CSSV: Towards a Realistic Tool for Statically Detecting All Buffer Overflows in C

Nurit Dor, Michael Rodeh, Mooly Sagiv

PLDI’2003

DAEDALUS project
Vulnerabilities of C programs

/* from web2c [strpascal.c] */

void foo(char *s)
{
    while ( *s != ' ' )
        s++;

    *s = 0;
}

Vuln: Null dereference
Vuln: Dereference to unallocated storage
Vuln: Out of bound pointer arithmetic
Vuln: Out of bound update
Is it common?

- General belief – yes!
- FUZZ study
  - Test reliability by random input
  - Tens of applications on 9 different UNIX systems
  - 18% – 23% hang or crash
- CERT advisory
  - Up to 50% of attacks are due to buffer overflow

**COMMON AND DANGEROUS**
CSSV’s Goals

- Efficient conservative static checking algorithm
  - Verify the absence of buffer overflow --- not just finding bugs
  - All C constructs
    - Pointer arithmetic, casting, dynamic memory, …
  - Real programs
  - Minimum false alarms
/* from web2c [fixwrites.c] */
define BUFSIZ 1024
char       buf[BUFSIZ];

char insert_long(char *cp)
{
    char       temp[BUFSIZ];
    ...
    for (i = 0; &buf[i] < cp ; ++i)
        temp[i] = buf[i];
    strcpy(&temp[i],"(long)");
    strcpy(&temp[i+6],cp);
    ...
}
/* from web2c [fixwrites.c] */
#define BUFSIZ 1024
char buf[BUFSIZ];

char insert_long(char *cp)
{
    char temp[BUFSIZ];
    ...
    for (i = 0; &buf[i] < cp ; ++i)
        temp[i] = buf[i];
    strcpy(&temp[i],"(long)");
    strcpy(&temp[i+6],cp);
    ...

Cleanness is potentially violated: 
7 + offset (cp) ≥ BUFSIZ
/* from web2c [fixwrites.c] */
#define BUFSIZ 1024
char buf[BUFSIZ];

char insert_long(char *cp)
{
    char temp[BUFSIZ];
    ...
    for (i = 0; &buf[i] < cp ; ++i)
        temp[i] = buf[i];

    strcpy(&temp[i],"(long)");
    strcpy(&temp[i+6],cp);
    ...

Cleanness is potentially violated:
offset(cp)+7 +len(cp) ≥ BUFSIZ
7 + offset (cp) < BUFSIZ
Verifying Absence of Buffer Overflow is non-trivial

```c
void safe_cat(char *dst, int size, char *src )
{
    (size > strlen(src) + strlen(dst)) ⇒ alloc(dst+len(dst)) > len(src))

    if ( size > strlen(src) + strlen(dst) )
    {
        dst = dst + strlen(dst);
        strcpy(dst, src);
        {string(src) ∧ alloc(dst) > len(src)}
    }
}
```
Can this be done for real programs?

- Complex linear relationships
- Pointer arithmetic
- Loops
- Procedures

- Use Polyhedra[CH78]
- Points-to-analysis
- Widening
- Procedure contracts

Very few false alarms!
C String Static Verifier

- Detects string violations
  - Buffer overflow (update beyond bounds)
  - Unsafe pointer arithmetic
  - References beyond null termination
  - Unsafe library calls
- Handles full C
  - Multi-level pointers, pointer arithmetic, structures, casting, …
- Applied to real programs
  - Public domain software
  - C code from Airbus
Operational Semantics

$p_1 = \text{alloc}(m)$

$p_2 = p_1 + i$

$p_3 = *p_2$
Domain Construction

• Given an abstract domains $D_1, D_2, \ldots, D_k$
• Construct a “composite domain” $c(D_1, D_2, \ldots, D_k)$
• Examples:
  – Cartesian Abstraction
• More later
CSSV’s Abstraction

- Ignore exact location
- Track base addresses

```
i 0x480000
p_1 0x480580
  0x5050510
p_2 0x480590
  0x5050518
p_3 0x490000
  0x5050510
  20
heap_1
```

Shadow memory

<table>
<thead>
<tr>
<th>base</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x480000</td>
<td>4</td>
</tr>
<tr>
<td>0x480580</td>
<td>4</td>
</tr>
<tr>
<td>0x480590</td>
<td>4</td>
</tr>
<tr>
<td>0x490000</td>
<td>4</td>
</tr>
</tbody>
</table>

```
0x5050510
  999
0x5050518
  20
```

Abstract locations
CSSV’s Abstraction

- Track sizes

Abstract locations

\[
\begin{align*}
  i & : 0x480000, 8 \\
  p_2 & : 0x480580, 0x5050518 \\
  p_1 & : 0x480590, 0x5050510 \\
  p_3 & : 0x490000, 20 \\
  \text{heap}_1 & : 0x5050510, 20, 999 \\
  \text{shadow memory base} & : 0x480000, 0x480580, 0x480590 \\
  \text{shadow memory size} & : 4, 4, 4
\end{align*}
\]
CSSV’s Abstraction

- Track pointers from one base to another (may)

```plaintext
Abstract locations

<table>
<thead>
<tr>
<th>i</th>
<th>0x480000</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>0x480590</td>
</tr>
<tr>
<td>p2</td>
<td>0x480580</td>
</tr>
<tr>
<td>p3</td>
<td>0x490000</td>
</tr>
</tbody>
</table>

Shadow memory

<table>
<thead>
<tr>
<th>base</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x480000</td>
<td>4</td>
</tr>
<tr>
<td>0x480580</td>
<td>4</td>
</tr>
<tr>
<td>0x480590</td>
<td>4</td>
</tr>
<tr>
<td>0x490000</td>
<td>4</td>
</tr>
</tbody>
</table>
```

m

heap1

| 0x5050510 | 999 |
| 0x5050518 | 20  |
Pointer Validation

• How can we validate pointer arithmetic?

\[ p_2 = p_1 + i \]

• Track offsets from origin
• Track numeric values
Numeric values are unknown

- Track integer relationships

\[ p_2 = p_1 + i \]

\[ p_2.\text{offset} = p_1.\text{offset} + i \]
Validation

• Pointer arithmetic

\[ p_2 = p_1 + i \]

\[ *p_1.size \geq p_1.offset + i \]

• Pointer dereference

\[ p_3 = *p_2 \]

\[ *p_2.size \geq p_2.offset \]
The null-termination byte

• Many expressions involve the ‘\0’ byte

\texttt{strcpy(dst, src)}

• Track the existence of null-termination
• Track the index of the first one
Abstract Transformers

- Defines the effect of statements on the abstract representation

\[ p_1 = \text{alloc}(m) \]

\[ p_2 = p_1 + i \]
Abstract Transformers

- Unknown value

\[ p_3 = *p_2 \]

\[
\begin{align*}
p_3 &= 0 \\
p_3 &= \text{unknown} \\
\end{align*}
\]

\[
\begin{align*}
p_3 &= 0 & *p_2.\text{is\_nullt} \land *p_2.\text{len} &= p_2.\text{offset} \\
p_3 &= \text{unknown} & \text{otherwise}
\end{align*}
\]
Overly Conservative

- Representing infeasible concrete states
- Infeasible pointer aliases
- Infeasible integer variables
Procedure Calls – Contracts

char* strcpy(char* dst, char *src)

requires ( string(src) \&\&
alloc(dst) > len(src) )

mod len(dst), is_nullt(dst)

ensures ( len(dst) == pre@len(src) \&\&
return == pre@dst )
Advantages of Procedure Contracts

• Modular analysis
  – [Not all the code is available]
  – Enables more expensive analyses

• User control of the verification
  – Detect errors at point of logical error
  – Improve the precision of the analysis
  – Check additional properties
    • Beyond ANSI-C
Specification and Soundness

- All errors are detected
- Violation of procedure’s precondition
  - Call
- Violation of procedure's postcondition
  - Return
- Violation of statement’s precondition
  - ..a[i]...
Procedure Calls – Contracts

char* strcpy(char* dst, char *src)

requires ( string(src) ∧
            alloc(dst) > len(src)
        )

mod len(dst), is_nullt(dst)

ensures ( len(dst) == pre@len(src) ∧
          return == pre@dst
        )
safe_cat’s contract

void safe_cat(char* dst, int size, char* src)

requires ( string(src) \land string(dst)
                alloc(dst) == size
          )

mod dst

ensures ( len(dst) <= pre@len(src) +
                pre@len(dst) \land
                len(dst) >= pre@len(dst) )
Specification – insert_long()

/* insert_long.c */
#include "insert_long.h"

char buf[BUFSIZ];
char * insert_long (char *cp) {
    char temp[BUFSIZ];
    int i;
    for (i=0; &buf[i] < cp; ++i){
        temp[i] = buf[i];
    }
    strcpy (&temp[i],"(long)");
    strcpy (&temp[i + 6], cp);
    strcpy (buf, temp);
    return cp + 6;
}

char * insert_long(char *cp)
    requires( string(cp) ∧
    buf ≤ cp < buf + BUFSIZ
    )
    mod cp.len
    ensures ( len(cp) = = pre@len(cp) + 6
    ∧
    return_value = = cp + 6 ;
    )
/* from web2c [fixwrites.c] */
#define BUFSIZ 1024
char buf[BUFSIZ];

char insert_long(char *cp)
{
    char temp[BUFSIZ];
    ...
    for (i = 0; &buf[i] < cp; ++i)
        temp[i] = buf[i];
    strcpy(&temp[i], "(long)");
    strcpy(&temp[i+6], cp);
    ...

Cleanness is potentially violated:
7 + offset (cp) ≥ BUFSIZ
/* from web2c [fixwrites.c] */
#define BUFSIZ 1024
char buf[BUFSIZ];

char insert_long(char *cp)
{
    char temp[BUFSIZ];
    ...
    for (i = 0; &buf[i] < cp ; ++i)
        temp[i] = buf[i];
    strcpy(&temp[i],"(long)");
    strcpy(&temp[i+6],cp);
    ...

Cleanness is potentially violated:
    offset(cp)+7 + len(cp) ≥ BUFSIZ
    7 + offset (cp) < BUFSIZ
CSSV – Technical overview

C files

Procedure name

Contracts

Pointer Analysis

Procedure’s Pointer info

C2IP

Integer Proc

Integer Analysis

Potential Error Messages
Used Software

• ASToolKit [Microsoft]
  – LLVM, Soot
• Core C [TAU - Greta Yorsh]
  – CIL [Berkeley, LLVM]
• GOLF [Microsoft - Manuvir Das]
• New Polka [Inria - Bertrand Jeannet]
  – Apron
CSSV Static Analysis

1. Inline contracts
   • Expose behavior of called procedures

2. Pointer analysis (global)
   • Find relationship between base addresses
   • Project into procedures

3. Integer analysis
   • Compute offset information
## Preliminary results (web2C)

<table>
<thead>
<tr>
<th>Proc</th>
<th>line</th>
<th>coreC line</th>
<th>time (sec)</th>
<th>space (Mb)</th>
<th>errors</th>
<th>FA</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert_long</td>
<td>14</td>
<td>64</td>
<td>2.0</td>
<td>13</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>fprintf_pascal_string</td>
<td>10</td>
<td>25</td>
<td>0.1</td>
<td>0.3</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>space_terminate</td>
<td>9</td>
<td>23</td>
<td>0.1</td>
<td>0.2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>external_file_name</td>
<td>14</td>
<td>28</td>
<td>0.2</td>
<td>1.7</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>join</td>
<td>15</td>
<td>53</td>
<td>0.6</td>
<td>5.2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>remove_newline</td>
<td>25</td>
<td>105</td>
<td>0.6</td>
<td>4.6</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>null_terminate</td>
<td>9</td>
<td>23</td>
<td>0.1</td>
<td>0.2</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>
## Preliminary results (EADS/RTC_Si)

<table>
<thead>
<tr>
<th>Proc</th>
<th>line</th>
<th>coreC line</th>
<th>time (sec)</th>
<th>space (Mb)</th>
<th>errors</th>
<th>FA</th>
</tr>
</thead>
<tbody>
<tr>
<td>FiltrerCarNonImp</td>
<td>19</td>
<td>34</td>
<td>1.6</td>
<td>0.5</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>SkipLine</td>
<td>12</td>
<td>42</td>
<td>0.8</td>
<td>1.9</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>StoreIntInBuffer</td>
<td>37</td>
<td>134</td>
<td>7.9</td>
<td>21</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
CSSV: Summary

• Semantics
  – Safety checking
  – Full C
  – Enables abstractions

• Contract language
  – String behavior
  – Omit pointer aliasing

• Procedural points-to
  – Scalable
  – Improve precision

• Static analysis
  – Tracks important string properties
  – Utilizes integer analysis
Related Projects

- SAL Microsoft
- Splint: David Evans
- Sage: Microsoft
- Brian Hacket static analysis, ICSE’2006
- Vinod Ganapathy: CCS’2013
Ambitious sound analyses

Very few false alarms

Scaling is an issue
  – Use staged analyses
  – Use modular analysis
  – Use encapsulation