6.858 Fall 2014 Lab 3: Symbolic execution

Handed out: Lecture 10
Part 1 due: Two days after Lecture 11 (5:00pm)
All parts due: Two days after Lecture 13 (5:00pm)

Introduction

This lab will introduce you to a powerful technique for finding bugs in software: *symbolic execution*. This can be a good way to audit your application for security vulnerabilities so that you can then fix them. By the end of this lab, you will have a symbolic execution system that can take the Zoobar web application and mechanically find inputs that trigger different kinds of bugs in Zoobar that can lead to security vulnerabilities. (More precisely, this lab will be building a concolic execution system; we will explain what this means later on.)

The KLEE paper describes a symbolic execution system for C programs. For simplicity, this lab will focus on building a symbolic/concolic execution system for Python programs, by modifying Python objects and overloading specific methods. Much like KLEE, we will be using an SMT solver to check for satisfiable constraints and come up with example inputs to the program we are testing. (As an aside, SMT stands for Satisfiability Modulo Theories, which means the solver is able check constraints that involve both traditional boolean satisfiability expressions as well as constraints that refer to other "theories" like integers, bit vectors, strings, and so on.)

In this lab, you will first familiarize yourself with the use of Z3, a popular SMT solver, by using it to find a correct way to compute the unsigned (and signed) average of two 32-bit values. You will then create wrappers for integer operations in Python (much like KLEE provides replacements for operations on symbolic values), and implement the core logic of invoking Z3 to explore different possible execution paths. Finally, you will explore how to apply this approach to web applications, which tend to handle strings rather than integer values. You will wrap operations on Python strings, implement symbolic-friendly wrappers around the SQLalchemy database interface, and use the resulting system to find security vulnerabilities in Zoobar.

Getting started

To fetch the new source code, first use Git to commit your solutions from lab 2, then run `git pull` to fetch the new code, and then check out the `lab3` branch. If you are using the provided .zip files, please download lab3.zip from the Labs section on the MIT OpenCourseWare site.

```
httpd@vm-6858:~$ cd lab
httpd@vm-6858:~/lab$ git commit -am 'my solution to lab2'
[lab1 f54fd4d] my solution to lab2
 1 files changed, 1 insertions(+), 0 deletions(-)
httpd@vm-6858:~/lab$ git pull
...
httpd@vm-6858:~/lab$ git checkout -b lab3 origin/lab3
Branch lab3 set up to track remote branch lab3 from origin.
Switched to a new branch 'lab3'
httpd@vm-6858:~/lab$
```

WARNING: do *not* merge your lab 2 solutions into the lab 3 source code! Our basic symbolic execution system cannot track symbolic constraints across RPC between multiple processes, so we will focus on using symbolic execution on a non-privilege-separated Zoobar site. The symbolic execution system will also help us find bugs that privilege separation does not fully mitigate.

Next, make sure that the lab 3 source code is running correctly in your VM, by running `make check`. As shown below, this command should report that all checks fail, but if you get other error messages, stop and figure out what went wrong (perhaps by contacting the course staff or asking on Piazza).

```
httpd@vm-6858:~/lab$ make check
./check_lab3.py
FAIL Exercise 1: unsigned average
FAIL Challenge 1: signed average
FAIL Exercise 2: concolic multiply
FAIL Exercise 2: concolic divide
FAIL Exercise 2: concolic divide+multiply+add
FAIL Exercise 3: concolic execution for integers
FAIL Exercise 4: concolic length
FAIL Exercise 4: concolic contains
FAIL Exercise 4: concolic execution for strings
FAIL Exercise 5: concolic database lookup (str)
FAIL Exercise 5: concolic database lookup (int)
FAIL Exercise 5: eval injection not found
FAIL Exercise 6: balance mismatch not found
FAIL Exercise 6: zoobar theft not found
PASS Exercise 7: eval injection not found
PASS Exercise 7: balance mismatch not found
PASS Exercise 7: zoobar theft not found
httpd@vm-6858:~/lab$
```

One common failure is that you have a leftover `zoobar/db` directory from lab 2, owned by a user other than `httpd`. If you are getting an exception when running `make check`, try removing the `zoobar/db` directory as `root`, by running `sudo rm -rf zoobar/db`.

Note that parts of the concolic execution system are fairly CPU-intensive. If you are running QEMU without KVM support, you might observe that the last check (for exercise 6) can take a very long time (over 5 minutes). Consider enabling KVM or using some other low-overhead VMM (like VMware).

Using an SMT solver

A key piece of machinery used by symbolic execution is an SMT solver. For this lab, you will be using the Z3 solver from Microsoft Research. Since our goal is to look for bugs in Zoobar, which performs a lot of string processing, we will use the Z3-str extension of Z3 which adds support for reasoning about constraints on strings. We will invoke Z3 using its Python-based API; you may find it useful to consult the documentation for Z3's Python API. The lab comes with a pre-built binary of Z3-str; it was built from our modified version of Z3-str.

As a first step to learn about Z3, we will use it to help us implement a seemingly simple but error-prone piece of code: *computing the average of two 32-bit integers*. This is surprisingly subtle to do correctly. One naive approach to compute the average of `x` and `y` might be to use `(x+y)/2`. However, if both `x` and `y` are large, their sum `x+y` might overflow and wrap around modulo 2^{32}, so `(x+y)/2` will not be the correct average value. Infact, integer overflow errors are a significant source of security problems for systems code (check out the KINT paper if you are curious to learn more about that), so it is important to know how to write this code correctly.

Z3 can help us get a correct implementation of the averaging function by checking whether a particular implementation we have in mind is correct. In particular, given a boolean expression, Z3 can tell us whether it's possible to make that boolean expression true (i.e., satisfy it). Moreover, if it's possible to make the expression true, and the expression contains some variables, Z3 will give us an example assignment of values to these variables which makes the expression true.

To see how we can use Z3, take a look at the code we provided for you in `int-avg.py`. The first few lines construct two 32-bit variables called `a` and `b`. The next line tries to compute the *unsigned* average of `a` and `b` (that is, treating both `a` and `b` as unsigned integers) and stores it in `u_avg`. Note that this code does *not* actually perform the addition and division. Instead, it constructs a symbolic expression representing these operations, and Z3 will reason about the possible values of this expression later on. You can observe this just by printing out the `u_avg` variable:

```
httpd@vm-6858:~/lab$ python
Python 2.7.6 (default, Mar 22 2014, 22:59:38)
[GCC 4.8.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import symex.z3str
>>> import z3
>>> a = z3.BitVec('a', 32)
>>> b = z3.BitVec('b', 32)
>>> u_avg = z3.UDiv(a + b, 2)
>>> print u_avg
UDiv(a + b, 2)
>>> s_avg = (a + b) / 2
>>> print s_avg
(a + b)/2
>>>
```

As the comment in the source code says, you should beware of the difference between signed and unsigned integer operations. In Z3, the default Python operators for division (`/`) and right-shifting (`>>`) treat bit vectors as signed values. When you want to perform unsigned operations, you should use the z3.UDiv and z3.LShR functions instead.

The initial code computes `u_avg` in the naive way that we discussed above, and is not always correct. Let's now see how Z3 can help us spot this mistake. In the rest of `int-avg.py`, we compute a reference value representing the expected, correct average of `a` and `b`. To compute this reference value, we "cheat": we actually turn both `a` and `b` into 33-bit integers (one bit more than before) using z3.ZeroExt. We then compute their average using the naive method, but since we are using 33-bit arithmetic, there is no overflow, and the naive method works correctly. We finally truncate the 33-bit value back down to 32 bits, which is also safe to do (because the average will always fit in 32 bits), and store the resulting symbolic expression in `real_u_avg`.

Now, to check whether `u_avg` computed the average correctly, we just ask Z3 whether it's possible to satisfy the expression `u_avg != real_u_avg`, meaning our `u_avg` value is not correct. In this case, since our naive 32-bit averaging is broken, the expression is satisfiable, and Z3 tells us so (for now, you can ignore the second part about the signed average):

```
httpd@vm-6858:~/lab$ ./int-avg.py
Checking unsigned avg using Z3 expression:
    UDiv(a + b, 2) !=
    Extract(31, 0, UDiv(ZeroExt(1, a) + ZeroExt(1, b), 2))
  Answer for unsigned avg: sat
  Example solution: [b = 2147483616, a = 4261412832]
Checking signed avg using Z3 expression:
    (a + b)/2 !=
    Extract(31, 0, (SignExt(1, a) + SignExt(1, b))/2)
  Answer for signed avg: sat
  Example solution: [b = 402641344, a = 1744842304]
httpd@vm-6858:~/lab$
```

As you can see, Z3 tells us that the expression is satisfiable, and gives us an example assignment of values to both `a` and `b` for which the expression is true. Indeed, these example values look quite large, and their sum clearly exceeds 2^{32}, breaking our naive averaging method.

Exercise 1. Implement a correct function to compute the unsigned average of `a` and `b`, by modifying the `u_avg = ...` line in `int-avg.py`.

For the purposes of this exercise, you are not allowed to change the bit-widths of your operands. This is meant to represent the real world, where you cannot just add one more bit to your CPU's register width.

You may find it helpful to search online for correct ways to perform fixed-width integer arithmetic. The book Hacker's Delight by Henry S. Warren is a particularly good source of such tricks.

Check your averaging function by re-running **`./int-avg.py`** or **`make check`**. If your implementation is correct, `int-avg.py` should produce the message `Answer for unsigned avg: unsat`.

Challenge! (optional) For extra credit, figure out how to compute the average of two 32-bit *signed* values. Modify the `s_avg = ...` line in `int-avg.py`, and run **`./int-avg.py`** or **`make check`** to check your answer. Keep in mind the direction of rounding: 3/2=1 and -3/2=-1, so so the average of 1 and 2 should be 1, and the average of -2 and -1 should be -1.

Interlude: what are symbolic and concolic execution?

As you probably recall from the KLEE paper, symbolic execution is an approach for testing a program by observing how the program behaves on different possible inputs. Typically, the goal of symbolic execution is to achieve high code coverage or path coverage on the program. In the context of security, this is useful because it helps explore rare code paths that might contain vulnerabilities but that aren't being triggered in typical executions of the code. At a high level, if we are building a symbolic execution system, we have to address several points:

1. As the program constructs intermediate values based on the input (e.g., taking two input integer values, computing their average, and storing that in some variable), we need to remember the relation between the input and these intermediate values. Typically this is done by allowing variables or memory locations to have either *concrete* or *symbolic* values. A concrete value is what an ordinary program would store in a variable or memory location: some specific value, such as the integer 42. A symbolic value is not a specific value but rather a symbolic expression describing what the value *would be* as a function of the inputs, such as `(a+b)/2`. This is similar to the symbolic Z3 expression you constructed for `u_avg` in the first exercise above.

2. We need to determine what control flow decisions (branches) the application makes based on the input. This boils down to constructing a symbolic constraint every time the program branches, describing the boolean condition (in terms of the program's original input) under which the program takes some particular branch (or does not). Since we are keeping track of how all intermediate values are related to the program's original input, using symbolic values, we typically do not need to look back at the original input to make these constraints. These constraints are similar to the constraint you used above to look for bugs in the integer average

function. Determining these control flow constraints is important because if the program initially goes one particular way at some branch, we would like to figure out how to get it to go down the other way, to see if there are interesting bugs in that other code. In KLEE's case, they build an interpreter for LLVM bytecode, and this interpreter knows the behavior of all branching instructions.

3. For each of the above branches, we need to decide if there's a possible input that will cause the program to execute the other way at a branch. (More generally, we often think of entire control flow paths, rather than individual branches in isolation.) This helps us find control flow decisions in a program that we can affect by tweaking the input (as opposed to control flow decisions that will always go a certain way in a program, regardless of the input we are considering). All symbolic execution systems rely on some kind of SMT solver to do this.

4. We need to specify what we are looking for in our testing. Typically this is best thought of in terms of some invariant that you care about ensuring in your program, and symbolic execution looks for inputs that violate this invariant. One thing we could look for is crashes (i.e., the invariant is that our program should never crash). Looking for crashes makes a lot of sense in the context of C programs, where crashes often indicate memory corruption which is almost certainly a bug and often could be exploited. In higher-level languages like Python, memory corruption bugs are not a problem by design, but we could still look for other kinds of issues, such as Python-level code injection attacks (some part of the input gets passed into `eval()`, for example), or application-specific invariants that matter for security.

5. Finally, given all of the control flow paths though the program that are possible to execute, we need to decide which path to actually try. This is important because there can be exponentially many different paths as the size of the program gets larger, and it quickly becomes infeasible to try all of them. Thus, symbolic execution systems typically include some kind of *scheduler* or *search strategy* that decides which path is the most promising in terms of finding violations of our invariant. A simple example of a search strategy is trying branches that we haven't tried before, in hopes that it will execute new code that we haven't run yet; this will lead to higher code coverage, and perhaps this new code contains a bug we haven't run into yet.

An alternative to symbolic execution is fuzzing (also called fuzz-testing). Fuzzing takes a randomized approach: instead of trying to carefully reason about what inputs will trigger different code paths in the application, fuzzing involves constructing concrete random inputs to the program and checking how the program behaves. This has the advantage of being relatively easy, but on the other hand, it can be difficult to construct precise inputs that hit some specific corner case in the application code.

One challenge in building a symbolic execution system, such as KLEE, is that your system has to know how to execute all possible operations on symbolic values (steps 1 and 2 above). In the case of KLEE, which works at the level of LLVM bytecode, this means that KLEE has to understand how every LLVM opcode works. In this lab, we are going to interpose at the level of Python objects (in particular, integers and strings). This is challenging for symbolic execution because there are a very large number of operations that one can do on these Python objects, so building a complete symbolic execution system for such a high-level interface would be a tedious process.

Luckily, there is an easier option, called *concolic execution*, which you can think of as somewhere in the middle between completely random fuzzing and full symbolic execution. The idea is that, instead of keeping track of purely symbolic values (like in KLEE), we can store both a concrete *and* a symbolic value for variables that are derived from the input. (The name *concolic* is a combination of *concrete* and *symbolic*.) Now that we have both a concrete and a symbolic value, we can almost get the best of both worlds:

- If the application performs some operation that our concolic system knows about, we will run pretty much like symbolic execution (except that we will also propagate the concrete part of every value). For instance, suppose we have two concolic integer variables `aa` and `bb`, whose concrete values are 5 and 6, and whose symbolic expressions are `a` and `b`. If the application stores `aa+bb` into variable `cc`, variable `cc` will now have concrete value 11 and symbolic expression `a+b`. Similarly, if the application branches on `cc==12`, the program can execute as if the branch condition was false (since 11 != 12) and record the corresponding symbolic branch condition (`a+b != 12`).

- If, on the other hand, the application performs some operation that our concolic system does not know about, the application will just get the concrete value. For example, if the application writes the variable `cc` to a file, or perhaps passes it to some external library that we don't instrument, the code can still execute, using the concrete value 11 as if the application was just running normally.

The benefit of concolic execution, for the purposes of this lab, is that we do not need to be complete in terms of supporting operations on symbolic values. As long as we support enough operations to find interesting bugs that we care about in the application, the system will be good enough (and in practice, most bug finding systems are approximate anyway, since it's usually infeasible to find all bugs). The trade-off is of course that, if the application performs some operations we do not support, we will lose track of the symbolic part, and will not be able to do symbolic-execution-style exploration of those paths.

Concolic execution for integers

To start with, you will implement a concolic execution system for integer values. The skeleton code that we provide you for concolic execution is in `symex/fuzzy.py` in your lab directory. There are several important layers of abstraction that are implemented in `fuzzy.py`:

- **The AST.** Instead of using Z3 expressions to represent symbolic values, as you did in the `int-avg.py` exercise above, we build our own abstract syntax tree (AST) to represent symbolic expressions. An AST node could be a simple variable (represented by a `sym_str` or `sym_int` object), a constant value (represented by a `const_int`, `const_str`, or `const_bool` object), or some function or operator that takes other AST nodes as arguments (e.g., `sym_eq(a, b)` to represent the boolean expression `a==b` where `a` and `b` are AST nodes, or `sym_plus(a, b)` to represent the integer expression `a+b`).

  Every AST node `n` can be converted into a Z3 expression by calling `z3expr(n)`. This works by calling `n._z3expr()`, and every AST node implements the `_z3expr` method that returns the corresponding Z3 representation.

  The reason we introduce our own AST layer, instead of using Z3's symbolic representation, is that we need to perform certain manipulations on the AST that are difficult to do with Z3's representation. Furthermore, we need to fork off a separate process to invoke Z3's solver, so that in case the Z3 solver takes a really long time, we can time out, kill that process, and assume the constraint is just unsolvable. (In this case, we might miss those paths, but at least we will make progress exploring other paths.) Having our own AST allows us to cleanly isolate Z3 state to just the forked process.

- **The concolic wrappers.** To intercept Python-level operations and perform concolic execution, we replace regular Python `int` and `str` objects with concolic subclasses: `concolic_int` inherits from `int` and `concolic_str` inherits from `str`. Each of these concolic wrappers stores a concrete value (in `self.__v`) and a symbolic expression (an AST node, in `self.__sym`). When the application computes some expression derived from a concolic value (e.g., `a+1` where `a` is a `concolic_int`), we need to intercept the operation and return another concolic value containing both the concrete result value and a symbolic expression for how the result was computed.

  To perform this interception, we overload various methods on the `concolic_int` and `concolic_str` classes. For example, `concolic_int.__add__` is invoked when the application computes `a+1` in the above example, and this method returns a new concolic value representing the result.

  In principle, we should have a `concolic_bool` that is a subclass of `bool` as well. Unfortunately, `bool` cannot be subclassed in Python (see here and here). So, we make `concolic_bool` a function that logically pretends that, once you construct a concolic boolean value, the program immediately branches on its value, so `concolic_bool` also adds a constraint to the current path condition. (The constraint is that the symbolic expression of the boolean value is equal to the concrete value.) The `concolic_bool` function then returns a concrete boolean value.

- **The concrete inputs.** The input to the function being tested under concolic execution is represented by a Python dictionary, which maps input variable names (regular Python strings) to the value of that variable. The value is a regular Python integer (for integer variables) or a regular Python string (for string variables). The current input is always stored in the `concrete_values` dictionary.

  The reason this is a global variable is that applications create concolic values by invoking `fuzzy.mk_str(name)` or `fuzzy.mk_int(name)` to construct a concolic string or integer, respectively. This returns a new concolic value, whose symbolic part is a fresh AST node corresponding to a variable named `name`, but whose concrete value is looked up in the `concrete_values` dictionary. If there is no specific value assigned to that variable in `concrete_values`, the system defaults to some initial value (0 for integers and the empty string for strings).

The concolic execution framework maintains a queue of different inputs to try, in an `InputQueue` object (also defined in `symex/fuzzy.py`). The concolic execution framework first adds an initial input (the empty dictionary `{}`), and then runs the code. If the application makes any branches, the concolic execution system will invoke Z3 to come up with new inputs to test other paths in the code, add those inputs to the input queue, and keep iterating until there are no more inputs to try.

- **The SMT solver.** The `fork_and_check(c)` function checks whether constraint `c` (an AST) is a satisfiable expression, and returns a pair of values: the satisfiability status `ok` and the example model (assignment of values to variables) if the constraint is satisfiable. The `ok` variable is `z3.sat` if the constraint is satisfiable, and `z3.unsat` or `z3.unknown` otherwise. Internally, this function forks off a separate process, tries to run the Z3 solver, but if it takes longer than a few seconds (controlled by `z3_timeout`), it kills the process and returns `z3.unknown`.
- **The current path condition.** When the application executes and makes control flow decisions based on the value of a concolic value (see discussion of `concolic_bool` above), the constraint representing that branch is appended to the `cur_path_constr` list. To help with debugging and search heuristics, information about the line of code that triggered this branch is added to the `cur_path_constr_callers` list.

Now, your job will be to finish the implementation of `concolic_int`, and then implement the core of the concolic execution loop. We provide two test programs for you, `check-concolic-int.py` and `check-symex-int.py`. Take a look at these programs to get a sense of how we are using concolic execution, and what code these test cases are invoking.

Exercise 2. Finish the implementation of `concolic_int` by adding support for integer multiply and divide operations. You will need to overload additional methods in the `concolic_int` class, add AST nodes for multiply and divide operations, and implement `_z3expr` appropriately for those AST nodes.

Look for the comments `Exercise 2: your code here` in `symex/fuzzy.py` to find places where we think you might need to write code to solve this exercise.

Run **`./check-concolic-int.py`** or **`make check`** to check that your changes to `concolic_int` work correctly.

Exercise 3. Implement the core concolic execution logic in `concolic_test()` in `symex/fuzzy.py` to get concolic execution working. Look for the comment `Exercise 3: your code here` and read the comment below that for a proposed plan of attack for implementing that loop.

Run **`./check-symex-int.py`** or **`make check`** to check that your changes to `concolic_test()` work correctly.

Beware that our check for this exercise is *not* complete. You may well find that later on something does not work, and you will have to revisit your code for this exercise.

This completes part 1 of this lab.

Submit your answers to the first part of this lab assignment by running **`make submit-a`**. Alternatively, run **`make prepare-submit-a`**. The resulting `lab3a-handin.tar.gz` file will be graded.

Concolic execution for strings and Zoobar

Before we can run the entire Zoobar application through our concolic system, we have to first add support for strings, in addition to support for integers we added above. This is now your job.

Exercise 4. Finish the implementation of concolic execution for strings in `symex/fuzzy.py`. We left out support for two operations on `concolic_str` objects. The first is computing the length of a string, and the second is checking whether a particular string `a` appears in string `b` (i.e., `a` is contained in `b`).

Look for the comment `Exercise 4: your code here` to find where you should implement the code for this exercise. We have already implemented the `sym_contains` and `sym_length` AST nodes for you, which should come in handy for this exercise.

Run **`./check-concolic-str.py`** *and* **`./check-symex-str.py`** (or just run **`make check`**) to check that your answer to this exercise works correctly.

In addition to performing string operations, Zoobar's application code makes database queries (e.g., looking up a user's `Profile` object from the profile database). Our plan is to supply a concolic HTTP request to Zoobar, so the username being looked up is going to be a concolic value as well (coming through the HTTP cookie). But how can we perform SQL queries on a concolic value? We would like to allow the concolic execution system to somehow explore all possible records that it could fetch from the database, but the SQL query goes into SQLite's code which is written in C, not Python, so we cannot interpose on that code.

Exercise 5. Figure out how to handle the SQL database so that the concolic engine can create constraints against the data returned by the database. To help you do this, we've written an empty wrapper around the sqlalchemy `get` method, in `symex/symsql.py`. Implement this wrapper so that concolic execution can try all possible records in a database. Examine **`./check-symex-sql.py`** to see how we are thinking of performing database lookups on concolic values.

You will likely need to consult the reference for the SQLalchemy query object to understand what the behavior of `get` should be and what your replacement implementation should be doing.

Run **`./check-symex-sql.py`** (or just run **`make check`**) to check that your answer to this exercise works correctly.

Now that we can perform concolic execution for strings and even database queries, we can finally try running Zoobar under concolic execution. Take a look at the `check-symex-zoobar.py` program to see how we can invoke Zoobar on symbolic inputs. To make sure that concolic execution of Zoobar finishes relatively quickly, we over-constrain the initial inputs to Zoobar. In particular, we specify that the HTTP method (in `environ['REQUEST_METHOD']`) is always `GET`, and that the URL requested (in `environ['PATH_INFO']`) always starts with `trans`. This greatly cuts down the number of possible paths that concolic execution must explore; making these inputs arbitrary leads to about 2000 paths that must be explored, which takes about 10 minutes.

Try running `check-symex-zoobar.py` to make sure that all of the code you've implemented so far in this lab works properly. We suggest running it inside of `script` so that the output is saved in the `typescript` file for later inspection:

```
httpd@vm-6858:~/lab$ script -c ./check-symex-zoobar.py
Script started, file is typescript
Trying concrete values: {}
startresp 404 NOT FOUND [('Content-Type', 'text/html'), ('Content-Length', '233'), ('X-XSS-Protection', '0')]
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
<title>404 Not Found</title>
```

```
<h1>Not Found</h1>
<p>The requested URL was not found on the server.  If you entered the URL manually please check your spelling and try again.</p>

Trying concrete values: {'referrer': '', 'cookie': '', 'path': 'fer'}
/home/httpd/lab/zoobar/debug.py:23 :: __try : caught exception in function transfer:
 Traceback (most recent call last):
  File "/home/httpd/lab/zoobar/debug.py", line 20, in __try
    return f(*args, **kwargs)
  File "/home/httpd/lab/zoobar/login.py", line 59, in loginhelper
    if not logged_in():
  File "/home/httpd/lab/zoobar/login.py", line 50, in logged_in
    g.user.checkCookie(request.cookies.get("PyZoobarLogin"))
  File "/usr/lib/python2.7/dist-packages/werkzeug/local.py", line 338, in __getattr__
    return getattr(self._get_current_object(), name)
  File "/usr/lib/python2.7/dist-packages/werkzeug/utils.py", line 71, in __get__
    value = self.func(obj)
  File "/home/httpd/lab/symex/symflask.py", line 44, in cookies
    fuzzy.require(hdr == name + '=' + val)
  File "/home/httpd/lab/symex/fuzzy.py", line 362, in require
    raise RequireMismatch()
RequireMismatch

startresp 500 INTERNAL SERVER ERROR [('Content-Type', 'text/html'), ('Content-Length', '291')]
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
<title>500 Internal Server Error</title>
<h1>Internal Server Error</h1>
<p>The server encountered an internal error and was unable to complete your request.  Either the server is overloaded or there is an er
</p>

Trying concrete values: {'referrer': '', 'path': 'fer', 'cookie_name': 'v', 'cookie': 'v=aC', 'cookie_val': 'aC'}
...
Stopping after 139 iterations
Script done, file is typescript
httpd@vm-6858:~/lab$
```

So what bugs did we find in these 139 different paths? Since Zoobar is written in Python, there are no memory corruption or crash bugs, so it's not immediately clear how we could tell that there's a problem. As a result, we have to write explicit invariants to catch bad situations indicative of a security problem.

One such invariant we have supplied for you is a check for eval injection; that is, arbitrary input being passed to the `eval()` function in Python. Examine `symex/symeval.py` to see how we check for eval injection. We make an approximation that seems reasonable in practice: if the string passed to `eval()` can ever contain `;badstuff();` somewhere in the string, it's a good bet that we have an eval injection vulnerability. Since the `eval` implementation is written in Python, the check `';badstuff();' in expr` invokes your overloaded method in `concolic_str`, and this tells the concolic execution system to try to construct an input that contains that substring.

You can see whether Zoobar contains any such bugs by looking for the string printed by that check in `symex/symeval.py` in the output from `check-symex-zoobar.py`:

```
httpd@vm-6858:~/lab$ grep "Exception: eval injection" typescript
Exception: eval injection
Exception: eval injection
httpd@vm-6858:~/lab$
```

It looks like the concolic execution system found two different inputs that lead to our "eval injection" check. Now you could look at the lines just before that message to see what concrete input triggers eval injection. This can greatly help a developer in practice to find and fix such a bug.

Now, your job will be to implement two additional invariant checks to see whether Zoobar balances can ever be corrupted. In particular, we want to enforce two guarantees:

- If no new users are registered, the sum of Zoobar balances in all accounts should remain the same before and after every request. (That is, zoobars should never be created out of thin air.)
- If a user `u` does not issue any requests to Zoobar, `u`'s Zoobar balance should not shrink. (That is, it should be impossible for requests by one user to steal another user's zoobars.)

Exercise 6. Add invariant checks to `check-symex-zoobar.py` to implement the above two rules (total balance preservation and no zoobar theft). Look for the comment `Exercise 6: your code here` to see where you should write this code. When you detect a zoobar balance mismatch, call the `report_balance_mismatch()` function. When you detect zoobar theft, call `report_zoobar_theft()`.

Recall that our check for exercise 3, where you implemented the core of the concolic execution system, was not complete. If you are having trouble with this exercise, it may be that you did not implement exercise 3 correctly, so you may need to go back and fix it.

To check whether your solution works correctly, you need to re-run **`./check-symex-zoobar.py`** and see whether the output contains the messages `WARNING: Balance mismatch detected` and `WARNING: Zoobar theft detected`. Alternatively, you can run **`make check`**, which will do this for you (run `check-symex-zoobar.py` and look for these magic strings).

Finally, your job is to fix these two bugs (zoobar balance mismatch and zoobar theft), and make sure that your symbolic execution system properly reports that these bugs are no longer reachable. To fix the bugs, we ask that you do *not* modify the original Zoobar source code in the `zoobar` directory, so that `make check` can continue to work. Instead, please make a copy of any `.py` file you want to fix from the `zoobar` directory into `zoobar-fixed`. Then use the `check-symex-zoobar-fixed.sh` script to see if your fixes work properly.

We already fixed the eval injection bug for you, in `zoobar-fixed/transfer.py`. You should also make sure that your concolic execution system does not report eval injection bugs anymore.

Exercise 7. Fix the two bugs you found in Exercise 6, by copying whatever `.py` files you need to modify from the `zoobar` directory to `zoobar-fixed` and changing them there.

Recall that our check for exercise 3, where you implemented the core of the concolic execution system, was not complete. If you are having trouble with this exercise, it may be that you did not implement exercise 3 correctly, so you may need to go back and fix it.

To check whether your solution works correctly, you need to run **`./check-symex-zoobar-fixed.sh`** and see whether the output still contains the messages `Exception: eval injection`, `WARNING: Balance mismatch detected` and `WARNING: Zoobar theft detected`. Alternatively, you can run **`make check`**, which will do this for you.

You are done! Submit your answers to the lab assignment by running **`make submit`**. Alternatively, run **`make prepare-submit`**. The resulting `lab3-handin.tar.gz` file

will be graded.

6.858 Computer Systems Security

Fall 2014