CS 261 – Static Analysis
Scribe Notes – 09/13 (Man-Kit Leung)

Administrative: Homework 1 is due tomorrow. Paper summaries were great. It seems like people are doing more than they are supposed to. You can skip the summarization and mention directly the main insights.

We continued our discussion on dynamic tainted analysis from last lecture. Professor gave the following example of tainted policy for preventing SQL injection. The policy matches all the expressions that have meta-characters. Something like this:

$$* ('|;|*|*|...)^t *$$

What wrong with this approach? This is basically blacklisting (e.g. if you forget one then, you are in trouble). They did not check for keywords. What if the user enters the following:

```
Q = "SELECT foo FROM tbl WHERE uid = " + id";
```

Attack can enter "99 OR 1"

Text matching does not work in general. What happen if id actually contains the sequence "or" (e.g. David Wagnor)? The solution is to parse the text into AST parse trees. This prevents over blocking because the text is associated with structure. Another vulnerability is XSS (cross-site scripting). They have the following policy:

$$* (<script)^t *$$

There are many ways to bypass this checking. One way is to insert space between "<" and "script". There are other synonyms to "script" (e.g. "javascript"). There are also meta-character for single character "s", like " ", to alter the text form other than "script". The solution again is to use AST parsing so one can interpret the meaning of the tag based on the node of the AST. Tainted analysis can be performed on the AST tree instead. This is useful on html doc where it allows the user to "fill out a hole". The user can insert a control characters to violate the intended usage.

This is related to hw1 where one can view the entire input as tainted for your filter. Does that mean regex is not useful in specifying security policy? Not exactly, but the lesson here is to use object instead of string for interpretation. You get typing information with objects. If you work with strings, it may work 95% of the time, but the attack can always find ways to by pass the checking. This is because the underlying language is not well-represented with regex (e.g. recursion cannot be expressed).

Today's topic is static analysis. There are basically two tricks in static analysis.

1. Augment the semantics
2. Abstract

To illustrate the first, let's use the tainted tracking programming as an example. The c programming language defines the functional semantics, and the tainted tool gives a different semantics to operate on shadow data structures.

The motivation for abstraction is because there are infinitely many ways to execute the program. I need abstract it by describing it in finite descriptions. There are two places where I would fail. One is omission (e.g. the analysis is not sound). One is over-conservative (e.g. the analysis produces many false positives).

Let `P = {x >= 0, x <= 0}` be a pair of boolean predicates. This is a description of the program state at any point of the execution. The next step is to analyze the program in respects to these predicates. There are two types of analysis: MAY analysis (which predicates **may** be true), and MUST analysis (which predicates **must** be true).

To illustrate as an example, let's do a MAY analysis for the following given code:

```
u = "SELECT foo FROM tbl WHERE user=";
if (moonPhase)
        v = request.getParameter("name");
else
        v = "default";
w = u + v;
db.execute(w);
```

We want to do a static tainted analysis. First, we need to augment the property of the program. The way to do that is to tag variables with a boolean value "tv" (tainted value). There is a boolean variable `t*` associated with each program variable. We want to introduce statements to update the values of these taint variables.

```
u = "SELECT foo FROM tbl WHERE user=";
tu = false;

if (phase)
        v = request.getParameter("name");
        tv = true;
else
        v = "default";
        tv = false;
w = u + v;
tw = tu || tv;
db.execute(w);
assert(!tw);
```

We can easily assign boolean value for tu and tv according the information given from the original program. But notice when updating tw, you may be tempted to say true. But these updates are computed using only localized info (e.g. imagine you cannot see the previous taint

values of other variables). To determine whether there is a security breach, one only needs to look at the bold statements!

Next is to build a control flow graph (CFG). First, we need to insert program points (after each merge and branch point). The CFG represents the possible execution paths of the program.

```
        u = "SELECT foo FROM tbl WHERE user=";
        tu = false;
(1)
        if (phaseMoon)
(2)
                v = request.getParameter("name");
                tv = true;
(3)
        else
(4)
                v = "default";
                tv = false;
(5)
(6)
        w = u + v;
        tw = tu || tv;
(7)
        db.execute(w);
        assert(!tw);
```

We can build a CFG from the code given above, and execute only the taint-value statements to determine all taint values. Notice that none of the original program statements is computed. The benefit of doing static analysis is avoiding the execution of the actual program. There are some considerations on the side. Since this is a MAY analysis, it may have (zero or many) false positives. Although building a CFG is a pretty much solved problem in PL, there is a bunch of PL issues we overlooked here (e.g. pointers, loops, procedure calls, recursions). How does the Java tool deals with procedures? In principle, it uses inlining techniques where a new copy of each function call is made. They weren't too specific about how they deal with recursion.

```
struct reg { char *p; …};

int sys_read(struct reg *r) {
        return *(r→p);
}
```

Imagine you found this code in the operating system. The user code requests service from the kernel through this system call. What are some of the potential vulnerabilities? The user can pass in arbitrary value for r and read into kernel memory (e.g. user can learn passwords that are stored in the kernel address space).

```
struct reg { char *p; …};

int sys_read(struct reg *r) {
```

```
        if (r >= 0xC000 0000 || r→p >= 0xC000 0000)
                return –EPERM;
        log(…);
        return *(r→p);
}
```

Now imagine we added some code to check the content of r. Any potential problems? There is a TOCTTOU problem. What happens if the kernel is preempted out of the CPU before the return statement? User can modify the content of r or r→p, and bypass the checking code. The canonical solution to this is to avoid using the passed-in object twice. So one would copy the value of r into a local variable, and later accesses would use the local variable private to the procedure. This is called copy-in. This is basically to build a programming idiom and build a static analysis to enforce these idioms to ensure safety.

We want to compare between dynamic monitoring (like what is mentioned in the run-time defense papers) and static analysis. When would I be better off doing static analysis? What are the benefits or drawbacks for each of them? Performance is a big plus for static analysis because it doesn't add overhead to the run-time. However, there may be more false positives for doing static analysis. Manual annotations may introduce more places for bugs.