FUZZING ATTACK

SUBJECT NAME: Cryptography and Network Security
SUBJECT CODE: CS6008
MODULE: 04

NAME: R.Aadharsh
REG.NO.: 2019103604
DATE: 06/06/2022

AIM:

Using libfuzzer and AFL to fuzz your own C/C++ implementations, to implement a c/c++ code using fuzzer tools

TOOLS INVOLVED:

LIBFUZZER
AFL FUZZER
VISUAL STUDIO CODE
KALI LINUX TERMINAL (WSL)
WINDOWS (OPERATION SYSTEM)

PROBLEM DESCRIPTION:

Fuzzing is the process of attempting a large number of random inputs to code in order to find a flaw. You design a testbench for the code in question, link it with a fuzzing engine that creates random data, and run it on a server. If your testbench is sound, it will return with a set of inputs that will cause the code to crash hours, days, or weeks later. This procedure can be sped up by:

• Using a sanitizer: compiler-supported sanitizers add extra code to binaries to check for unlawful circumstances like out-of-bounds memory accesses that don't always result in a crash. As a result, the code under test is more likely to fail, and the fuzzer runs faster.

• Coverage-driven fuzzing: fuzzers can track the programme states reached by various inputs and direct the inputs in a way that tends to generate new ones.

Check up libFuzzer from the LLVM Subversion repository and build it according to their instructions. A test driver named LLVMFuzzerTestOneInput is provided as a function with C linkage. As a consequence, you'll have a standalone application that runs the code contained within that function. It monitors which pathways are exercised using instrumentation provided by the Clang compiler via the -fsanitize-coverage option, thus gcc isn't an option. To guarantee no initialization is omitted, it must be built using -fsanitize=memory. AFL is a stand-alone application that employs binary rewriting to instrument the code under test. Wrapper compilers are provided, which call any of the two compilers.

INPUT:
Getting input from a file
OUTPUT:
Given input string is to check whether it is a palindrome or not

SCREENSHOTS:

FUZZING WITH AFL:

Step 1:Installing AFL:

```
$ apt-get install -y afl
$ apt-get source libxml2-utils
```

Step 2:

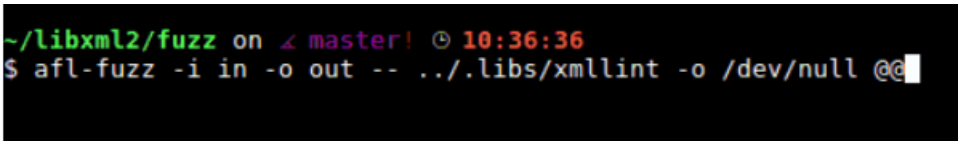
Next we configure libxml2 build to use AFL compilers and compile the xmllint utility.

```
$ cd libxml2/
$ ./configure CC=afl-gcc CXX=afl-g++
$ make xmllint
```

Step 3:Lastly we create a sample file with content "<a>" for AFL to start with and run the afl-fuzz.

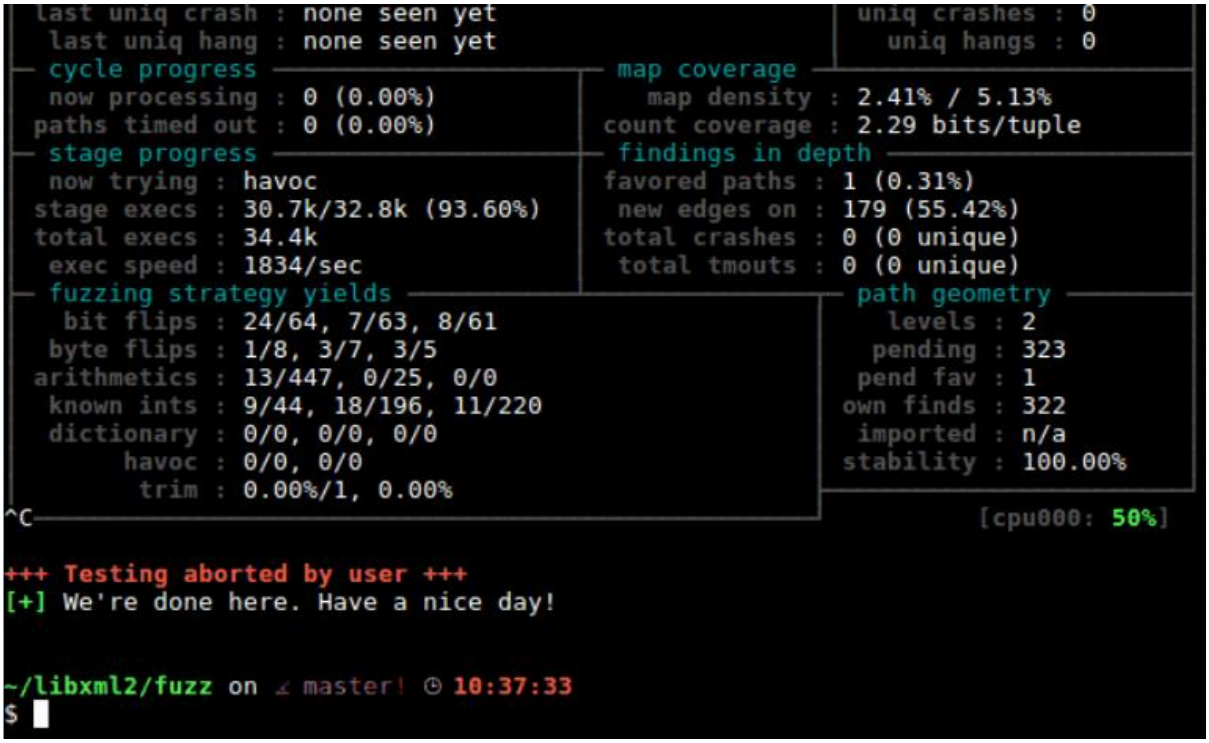
```
$ echo "" > in/sample
$ LD_LIBRARY_PATH=./.libs/ afl-fuzz -i ./in -o ./out -- ./.libs/lt-xmllint -o /dev/null @@
```

Compiling and running the program using AFL



```
afl-fuzz 2.43b by <lcamtuf@google.com>
[+] You have 4 CPU cores and 1 runnable tasks (utilization: 25%).
[+] Try parallel jobs - see /usr/local/share/doc/afl/parallel_fuzzing.txt.
[*] Checking CPU core loadout...
[+] Found a free CPU core, binding to #0.
[*] Checking core_pattern...
[*] Setting up output directories...
[*] Scanning 'in'...
[+] No auto-generated dictionary tokens to reuse.
[*] Creating hard links for all input files...
[*] Validating target binary...
[*] Attempting dry run with 'id:000000,orig:sample'...
[*] Spinning up the fork server...
[+] All right - fork server is up.
    len = 8, map size = 1582, exec speed = 575 us
[+] All test cases processed.

[+] Here are some useful stats:

    Test case count : 1 favored, 0 variable, 1 total
       Bitmap range : 1582 to 1582 bits (average: 1582.00 bits)
        Exec timing : 575 to 575 us (average: 575 us)

[*] No -t option specified, so I'll use exec timeout of 20 ms.
[+] All set and ready to roll!
```

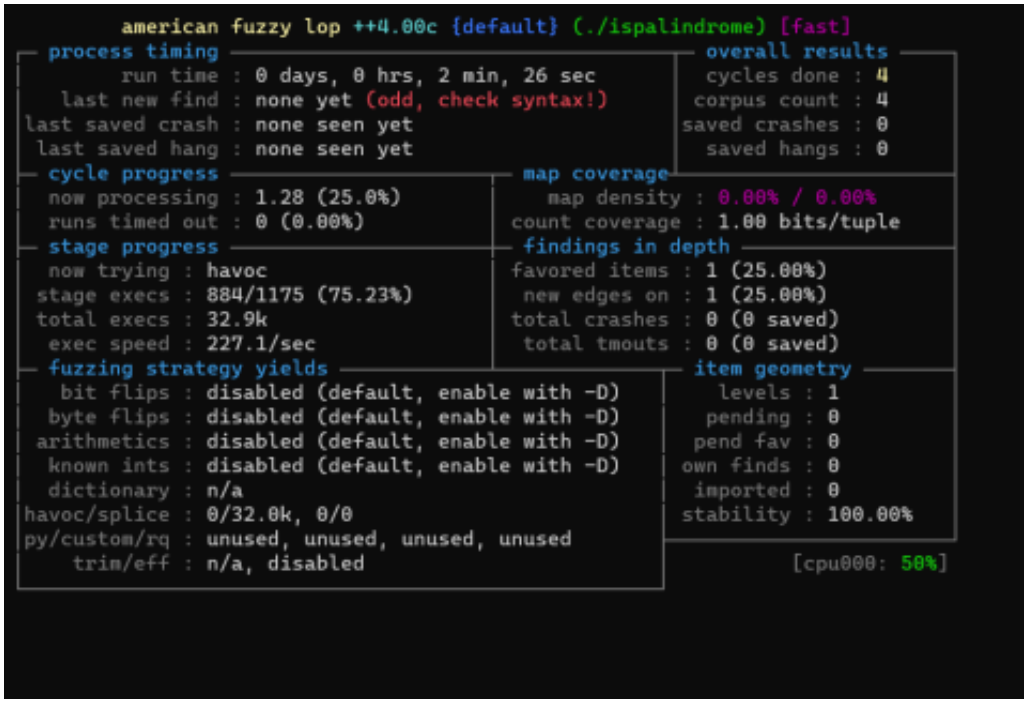
```
                    american fuzzy lop 2.43b (xmllint)

┌─ process timing ─────────────────────┬─ overall results ─────┐
│        run time : 0 days, 0 hrs, 0 min, 14 sec │  cycles done : 0      │
│   last new path : 0 days, 0 hrs, 0 min, 0 sec  │  total paths : 293    │
│ last uniq crash : none seen yet      │ uniq crashes : 0      │
│  last uniq hang : none seen yet      │   uniq hangs : 0      │
├─ cycle progress ─────┬─ map coverage ─┴───────────────────────┘
│  now processing : 0 (0.00%)          │   map density : 2.41% / 5.02%
│ paths timed out : 0 (0.00%)          │ count coverage : 2.27 bits/tuple
├─ stage progress ─────┼─ findings in depth ────────────────────┐
│  now trying : havoc                  │ favored paths : 1 (0.34%)
│ stage execs : 22.1k/32.8k (67.46%)   │  new edges on : 165 (56.31%)
│ total execs : 25.6k                  │ total crashes : 0 (0 unique)
│  exec speed : 1835/sec               │  total tmouts : 0 (0 unique)
├─ fuzzing strategy yields ────────────┴─ path geometry ──────┐
│   bit flips : 24/64, 7/63, 8/61      │    levels : 2        │
│  byte flips : 1/8, 3/7, 3/5          │   pending : 293      │
│ arithmetics : 13/447, 0/25, 0/0      │  pend fav : 1        │
│  known ints : 9/44, 18/196, 11/220   │ own finds : 292      │
│  dictionary : 0/0, 0/0, 0/0          │  imported : n/a      │
│       havoc : 0/0, 0/0               │ stability : 100.00%  │
│        trim : 0.00%/1, 0.00%         │                      │
└──────────────────────────────────────┘                      [cpu000: 50%]
```


last uniq crash : none seen yet
last uniq hang : none seen yet
uniq crashes : 0
uniq hangs : 0
cycle progress
now processing : 0 (0.00%)
paths timed out : 0 (0.00%)
map coverage
map density : 2.41% / 5.13%
count coverage : 2.29 bits/tuple
stage progress
now trying : havoc
stage execs : 30.7k/32.8k (93.60%)
total execs : 34.4k
exec speed : 1834/sec
findings in depth
favored paths : 1 (0.31%)
new edges on : 179 (55.42%)
total crashes : 0 (0 unique)
total tmouts : 0 (0 unique)
fuzzing strategy yields
bit flips : 24/64, 7/63, 8/61
byte flips : 1/8, 3/7, 3/5
arithmetics : 13/447, 0/25, 0/0
known ints : 9/44, 18/196, 11/220
dictionary : 0/0, 0/0, 0/0
havoc : 0/0, 0/0
trim : 0.00%/1, 0.00%
path geometry
levels : 2
pending : 323
pend fav : 1
own finds : 322
imported : n/a
stability : 100.00%
^C
[cpu000: 50%]
+++ Testing aborted by user +++
[+] We're done here. Have a nice day!
~/libxml2/fuzz on master! 10:37:33
$

Code:

```cpp
#include <iostream>
#include <bits/stdc++.h>
#include <fstream>
using namespace std;
bool isPalindrome(string data, int s)
{
for (int i = 0; i < s / 2; i++)
{
if (data[i] != data[s - i - 1])
return false;
}
return true;
}
void parse_file(char *filename,vector<string>&input){
string i;
ifstream input_file(filename);
if(!input_file.is_open()){
cerr << "Could not open the file - '"<< filename << "'" << endl;
exit(-1);
}
while(getline(input_file,i)){
input.push_back(i);
}
input_file.close();
}
int main(int argc, char **argv)
{
string data;
if (argc < 2)
{
cout << "enter filename";
return -1;
}
vector<string>input;
parse_file(argv[1],input);
for(auto i : input){
if(isPalindrome(i,i.size())){
cout<<i<<"\t --> TRUE\n";
}
else{
cout<<i<<"\t --> FALSE\n";
}
}
return 0;
}
```

Compiling and running with afl-g++



LIBFUZZER

Let’s now fuzz libxml2 with the LLVM libFuzzer. To start fuzzing, you’ll first need to introduce a target function, LLVMFuzzerTestOneInput, that receives the fuzzed input buffer from libFuzzer. The code looks like this.

```
extern "C" int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size) {
 DoSomethingInterestingWithMyAPI(Data, Size);
 return 0; // Non-zero return values are reserved for future use.
 }
```

For fuzzing libxml2, Google’s fuzzer test suite provides a good example which is as follows:

```
#include
 #include
 #include "libxml/xmlversion.h"
 #include "libxml/parser.h"
 #include "libxml/HTMLparser.h"
 #include "libxml/tree.h"
void ignore (void * ctx, const char * msg, ...) {}
extern "C" int LLVMFuzzerTestOneInput(const uint8_t *data, size_t
size) {
 xmlSetGenericErrorFunc(NULL, &ignore);
 if (auto doc = xmlReadMemory(reinterpret_cast(data), size,
"noname.xml", NULL, 0))
 xmlFreeDoc(doc);
 return 0;
 }
```

Before compiling our target function, we need to compile all dependencies with clang and -fsanitize-coverage=trace-pc-guard, to enable SanitizerCoverage coverage tracing. It is a good idea to also use -fsanitize=address,undefined in order to enable both the AddressSanitizer(ASAN) and the UndefinedBehaviorSanitizer(UBSAN) that catch many bugs that otherwise might be hard to find

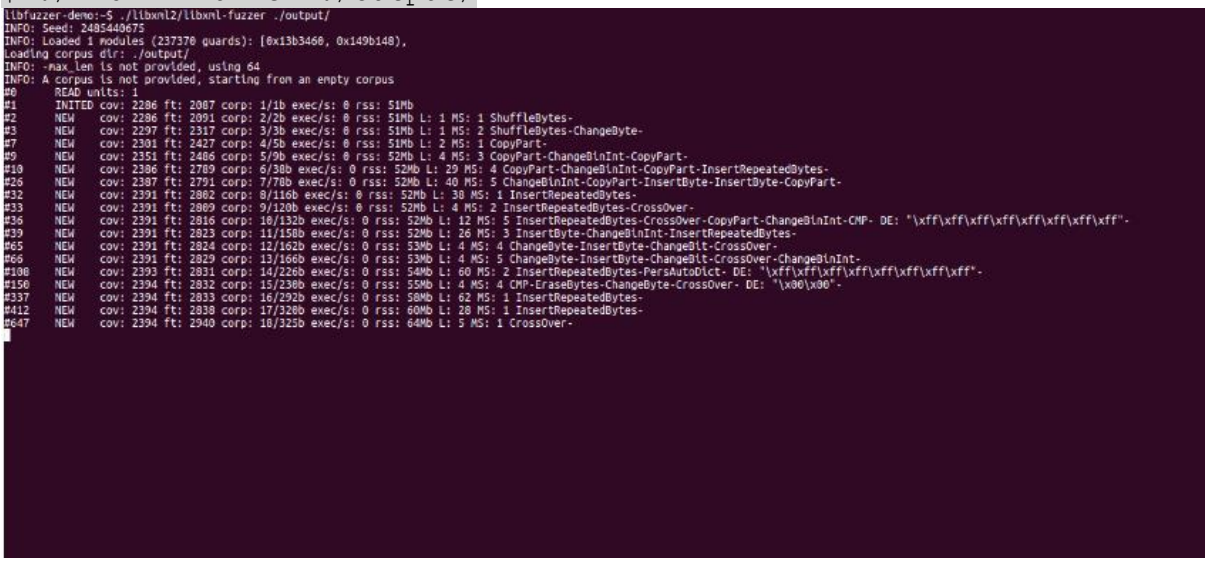
```
$ git clone https://github.com/GNOME/libxml2 libxml2
 $ cd libxml2
$ FUZZ_CXXFLAGS="-O2 -fno-omit-frame-pointer -g -
fsanitize=address,undefined -fsanitize-coverage=trace-pc-guard"
 $ ./autogen.sh
 $ CXX="clang++-5.0 $FUZZ_CXXFLAGS" CC="clang-5.0 $FUZZ_CXXFLAGS"
CCLD="clang++-5.0 $FUZZ_CXXFLAGS" ./configure
 $ make
```

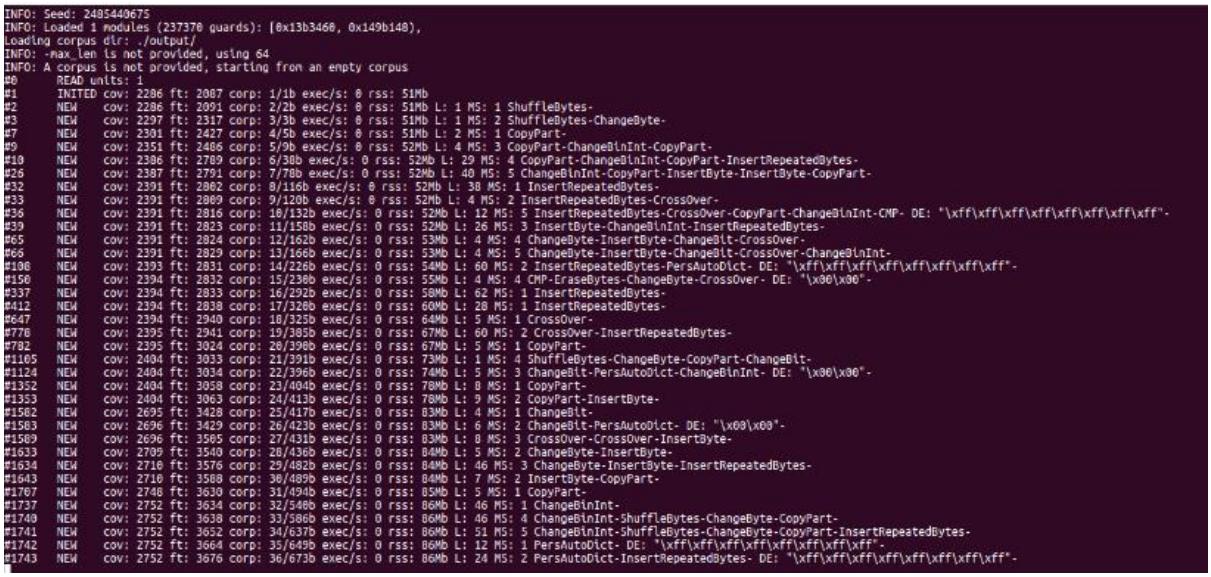
The second step is to compile our target function, with the same flags, and link it with both the libFuzzer runtime and the libxml2 we compiled earlier.

```
$ clang++-5.0 -std=c++11 $FUZZ_CXXFLAGS -lFuzzer ./libxml-test.cc -I ./include ./.libs/libxml2.a -lz -llzma -o libxml-fuzzer
```

Now we are ready to run our fuzzer

```
$ mkdir ./output
$ ./libxml-fuzzer ./output/
```



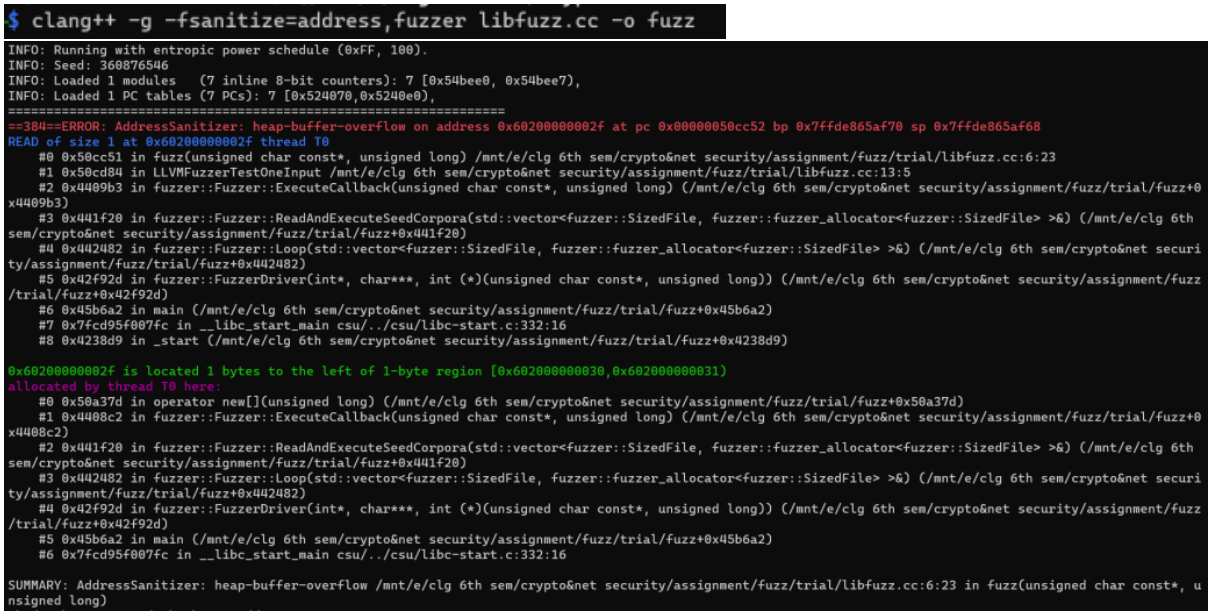


Code:

```
#include <stdint.h>
#include <stddef.h>
bool fuzz(const uint8_t *data, size_t s){
for(int i = 0 ; i <= s/2 ; i++){
if(data[i] != data[s-i-1])
return false;
}
return true;
}
extern "C" int LLVMFuzzerTestOneInput(const uint8_t *data, size_t s){
fuzz(data,s);
return 0;
}
```

When the value of I is equal to half of the value of size in the for loop of the function "fuzz," an error will occur since there is no condition for palindrome. To include libfuzzer in our software, we utilised Clang+. The libfuzzer driver LLVMFuzzerTestOneInput

is used to test the fuzz() function. We'll compile the results and see what occurs when Max_len = 20 with max ip size of 5. clang++ -g -



Here the error says that heap-overflow has occurred at address 0x6020000002f at instruction pc

0x000000050xx52 bp 0x7ffde865af70 bp 0x7ffde865af68.

Program after removal of discovered bug:

Code:

```cpp
#include <stdint.h>
#include <stddef.h>
#include <signal.h>
bool isPalindrome(const uint8_t *data, size_t s){
for(int i = 0 ; i < s/2 ; i++){
if(data[i] != data[s-i-1])
return false;
}
return true;
}
extern "C" int LLVMFuzzerTestOneInput(const uint8_t *data, size_t s){
isPalindrome(data,s);
return 0;
}
```

```
└─$ ./vuln -max_len=3
INFO: Running with entropic power schedule (0xFF, 100).
INFO: Seed: 3727234840
INFO: Loaded 1 modules   (7 inline 8-bit counters): 7 [0x54bee0, 0x54bee7),
INFO: Loaded 1 PC tables (7 PCs): 7 [0x524070,0x5240e0),
INFO: A corpus is not provided, starting from an empty corpus
#2      INITED cov: 4 ft: 4 corp: 1/1b exec/s: 0 rss: 29Mb
#6      NEW    cov: 5 ft: 5 corp: 2/3b lim: 3 exec/s: 0 rss: 30Mb L: 2/2 MS: 4 ShuffleBytes-ChangeBit-ChangeByte-CopyPar
t-
#7      NEW    cov: 7 ft: 7 corp: 3/5b lim: 3 exec/s: 0 rss: 30Mb L: 2/2 MS: 1 InsertByte-
#4194304        pulse  cov: 7 ft: 7 corp: 3/5b lim: 3 exec/s: 1398101 rss: 217Mb
#8388608        pulse  cov: 7 ft: 7 corp: 3/5b lim: 3 exec/s: 1398101 rss: 403Mb
#16777216       pulse  cov: 7 ft: 7 corp: 3/5b lim: 3 exec/s: 1290555 rss: 775Mb
#33554432       pulse  cov: 7 ft: 7 corp: 3/5b lim: 3 exec/s: 1118481 rss: 1159Mb
#67108864       pulse  cov: 7 ft: 7 corp: 3/5b lim: 3 exec/s: 1100145 rss: 1159Mb
#134217728      pulse  cov: 7 ft: 7 corp: 3/5b lim: 3 exec/s: 1109237 rss: 1159Mb
```

Crash reported at when the input string is null