CSE409, Rob Johnson, http://www.cs.stonybrook.edu/~rob/teaching/cse409-fa11/
Alin Tomescu, November 7th, 2011

Type-check-based static analysis

Type-check-based static analysis can be used to detect software vulnerabilities such as the buffer overflow/format string attack in homework #2.

By checking the “source” of the format string parameter given to `printf`, we can tell whether the data could be malicious.

`victim2.c:`

```
#include <stdio.h>

struct processor {
  int (*doit)(char *input);
  char *arg;
};

int default_doit(char *arg)
{
  printf("Received argument: %s\n", arg);
  return 0;
}

int run_processor(struct processor *p)
{
  // We can tell this is susceptible since the fmt string parameter
  // is a variable. Also the source is malicious (ENV var)
  printf(p->arg);
  return p->doit(p->arg);
}

int main(int argc, char **argv)
{
  struct processor p = { &default_doit, argv[1] };
  run_processor(&p);
  return 0;
}
```

We start from the basic idea that programmers always use types. Types are expressive, and they can encourage programmers to consider security.

If we could qualify types with “tainted” and “untainted,” in order to specify whether the data for that type is malicious or not, then we can redefine `printf` like this:

```
int printf(untainted char * format str, ...);
```

We can change the compiler to ensure that `printf` never takes tainted input from functions such as `read_from_user`:

```
tainted char * read_from_user();
```

Policy: Tainted variables cannot be passed as untainted parameters. However, the other way is okay, you can pass untainted variables as tainted parameters.

CSE409, Rob Johnson, http://www.cs.stonybrook.edu/~rob/teaching/cse409-fa11/
Alin Tomescu, November 7th, 2011

Subtyping

Qualifiers introduce subtyping:

$$q \leq q' \Rightarrow q\ \text{int} \leq q'\text{int}$$

Example: $untainted \leq tainted \Rightarrow untainted\ char \leq tainted\ char$

We want type qualifiers which you can customize and then introduce wherever you want in your code. Suppose we have two functions:

$$f\colon t_1' \to t_2'$$

$$g\colon t_1 \to t_2$$

How do you define subtyping for two functions, such that you can check that $f \leq g$:

$$t_1' \leq t_1\ \wedge\ t_2 \leq t_2' \Rightarrow f \leq g$$

Definition: f is a subtype of g, means anywhere you call g, you can also call f

- Anywhere you pass something to g, you can also pass it to f
- Anywhere you return something from g, you can also return it from f

What about pointers?

The right way: $t = t' \Rightarrow ref(t) \leq ref(t')$

The bad way: $t \leq t' \Rightarrow ref(t) \leq ref(t')$.

This is false, consider the following counter-example:

Let $p = untainted\ char, q = tainted\ char$ such that $p \leq q$

Since $p \leq q \Rightarrow ref(p) \leq ref(q)$, you can assign $ref(p)$ to $ref(q)$.

That is, you could assign an `untainted char *p_ref` to a `tainted char * q_ref`. As a result, you can now modify

```
untainted char * p_ref;
tainted char *q_ref;

// The bad rule lets you assign p_ref to q_ref
q_ref = p_ref

// Now you can write q_ref = evil data into the untainted p data
*q_ref = "evil input";

//    As a result, p_ref will point to "evil input"
```

Type qualifier inference

```
tainted char * getinput_from_user(char * option);
int printf(untainted const char * fmt, ...);

char * s, *t;
s = getinput_from_user("stdin");
t = s;
printf(t);
```

CSE409, Rob Johnson, http://www.cs.stonybrook.edu/~rob/teaching/cse409-fa11/
Alin Tomescu, November 7th, 2011
We have to check if there is a path from the tainted input to the untainted input:

- `getinput_from_user` returns tainted
- `printf`'s first argument `fmt` should be untainted (printf_arg0_p)
- `getinput_from_user`'s return value has to be a subtype of `s`
- `getinput_from_user`'s return pointee type has to be the same as `s`'s pointee's type
- `s <= t`, `s` has to be a subtype of `t` (since we're assigning `s` to `t`)
- `s`'s pointee = `t`'s pointee type (what `s` points to has to have the same type as what `t` points to)
- `t` <= `printf` 's first argument `fmt`
- `t`'s pointee type = `printf` 's first argument `fmt`'s pointee type

Runtime Integer Overflow Checker (RICH)

Subtyping check (check that you always put int8 into int16 or similar).

CCured: safe-typed pointers

CCured uses static type inference along with runtime checks on **fat-pointers.** Remember that in C you can declare arrays of fixed sizes.

```
int a[3];
int * p
p = a + 4;
```

We can **statically check safe-typed pointers** and only insert runtime checks for unsafe-typed pointers.

When `a` is declared we should give it bounds, such that when `p` is assigned `a` + 4, we can check that `a`'s bounds are respected in the arithmetic.

What if you have something like?

```
p = a + 4;
p--;
p--;
```

We can use an **out of bounds structure,** which keeps track of pointers that go out of bounds. Anytime you do an arithmetic operation on `p`, you can check if `p` is in bounds and then add it or remove it from the OOB structure.

Fat pointers: Put the bound information close to where the pointer is stored in memory (better locality).

```
int p[10];
int q[5];

//    p_ref's bounds are [p, p+10)
int * p_ref = p;
//    q_ref's bounds are [q, q+5)
int * q_ref = q;

//    Now p_ref's bounds are the same as q_ref's: [q, q+5)
p = q;
```

When you assign `q` to `p`, you also assign `q`'s bounds to `p`'s bounds.

CSE409, Rob Johnson, http://www.cs.stonybrook.edu/~rob/teaching/cse409-fa11/
Alin Tomescu, November 7th, 2011
CCured tries to not check pointers which do not need to be checked at runtime.