash.json
argv[3]: -m
argv[4]: AddressSanitizer: heap-buffer-overflow
argv[5]: -o
argv[6]: example/output/jsmn.json
argv[7]: ./example/jsmn/jsondump

[CIMIN] Parsed arguments:
crashingInput:
{
	"glossary": {
		~~~
		(Omitted)
		~~~
		}

}

crashMessage: AddressSanitizer: heap-buffer-overflow
outputPath: example/output/jsmn.json
program: ./example/jsmn/jsondump
execOptions[0]: ./example/jsmn/jsondump
```

(2) Interrupt handling through Ctrl + C key input during program execution

```
$ make
gcc cimin.c -o cimin
$ ./cimin -i example/jsmn/testcases/crash.json -m "AddressSanitizer: heap-buffer-overflow" -o example/output/jsmn.json ./example/jsmn/jsondump
[CIMIN] Minimizing the crashing input. It takes a while.
^C
[CIMIN] Interrupted by the user.
[CIMIN] The test is not started yet.
[CIMIN] No minimized input is created. Exiting…
```

(3) Other features cannot be demonstrated because the program does not properly generate reduced input.

4. Discussion

What we found:

- Brief concept and mechanism of “Delta debugging”.
- Various kinds of Process API functions.
- Numerous developer communities
- Get the data by receiving and parsing the arguments

The challenging part:

- Data exchange between parent and child processes
- Reflecting Delta debugging pseudocode behavior in C code

Unsolved question:

- How does the child process use the data received through the pipe as standard input for the new process?

What we learn:

- Write a basic Makefile
- Remode development using ssh
- How to use the Linux man pages and find the functions we need
- How to build existing source code and use it for testing
- Operation sequence and data exchange method of a new process branched through the Process API
- Aside from that, we learned a lot of things. It just didn't help much with this homework.

Any other interesting discussions:

- During the Delta debugging process, the larger the crash input, the longer the debugging time will be. Is there any way to reduce this time?
- Divergent Debugging is a method of finding and correcting points where the expected output of the test input differs from the actual output. We thought it would be easier to analyze the difference and correct it, although a method to reduce errors like the delta debugging algorithm is good.

5. Impression

I spent a lot of time first getting a good understanding of the assignment through Google Searching. After that, I implemented the code and algorithm for inputting arguments, and fully understood the contents of fork() and pipe(). However, I think I lacked understanding of the execl() function. I thought I used it properly, but there were problems such as infinite loops. I wondered if jsondump was the wrong executable, but when I ran "./jsondump < jsmn/testcases/crash.json", I confirmed that there was no problem with the program. Also, I think that I lacked understanding of fork() and pipe() as I saw that it was called well when execv was executed without using pipe(), but it did not return when used with pipe(). *(By. Hyunwoo Gu)*

I still remember how shocked I was when I first read the assignment. It seemed like she was given a paper, not an assignment, and spent a lot of time understanding the pseudocode. However, among the assignments I have done in HGU so far, I wonder if there was another assignment that I was motivated and wanted to work hard on. Overall, it seems that there was a lack of understanding of the functions defined in <unistd.h>. I looked up various online references and perused the manual, but it seems that there was a fundamental lack of understanding in applying the information I found to the task. However, I was able to learn various things while doing the assignments, and it seems that the time has come to learn a lot of things I did not know while collaborating. Although, it is very regretful to submit the assignment without being able to complete it, but it was an assignment that I would like to complete alone later. *(By. Hyunseo Lee)*