Baggy bounds checking

Periklis Akritidis, Manuel Costa, Miguel Castro, Steven Hand
C/C++ programs are vulnerable

- Lots of existing code in C and C++
- More being written every day
- C/C++ programs are prone to bounds errors
- Bounds errors can be exploited by attackers
Previous solutions are not enough

• Finding all bugs is unfeasible
• Using safe languages requires porting
• Existing solutions using fat pointers (Ccured, Cyclone) break binary compatibility
• Backwards compatible solutions are slow
• And performance is critical for adoption
Baggy bounds checking (BBC)

- Enforce allocation instead of object bounds

- Constrain allocation sizes and alignment to powers of two
  - Fit size in one byte
  - No need to store base address

- Fast lookup using linear table
BBC Benefits

• Works on unmodified source code
• Broad coverage of attacks
• Interoperability with uninstrumented binaries
• Good performance
  – 30% average CPU overhead
    • 6-fold improvement over previous approaches on SPEC
  – 7.5% average memory overhead
  – 8% throughput degradation for Apache
System overview

Unmodified C Source Code → Generate IR → Analyze

Baggy Bounds Checking

Runtime Support Library → Insert Checks

Binary Libraries

Link → Generate Code

Hardened Executable
Attack Example

• Pointers start off valid
  \[ p = \text{malloc}(200); \]

• May become invalid
  \[ q = p + 300; \]

• And then can be used to hijack the program
  \[ *q = 0x00000\text{BAD} \]
Traditional Bounds Checking

[Jones and Kelly]

• Use table to map allocated range to bounds
  \( p = \text{malloc}(200); \)
• Lookup bounds using source \( p \)
  \( q = p + 300; \)
• Check result \( q \) using bounds
• Note that source pointer \( p \) assumed valid
  – points to allocation or result of checked arithmetic
  – maintain this invariant throughout execution
• But keeping bounds information is expensive...
Baggy Bounds

• Pad allocations to power of 2
  – malloc(200) -> malloc(256)

• Align to allocation size
  – Upper bound: \texttt{0x888888}FF
  – Lower bound: \texttt{0x888888}00

• Can recover bounds using
  – The valid source pointer
  – The binary logarithm of the allocation size
Bound table implementation

• Previous solutions need e.g. splay tree to lookup bounds for a given source pointer
• If force allocations to be a multiple of 16 byte slots, can use an array with 1 byte per slot
Efficient table lookup

mov eax, p ; Copy pointer
shr eax, 4 ; Right-shift by 4
mov al, [TABLE+eax] ; One memory read

• Loads allocation size logarithm in register %al
• However:
  • No need to recover explicit bounds
  • Use valid pointer and allocation size directly
Efficient Checks

\[ q = p + 300; \]

```assembly
mov ebx, p ; copy source 0x88888800
xor ebx, q ; xor with result 0x8888892C
shr ebx, al ; right shift >> 8
; by table entry 0x00000001
jnz error ; check for zero !!!
```
(Legal) Out-of-bounds pointers

- C programs can use out-of-bounds pointers
- Cannot dereference
- Can use in pointer arithmetic
- C standard allows only one byte beyond object
  - Some programs go beyond, or below object e.g.
    ```c
    char *array = malloc(100) – 1;
    // now can use array[1..100]
    ```
Dealing with OOB pointers

• 1. Mark to avoid dereference
  – Set pointer top bit [Dhurjati et al.]
  – Protect top half of address space

• 2. Recover valid pointer if marked
  – Can use extra data structure [Ruwase and Lam]
  – BBC: support most cases without a data structure
  – (can support more in 64-bit mode – see later)
Common out-of-bounds pointers

[Diagram showing out-of-bounds pointers with labeled sections: bottom half of slot and top half of slot.]
Extra check in fast path

- Source marked? (YES/NO)
  - YES: Get valid pointer → Table lookup
  - NO: Result valid? (YES/NO)
    - YES: OK
    - NO: Mark result
Optimized fast path

Table lookup

Result valid?

Get valid pointer

OK

Source marked?

Mark result
Memory Allocations

• Heap using binary buddy system
  – Perfect fit for baggy bounds
• Align stack frames at runtime
  – Only if contains array or address taken variable
• Pad and align globals at compile time
• Memory allocated by uninstrumented code has default table entry
  – Default value 31: maximal bounds
Performance Evaluation

• Measured CPU and memory overhead
  – Olden and SPEC benchmarks
• Baggy
  – Baggy bounds checking as described
• Splay
  – Splay tree from previous solutions
  – Standard allocator
  – Same checks
Execution Time vs. Splay Tree

- 30% for baggy vs. 6x for splay tree on average
• 7.5% for baggy vs. 100% for splay on average
• 8% throughput decrease with saturated CPU
Effectiveness

- Evaluated using buffer overflow suite [Wilander and Kamkar]
- Blocked 17 out of 18 attacks
- Missed overflow between structure fields
Baggy bounds on x64

• Baggy bounds can fit inside pointers

Avoid memory lookup entirely:

```
mov rax, p   ; copy pointer
shr rax, 38  ; shift tag to %al
```
x64 Out-of-bounds pointers

- Adjust pointer by offset in spare bits
- Greatly increases out-of-bounds range
Conclusions

• Baggy Bounds Checking provides practical protection from bounds errors in C\C++
• Works on unmodified programs
• Preserves binary compatibility
• Good performance
  – Low CPU overhead (30% average)
  – Low memory overhead (7.5% average)
• Can protect systems in production runs