CS 161

Computer Security

Dawn Song — Lab 2

Fall 2012 — Program Verification & Vulnerability Discovery

Due Date: Monday October 1, 2012 by 11:59pm

Introduction

In this exercise, we will reason about the correctness of real-world programs by applying manual and automated verification techniques. This lab is split into three sections:

1. *Program Verification (20 pts)* - We will manually prove whether the Merge-Sort program contains certain vulnerabilities by reasoning on its source code.
2. *Constraint Solving (45 pts)* - By invoking a constraint solver (Z3) to help us reason about a program, we will gain hands-on insight into symbolic execution and constraint solving, the underlying techniques of whitebox fuzzing.
3. *Whitebox Fuzzing and (Optionally) Blackbox Fuzzing (35 pts)* - Using a whitebox fuzzing tool (FuzzBALL), we will learn how to automatically discover vulnerabilities.

All files and detailed submission instructions for this lab are at the course site in Bspace, under Assignments → Lab 2.

Files included in this exercise

You should use the Lab 2 VM iamge (lab2.vmx) for this exercise. The user is "ubuntu" and the password is "reverse". It is a 32-bit Ubuntu 10.04 LTS VM augmented with the following files:

- `Z3`, a constraint solver from Microsoft Research (Q2).
- `example{1,2}.c`, example source files (Q2).
- `example{1,2}.smt` the encoded forms of `example{1,2}.c` as logical formulas in SMT-LIB format (Q2).
- `elfhash.c`, an algorithm to hash objects in the ELF format (Q2).
- `elfhash.smt`, *to be completed and submitted* (Q2).
- `FuzzBALL`, a dynamic symbolic execution tool distributed as a 32-bit x86 ELF binary (Q3), and `run-fuzzball.sh`, a bash srcipt to run it,

- `testme`, an executable extracted from BIND[1] (Q3).
- `testcase.init`, an initial input test case for `testme` (Q3).
- `submitWeb.py`, a python script to submit your files (Q2 and Q3).

Should you wish to complete this exercise without the use of the VM, visit the "Extra Notes" section at the bottom of this document.

1 Program Correctness (20 points)

Figure 1 shows an implementation of mergesort published in Wikibooks, retrieved Jan 6, 2012. Notice that at `line 31`, its anonymous author assures us the program has been tested. Are we sufficiently confident to use it? Let us verify its correctness or otherwise by answering the following questions.

1. Consider `line 29`. Can a double-free occur?
2. Consider `line 12`. Can a null-pointer dereference occur?
3. Consider `line 12`. Can a read out-of-bounds occur?
4. Consider `line 32`. Can infinite recursion occur?

No programming nor testing is required for this question. Note that you are *not* required to concern yourself with functional correctness, i.e., whether the array is indeed sorted on termination. Also, we will assume that (1) the entry point (first call to `MergeSort` at `line 32`) is **valid**, i.e., it starts off with an pre-allocated `int` array `tab` of size `count` and `count > 0`; (2) `line 6` calls a standard libc implementation of `malloc` (lookup its `man` pages if it is not already familiar); and (3) the program runs on a 32-bit x86 machine.

You are required to fill your answers into the assignment submission form for Lab 2 in Bspace. For each question, if your answer is 'yes', provide the *entry point* inputs (first call to `MergeSort` at `line 32`) that trigger the violation; if your answer is 'no', provide the invariant(s) at the considered line number.

[1] http://www.isc.org/software/bind

```
1  /* Mix two sorted tables in one and split the result into
      these two tables. */
2  int *Mix(int *tab1,int *tab2,int count1,int count2)
3  {
4    int i,i1,i2;
5    i = i1 = i2 = 0;
6    int * temp = (int *)malloc(sizeof(int)*(count1+count2));
7
8    while((i1<count1) && (i2<count2))
9    {
10     while((i1<count1) && (*(tab1+i1)<=*(tab2+i2)))
11     {
12       *(temp+i++) = *(tab1+i1);
13       i1++;
14     }
15     if (i1<count1)
16     {
17       while((i2<count2) && (*(tab2+i2)<=*(tab1+i1)))
18       {
19         *(temp+i++) = *(tab2+i2);
20         i2++;
21       }
22     }
23   }
24   memcpy(temp+i,tab1+i1,(count1-i1)*sizeof(int));
25   memcpy(tab1,temp,count1*sizeof(int));
26
27   memcpy(temp+i,tab2+i2,(count2-i2)*sizeof(int));
28   memcpy(tab2,temp+count1,count2*sizeof(int));
29   free(temp);
30 }
31 /* MergeSort a table of integer of size count. (Has been
      tested.) */
32 void MergeSort(int *tab,int count)
33 {
34   if (count==1) return;
35
36   MergeSort(tab,count/2);
37   MergeSort(tab+count/2,(count+1)/2);
38   Mix(tab,tab+count/2,count/2,(count+1)/2);
39 }
```

Figure 1: A mergesort implementation published in Wikibooks.

2 Constraint Solving (45 points)

We now take the first step towards reasoning automatically about programs using constraint solvers. The whitebox fuzzer that we will see in Q3 is a fully automated tool that performs symbolic execution to model an execution path into a path constraints formula, and then invokes a *constraint solver* to solve that formula. In this question, we will get to understand how symbolic execution works, and directly apply constraint solving to an interesting problem.

The interesting problem that we explore here is to find collisions in hash functions. Non-cryptographic hash functions are commonly used to support hash table lookups or to speed up the comparison of data. For example, the ELFhash function, shown in Figure 2, is used in Linux to compare objects in the ELF format. Given two different inputs, a hash collision occurs when their hash outputs are identical. Obviously, hash collisions are undesirable[2]. A good hash function minimizes the likelihood of collisions by distributing inputs uniformly over its output space. Despite the simplicity of the concept, there are few universally-accepted 'good' hash functions, and the design of 'good' hash functions remains an open problem.

One way to evaluate a hash function is to attempt to generate hash collisions. The two most straightforward approaches are brute-force and random enumeration of inputs (akin to blackbox fuzzing); in this exercise, we will attempt a different approach — we will apply constraint solving to help us 'solve' for collisions (akin to whitebox fuzzing).

Files included in this exercise:

You should find the following files in the lab2 directory:

- `Z3`, a constraint solver from Microsoft Research.
- `example{1,2}.c`, example source files.
- `example{1,2}.smt` the encoded forms of `example{1,2}.c` as logical formulas in SMT-LIB format.
- `elfhash.c`, an algorithm to hash objects in the ELF format.
- `elfhash.smt`, to be completed and submitted.

How to do constraint solving:

Before we can use a constraint solver, we need to learn how symbolic execution works, i.e., how it *systematically* constructs a constraint formula from code. The

[2]As you will learn later in this course, *cryptographic* hash functions are required to be *collision resistant*, i.e., it is computationally hard to find two different inputs with identical hash outputs. ELFhash is non-cryptographic.

```
unsigned int ELFHash(char* str, unsigned int len)
{
   unsigned int hash = 0;
   unsigned int x    = 0;
   unsigned int i    = 0;

   for(i = 0; i < len; str++, i++)
   {
      hash = (hash << 4) + (*str);
      x = hash & 0xF0000000L; /*hexadecimal long data type
         */
      if(x != 0)
      {
         hash ^= (x >> 24);
      }
      hash &= ~x;
   }

   return hash;
}
```

Figure 2: An ELFhash implementation

supplementary document (constraint_solving.pdf) will guide you through this construction. Be sure to thoroughly understand the material before you proceed.

Problem Instructions:

We are now ready to apply constraint solving to help us generate ELFhash collisions. To reduce your work, let us fix all ELFhash inputs to be of length 3 in this exercise (thus the `for` loop in Figure 2 is unrolled 3 times). Then perform the following steps.

1. Let's choose `"sec"` as a 3-letter input and ELFhash it (using `elfhash.c`). What is its ELFhash output? Be sure to fill your answer into the assignment submission form for lab 2 in Bspace.

2. Now construct an assertion at the end of the ELFhash function in `elfhash.c` such that the solver would print out an input that collides with `"sec"`. Complete `elfhash.smt` by encoding the ELFhash function in SMT-LIB. Then query the Z3 solver. What is the colliding input string that you found? You might want to lookup the ASCII table to convert ASCII values to characters. Be sure to fill your answer into the assignment submission form in Bspace.

3. Feed the colliding input back into ELFhash as done in step 1. Does it indeed colide with "`sec`"? If so, congratulations! You're now ready to submit your solution to Q2. Otherwise, you should debug your SMT-LIB encoding (say, by repeating the process from step 1 with a single letter).

4. Now submit your Q2 solution in two steps. First, type "`./submitWeb.py -q2`" in the lab directory at command-line and follow the instructions. This should produce a submission file `submit_lab2_q2.txt`. Then, upload this file as attachment to the Bspace assignment submission system.

5. Now modify `elfhash.smt` to obtain another colliding input. Then fill your answer into the assignment submission form in Bspace (if you do not find any more colliding inputs, enter "None"). You should *not* submit the modifications in this part.

3 Whitebox and Blackbox Fuzzing (35 points)

Security fuzz testing (or fuzzing, in short) is a critical step in the software development process for any program or system. It attempts to generate inputs that are not expected under typical usage conditions, which tend to be overlooked or poorly handled by programmers. Blackbox fuzzing is a common technique used by security researchers to evaluate the security of programs and systems without access to source code, and accounts for the vast majority of existing vulnerability disclosures. In contrast, whitebox fuzzing is a relatively newer approach that applies symbolic execution and constraint solving to systematically explore the program state space and to generate invalid inputs. In this exercise, we will experiment with whitebox followed (optionally) by blackbox fuzzing.

```
Potential Vulnerability : Callsite addr :   Callee   : Function
Write out-of-bounds     :  0x08048f6c   : 0x08049549 :  memcpy
Write out-of-bounds     :  0x080489ed   : 0x08049549 :  memcpy
Write out-of-bounds     :  0x080488c1   : 0x08049549 :  memcpy
```

Figure 3: Static analysis warnings

Figure 3 shows a subset of warnings generated from an initial static analysis on `testme`. It contains the locations of potential vulnerabilities found by a static analyzer, which cannot be confirmed due to limited precision. For simplicity, Figure 3 shows a subset of warnings that are potential write out-of-bounds limited to calls to the C library function `memcpy`. Simply treating all static analysis warnings as bugs or vulnerabilities would result in high false positives, so one major application

of symbolic execution is to check the validity of static analysis warnings to prune false positives, and generate inputs that trigger the true vulnerabilities. This is how we will use FuzzBALL in this exercise.

Files included in this exercise:

You should find the following files in the lab2 directory:

- `FuzzBALL`, an experimental research tool implementing dynamic symbolic execution, which can be applied to whitebox fuzzing.[3]
- `run-fuzzball.sh`, a bash srcipt to run `FuzzBALL`.
- `testme`, an executable extracted from BIND[4] and is the binary executable we are trying to test. It is compiled using `gcc` from a subset of BIND source code, without stripping its debug symbols (you might find this useful later).
- `testcase.init`, an initial input test case for `testme`.

Using FuzzBall:

To use FuzzBALL, we would have to construct 'vulnerability conditions'[5] from these warnings, so that FuzzBALL checks them when it encounters these locations during its systematic exploration of `testme`. Such vulnerability conditions can be generated automatically, but we will construct them manually in this exercise to gain a better understanding of how it works.

Example vulnerability condition	Description
`0x08048b41:` `R_EAX:reg32_t <> 0x01:reg32_t`	Register `EAX` is not equal to 1 when execution reaches (`EIP` is) `0x08048b41`

`0x080497dc:`
`mem[R_EBX:reg32_t + 0x21:reg32_t]`
`:reg32_t == 0:reg32_t`
	The value at memory location referenced by `EBX+0x21` is equal to 0 when execution reaches `0x080497dc`

`0x08049f77:`
`mem[R_EBP:reg32_t + 0x08:reg32_t]`
`:reg32_t <= R_EBX:reg32_t`
	The value at memory location referenced by `EBP+0x08` is less than or equal to `EBX` when execution reaches `0x08049f77`

Table 1: Example vulnerability conditions and descriptions.

Vulnerability conditions are specified to FuzzBALL in the Vine Intermediate Language[6]. Table 1 provides some example vulnerability conditions and the descriptions of what they mean. When a vulnerability condition is specified to FuzzBALL,

[3]FuzzBALL is a symbolic execution tool in the BitBlaze binary analysis platform http://bitblaze.cs.berkeley.edu/.

[4]http://www.isc.org/software/bind

[5]A vulnerability condition is the negation of a verification condition.

[6]The full syntax is described at http://bitblaze.cs.berkeley.edu/release/vine-1.0/howto.html#htoc19

it checks the satisfiability of the given condition when it is encountered during program exploration (the register `EIP` 'hits' the specified instruction address). It terminates when the condition is **satisfiable**, i.e., there exists an input that satisfies this condition in that execution path to trigger the vulnerability. Otherwise, it continues exploration. It explores different execution paths up to a pre-configured bound. You should first try running FuzzBALL with at least one of the conditions in Table 1 to familiarize yourself with FuzzBALL and its vulnerability conditions. Be sure to check your input syntax if you encounter exceptions. To run FuzzBALL, enter the following in its local directory:

```
./run-fuzzball.sh -check-condition-at '<condition>'
```

Do **not** modify the options within the script nor delete the `fuzzball.log` log that it generates, as your submission score will depend on it.

Problem Instructions:

We are now ready to embark on our bug-finding quest by answering the following questions.

1. Without knowing the precise upper bounds on the destination buffer sizes in `memcpy`, can you establish a crude upper-bound value on the `memcpy` length beyond which the call to `memcpy` *must* be an error? (Hint 1: `testme` is a 32-bit program. Hint 2: programs commonly assign `int` variables to `size_t`.) Fill your answer into the assignment submission form in Bspace.

2. Hence or otherwise, construct a vulnerability condition for each of the 3 warnings in Figure 3. (Hint: you might find running a disassembler such as "`objdump -D testme`" useful. Also lookup the `cdecl` calling convention.)

3. Now run FuzzBALL to check each of your vulnerability conditions separately. Which of the 3 warnings are true vulnerabilities? Fill your answer into the assignment submission form in Bspace.

4. Now submit your Q3 solution in two steps. First, type "`./submitWeb.py -q3`" in the lab directory at command-line and follow the instructions. This should produce a submission file `submit_lab2_q3.txt`. Then, upload this file as attachment to the Bspace assignment submission system.

Optional. You are encouraged to try out the blackbox fuzzing approach to find vulnerabilities in `testme`. Unlike whitebox fuzzing, the way it detects vulnerabilities is through crashing (e.g., on segmentation fault). We have left this as an optional exercise because whether you find vulnerabilities at all depends on your initial test cases and your fuzzing strategy (topped with a stroke of luck), so assigning scores based on that would be subjective. Nevertheless, we believe this would

be a fun exercise that does not take too much effort. For example, mutation-based fuzzers can be written in as little as five lines of code[7].

Submission and Grading

As you complete each question in this Lab, be sure to enter your answers into the assignment submission form in Bspace *and* submit your work using the submission script `submitWeb.py`. The following table describes the composition of scores in this lab.

Question	#. Bspace Form Entries to fill	Submission Data	Points
Q1: Program Correctness	8	-	20 pts
Q2: Constraint Solving	3	`elfhash.smt`	45 pts
Q3: Whitebox & Blackbox Fuzzing	2	`fuzzball.log`	35 pts
Total			100 pts

You are allowed to submit your solutions multiple times, and we will take the highest score into consideration.

Extra Notes

Installation for Other Platforms

Even if you choose not to use the provided VMWare virtual machine image, you should still work on this lab on Linux. FuzzBALL, the dynamic symbolic execution tool that you'll use in Q3, is a 32-bit x86 ELF binary that runs only on Linux and has been tested on 32-bit Ubuntu 10.04 LTS. If you prefer to use Virtual Box, you may obtain a bare Ubuntu image[8] and run that image as a virtual machine. Be sure to use only **32-bit VM images**. You may obtain a virtualbox image that works at `http://tinyurl.com/6my6tko` (660 MB with guest username/password: `ubuntu/reverse`).

On all platforms, you'll also need the following programs to be installed.

- `python`, used to submit your work, it may already be installed by default on Linux. Otherwise, it may be downloaded from `http://python.org`.

[7] Charlie Miller, "Babysitting an Army of Monkeys", CanSecWest 2010, available at http://securityevaluators.com/files/slides/cmiller_CSW_2010.ppt

[8] http://virtualboxes.org/images/ubuntu/

- `Z3`, used in Q2, it is distributed as part of the lab archive as a Linux binary (under the terms of Microsoft's license) obtained from `http://research.microsoft.com/projects/z3`.

Acknowledgments

This lab was prepared by Chia Yuan Cho. Thanks to Stephen McCamant for making FuzzBALL available in this lab, and Kunal Agarwal and Dylan Jackson for revisions.