

Detecting Format String Vulnerabilities with Type Qualifiers

Shankar, Talwar, Foster, & Wagner (May 2001)

James Ezick
CS 711: Advanced PL Seminar on
Language Based Security and Information Flow
1 October 2003

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

`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

`printf(buf);`

If `buf` contains a format specifier (e.g., “%s”), `printf()` will naively **attempt to read non-existent arguments off the stack**, most likely causing the program to crash!

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:
 $\text{untainted } P < \text{tainted } P$
 - Tainted \approx "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

```
tainted char *getenv(const char *name);  
int printf(untainted const char *fmt, ...);  
  
char *a, *t;  
a = getenv("LD_LIBRARY_PATH");  
t = #;  
printf(t);
```

```
genenv_ret_p = tainted  
printf_arg0_p = untainted  
  
genenv_ret_p < t  
genenv_ret_p == t_p  
t < #  
t_p == #  
t < printf_arg0_p  
t_p < printf_arg0_p
```

Figure 3: An example of constraint generation. The left column is a code fragment; the right column gives the inferred constraints on the qualifier variables.

- By transitivity:
 $\text{tainted} = \text{genenv_ret_p} = \text{s_p} = \text{t_p} \leq \text{printf_arg0_p} = \text{untainted}$
- Incorrect, since $\text{tainted} \leq \text{untainted}$ 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 $b = \text{ret_id} = x$;
 - This problem is trivially solved by introducing polymorphism on the function's qualified type
- Example:
- ```
char id(char x) {
 return x;
}
...
tainted char t;
untainted char u;
char a, b;
a = id(t);
b = id(u);
```

## Explicit Type Casts

- Taint-qualifier is preserved through ordinary type-casts
- Casts to  $(\text{void } *)$  are matched as deeply as possible, then all remaining qualifiers are "collapsed" and equated
- Programmer can "cast-away" taint:  
 $\text{char } *x = (\text{untainted char } *) y$ ;  
 $x$  is 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

```
char *x, *y;
int a, b;

a = (int) x; (1)
b = a; (2)
y = (char *) b; (3)
```

For line (1), we generate the constraint  $x_p = a$ . For line (2), we generate the constraint  $a \leq b$ . And for line (3), we generate the constraint  $b = y_p = y$ . Notice that we have  $x_p \leq y_p$  but we do not have  $y_p \leq x_p$ , so our deductions are unsound.

## Variable Argument Functions

- ❑ Cannot deal individually with variable arguments
- ❑ Grammar extended to qualify "..."
- ❑ `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 ≤ s` and `t_p ≤ s_p`"

## Empirical Results

| Name     | Version | Description                     | Lines | Preproc. | Time | Warnings | Bugs |
|----------|---------|---------------------------------|-------|----------|------|----------|------|
| cfengine | 1.5.4   | System administration tool      | 24k   | 126k     | 28s  | 5        | 1    |
| mutl     | 2.05d   | IRC proxy                       | 3k    | 103k     | 5s   | 12       | 1    |
| ftpd     | 1.0.11  | FTP server                      | 2k    | 34k      | 2s   | 2        | 1    |
| msr_svr  | 0.99    | Novell Network emulator         | 21k   | 73k      | 21s  | 0        | 0    |
| mingetty | 0.9.4   | Remote terminal control utility | 0.2k  | 2k       | 1s   | 0        | 0    |
| apache   | 1.3.12  | HTTP server                     | 33k   | 136k     | 43s  | 0        | 0    |
| sshd     | 2.3.0p1 | OpenSSH ssh daemon              | 26k   | 221k     | 115s | 0        | 0    |
| imapd    | 4.7c    | Univ. of Wash. IMAP4 server     | 43k   | 82k      | 268s | 0        | 0    |
| ipopd    | 4.7c    | Univ. of Wash. POP3 server      | 40k   | 78k      | 373s | 0        | 0    |
| identd   | 1.0.0   | Network identification service  | 0.2k  | 1.2k     | 3s   | 0        | 0    |

- ❑ 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
  - LCLint
  - 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