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 vulerability 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

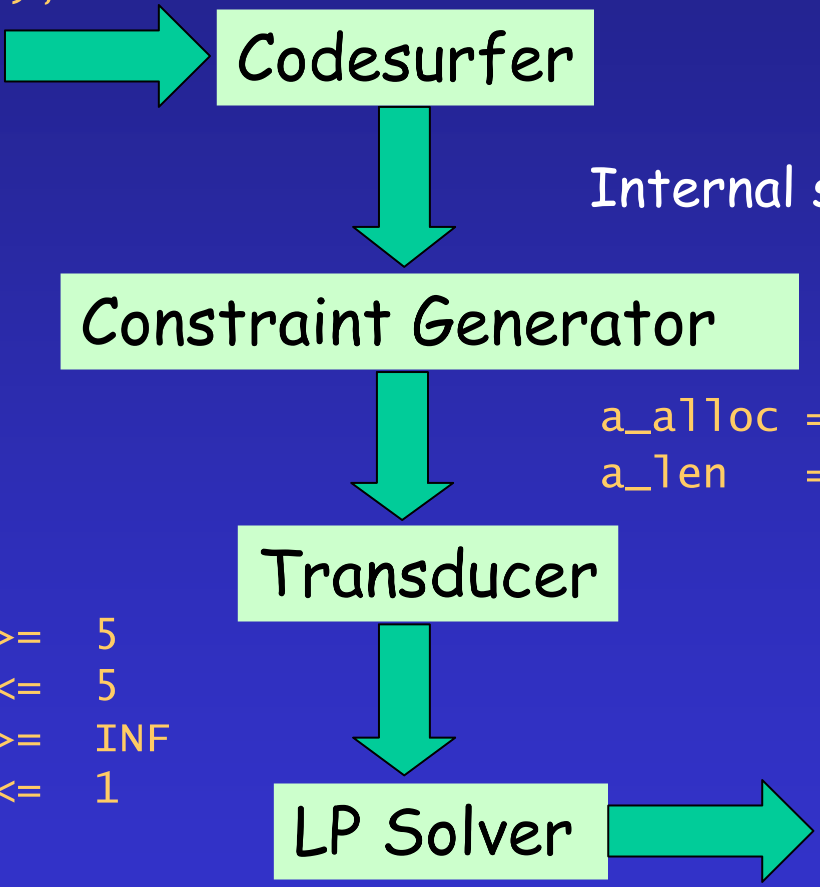
- Strings -- Abstract Data Types
  - Operations allowed – `strcpy(), strcat(),...`
- Associate string `s` with two variables:
  - `s_alloc` : space allocated for `s`
  - `s_len` : length of `s`
- Safety Property: `s_alloc >= s_len`

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.

Tool Layout



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

Codesurfer

Internal structures

Constraint Generator

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

Transducer

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

LP Solver

Buffer Overrun
on a

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
– Why assignments?	`a = strcpy(b,c);`
– Why declarations?	`char a[5];`

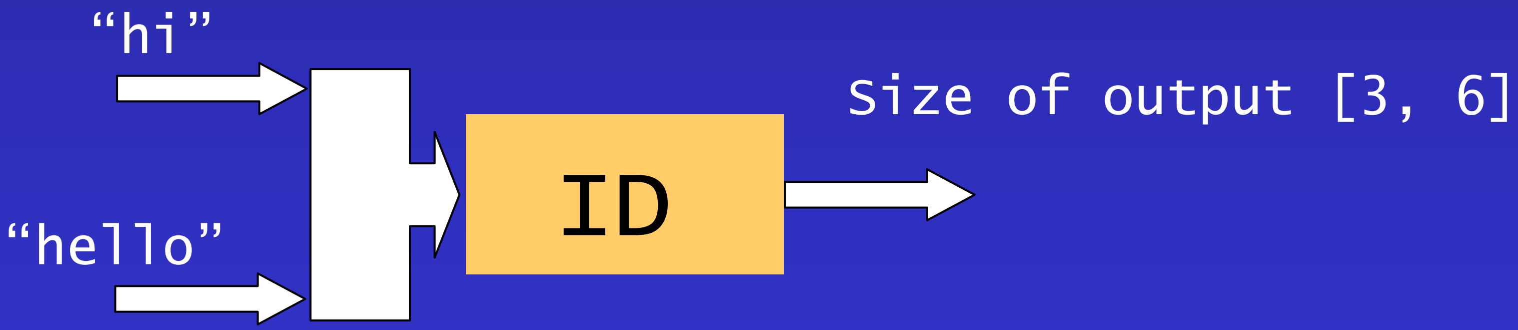
Modeling Functions

- Context Sensitive vs. Context Insensitive
  - Sensitive: differentiate call sites.
  - Insensitive: Merge information across call sites
- Speed vs. Precision
  - More Computation => Slower constraint generation
  - Greater Precision => fewer false alarms.

Modeling Functions

- False Alarms?

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



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

```
…
strcpy(a, ID("hi"));
…
```

```
char *ID(char *formal){
  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?

```
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:

```
gets(a) : a_len = choose(1..INF)
          a_len_max >= INF
          a_len_min <= 1
```

Linearizing Constraints

- More examples:
  - Multiple assignments to a variable
    - `strcpy(a,”hi”);` `a_len = 3`

      `strcpy(a,”hello”);` `a_len = 6`

```
a_len_max >= 3
a_len_min <= 3
a_len_max >= 6
a_len_min <= 6
```

Linearizing Constraints

- Even more examples:
  - Min/max constraints
    - `strncpy(a, b, n);`   `a_len = min(b_len, n)`

```
a_len_max >= fresh_var
a_len_min <= fresh_var
fresh_var <= b_len
fresh_var <= n
```

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 tighest possible range: How?

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

- Our case: minimize the range size :

  `a_len_max - a_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