Towards automated detection of buffer overrun vulnerabilities: a first step

David Wagner   Jeffrey S. Foster
Eric A. Brewer   Alexander Aiken

NDSS 2000   Feb 3, 2000

Introduction

- The state of computer security today is depressing

  . . . and most holes arise from simple programming errors in legacy C code

- 'Buffer overruns' are one of the worst offenders
  - A common coding error with uncommonly-devastating effects

Goal: eliminate buffer overruns from security-critical source code.

A puzzle: spot the bug

Here's sendmail-8.9.3 source; can you spot the coding error?

Organization

- Introduction
- Background and motivation
- Techniques for automated detection of buffer overruns
- Evaluation of our prototype
- Summing up

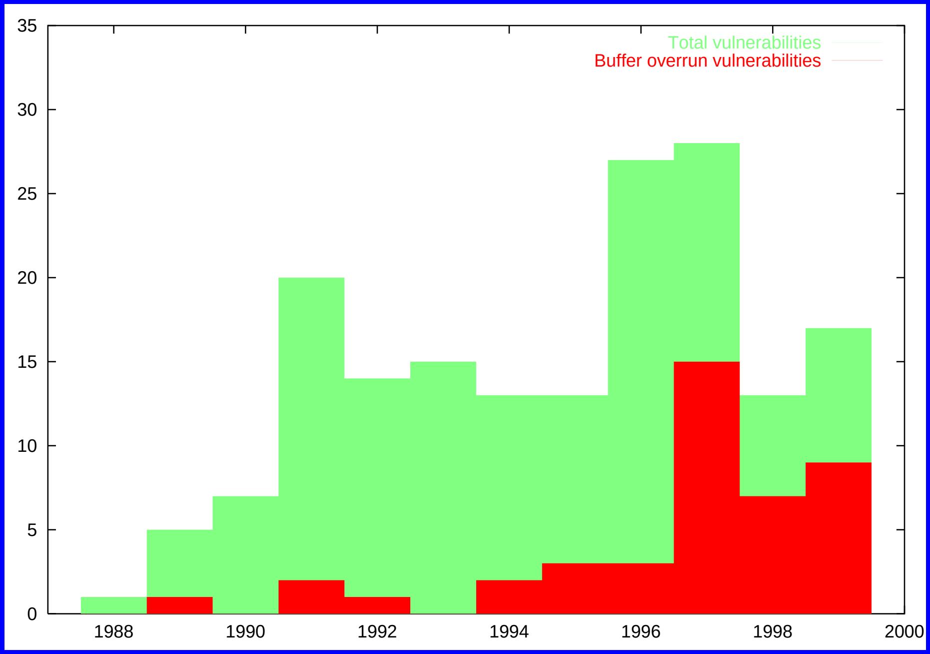
Review

- An example code fragment vulnerable to buffer overruns:

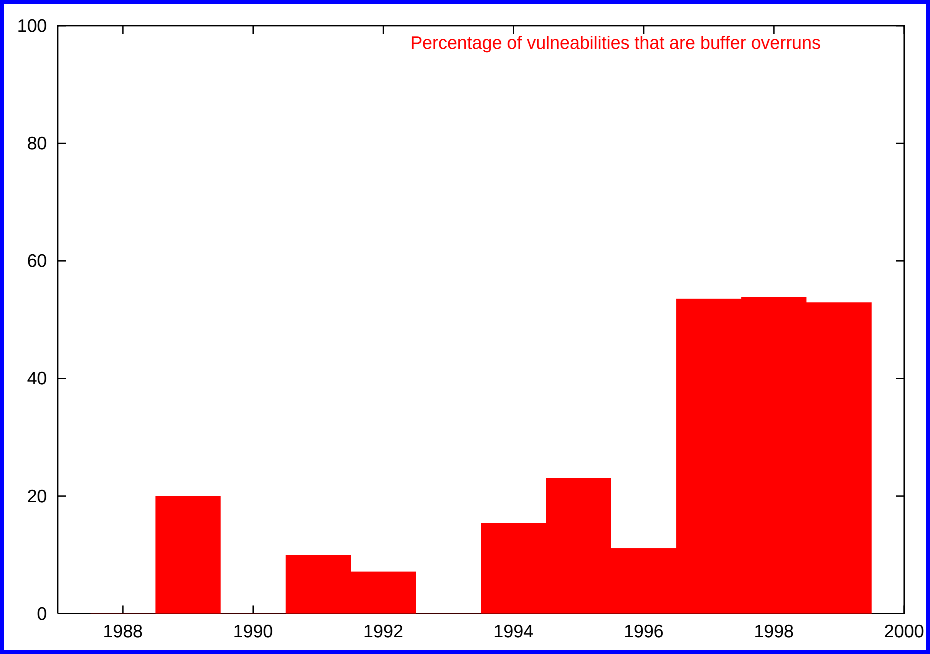
```
void foo(void) {
    char buf[80];
    strcpy(buf, gethostbyaddr(...)->hp_hname);
}
```

- Exploits are possible by writing past the end of `buf`.
  - Typically allows attacker to execute arbitrary code
  - Hacker tools are very good; even an off-by-one error can be exploited

Why are buffer overruns important?



Absolute number of vulnerabilities reported



Relative frequency of buffer overruns

Overruns account for 40%–50% of recent holes!

- Compare: this is 2× what can be blamed on poor crypto
- Upwards trend due to development of hacker tools

Organization

- Introduction
- Background and motivation
- Techniques for automated detection of buffer overruns
- Evaluation of our prototype
- Summing up

Overview

Our approach:

- A lint-like tool for analyzing C source code
  - Finds potential buffer overruns
  - But might issue false alarms, and might miss some bugs—no guarantees!
- Key technique: whole-program static analysis
  - Borrow ideas from program analysis and theory literature
    (Avoid unnecessary innovation.)

Why static analysis?

How do you look for potential vulnerabilities?

- Runtime testing? (i.e., dynamic checking)
  - \+ Some tools already exist [fuzz,Purify, . . .]
  - – But hard to generate test cases, and hard to know when you're done
- Compile time warnings? (i.e., static checking)
  - \+ Opportunity to find and eliminate holes *proactively*
  - – But implementation is a challenge

$\Rightarrow$ Static analysis is potentially very attractive, but how to do it?

Our tool

Approach:

- Simplify!
  - e.g.: flow-insensitive analysis

$\Rightarrow$ Trade off precision for *ease of prototyping* and *scalability*.

Architecture:

- *Constraint-based* analysis
  - Two phases: constraint generation, constraint solving

Notation

Each dynamic quantity of interest gets a set-variable.

If $\mathtt{s}$ is a string variable, let len($\mathtt{s}$) (resp., alloc($\mathtt{s}$)) denote the set of possible lengths (resp., number of bytes allocated) for $\mathtt{s}$ during a run of the program.

We find a *conservative approximation* for len($\mathtt{s}$) and alloc($\mathtt{s}$).

- Then, checking the safety condition len($\mathtt{s}$) $\leq$ alloc($\mathtt{s}$) is easy.

Constraints

Let $[m, n]$ denote the range $\{m, m + 1, \ldots, n\}$.

Constraints take the form, e.g., $X \subseteq Y$, where X, Y are range-variables.

For example,

`strcpy(dst,src);` $\Rightarrow$ len(`src`) $\subseteq$ len(`dst`)

Constraint generation

- Constraint generation is best described by example
  - So here is a code snippet to illustrate the analysis:

```
char buf[128];
while (fgets(buf, 128, stdin)) {
    if (!strchr(buf, '\n')) {
        char error[128];
        sprintf(error, "Line too long: %s\n", buf);
        die(error);
    }
    ...
}
```

The example, with annotations

Original source code	The constraints we generate
`char buf[128];`	$[128, 128] \subseteq \text{alloc}(\texttt{buf})$
`while (fgets(buf, 128, stdin)) {`	$[1, 128] \subseteq \text{len}(\texttt{buf})$
`    if (!strchr(buf, '\n')) {`	
`        char error[128];`	$[128, 128] \subseteq \text{alloc}(\texttt{error})$
`        sprintf(error, "Line too long: %s\n", buf);`	$\text{len}(\texttt{buf}) + 16 \subseteq \text{len}(\texttt{error})$
`        die(error);`	
`    }`	
`    ...`	
`}`	

Notice how we focus on primitive string operations?

- We largely ignore pointer ops; we treat strings as abstract datatypes (We don't always catch missing `'\0'` terminators or unsafe pointer dereferences, but in principle we could, with more effort)

The constraint solver

- Uses graph-based algorithms
- Fast, precise, and scalable

$\Rightarrow$ Runs in linear time in practice

And that's all I'll say. See the paper for more.

Organization

- Introduction
- Background and motivation
- Techniques for automated detection of buffer overruns
- Evaluation of our prototype
- Summing up

Results

- We implemented the analysis
- We used the tool to find *new* vulnerabilities in *real* programs
  - Linux nettools: 7k lines, previously hand-audited
    Found several new holes, *exploitable from remote hosts*
  - Latest sendmail: 32k lines, previously hand-audited
    Found several new buffer overruns, most likely not exploitable
  - Re-discovered old serious holes in e.g. sendmail-8.7.5, popd, . . .
    (Could have prevented some widespread attacks, if tool had been available)
- Just a prototype, many rough edges, but it's already useful

Limitations

Lots of false alarms:

- Example: 44 warnings for sendmail, only 4 real coding errors
  - Mostly because we traded precision for simplicity; see next slide.
- But this still compares quite favorably to the alternatives
  - Comparison: `grep` shows ~ 700 calls to unsafe string ops,
    so we reduce the manual auditing effort by $15\times$ over `grep`

A few false negatives:

- But false negatives appear to be relatively rare.
  - Of the ($\geq$ 10) bugs in sendmail 8.7.5 that have been fixed,
    the tool missed only one

Possibilities for future improvements

Classifying the cause of false alarms in sendmail:

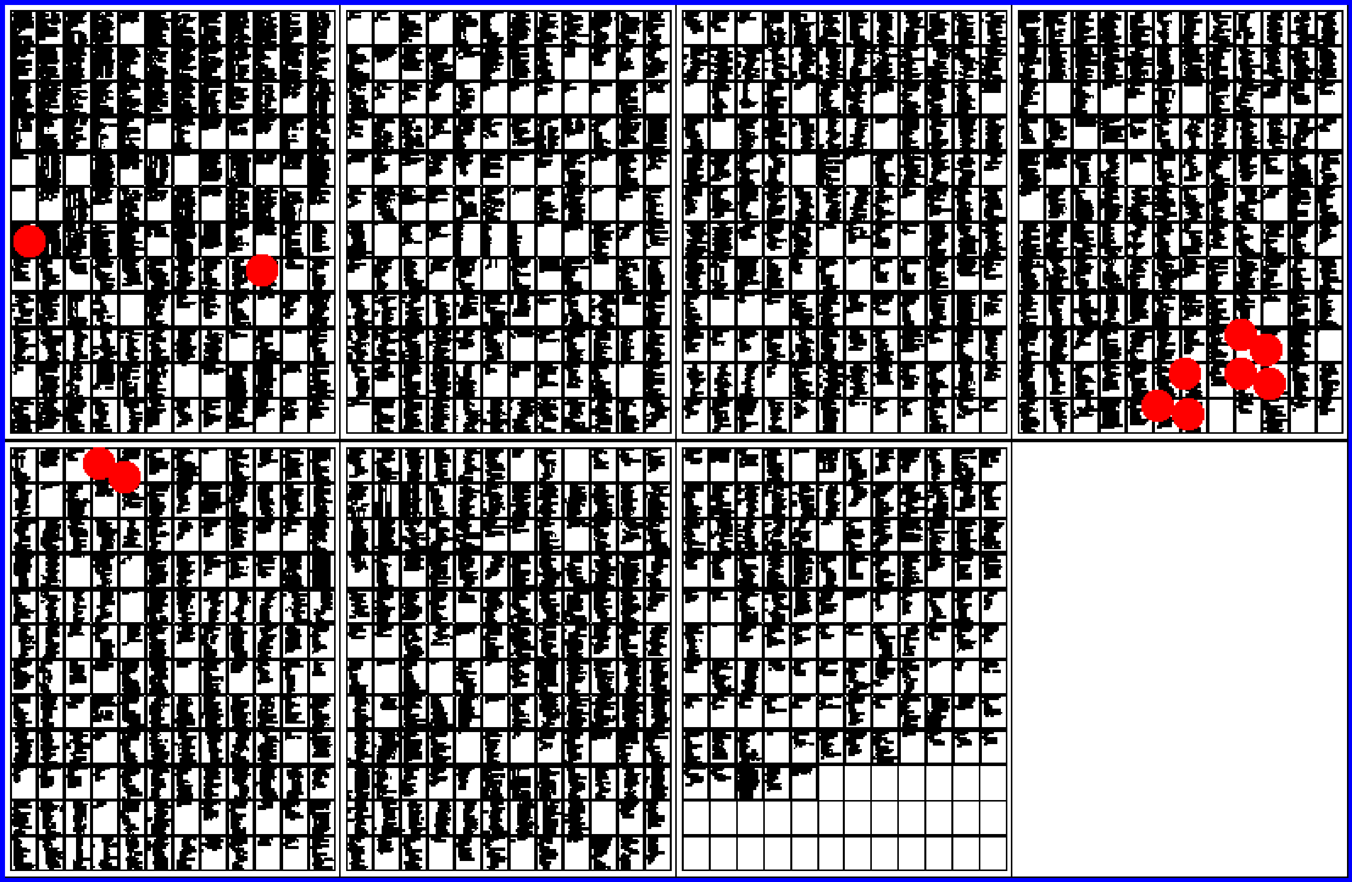
Improved analysis	False alarms eliminated
flow-sensitive	47.5%
flow- and context-sensitive, with pointer analysis and inter-variable invariant inference	95%

(flow-sens. = models control flow;
context-sens. = doesn't merge function call sites)

- Might do $20\times$ better, using only known techniques?

$\Rightarrow$ Know how to build a much better second system.

Solution to the puzzle



Shows an overrun. Red spots = lines of code you must understand to find it.

Bug has been there for $>$ 3 years, and has survived several hand audits.

Summary

- A successful research prototype
  - Already finding new vulnerabilities in real programs
  - But lots of room for improvement
- A promising new methodology: static analysis for code auditing
  - Key advantages: *proactive security* for *legacy code*; possibility of *compensating for language deficiencies*