Lab 2
Information Flow

Due Tuesday, April 7, 2026
125 points

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

In Part 1 you have to **attack** five server configurations in order to extract secrets. Each server is available as a binary on the `linux.andrew.cmu.edu` machines. You will then submit the secrets as text files to Gradescope to receive credit. Your file names should be `flow_serve1.txt`, `flow_serve2.txt`, etc., where each file contains a single line with the secret you discovered as discussed below.

In Part 2 you have to implement a **defense**: a termination-sensitive information flow type checker for a subset of the C0 programming language. You will submit your checker to the autograder, where we will test it against a collection of programs. The file `lab2.zip` should contain your checker code. As in Lab 1, we will call

```
make
```

after unzipping this file. This should create an executable `c0_serve` satisfying the spec given below. For this lab you are encouraged to work in pairs. Each pair should hand in only one solution after possibly multiple submissions. Identify your partner in Gradescope.

1 C0 Subset for Information Flow

The programs in this lab are a subset of C0 with the following signature:

```
int main(int input, int secret) { ... }
```

The function takes two integer parameters: `input` (a low-security value chosen by the attacker) and `secret` (a high-security value hidden from the attacker). The function returns an `int`, which is observable (a low observation).

1.1 Security Levels and Labels

The language has two security levels, high (H) and low (L), with $\mathsf{L} \sqsubset \mathsf{H}$.

- `input` is labeled L (low, attacker-controlled).
- `secret` is labeled H (high, to be protected).
- The return value of `main` is a **low observation**.

Newly declared variables and arrays default to `L` (low). To declare a high-security variable or array, place a label annotation immediately before the declaration:

```
//@label H;
int h = secret + 1;

//@label H;
int[] A = alloc_array(int, 10);

int x = 0;
```

For arrays, the label applies to the entire array—there is no per-element labeling. Unlabeled declarations are low by default.

1.2 Execution Semantics

Because safety is no longer the focus it was in Lab 1, we conflate unsafe behavior with abort. In other words, any unsafe behavior (division by zero, modulo by zero, out-of-bounds array access) will abort the program. In addition, `assert(e)` aborts if `e` is false, and `error("msg")` always aborts.

Contracts (`//@requires` and `//@loop_invariant`) are ignored during execution and information flow checking, as they are designed for proving programs safe or correct, which is not in scope for this lab.

2 Part 1: The Attack [60 points]

Five server binaries are provided (`c0_serve1` through `c0_serve5`), each implementing a different information flow policy. These are based on taint analysis, termination-insensitive, termination-sensitive, and timing-sensitive noninterference, but may in addition have bugs. We do not reveal which server implements which policy. One of the servers is intended to be secure against 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!

The servers are invoked as follows on the `linux.andrew` machines:

```
% ~mfredrik/bin/c0_serve<n> <userid> <file.c0> <input>
```

where $n \in \{1, 2, 3, 4, 5\}$, *userid* is your Andrew ID (the part before `@andrew.cmu.edu`), *file.c0* is the name of the C0 file containing your attack, and *input* is an integer. The secret will always be an unsigned 62-bit number and is compared with your return value. Correspondingly, the text files you submit to Gradescope should be named `flow_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 pair, each partner has a different secret; the autograder will accept secrets generated from either partner's ID.

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

- `success <value>` (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 <value>` (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.

For some servers, a single submission may be sufficient. For others you may need to write a script that submits many queries and combines the results.

3 Part 2: The Defense [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, `c0_serve`, is invoked with

```
% ./c0_serve <filename>
```

and should parse *filename* and classify the program. Your checker should return one of the following:

- `secure` (with exit code 0). The program is accepted as secure.
- `error` (with exit code 1). The file does not exist, or does not parse correctly.
- `insecure` (with exit code 2). The program is rejected as insecure.

Your checker should not execute the given program. You are not responsible for implementing an interpreter in your server. Executing the program may be helpful for you in debugging, and you may do so if you wish, but the autograder will simply check if your server classifies each test file correctly as secure or insecure.

A parser is provided in the starter code. You should implement the information flow type system from lecture, extended to handle the C0 constructs in this lab. Note that we did not cover information flow typing rules for arrays in lecture. You are responsible for designing suitable rules for array reads and writes.

3.1 Using the Reference Implementation

You can test your Part 2 checker against the reference implementation:

```
% ~mfredrik/bin/c0_serve anyid your_test.c0 0
```

If the output is `insecure` (exit code 2), the reference rejects the program. Any other result means the program passed the IFC check.

4 Language Reference

Below is a specification of the grammar of the C0 subset used in this lab.

```
<program> ::= 'int' 'main' '(' 'int' 'input' ',' 'int' 'secret' ')'
              <contract>* <block>

<contract> ::= '//@requires' <exp> ';'

<label>   ::= '//@label' ('H' | 'L') ';'

<type> ::= 'int' | 'bool' | 'int' '[]'

<exp> ::= <num>
        | <var>
        | 'true' | 'false'
        | <exp> '+' <exp> | <exp> '-' <exp>
        | <exp> '*' <exp> | <exp> '/' <exp> | <exp> '%' <exp>
        | '!' <exp>
        | <exp> '&&' <exp> | <exp> '||' <exp>
        | <exp> '==' <exp> | <exp> '!=' <exp>
        | <exp> '<' <exp>  | <exp> '<=' <exp> ...
        | <exp> '[' <exp> ']'
        | '\length' '(' <exp> ')'
        | '(' <exp> ')'

<stmt> ::= <label>? <type> <var> [ '=' <exp> ] ';'
         | <label>? <type> <var> '=' 'alloc_array' '(' 'int' ',' <exp> ')' ';'
         | <exp> '=' <exp> ';'
         | '{' <stmt>* '}'
         | 'if' '(' <exp> ')' <stmt> [ 'else' <stmt> ]
         | 'while' '(' <exp> ')' <inv>* <stmt>
         | 'return' <exp> ';'
         | 'assert' '(' <exp> ')' ';'
         | 'error' '(' <string> ')' ';'

<inv> ::= '//@loop_invariant' <exp> ';'
```

Operator precedence and associativity follow standard C0 conventions.