

Feb 23, 2009
CSE, 409/509

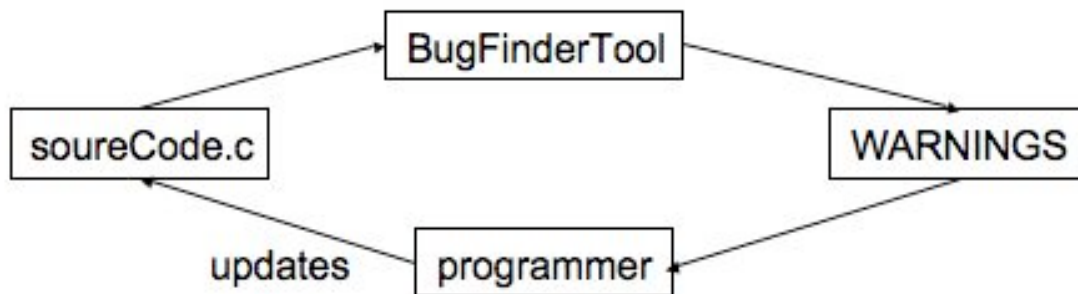
Mitigation of Bugs, Life of an exploit

- 1) Bug inserted into code
 - 2) Bug passes testing
 - 3) Attacker triggers bug
 - 4) The Attacker gains control of the program
 - 5) Attacker causes damage
- Different to gaining control. For example, the attacker might quit after gaining control.

Each of the above can be divided into different stages.

Static Analysis for Security

This area of research comes from the programming languages and compiles community



Depending on how the bug finder tool (BFT) was implemented, one might be able to prove that there is actually no more bugs after BFT says "no more bugs"

If BFT outputs a lot of warnings → lots of potential bugs → lot of human work.
We want to minimize warnings, and we would like to get as many warnings as there are bugs

Minimize number Warnings

- ideal number of warnings = number of bugs
- **Complete**: if BFT outputs zero bugs, this implies zero warnings

Minimize number of missed bugs

ideal zero missed bugs

Sound: Zero warnings implies zero bugs

Determining if a turing machine halts or not is undecidable
sound and complete bug finding is not possible

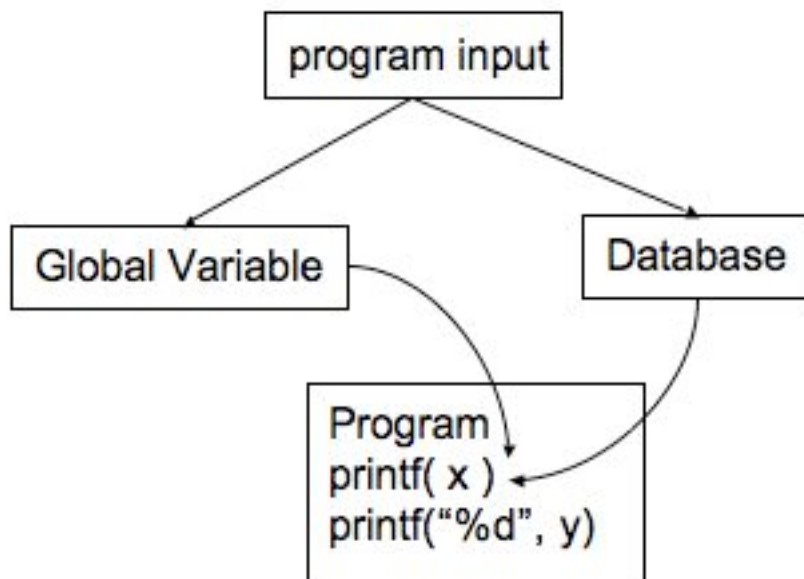
We can choose two out of three, but not all three aspects at once: terminate, sound, or complete

Cqual and type qualifier inference

(if used correctly and carefully, it can be sound)

Format string bug

Data Flow Problem



For example, we would like to stop the flow of data from possible tainted inputs to unsafe operations like printf(x). printf(\"%d\" , y) is a safer operation.

Analyse Data Flow

Can any inputs flow to the format argument of a printf function call?
(If an input flows to other arguments, it is ok)

Equal Labels Type qualifiers

Append two new qualifiers to basic types:
\$untainted and \$tainted

```
$tainted char  
$untainted struct foo;  
$tainted char untainted *;
```

```
int printf( $untainted char *, ... );  
void main( int argc, char **argv )  
{  
    printf( argv[ 1 ] );  
}
```

Assume the attacker has control of argv.
then the main function should really look like

```
void main( $untainted int argc, $tainted char **argv )  
{  
    printf( argv[ 1 ] );  
}
```

The above code will not compile because we are passing a tainted argument to printf, which only accepts untainted arguments.

fix:

```
safe_printf( $tainted char *, ... );  
...  
...  
$untainted char *buf;  
safe_print( buf );  
$tainted char *buf2;  
safe_print( buf );
```

safe_print should be able to accept both tainted and untainted variables.

Note: Class hierarchy and subclasses. For example, in java we can pass a Vector Object to a function that accepts inputs of the type Object

Should we be able to pass an untainted pointer to safe_print ? for now, assume yes

```
$tainted > $untainted t1 = t2  
$tainted ref t1 > $untainted ref t2
```

can we relax the above rule to get a weaker rule ?

```
$tainted > $untainted tau1 > tau2  
$tainted ref t1 > $untainted ref t2
```

if we allow the weaker rule, then the following is possible

```
$tainted char * $untainted t;  
$untainted char * $untainted u;  
t = 0; // ok, types check  
*t = <some tainted data>; // type checks  
printf( u ); // type checks
```

printf accepts untainted data only, but we just passed it tainted data through u.

If given the following struct

```
struct foo{  
    $tainted x  
}
```

then foo a.x is tainted, and foo b.x is also tainted. It depends on how you implement the struct. Example:

```
struct foo{  
    char buf [64]  
}
```

```
struct foo A, B;  
1: read( fromNetwork, A.buf, 64 )  
2: memcpy( B.buf, "hello", 6 )  
3: printf( B.buf )
```

line 1 is receiving input from the network, possibly tainted, but lines 2 and 3 accept untainted data only.
possible solution, create two different types of struct foo.

How to figure out what annotation should be given to each variable:

Tainted Qualifier Inference

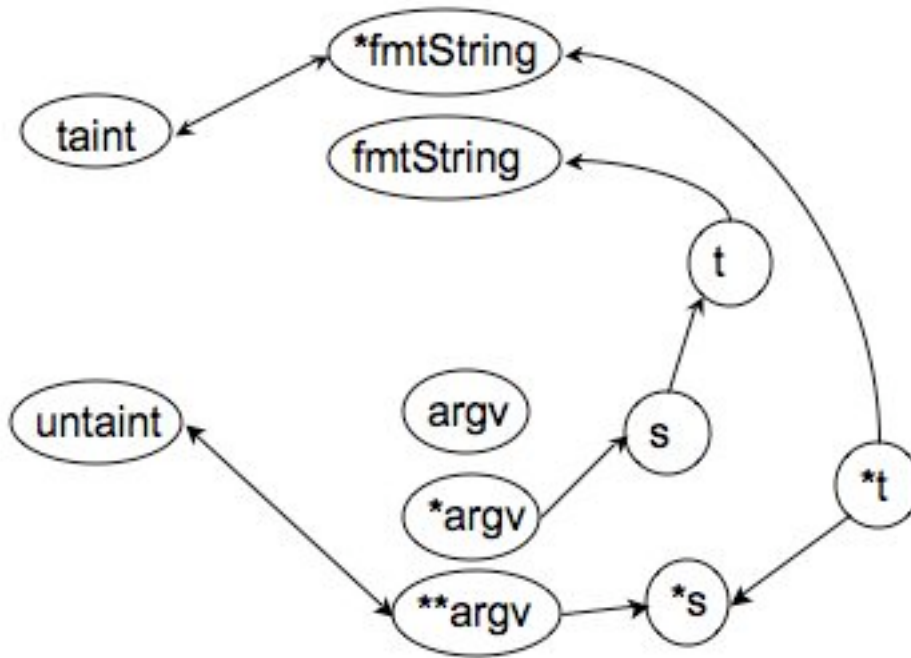
Given the following program, what variable should be tainted, which should be untainted?

```

int main( int argc, char **argv )
{
    char *s *t
    s = argv[1]
    t = s
    printf( t )
}

```

To add annotations, we start with a few, and the algorithm will find the rest for us by building a flow graph. We start with:
int printf(\$untainted const char *format, ...);
int main(int argc, \$tainted char ***argv);
Bug Finder reads printf and main declarations, and finds the data flow



If there exist a path from tainted to untainted, then this implies there is a bug in the program. Notice that argv is neither tainted or untainted.

cqual made all annotation to all C libraries
library functions = 200 annotations (approx).
analyzed 7 programs found 3 bugs (approx).
false positive: 12 to 15 (no real bugs)
this is a high false positive rate