Program Verification & Other Types of Vulnerabilities

Dawn Song

dawnsong@cs.berkeley.edu

Review

- Memory-safety vulnerabilities
- Runtime detection
- Fuzzing for bug finding
  - Blackbox fuzzing
  - Whitebox fuzzing

This Class

- Program verification
- Other types of vulnerabilities

Static Analysis

- Instead of running the code to detect attacks or find bugs, we statically analyze code
- Simple pattern match:
  - Whether program uses unsafe APIs: gets, sprintf, etc.
- Simple checks:
  - E.g., variable use before def or initialization
- More sophisticated analysis
  - E.g., potential array-out-of-bounds check
- Many tools available
  - Open source: http://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis
  - Commercial tools: Coverity, Fortify, etc.

Program Verification

- Can we prove a program free of buffer overflows?

- How to prove a program free of buffer overflows?
  - Precondition
  - Postcondition
  - Loop invariants

Precondition

- Functions make certain assumptions about their arguments
  - Caller must make sure assumptions are valid
  - These are often called *preconditions*
- Precondition for `f()` is an assertion (a logical proposition) that must hold at input to `f()`
  - Function `f()` must behave correctly if its preconditions are met
  - If any precondition is not met, all bets are off
- Caller must call `f()` such that preconditions true – an obligation on the caller, and callee may freely assume obligation has been met
- The concept similarly holds for any statement or block of statements

Simple Precondition Example

-
  ```
  int deref(int *p) {
      return *p;
  }
  ```
- **Unsafe to dereference a null pointer**
  - **Impose precondition that caller of `deref()` must meet: *$p \neq NULL$* holds at entrance to `deref()`**
- **If all callers ensure this precondition, it will be safe to call `deref()`**
- **Can combine assertions using logical connectives (and, or, implication)**
  - **Also existentially and universally quantified logical formulas**

Another Example

-
  ```
  int sum(int *a[], size_t n) {
      int total = 0, i;
      for (i=0; i<n; i++)
          total += *(a[i]);
      return total;
  }
  ```

- **Precondition:**
  - **a[] holds at least n elements**
  - **Forall j.$(0 \leq j < n) \rightarrow a[j] \neq NULL$**

Postcondition

- ***Postcondition* for `f()` is an assertion that holds when `f()` returns**
  - **`f()` has obligation of ensuring condition is true when it returns**
  - **Caller may assume postcondition has been established by `f()`**
- **Example:**

```
void *mymalloc(size_t n) {
    void *p = malloc(n);
    if (!p) {
        perror("Out of memory");
        exit(1);
    }
    return p;
}
```

- `Post condition: retval != NULL`

Proving Precondition→Postcondition

- Given preconditions and postconditions
  - Which specifies what obligations caller has and what caller is entitled to rely upon
- Verify that, no matter how function is called, if precondition is met at function's entrance, then postcondition is guaranteed to hold upon function's return
  - Must prove that this is true for all inputs
  - Otherwise, you've found a bug in either specification (preconditions/postconditions) or implementation

Proving Precondition→Postcondition

- Basic idea:
  - Write down a precondition and postcondition for every line of code
  - Use logical reasoning
- Requirement:
  - Each statement's postcondition must match (imply) precondition of any following statement
  - At every point between two statements, write down *invariant* that must be true at that point
    - Invariant is postcondition for preceding statement, and precondition for next one

Example

- Easy to tell if an isolated statement fits its pre- and post-conditions
- postcondition for "`v=0;`" is
  - `v=0` (no matter what the precondition is)
  - Or, if precondition for "`v=v+1;`" is `v≥5`, then a valid postcondition is
    - `v≥6`
- If precondition for "`v=v+1;`" is `w≤100`, then a valid postcondition is
  - `w≤100`
  - Assuming `v` and `w` do not alias

Loop Invariant

- An assertion that is true at entrance to the loop, on any path through the code
  - Must be true before every loop iteration
    - » Both a pre- and post-condition for the loop body
- Example: Factorial function code

```
/* Requires: n >= 1 */
int fact(int n) {
    int i, t;
    i = 1;
    t = 1;
    while (i <= n) {
        t *= i;
        i++;
    }
    return t;
}
```

  - Prerequisite: input must be at least 1 for correctness
  - Prove: value of `fact()` is always positive

Verifying Invariant Correctness

```
/* Requires: n >= 1
   Ensures: retval >= 0 */
int fact(int n) {
    int i, t;          /* n>=1 */
    i = 1;             /* n>=1 && i==1 */
    t = 1;             /* n>=1 && i==1 && t==1 */
    while (i <= n) {
       /* 1<=i && i<=n && t>=1   <-- loop invariant */
       t *= i;          /* 1<=i && i<=n && t>=1 */
       i++;             /* 2<=i && i<=n+1 && t>=1 */
    }                   /* i>n && t>=1 */
    return t;
}
```

- Easy if we examine each step:
  - Function's precondition implies invariant at function body start
  - Invariant at end of function body implies function's postcondition
  - If each statement matches invariant immediately before and after it, everything's OK
- That leaves the loop invariant…

Verifying the Loop Invariant

- Loop invariant: `1<=i && i<=n && t>=1`
- Prove it is true at start of first loop iteration
  - Follows from:
    - » `n≥1 ∧ i=1 ∧ t=1 → 1≤i≤n ∧ t≥1`
    - » if `i=1`, then certainly `i≥1`
- Prove that if it holds at start of any loop iteration, then it holds at start of next iteration (if there's one)
  - True, since invariant at end of loop body `2≤i≤n+1 ∧ t≥1` and loop termination condition `i≤n` implies invariant at start of loop body `1≤i≤n ∧ t≥1`
- Follows by induction on number of iterations that loop invariant is always true on entrance to loop body
  - Thus, `fact()` will always make postcondition true, as precondition is established by its caller

Function Post-/Pre-Conditions

- **Any time we see a function call, we have to verify that its precondition will be met**
  - **Then we can conclude its postcondition holds and use this fact in our reasoning**
- **Annotating every function with pre- and post-conditions enables *modular reasoning***
  - **Can verify function `f()` by looking only its code and the annotations on every function `f()` calls**
    - **Can ignore code of all other functions and functions called transitively**
  - **Makes reasoning about `f()` an almost purely local activity**

Documentation

- **Pre-/post-conditions serve as useful documentation**
  - **To invoke Bob's code, Alice only has to look at pre- and post-conditions – she doesn't need to look at or understand his code**
- **Useful way to coordinate activity between multiple programmers:**
  - **Each module assigned to one programmer, and pre-/post-conditions are a contract between caller and callee**
  - **Alice and Bob can negotiate the interface (and responsibilities) between their code at design time**

Avoiding Security Holes

- **To avoid security holes (or program crashes)**
  - **Some implicit requirements code must meet**
    - **Must not divide by zero, make out-of-bounds memory accesses, or deference null ptrs, …**
- **We can try to prove that code meets these requirements using same style of reasoning**
  - **Ex: when a pointer is dereferenced, there is an implicit precondition that pointer is non-null and in-bounds**

Proving Array Accesses are in-bounds

```
/* Requires: a != NULL and a[] holds n elements */
int sum(int a[], size_t n) {
    int total = 0, i;
    for (i=0; i<n; i++)
            /* Loop invariant: 0 <= i < n */
        total += a[i];
    return total;
}
```

- **Loop invariant true at entrance to first iteration**
  - **First iteration ensures** `i=0`
- **It is true at entrance to subsequent iterations**
  - **Loop termination condition ensures** `i<n`**, and** `i` **only increases**
- **So array access** `a[i]` **is within bounds**

Buffer Overruns

- **Proving absence of buffer overruns might be much more difficult**
  - **Depends on how code is structured**
- **Instead of structuring your code so that it is hard to provide a proof of no buffer overruns, restructure it to make absence of buffer overruns more evident**
- **Lots of research into automated theorem provers to try to mathematically prove validity of alleged pre-/post-conditions**
  - **Or to help infer such invariants**

Administravia

- **Hw3 out**
- **Project partner**

User/Kernel Pointer Bugs

- An important class of bugs
-
  ```
  int x;
  void sys_setint (int *p)
  { memcpy(&x, p, sizeof(x));
  }
  void sys-getint (int *p)
  { memcpy(p, &x, sizeof(x));
  }
  ```
- Can cause system hang, crash kernel, gain root privileges, read secret data from kernel buffers

Non-Language-Specific Vulnerabilities

-
  ```
  int openfile(char *path) {
      struct stat s;
      if (stat(path, &s) < 0)
          return -1;
      if (!S_ISRREG(s.st_mode)) {
          error("only regular files allowed!");
          return -1;
      }
      return open(path, O_RDONLY);
  }
  ```
- Code to open only regular files
  - Not symlink, directory, nor special device
- On Unix, uses `stat()` call to extract file's meta-data
- Then, uses `open()` call to open the file

The Flaw?

- Code assumes FS is unchanged between `stat()` and `open()` calls – Never assume anything…
- An attacker could change file referred to by `path` in between `stat()` and `open()`
  - From regular file to another kind
  - Bypasses the check in the code!
  - If check was a security check, attacker can subvert system security
- Time-Of-Check To Time-Of-Use (TOCTTOU) vulnerability
  - Meaning of `path` changed from time it is checked (`stat()`) and time it is used (`open()`)

TOCTTOU Vulnerability

- **In Unix, often occurs with filesystem calls because system calls are not atomic**
- **But, TOCTTOU vulnerabilities can arise anywhere there is mutable state shared between two or more entities**
  - **Example: multi-threaded Java servlets and applications are at risk for TOCTTOU**