Light-weight Bounds Checking

Niranjan Hasabnis, Ashish Misra, and R. Sekar
Memory Errors

2003
- Lock IT Down: Sapphire/Slammer worm attacks SQL Server and the Internet
  By John McCormick
  January 26, 2003, 8:00am PST

2005
- Linux introduced ASLR. Other OS followed it later.

2010
- Adobe PDF exploits using signed certificates, bypasses ASLR/DEP
  By Ryan Naraine | September 9, 2010, 11:54am PDT

2012
- NIST National Vulnerability Database
  There are 530 matching records. Displaying matches 1 through 20.
ASLR: Problem

It is a probabilistic defense!

To overcome this challenge, we need deterministic and prompt defense.
Research Efforts

• Debugging techniques
  – Purify [Hastings et al 92]
  – Valgrind [Nethercote et al 07]

• Type-safe C dialects
  – Cyclone [Morrisett et al 02]
  – CCured [Necula et al 05]

• Bounds checking
  – Backwards-compatible bounds checking for arrays and pointers in C programs [Jones et al 97]
  – CRED [Ruwase et al 04]
  – Baggy bounds [Akritidis et al 09]
Why Do We Need A New Method?

Most existing solutions are not widely deployed!

- **Performance**
  - Fastest previous bounds checker: 60% runtime overhead

- **Compatibility**
  - Breaks some programs
  - Checks pointer *arithmetic*
    - Problems when
      - pointers are cast into integers
      - invalid pointers are created but not used

Check pointer dereference and not arithmetic!
Our Approach: LBC

- **Limitation**: jumping past guard zones
- **Challenge**: how to make it efficient
Contributions

• **Light-weight Bounds Checking** is a source-to-source transformation based memory error detection technique
  – **Efficient:**
    • runtime overhead: 23% (half of fastest previous bounds checker)
    • space overhead: 8.5%
  – **Compatible:** compiled 7M LOC

• **Allows arbitrary pointer manipulations, but checks pointer dereferences**
Achieving Efficiency And Compatibility: Attempt 1

```c
if (guardmap[p] == 1)
    flag_error()
else
    *p
```

Guard map

Incompatible!
Achieving Efficiency And Compatibility: Attempt 2

```c
if (guardmap[x][y] == 1)
    flag_error()
else
    *p
```

Compatible but too slow!
Achieving Efficiency And Compatibility: Attempt 3

```c
if (*p == guard_zone_value)
    if (guardmap[x][y] == 1)
        flag_error()
else
    *p
```

Compatible and efficient!
Guardzone Size

• Tradeoff between security and performance

max (k * element_size, request_size / n)

k, n: configurable

Arrays

Non-arrays
Getting Even More Efficiency: Optimizations
Optimization (1): Compiler Optimizations

• Remove check for “safe” variables
  – Variable whose address is not taken cannot be involved in overflow (array is exception)

• All compiler-supported optimizations
  – CSE

\[ p->q->r = p->q->r + p->q->s; \]

Lots of optimizations from compiler … for FREE!
Optimization (2): Guard zone Placement

Before optimization

Guard of x

Guard of y

After optimization

Guard of x

Guard of y

Guard of x & y

Combined guard of x & y

RA

x

y

Activation record on x86

front and rear combined
Optimization (3): Lock Optimization For Multi-threaded Programs

• Naïve approach is to use lock for all concurrent accesses to guard map.

Develop an approach to reason about the cases when lock is not needed. Prove correctness.
Optimization (4): Static Analysis

• Observation
  – Most pointers are not involved in arithmetic.
  – Guard zone checks can be eliminated for such pointers.

• Approach
  – Classify pointers into
    • SAFE (cannot cause overflow)
    • UNSAFE (may cause overflow)
Static Analysis …

• CCured [Necula et al 05]
  – **SAFE**: not involved in arithmetic and unsafe typecasting

• LBC SAFE = CCured SAFE

Don’t check SAFE pointers and SAFE objects.
But we made an important change to CCured’s algorithm.

```c
int main()
{
    int a[2] = {100, 200};
    int* q = &a[0];

    for (int j = 0; j < 2; j++)
        q++;

    int* p = q; /* Line 36 */

    return 0;
}
```

Failure UBOUND at eager.c:36:
main(): Ubound
Aborted

No assignment from UNSAFE pointer to SAFE pointer.
Implementation

• Package
  – Source-to-source transformer:
    • OCaml (5KLOC) + CIL [McPeak et al 02]
  – Runtime support library: C (1KLOC)

• Supports 32-bit Linux
  – 64-bit release is on the way

http://seclab.cs.sunysb.edu/download.html
Evaluation

• Compatibility

• Security: effectiveness in stopping exploits

• Runtime and Space overhead

• Effectiveness of optimizations
Compatibility

7 Million
Detecting Memory Errors And Stopping Exploits

- **Bugbench** [Lu et al 2005]
  - Overflows in `bc`, `man`, `polymorph`, `gzip`, `ncompress`
  - LBC detected and prevented all the overflows.
  - `cvs`, `mysql1`, `mysql2`, `mysql3` contain other kinds of bugs which are not bounds errors.

- **RIPE** [Wilander et al 2011]
  - 850 different attacks
  - 4 attack areas: stack, heap, data, bss
  - 5 target code pointers (return address, base pointer, etc)
  - multiple overflow techniques: direct, indirect
  - ...

LBC detected and prevented 770 attacks.
It missed 80 which are intra-object overflows.
Runtime And Space Overhead For SPECINT 2000

Runtime Overhead

- LBC (23%)
- Baggy (60%)

Space Overhead (8.5 %)

- Fixed
- Variable
Effectiveness Of Optimizations

- No Optimization (556%)
- Fast Check (86%)
- Fast Check + Lock Optimization (41%)
- Fast Check + Lock Optimization + Static Analysis (23%)
Related Work

• Debugging techniques
  – Purify [Hastings et al 92]
  – Valgrind [Nethercote et al 07]

• Comprehensive memory error detection
  – RTCC [Steffen et al 92]
  – CCured [Necula et al 05]

• Bounds checking
  – Backwards-compatible bounds checking for arrays and pointers in C programs [Jones et al 97]
  – Baggy bounds checking [Akritidis et al 09]

• Security-targeted techniques
  – ASLR [PaX group 00]
  – Write-integrity testing (WIT) [Akritidis et al 08]
Conclusion

• Light-weight backwards compatible memory error detection technique
  – Runtime overhead: 23% (half of fastest previous bounds checking technique)
  – Compatibility: compiled 7 Million lines of code

• Favorable factors for easy deployment
Thank You

nhasabni@cs.stonybrook.edu

http://seclab.cs.sunysb.edu/download.html
Backup Slides
Compatibility

```c
struct S {
    int* p;
} *q;
q-> p = &i;
foo (q);

q >= q.base && (char*) q + sizeof(q) <= q.size
q->p >= p.base && (char*) q->p + sizeof(*q->p) <= p.size
*q->p

void foo (struct S* q) {
    static int j;
    q->p = &j;
}
```

False positives cause compatibility issues
Static Analysis

• But … CCured works in “merged mode”..

• For compatibility, we support “separate compilation” also, and for that we must do conservative analysis.

All pointers, by default, are WILD. A pointer is made SAFE iff it is not assigned from WILD, and not involved in arithmetic and unsafe cast.
### Instrumentation

<table>
<thead>
<tr>
<th>Original Program</th>
<th>LBC-Transformed Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>void f() {</td>
<td>void f() {</td>
</tr>
<tr>
<td></td>
<td>struct x_type {</td>
</tr>
<tr>
<td></td>
<td>char front[24];</td>
</tr>
<tr>
<td></td>
<td>int orig;</td>
</tr>
<tr>
<td></td>
<td>char rear[24];</td>
</tr>
<tr>
<td></td>
<td>} x_gz;</td>
</tr>
<tr>
<td></td>
<td>int* y;</td>
</tr>
<tr>
<td></td>
<td>init_guardzone(x_gz);</td>
</tr>
<tr>
<td></td>
<td>init_guardmap(x_gz);</td>
</tr>
<tr>
<td></td>
<td>y = &amp;x_gz.orig;</td>
</tr>
<tr>
<td></td>
<td>if (*y == guard_zone_value)</td>
</tr>
<tr>
<td></td>
<td>if (slowcheck(y))</td>
</tr>
<tr>
<td></td>
<td>flag_error();</td>
</tr>
<tr>
<td></td>
<td>*y = 100;</td>
</tr>
<tr>
<td></td>
<td>uninit_guardmap(x_gz);</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
</tbody>
</table>

**Original Program**

```c
void f() {
    int x;

    int* y;

    y = &x;

    *y = 100;
}
```

**LBC-Transformed Program**

```c
void f() {
    struct x_type {
        char front[24];
        int orig;
        char rear[24];
    } x_gz;

    int* y;

    init_guardzone(x_gz);
    init_guardmap(x_gz);

    y = &x_gz.orig;

    if (*y == guard_zone_value)
        if (slowcheck(y))
            flag_error();
        *y = 100;

    uninit_guardmap(x_gz);
}
```