Lab 2
Information Flow

15-316: Software Foundations of Security & Privacy
Frank Pfenning

Due Wednesday, Nov 13, 2024
125 points

In this lab you will explore information flow from two perspectives.

In Part 1 you have to **attack** several servers that we make available on the `linux.andrew.cmu.edu` machines in order to extract secrets. You will then submit the secrets as text files to Gradescope to receive credit. Your file names should be `tiny_serve1.txt`, `tiny_serve2.txt`, etc, where each file contains a single line with the secret you discovered as discussed below.

In Part 2 you have to write a server that defends against a specific class of attacks. You will submit your server code to the autograder, where we will test it against a collection of attacks. The file `lab2.zip` should contain your server code. As in Lab 1, we will call

```
make
```

after unzipping this file. This should create an executable `tiny_serve` satisfying the spec given below.

Changes in the Tiny Language

It is likely to be useful to start with your solution to Lab 1, since there is no separate starter code for Lab 2. You can also download the starter code for Lab 1 in a different language and port some of your code from Lab 1 or start from scratch.

There some changes to the syntax and the semantics, so you will have to change the lexer and parser. We found the changes to be minor and straightforward.

Syntax

We add two new commands

```
<prog> ::= ... (all constructs so far) ...
         | <var> ':=' 'div' '(' <exp> ',' <exp> ')'
         | <var> ':=' 'mod' '(' <exp> ',' <exp> ')'
```

In addition to `#input` and `#output` we also have a new special variable `#secret`.

Evaluation

Because safety is no longer the focus it was in Lab 1, we conflate unsafe behavior with abort. In other words, any unsafe behavior (which comes only from memory read/write, div, and mod) must be detected and abort the program. `x := div(e1, e2)` and `x := mod(e1, e2)` abort unless $e_2 > 0$. Memory access performs the same bounds check as in Lab 1, but aborts instead of signaling unsafe behavior. In addition, the **test** P aborts as before when P evaluates to $\bot$.

Assertions, loop invariants, and preconditions are ignored, as they are designed for proving the program safe (or correct), which is not in scope for this lab.

Security Policy

The language has two security levels, high (H) and low (L) with $\mathsf{L} \sqsubset \mathsf{H}$. The variable `#secret` is of level H, as is the contents of memory. All other variables you might use (including `#input` and `#output`) are of level L.

We do not reveal which form of noninterference our servers implement. These are based on taint analysis, termination-insensitive, termination-sensitive, and timing-sensitive noninterference, but may in addition have bugs. One of the servers is intended to be secure with all channels we covered in lecture (implicit flows, termination, and timing), so you should not expect to crack them all. If you do, please let us know how!

Part 1: The Attack [4 * 15 = 60 points]

The servers `tiny_serve<n>` are invoked as follows on the `linux.andrew` machines:

```
% ~fp/bin/tiny_serve<n> <userid> <filename> <input>
```

where $n \in \{1, 2, 3, 4, 5\}$, *userid* is you or your partner's Andrew id, *filename* is the name of the TINY file containing your attack, and *input* is an integer containing some input. The secret will always be an unsigned 62 bit number (don't ask) and is compared with your output. Correspondingly, the text files you submit to Gradescope should be named `tiny_serve<n>.txt`, each containing a single line with the secret from the corresponding server.

Important: We use the *userid* to generate a unique secret for each server and user. If you work in a team, you both must use the same user id (although it doesn't matter who runs the server). Furthermore, this user should hand in the discovered secrets to Gradescope.

The server may not terminate on your input (and you have to interrupt it). If it does terminate, it will print one of the following:

- `success <output>` (with exit code 0). Your output was the secret.
- `error` (with exit code 1). The file does not exist, or the file or input does not have the correct format.
- `failure <output>` (with exit code 2). Your program was judged secure, but your output was **not** the secret.
- `abort` (with exit code 3). The program aborted.
- `insecure` (with exit code 4). The program was rejected as insecure.
- `undefined` (with exit code 5). Undefined variable.

Part 2: The Server [65 points]

Your implementation should be termination-sensitive, where nontermination (that is, the absence of a proper poststate) may be due to an abort or an infinite loop. In other words, neither an abort nor an infinite loop should leak information, but a timing channel might still exist.

Your server, `tiny_serve`, should parse a file and perform the following steps:

1. Verify that the program does not reference an undefined variable, where `#input` and `#secret` should be assumed to be defined, and `#output` must be defined at the end of the program. This is almost exactly as in Lab 1.
2. Use the information flow type system to determine whether your program is secure in the termination-sensitive way explained above.
3. **Your server should not execute the given program.** Executing the program may be helpful for you in debugging, but the autograder will simply check if your server classifies each test file correctly as secure or insecure.

Your server should return one of the following exit codes:

- `secure` (with exit code 0). The program is secure.
- `error` (with exit code 1). The file does not exist, or the file or input does not have the correct format.
- `insecure` (with exit code 4). The program is rejected as insecure.
- `undefined` (with exit code 5). Undefined variable.

The exit codes 2 and 3 are not relevant, since your server does not execute the program. The exit code 0 does not print the value of `#output` since it does not execute the program.

Language Reference

Below is a specification of the grammar of Tiny.

```
<idstart> := [a-z_]
<idchar> := [a-z_0-9]

<var> := <idstart> <idchar>*
       | '#input' | '#output'
       | '#secret'              (* new *)

<num> := [0-9]+

<exp> ::= <num>
        | <var>
        | <exp> '+' <exp>
        | <exp> '-' <exp>
        | <exp> '*' <exp>
        | '(' <exp> ')'
```

```
<form> ::= 'true'
         | 'false'
         | <exp> '<' <exp>
         | <exp> '<=' <exp>
         | <exp> '==' <exp>
         | <form> '/\' <form>
         | <form> '\/' <form>
         | <form> '->' <form>
         | '~' <form>
         | '(' <form> ')'

<prog> ::= <var> ':=' <exp>
         | <var> ':=' 'div' '(' <exp> ',' <exp> ')'   (* new *)
         | <var> ':=' 'mod' '(' <exp> ',' <exp> ')'   (* new *)
         | <var> ':=' 'M' '[' <exp> ']'
         | 'M' '[' <exp> ']' ':=' <exp>
         | <prog> ';' <prog>
         | 'assert' <form>
         | 'test' <form>
         | 'if' <form> 'then' <prog> 'else' <prog> 'endif'
         | 'while' <form> 'invariant' <form> 'do' <prog> 'done'

<file> ::= 'requires' <form> ';' <prog>
```

Ambiguities are resolved as follows:

- Arithmetic operators are left associative, with precedence $\star > + - $ (where $+$ and $-$ have equal precedence).
  This means that `3 * 4 + 5 * 1 - 2` is parsed as `((3 * 4) + (5 * 1)) - 2`. You can get `-x` by writing `0-x`.
- Logical operators are right associative, with precedence `~ > /\ > \/ > ->`.
  This means `~ x < 5 \/ x <= 5 /\ ~ y < 2` is parsed as `(~(x < 5)) \/ ((x <= 5) /\ ~(y < 2))`.
- Sequential composition `<prog> ; <prog>` is right associative.
  This means `x := 1 ; x := x+1 ; y := x-1` is parsed as `x := 1` followed by `x := x+1 ; y := x-1`.