18-739L - Research Proposal

Eliot Wong - ecwong

March 14th, 2015

1 Introduction

Many CTF or war games will often be littered with a small amount of binary exploitation problems which are often very straight forward to solve. These problems may often take time to trace through the correct conditional branches to determine the input string. Through automation the time spent on these problems could be reduced, and more focused could be placed on other problems.

One example is the problem "This is Endian" that is a low value binary exploitation problem. The program accepts one input and goes through various conditional branches to determine if the input matched a pre-determined string.

```
if(answer[0] == 0x52657663 && answer[1] == 0x30646521)
```

This is an easy example that could be solved just by taking a look at this line. However, in other programs, there may be more conditional branches and misdirection which can cause the solution to be less obvious. With all the branching paths, it is possible to make a simple mistake which will lead to the exploit not working. By automating these CTF problems, the possibilities of these mistakes are removed, and will allow more focus to be placed on higher valued problems which require for creative and innovative solutions which may not be easy to automate.

The automation tool will be built upon the symbolic execution tool, KLEE. The original intention of this tool is to provide a tool to automate test cases which provide consistently high code coverage. This is beneficial for the automation of solving CTF problems because often times it will be necessary to find a set of specific input sequence that will trigger a specific path through the program.

By creating a tool to solve the low point value CTF problems, this will hopefully improve my understanding of how to use KLEE. The next step of this tool is to create a tool which can automatically create a exploit string for buffer overflow vulnerabilities and possibility format string vulnerabilities.

In buffer overflow vulnerabilities CTF problems, many times the difficulty is identifying the location of the vulnerable code. Once the vulnerability is found, the remainder of a problem typically ends up being some math to manipulate the exploit string to properly overwrite a specified address, for example the return address. Automation of this process would ease the creation of the exploit string and removal trivial mistakes that can often occur.

2 Approach

The backbone of this project will be relient on the symbolic execution tool, KLEE. By taking use of KLEE's main feature which is to generate test cases with consistently high code coverage, it

could be possible to use this to automate the solving of CTF problems. This is because certain low point CTF problems will often have conditional branches which are dependent on the user specified input. By letting KLEE control this input, it would be possible to discover the input that allows the attacker to reach the required path, typically leading to the flag or a shell spawning.

Demonstrations of KLEE on various real-world programs has resulted in code coverages as high as a 90% average. Given that these programs often contained thousands of lines of code, and had many branching paths the complexity is much more than a typical CTF problem. As a result, this would give an indication that it would be possible to get results of code coverage just as good if not even better. The higher the code coverage, the more likely the tool has found the solution of the CTF problem.

However, KLEE cannot simply find and discover these paths just by itself. In order to build upon it, it is necessary to determine which inputs will be checked, the type of input variable and the size of the variable. Here is the source code for the CTF problem "This is the Endian". It will demonstrate the variables that need to be properly determined before it can be passed through to the KLEE tool.

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

    size_t answer_size = sizeof(int32_t)*2+2;
    char* str_answer = calloc(1, answer_size);

    printf("Access Code: ");
    fgets(str_answer, answer_size, stdin);

    trim(str_answer);
    int32_t* answer = (int32_t*)str_answer;

    if(answer[0] == 0x52657663 && answer[1] == 0x30646521) {
        printf("Access Granted!\n");
    } else {
        printf("You supplied: 0x%x and 0x%x\n", answer[0], answer[1]);
    }

    free(str_answer);
    return 0;
}
```

In this problem, the input that will be checked is acquired using the fgets function. This is helpful as it gives us a maximum input size that needs to be tested. Without an input size given, it would be necessary to come up with a reasonable input size, and increase the size if no working exploit cannot be found. An example of this would be the PicoCTF problem "Overflow1". Here is a snippet of the source code

```
void vuln(char *input){
    char buf[16];
    int secret = 0;
    strcpy(buf, input);

    if (secret == 0xc0deface){
        give_shell();
```

As the strcpy function is used in this case, the input size is technically unbounded, as a result

this does not provide a good indication of what the input size will be. The starting approach to finding hints as to the exploit string size would be to look at other parts of the code that check the string buffer, or possible what values could be overwritten on the stack.

In this example, the conditional branch is dependent on the value of the integer secret. However, this is not a value that can be directly control with the input string. However, it should be noted that it would be possible to overwrite this value as a result of a buffer overflow. This would allow the tool to set a initial input size of 20 bytes.

Finally, another aspect that will need to be identified is the type of variable that the input is stored in. For example, here is a line from the source code for level 2 of IO.

```
return abs(atoi(argv[1])) / atoi(argv[2]);
```

It is apparent from this line that the input arguments will eventually be represented as integers. This is helpful to limit the scope of what sort of input arguments need to be generated in order to solve this CTF. As a result, the number of inputs that will need to be tested would be significantly less.

A problem that is mentioned by others who have been working with KLEE is the potential for an infinite explosion of branching paths in real programs. Since CTF problems typically are self contained and small code size this should not be a problem as the number of paths will be limited. The only case where it may still be a problem is in a CTF problem which makes use of a lot of loops.

Taking a look once again at the "This is the Endian" problem, there is only one conditional branch within the program. Regardless the number of inputs, there will only be two paths through the program which should make KLEE very effective at testing this problem.

3 Implementation and Evaluation

3.1 Implementation

In order to successfully be able to run KLEE on the CTF problem the source code will need to be available. The program will then be compiled using LLVM-gcc with the option to output LLVM bitcode. This will allow KLEE to test the program.

Before this can be done, the program will need to be modified in order to identify the inputs which will be marked as symbolic for KLEE. Taking a look at the "This is the Endian" problem again, there is only one input which is obtained.

```
fgets(str_answer, answer_size, stdin);
```

Since this is the only input that the attacker will be able to modify, it is the obvious choice for becoming a symbolic variable. A new function will be created to act as the test harness for KLEE. Within this function, a new input variable will be created and marked as symbolic. In this example, a new character array is created and marked as symbolic. As state in the previous section, the size of the new input was determined by looking at the original source code.

```
int main() {
    char in[sizeof(int32_t)*2+2];

    klee_make_symbolic(in, sizeof(in), "in");

    return orig_main(in);
```

```
}
```

In order to automate this approach to allow it to solve a variety of CTF problems there needs to be a way to identify inputs to the program. The first place to look would be to see if the program makes use of any of the command line arguments. There are many other functions that can be used to get user input such as fgets(), get(), scanf(), getchar(). A list of all these functions will need to be compiled and the source code will need to be searched for occurrences of such functions. Another possibility is the program reading input from a pre-determined file path.

The next step, once the input has been identified, is to determine information about the size of the information. Sometimes there may be information about the minimum size of the input string. For example, here is a snippet from the level03 problem of IO. The input is identified to be argv[1], and the minimum length is 4.

```
if(argc != 2 || strlen(argv[1]) < 4)
    return 0;
```

In other cases, there may be a maximum length specified similar to the previous fgets example. The function will have a maximum number of characters it will attempt to read before terminating.

Finally, the last and most difficult case is if no input size is specified. The example in the previous section in the problem "Overflow1" is an example of this. The input size is not limited and is copied into a buffer using strcpy(). In this case, it will be important to recognize what the vulnerability of the program is. In the "Overflow1" example, the easiest solution is for the attacker to overwrite the value of secret which is on the stack.

However, in another problem, the solution might be to overwrite the return address to jump to a user-specified location. This will be significantly harder to automate. For the scope of this project, it will most likely not be possible to fully automate this process. It is possible the attacker will need to specify certain attributes or constraints for this to be possible.

A nice feature that KLEE has included is the klee_assert function. This will allow an assert statement to be placed on the specified path will solve the problem. If a test case is discovered which will run this assert statement, it will be easier to identify. KLEE will create a separate file, not just the test case, which indicates an assertion statement was triggered.

The most difficult part of the automation will be to ensure that it can work on a variety of problems. There is a danger of creating an "automation" program which is too closely modeled after the evaluation problems that it will not work on other CTF problems.

3.2 Evaluation

The first CTF problem to evaluate will be "This is the Endian". This is a very straightforward problem and the input size is easily determined. There are only two possible paths within the program and this will provide a good basic problem to get the automation working.

The next problem to test will be the "Binary Demo1" of the PicoCTF 2014 platform. This provides a nice challenge because the input size is unbounded and it will be useful to figure out how to determine a good starting size. This also has more conditional statements and thus more paths throughout the program.

Level02 of IO is a good problem to tackle the problem of identifying the different types of input and how to properly limit the symoblic variable to match this. The problem takes in command line arguments strings, however the input is passed through the atoi function. So it will be important to identify this to limit the scope of the test cases.

Other CTF problems that will be tested include Level03 of IO, Overflow1 of PicoCTF and Level0 in Narnia.

4 Related Work

Cristian Cadar, Daniel Dunbar, Dawson Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs

`http://llvm.org/pubs/2008-12-OSDI-KLEE.pdf`

This paper provides a nice introduction on how to use KLEE and the features that it includes. There is information included on topics such as query optimization and state management. They were able to archive 100% code coverage in some cases, looking more into this will be extremely useful for solving the CTF problems.

Thanassis Avgerinos, Sang Kil Cha, Brent Lim Tze Hao and David Brumley. AEG: Automatic Exploit Generation

`http://security.ece.cmu.edu/aeg/aeg-current.pdf`

The inspiration for this proposal came after watching the video demonstration of the AEG tool. This paper has a lot of useful information including the steps they took in order to have a fully automated tool to exploit programs with an end result of a shell. This will be a very useful resource.

`https://klee.github.io/`

This is the main page of the KLEE project and has lots of useful information about the tool such as tutorials, projects and documentation.

5 Updates

Update 1 - April 1st, 2014

Using KLEE

When I initially wrote the proposal, I had seen the Automatic Exploit Generation video as well as skimmed over the paper. At the time, it seems I had overestimated the contribution of KLEE to the AEG process. It seems that KLEE was used or a starting point for discovering the likely points of vulnerabilities within the program. However, the next step of AEG was to perform dynamic run-time analysis on the program. This is clearly an important step to determine information about the running program such as stack information.

Given this new understanding, it is difficult to determine whether the goal of being able to automatically solve binary exploitation CTF problems is appropriate for the scope of this project. There seems to be a small subset of binary exploitation problems where if the inputs and the respective sizes are properly identified, KLEE will be able to automatically solve the problem on its own. However, this small subset is only representative of a few problems which would represent trivial or low-point problems.

Looking at the other problems, with the input and input sizes identified, KLEE will be able to discover a likely point of vulnerability within a short period of time. For example, taking a look at the code of Stack0, a simple problem from Protostar.

```
int main(int argc, char **argv)
{
  volatile int modified;
  char buffer[64];

  modified = 0;
  gets(buffer);

  if(modified != 0) {
      printf("you have changed the 'modified' variable\n");
  } else {
      printf("Try again?\n");
  }
}
```

This program differs from a program such as binary_demo1 because the variable that needs to be modified is a result of a buffer overflow, rather than a direct input from the user. As a result, it is questionable whether KLEE could use the symbolic input and solver to get to the path where modified != 0.

When running the following command, KLEE will be able to identify a bug that there is a memory error within the gets function. However, it may not immediately clear where within the actual program the error is occurring. For example, if the program actually contained several calls to gets, it would not be clear which function the error was occurring in. More time will need to be spent investigating the features of KLEE to see if this information is attainable through KLEE.

```
klee -max-time=60 --libc=uclibc -posix-runtime stack0.o -sym-files 2 68
```

```
KLEE: ERROR: /home/ecwong/klee-uclibc/libc/string/strcpy.c:27: memory error: out of
    bound pointer
```

If KLEE is only able to identify the location of vulnerabilities, I wonder if it would be better to shift the project to automatically identifying inputs to the program, the minimum/maximum size of the input, and using KLEE to find locations within the code that an exploit could be inserted. Without performing dynamic analysis after using KLEE, it seems it would be difficult to automatically solve CTF problems. Trying to add dynamic analysis seems like it would greatly increase the scope of this project.

Input and Input Sizes

It seems the two main methods of getting input from a user in CTF problems will be either using command-line arguments or using a method to retrieve input from stdin. Although there are other methods such as reading from a file, these methods are not often present in CTF problems. Command-line arguments is the easier method to identify because of the two variables argc and argv which will be present in the function declaration of main. Using KLEE it will be easy to use symbolic input for command-line arguments without having to modify the source code of the problem.

For example, the following command will test binary_demo1 with a minimum of 1 and a maximum of 1 symbolic input which has a maximum size of 25. Running this command results in KLEE finding an input which will lead to finding the solution for this problem.

```
klee --optimize --libc=klee -posix-runtime binary_demo1.o -sym-arg 25
```

The second method of getting user input through different functions such as fgets, scanf and more is much more difficult to automatically identify. First, there are numerous different functions that can be used to collect input and each function may have different function arguments which will need to be identified. The function gets accepts only a single argument which is a pointer to where the input will be stored. Fgets on the other hand takes in 3 different arguments including a string pointer, number of bytes and a stream pointer. All the different functions and combination of arguments will lead to a more robust and complex identification process.

More coming soon!