Static Analysis Techniques to detect Buffer Overrun Vulnerabilities.

Vinod Ganapathy
University of Wisconsin
Overview

• Buffer Overrun Vulnerability

  - String length more than space allocated for it

  - char *a;
    a = (char *)malloc(5);
    gets(a);

  - Variable a has 5 bytes allocated, occupies ??
Overview

• Significance?
  - 10 out of 37 of CERT advisories in 2001
  - > 50% of vulnerabilities over last decade
    [Wagner et al, 2000 : CERT DB]
  - Internet worm - exploited fingerd
  - Buffer overruns in RPC services ranked as the top vulnerability to UNIX systems
    [SANS Institute 2001]
Overview

- Why is C so vulnerable?
  - Array references not automatically bounds checked
  - C library functions inherently unsafe: `strcpy()`, `gets()`, `strcat()`, `sprintf()` etc.
  - Very easy to get "off-by-one" bugs
Our Goal

• Automate buffer overrun detection
  - Use Static Analysis

• State of the Art: Research Prototype
  - Very good results on real life applications
  - No pointer analysis

• Our Contributions:
  - Points to Analysis
  - Use of Commercial Linear Program solvers
  - Modular Design
Ideas Involved

• Strings -- Abstract Data Types
  - Operations allowed — `strcpy()`, `strcat()`, ...

• Associate string \( s \) with two variables:
  - \( s\text{\_alloc} \): space allocated for \( s \)
  - \( s\text{\_len} \): length of \( s \)

• Safety Property: \( s\text{\_alloc} \geq s\text{\_len} \)
Ideas Involved

• A constraint for each string operation
  • `strcpy(a, b)`: `a_len = b_len`
  • `a = (char*)malloc(5);`: `a_alloc = 5`
  • `gets (a);`: `a_len = choose(1..INF)`

• Constraints for whole program:
  - Produce equations at each program point.
  - Solve as a Linear Program.
a = char(*) malloc(5);
gets(a);

Codesurfer

Constraint Generator

Transducer

Internal structures

LP Solver

Buffer Overrun on a

a_alloc = 5
a_len = choose(1 .. INF)

a_alloc_max >= 5
a_alloc_min <= 5
a_len_max >= INF
a_len_min <= 1
Codesurfer

• Why Codesurfer?
  - Capable of points to analysis (3 precision levels)
  - Type information available

• How we use Codesurfer:
  - Builds a number of structures – use PDG nodes
  - Walk the PDG nodes for each procedure
  - Walk for each procedure
  - Constraint generated based on semantics
Constraint Generation

• Various classifications of program points

• Interested in call-sites, assignments & declarations
  - Why call sites? Calls to Functions
    a = strcpy(b,c);
  - Why assignments? char a[5];
  - Why declarations?
Modeling Functions

• **Context Sensitive vs. Context Insensitive**
  - Sensitive: differentiate call sites.
  - Insensitive: Merge information across call sites

• **Speed vs. Precision**
  - More Computation \(\Rightarrow\) Slower constraint generation
  - Greater Precision \(\Rightarrow\) fewer false alarms.
Modeling Functions

- False Alarms?

```c
char a[3], b[6];
strcpy(b, ID("hello"));
strcpy(a, ID("hi"));
```

![Diagram showing ID function with inputs "hi" and "hello" and output size [3, 6]]
**Constraint Generation**

- **Our Model**
  - **Context sensitive**: Commonly used library functions.
  - **Context insensitive**: User defined functions.

- **Using type information**
  - Produce only relevant constraints.
  - Limit interest to strings and integers.
Constraint Generation

• An Example

```c
char *ID(char *formal) {
    strcpy(a, ID("hi"));
    return formal;
}
```

• What do we have here?

- call site : `formal_len = 3`
  `ID_return_len = formal_len`
- assignment : `param2_len = ID_return_len`
- call to `strcpy` : `a_len = param2_len`
Flow Insensitivity

• How to “walk” the PDG?

• Flow Sensitive Analysis:
  - Respect program order
  - Space vs. Time concerns

• Flow Insensitivity:
  - Approach adopted here.
  - Loss in precision - False Alarms
  - Ease of implementation and faster code
Flow Insensitivity

- **False Alarms?:**
  
  ```c
  char *a, b[3], c[6];
  a = "hi";
  strcpy(b, a);
  a = "hello";
  strcpy(c, a);
  ```

- **Way around?:**
  - Copy of store at each CFG node
The Transducer

- Constraints produced in an Intermediate Representation (IR)
  - Simple Mathematical equations
  - Easy for debugging purposes

- Converts IR to the input format of Linear Program Solver
Linearizing Constraints

• Transducer linearizes constraints

• Only “simple” constraints

• Example:

\[
\text{gets}(a) : \quad \text{a\_len} = \text{choose}(1..\text{INF}) \\
\text{a\_len\_max} \geq \text{INF} \\
\text{a\_len\_min} \leq 1
\]
Linearizing Constraints

- More examples:
  - Multiple assignments to a variable
    - `strcpy(a,"hi");` \(a_{\text{len}} = 3\)
    - `strcpy(a,"hello");` \(a_{\text{len}} = 6\)

\[
\begin{align*}
  a_{\text{len}}_{\text{max}} & \geq 3 \\
  a_{\text{len}}_{\text{min}} & \leq 3 \\
  a_{\text{len}}_{\text{max}} & \geq 6 \\
  a_{\text{len}}_{\text{min}} & \leq 6
\end{align*}
\]
Linearizing Constraints

• Even more examples:

  - Min/max constraints
    • `strncpy(a, b, n);` \( \text{a\_len} = \min(\text{b\_len}, n) \)

      \[
      \begin{align*}
      \text{a\_len\_max} & \geq \text{fresh\_var} \\
      \text{a\_len\_min} & \leq \text{fresh\_var} \\
      \text{fresh\_var} & \leq \text{b\_len} \\
      \text{fresh\_var} & \leq n
      \end{align*}
      \]

      Try to make `fresh\_var` as large as possible
Linearizing Constraints

- Each variable from IR associated with 2 variables
  - Denote the range of the variable
  - Get the tightest possible range: How?

- A Linear Program:
  - minimize: an objective function
  - Subject to: a set of constraints

- Our case: minimize the range size:
  \[a_{\text{len\_max}} - a_{\text{len\_min}}\]
The LP Solver

- Takes in:
  - The Objective Function
  - The constraints

- Gives out:
  - Solution satisfying all the above constraints

- Using SoPlex: Off the shelf solver
  - Takes input in MPS format
  - Modular design simplifies plugging in any solver
Current Status

• **Completed:**
  - Handled a number of library functions
  - Handled generalized calls and assignments
  - Incorporated the use of types
  - Linearizing constraints and producing the MPS file for SoPlex.
  - Constraint generation for real life programs
    sendmail-8.7.6 ~40K lines before macro expansion.
Current Status

• To be done:
  - Dictionary – for library functions.
    • Function prototype available
    • Source code of function body unavailable
    • Can mimic constraint generation
  - Wrapper around the LP solver.
  - Stress testing
  - Results on widely used software packages
Current Status

• BSD Talk Daemon
  - ~900 lines of code before macro expansion
  - ~5000 lines of code after macro expansion
  - Dictionary not yet written
  - 157 variables in the linear program
  - 222 equations
  - SoPLex takes negligible time to solve
Future Work

- **Context Sensitive Handling of user defined functions**
  - Compute the transfer functions

- **Identifying difference constraints**
  - Fast Solvers Exist
  - How to incorporate this with the LP Solver?

- **Apply concepts developed to Assembly Code**
Demo

- **Constraint Generation**
- **Linearized Constraints** and **Map File**
- **MPS File**
- **Results** Overrun Observed on “hname” in main()
- **Talk Daemon:**
  - **Constraint Generation**