SWEN90006 Software Testing and Reliability

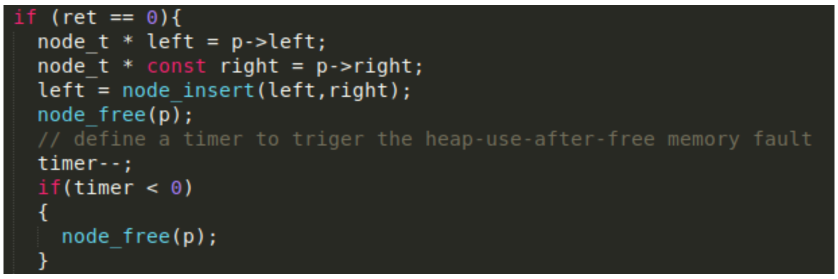
Assignment 2 Report

Team members and Responsibilities

Name	Responsibility
Saier Ding	Vulnerabilities analysis and implementation, coverage evaluation
Jinxin Hu	Vulnerabilities analysis and implementation, coverage evaluation
Boyu Zhou	Fuzzer analysis and implementation, design and evaluation
Dongming Liu	Fuzzer design and implementation, evaluation design.

Vulnerabilities

1. **Vulnerability_1:** heap-use-after-use error in remove function



```
if (ret == 0){
  node_t * left = p->left;
  node_t * const right = p->right;
  left = node_insert(left,right);
  node_free(p);
  // define a timer to triger the heap-use-after-free memory fault
  timer--;
  if(timer < 0)
  {
    node_free(p);
  }
```

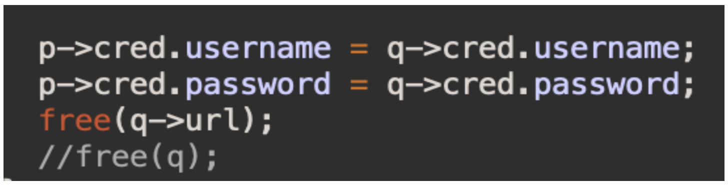
Figure1: vul_1

Description: This vulnerability part is in the remove function with a timer judgement for node_free(p), and the timer variable is defined as a global variable with the value of 511**.**

Trigger: When calling PUT and REM instruction pairs for more than 511 times, each of which has the same url and different username or password**.**

Potential Consequences: This should cause a heap-use-after-use error because the heap memory for node p has been used after its free action. Therefore, there is no guarantee that the memory to read is in legal.

2. **Vulnerability_2**: memory leaks error in node_edit_cred function

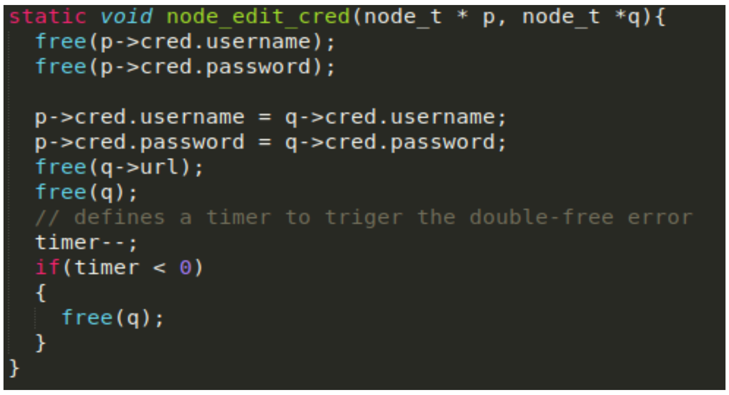


```
p->cred.username = q->cred.username;
p->cred.password = q->cred.password;
free(q->url);
//free(q);
```

Figure2: vul_2

Description: The free(p) function is removed in the node_edit_cred function.
Trigger: When calling PUT instruction after another PUT instructions which has the same url and different username or password.
Potential Consequences: This should cause a '???' error because the guard has been commented. Thus there is no guarantee that the memory to write in is in legal.
Heap-buffer-overflow can cause the illegal rewriting data thus could be exploited by malicious attackers to delete important data or trigger the downtime of system.

3. **Vulnerability_3**: double-free error in node_edit_cred function

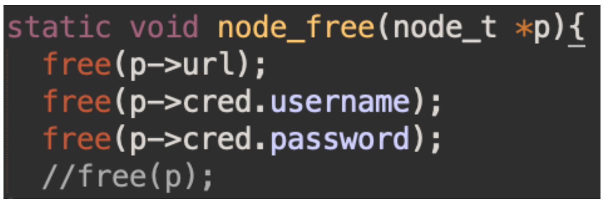


```c
static void node_edit_cred(node_t * p, node_t *q){
  free(p->cred.username);
  free(p->cred.password);

  p->cred.username = q->cred.username;
  p->cred.password = q->cred.password;
  free(q->url);
  free(q);
  // defines a timer to triger the double-free error
  timer--;
  if(timer < 0)
  {
    free(q);
  }
}
```

Figure3: vul_3
Description: Similar like the first one, this program part is added in the node_edit_cred function which will be called by put instruction with a global timer valued 1022.
Trigger: When calling PUT instruction for more than 1023 times which have the same url and different username or password.
Potential Consequences: This should cause a double-free error because the node q is required to be freed when it has been already freed. Therefore, there is no guarantee that the memory to free is in legal.

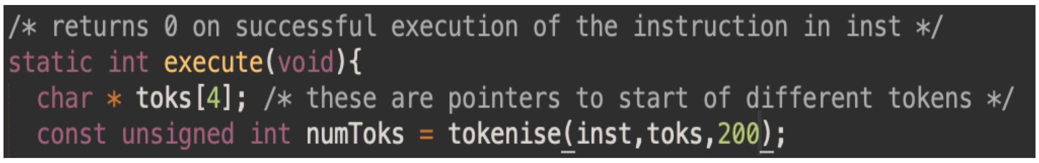
4. **Vulnerability_4**: memory leaks error in node_free function



```c
static void node_free(node_t *p){
  free(p->url);
  free(p->cred.username);
  free(p->cred.password);
  //free(p);
```

Figure4: vul_4
Description: The free(p) function is removed in the node_free function.
Trigger: When calling PUT instruction after another PUT instructions which has the same url and different username or password.
Potential Consequences: This should cause a '???' error because the guard has been commented. Therefore, there is no guarantee that the memory to write in is in legal.
Heap-buffer-overflow can cause the illegal rewriting data thus could be exploited by malicious attackers to delete important data or trigger the downtime of system.

5. **Vulnerability_5**: stack-buffer-overflow in tokenise function



```
/* returns 0 on successful execution of the instruction in inst */
static int execute(void){
  char * toks[4]; /* these are pointers to start of different tokens */
  const unsigned int numToks = tokenise(inst,toks,200);
```

Figure5: vul_5

Description: In function tokenize, change the maximum number of token in one line of instruction from 4 to 200

Trigger: Write instruction with more than 4 tokens per line.

Potential Consequences: This can cause stack-buffer-overflow error because the memory allocation for the token is only 4, and there is no space for new stack frame. This condition may lead for the attacker to rewrite functions loaded into the stack as well.

Fuzzer Design

Fuzzer Strategy

One of the most important criteria for a high quality fuzzer is high coverage. Generation-based fuzzer and mutation-based fuzzer are used for our fuzzers to try to reach a higher coverage. The invalid input could make the program exit/fail immediately, but some specific vulnerabilities may only be triggered by invalid input. Hence, we divided each fuzzer file into three parts, we ensure the first two parts inputs are all valid but only generate invalid inputs in third part. Besides this, some vulnerabilities may require more special inputs, but normally these kinds of vulnerabilities are rarely appeared, so we allocate 10 files for these special inputs and 90 files for the regular one. Now we introduce more details for these two different types of fuzzers.

Part 1:

We choose generation method because we have already known the protocol and constrain of the input. According to the specification, the instruction 'masterpw' will let the program exit immediately when the user enters a wrong master password, so we will not use this instruction in the first part, because all the inputs are randomly generated, when we generate the 'masterpw' instruction, the possibility for program to exit is infinitely close to 100%. For the other five instructions, we just randomly generate one of the instructions with correct format and legal length each lines. According to the specification, we know that the program allow one or more whitespace characters, namely space, tab, carriage-return and line-feed between instruction and parameters, but we will not choose to generate carriage-return because we will run the program in command where carriage-return means to execute this line. Overall, for each line we will generate an input like:

"instruction"[white space]<String>[white space]<String>

(The parameters may various according to different instructions.)

Part 2:

Actually, there exist rich logic among 'put', 'get' and 'rem' which we should consider in our fuzzers. For example, the user could put the same url1, user1, password1 for twice, and then use 'get' instruction to the information of url1, finally use 'rem' instruction to remove the information of url1. Similar to part 1, we also just generated valid input and follow the correct format of these three instructions. What special is that only 'put', 'get' and 'rem' instructions

will be used in part 2 and we restricted the length of each line to 1023 and the combinations of these three instructions followed by the below 8 rules:
1)
“put”[white space]url1[white space]user1[white space]pwd1
“put”[white space]url1[white space]user1[white space]pwd1
“get”[white space]url1
“rem”[white space]url1
“get”[white space]url1
2)
“put”[white space]url1[white space]user1[white space]pwd1
“put”[white space]url1[white space]user1[white space]pwd2
“get”[white space]url1
“rem”[white space]url1
“get”[white space]url1
3)
“put”[white space]url1[white space]user1[white space]pwd1
“put”[white space]url1[white space]user2[white space]pwd1
“get”[white space]url1
“rem”[white space]url1
“get”[white space]url1
4)
“put”[white space]url1[white space]user1[white space]pwd1
“put”[white space]url2[white space]user2[white space]pwd1
“get”[white space]url1
“rem”[white space]url1
“get”[white space]url1
“get”[white space]url2
“rem”[white space]url2
“get”[white space]url2
5)
“put”[white space]url1[white space]user1[white space]pwd1
“put”[white space]url1[white space]user2[white space]pwd2
“get”[white space]url1
“rem”[white space]url1
“get”[white space]url1
6)
“put”[white space]url1[white space]user1[white space]pwd1
“put”[white space]url2[white space]user1[white space]pwd2
“get”[white space]url1
“rem”[white space]url1
“get”[white space]url1
“get”[white space]url2
“rem”[white space]url2
“get”[white space]url2

7)
“put”[white space]url1[white space]user1[white space]pwd1
“put”[white space]url2[white space]user2[white space]pwd1
“get”[white space]url1
“rem”[white space]url1
“get”[white space]url1
“get”[white space]url2
“rem”[white space]url2
“get”[white space]url2
8)
“put”[white space]url1[white space]user1[white space]pwd1
“put”[white space]url2[white space]user2[white space]pwd2
“get”[white space]url1
“rem”[white space]url1
“get”[white space]url1
“get”[white space]url2
“rem”[white space]url2
“get”[white space]url2
By combining the part 1 and part 2, we could cover the different length of input even for the boundary length. We also covered as much as possible situations in our part 2 so that we could reach a high coverage.
Part 3:
Mutation-based fuzzer can allow us to do a slight change in the valid input. Regarding to the third part of the fuzzer, we combined the mutation based fuzzer and generation based fuzzer to ensure that we could firstly correct run the program, and for the last line command, all these following invalid inputs are used to mutate the generated input instructions and trigger some specific situation which only appeared for the invalid input:
a) Generate instructions more than 1024 lines.
b) Instruction ”put” has four parameters.
c) Generate one instruction with 1023 characters in one line.
d) Instruction ”put” only has one parameter.
e) Instruction ”rem” does not have the parameter
f) Instruction ”put” has four parameters.
g) Instruction “save” only has one parameter.
h) Instruction “list” has one parameters.

Evaluation
Trigger Vulnerabilities by our Fuzzers
One of the most important roles for our fuzzers is to try to trigger our own 5 vulnerabilities. Because our fuzzers include many random situations, we run our fuzzers for 5 times and generate 100 files each time. Then, we use our fuzzers as input to the 5 different vulnerabilities and the results are shown below.

Vulnerabilities	Triggered or not	Triggered in Group	Description
Vuln1	√	3	The fuzzer triggered the vulnerability that causes the heap-use-after-use error in remove function
Vuln2	√	1,2,3,4,5	The fuzzer triggered the vulnerability that causes the memory leak error in node_edit_cred function
Vuln3	x	0	The fuzzer triggered the vulnerability that causes the double-free error in node_free function
Vuln4	√	1,2,3,4,5	The fuzzer triggered the vulnerability that causes the memory leak error in node_free function
Vuln5	√	1,2,3,4,5	The fuzzer triggered the vulnerability that causes stack-buffer-overflow in tokenise function

Table 1: Check list of Vulnerabilities Triggered by our Fuzzer

Comparing with Greybox (libFuzzer)

For this project, we used libFuzzer as the greybox tool to do the experiment. The libFuzzer rely on a corpus of sample inputs for the code under test, so we feed the libFuzzer with a valid sample input and set the default seed then generate the libFuzzer 5 times and each time we also generate as much as 100 input test cases to find if the libFuzzer can trigger our vulnerabilities. LibFuzzer could trigger the vulnerabilities when it causes a memory crash or memory leak. The results are shown below:

Vulnerabilities	Cause crash	Crash time	Description
Vuln1	√	7.351s	ERROR: AddressSanitizer: heap-use-after-free on address 0x60400037ebd0 at pc 0x00000054f661 bp 0x7ffd937678e0 sp 0x7ffd937678d8
Vuln2	√	1.635s	ERROR: LeakSanitizer: detected memory leaks node_edit_cred function

Vuln3	√	2.057s	ERROR: AddressSanitizer: attempting double-free on 0x6040000d3d90 in thread T0:
Vuln4	√	2.392s	ERROR: LeakSanitizer: detected memory leaks
Vuln5	√	2.674s	ERROR: AddressSanitizer: stack-buffer-overflow on address 0x7ffc543ced60 at pc 0x00000054a814 bp 0x7ffc543cec40 sp 0x7ffc543cec38

Table 2: Check list of Vulnerabilities Triggered by libFuzzer

Coverage

For this part, we ran all of our five group of fuzzers for all of the vulnerabilities, but we still can't find the first vulnerability and the third vulnerability errors in the five times of generating data, we manually generated two test cases that can solve them. And then we fuzzed libFuzzer based on each vulnerability. Finally we recorded the coverage and calculated the average coverage of regions, functions and lines coverage.

In the data obtained from the experiment, we chose function coverage which measures the percentage of functions executed at least once when the program takes the test suite provided by fuzzer as an input of the program, line coverage which measures the percentage of statements executed at least once when the program takes the test suite provided by fuzzer as an input of the program and region coverage which measures the percentage of code regions executed at least once when the program takes the test suite provided by fuzzer as an input of the program. The results of the experiment are shown as below.

	Corpus generated with Vuln1	**Corpus generated with Vuln2**	**Corpus generated with Vuln3**	**Corpus generated with Vuln4**	**Corpus generated with Vuln5**	**Average coverage**
Regions	205	205	205	205	205	
Missed Regions	128	15	92	15	15	
Regions Coverage	37.6%	92.7%	55.1%	92.7%	92.7%	74.2%
Functions	20	20	20	20	20	
Missed Functions	10	0	5	0	0	
Functions Coverage	50%	100%	75%	100%	100%	85%
Lines	434	434	434	434	434	

Missed Lines	288	46	193	46	46	
Lines Coverage	33.6%	89.4%	55.5%	89.4%	89.4%	71.4%

Table 3 : Coverage with corpus that Fuzzer generated for each vulnerability

	Corpus generated with Vuln1	**Corpus generated with Vuln2**	**Corpus generated with Vuln3**	**Corpus generated with Vuln4**	**Corpus generated with Vuln5**	**Average coverage**
Regions	205	205	205	205	205	
Missed Regions	80	136	75	166	166	
Regions Coverage	61.0%	33.7%	63.4%	19.0%	19.0%	39.2%
Functions	20	20	20	20	20	
Missed Functions	3	10	4	15	15	
Functions Coverage	85%	50%	80%	25%	25%	53%
Lines	434	434	434	434	434	
Missed Lines	179	282	167	363	363	
Lines Coverage	58.8%	35.0%	61.5%	16.4%	16.4%	37.6%

Table 4: Coverage with corpus that LibFuzzer generated for each vulnerability

According to Table 2 and Table 3, we could find that the average coverage of Fuzzer is greater than LibFuzzer in Regions Coverage (74.2%), Functions Coverage (85%) and Lines Coverage (71.4%).

Evaluation Strategy

Based on the understanding of LibFuzzer and the experimental results of Fuzzer, we are going to design two experiments to analyze the advantages and disadvantages of our designed Fuzzer and LibFuzzer, and improve our design based on the results. Moreover, we also analyze the factors that affect LibFuzzer.

Firstly, in order to compare the advantages and disadvantages of LibFuzzer and Fuzzer, we designed the above table, respectively, by calculating Region Coverage, Function Coverage and Lines Coverage. In this experiment, we first use the get_coverage method to analyze the coverage for five vulnerabilities, and then for the control variables, a total of five vulnerabilities selected multiple Fuzzer generated test cases simultaneously use get_coverage method to analysis. Then, we analyze LibFuzzer and Fuzzer based on the returned data, and finally come to a conclusion.

Secondly, we made assumptions after the experiment one. As the LibFuzzer generating time increases, the larger the number of samples generated by LibFuzzer, the more code lines and functionality it will cover. So as time goes on, LibFuzzer's coverage will get closer to Fuzzer, and even surpass.

Discussion

From the table 1, it shows that our fuzzer could trigger most of vulnerabilities but would not trigger each vulnerability every time because whenever the vulnerabilities triggered, the program will stop executing. Therefore, the strategy which generates the instructions that may trigger one vulnerability with probability can reach our expectations, according to the result of random function.

Additionally, it can be seen from the table 2 that our fuzzer performed better than Libfuzzer since the average coverage of our fuzzer is greater than LibFuzzer in Regions Coverage (74.2%), Functions Coverage (85%) and Lines Coverage (71.4%).

Apart from this, the coverage of the output fuzzer is achieved by considering the behaviour of each instruction and the input format, not only mutating the instruction. During a series of experiment, we found that the coverage of the fuzzer can get over 90% coverage rate for the regions, which means the output of our fuzzer can cover over 90% codes.

Based on the performance on the leaderboard, the performance of our fuzzer can directly detect over 110 bugs. the main reason is that we combined both mutation-based fuzzing and generation-based fuzzing. The output of our fuzzer can cover more situations.

It also can be seen that the performance of fuzzer testing is based on the designer’s experience, the key is how to generate test cases according to the valid grammar but results the program break down. Thus, the program specification is important for designing a fuzzer.