Detecting Format String Vulnerabilities with Type Qualifiers
Shankar, Talwar, Foster, & Wagner (May 2001)

Contributions
- Type System for detecting "format string vulnerabilities" in C
- Technique for presenting the results of the analysis to a user
- Empirical results demonstrating effectiveness at finding previously unknown bugs with a low rate of false positives

Format String Vulnerabilities
- Arise from "design misfeatures" in the C Standard Library + "problematic implementation" of var-arg functions

```c
printf("%s", buf);  (correct)
printf(buf);  (may be incorrect)
```

No checking is done, either at run-time or compile-time, to verify that `printf()` is called with the correct number and types of args.

Format String Vulnerabilities
- Other Examples Include:
  - `syslog()` : message logging function
  - `setproctitle()` : set X-window name
- When combined with other tricks this bug can be used to write to arbitrary memory locations (see: "Format String Attacks", Tim Newsham, 2000)

Approach
- A Type System!
  - Static, Type-theoretic Analysis
  - Combine user-supplied type quantifiers (annotations) with a constraint-based inference engine
- Claim: This is superior to testing and manual code inspection
  - All paths are created equal
  - Bugs manifest from remote code
Type System

- Introduce two C type quantifiers (tainted, untainted)
- Syntax rules mirror `const`
- Induce a subtyping relationship:
  - `untainted P < tainted P`
- Tainted = "may be tainted"

Examples:
- `tainted int foo();` return value should be considered tainted
- `int bar(untainted int x);` Argument must not be tainted

Static Analysis

- Input
  - A few user-provided taint-qualifiers
  - Type constraints inferred from syntax
- Algorithm
  - Constraint solver to assign taint-qualifiers to every variable (+ implicit pointer targets)
- Output
  - Report if a solution to constraint system exists
  - Report any instance where a format string command has a tainted argument

Example Constraint System

- By transitivity:
  - `tainted = genenv_ret_p = s_p = t_p ≤ printf_arg0_p = untainted`
  - Incorrect, since `tainted ≤ untainted` does not hold

Example Generation

- Identifiers are colored by inferred qualifiers (tainted, untainted, either)
- Constraint Dependence Graph
- Paths in dependence graph from tainted to untainted indicates a type error
- Display shortest paths via BFS, list "hotspot” qualifiers

Polymorphism

- As presented, algorithm is both context- and flow-insensitive
- `x` is tainted by actual parameter `t`, therefore `b` is also tainted since `b = ret_id = x;`
- This problem is trivially solved by introducing polymorphism on the function's qualified type

Example:
- `char id(char x) { return x; }

Explicit Type Casts

- Taint-qualifier is preserved through ordinary type-casts
- Casts to `(void *)` are matched as deeply as possible, then all remaining qualifiers are "collapsed” and equated
- Programmer can “cast-away” taint: `char *x = (untainted char *) y;` `x` in now untainted regardless of `y`
Unsoundness of Casting

- Collapsing qualifiers on structure fields generated false-positives
- Qualifier-collapsing does not fully model casts from pointers to ints

variable *x, *y;
and x, y;

x = (int) y;
f = x;
y = (int) f;

For line (1), we guarantee the constraint s_p = s
For line (2), we ensure the constraint s < 8
But for line (3), we ensure the constraint s_p < 8
Notice that we lose a constraint s < 8

Variable Argument Functions

- Cannot deal individually with variable arguments
- Grammar extended to qualify "..."

\[ \text{printf}(s, "\%s", t) \]
Would like to infer s is tainted if t is
Add a constraint!

const Allows Deep Subtyping

- Take advantage of "const" to relax constraints

Example:
const char *s;
char *t;
...
s = t;
Replace "s_p = t_p" constraint with "t \leq s and t_p \leq s_p"

Empirical Results

- Preparation took 30-60 minutes each
- System reliably found "all known bugs"
- "Hotspots pinpointed the actual bug in most cases" (2 out of 3?)

Other Techniques

- Lexical Techniques
- Perl's taint mode
- Static Bug Detection
  - LCInt
  - Meta-level compilation
- Run-time techniques

Discussion

- How much time is wasted dealing with untainted data?
- Analysis suffers from flow-insensitivity
- Why not just use data-flow analysis augmented with an OTS pointer-analysis?
- Values: sets of tainted variables
- Could use standard techniques to get context-sensitivity, flow-sensitivity