Hybrid Techniques for Preventing Memory Corruption Attacks in C

Prachi Gauriar
prachi_gauriar@ncsu.edu

- Background
- Survey
  - CRED
  - Cyclone
  - CCured
- Lightweight 2-version Software Development
  - Lightweight analysis for signature generation
  - Heavyweight analysis for fast debugging

Background

- Memory corruption attack
  - Exploit a memory access error to attack a system
- Memory access errors
  - Referent and accessed memory object differ
  - Spatial - pointer is outside bounds of its referent
    - Buffer overflow, format string, deref NULL
  - Temporal - referent no longer exists
    - Dereference freed variables, dangling pointers

Background

- C is susceptible because it is not type-safe
- Type-safety (strong typing)
  - Can’t operate on a value like it has a different type
  - Requires that memory is accessed safely
- C’s safety shortcomings
  - Pointer arithmetic, unsafe casts
  - No garbage collection or explicit memory rules
  - Bad variable argument implementation

Approaches

- Static Analysis
  - Analyze source code to find errors
  - Usually sound, all analysis is done offline
  - Lots of false positives, only sound relative to model
- Dynamic Checking
  - Modify runtime to prevent errors
  - Usually no restrictions on code, no false positives
  - Targets specific error types, adds overhead

Approaches

- Static/Dynamic Hybrid Approaches
  - Goal: best of both worlds
    - No false positives and catch most/all errors
    - Minimize runtime overhead
- Basic pattern
  - Analyze source code to find unsafe portions
  - Add runtime checks to regions where analysis is insufficient

Survey

CRED

- Adds bounds checking to string buffers
- Does not modify the pointer representation
  - Meshes better with uninstrumented code
- Records the base address, size of all memory objects
  - Static, heap, and stack
  - Stores them in the *object table*
- At dereference, looks up pointer in object table
  - Perform bounds checks

CRED

- Pointer arithmetic presents special problems
- In-bounds pointer arithmetic
  - Address computed from an in-bounds pointer must refer to the same object as that pointer
  - Check if pointer is in-bounds
    - If so, find its referent
    - Final result of arithmetic must fall within the referent's bounds

CRED

- Out-of-bounds (OOB) pointer arithmetic
  - Address computed from an OOB pointer must refer to the same object as that pointer's referent
  - For each OOB pointer
    - CRED creates an OOB object in the heap
    - OOB pointer points to that OOB object
    - OOB object contains the original OOB pointer's address and referent

CRED

- After each address computation, check if result is OOB
- If so, create OOB object and store it in the OOB *table*
- If a pointer is used in pointer arithmetic
  - In bounds? If not, perform OOB table lookup
  - Use the OOB object's address and referent to perform a bounds check
- When an object is deallocated, remove it from the OOB

CRED

- Only checks strings
  - String buffers are most often overflowed
- Unsound
  - Can't detect non-string attacks
  - Does not analyze type casts
- Performance
  - Overhead < 26% in apps w/o heavy string use
  - Overhead of 60-130% otherwise

Cyclone

- "Safe" dialect of C
- Attempts to solve safety shortcomings, keep flexibility
  - Restricts pointer arithmetic, disallows unsafe casts
  - Inserts NULL checks before pointer dereferences
  - Requires pointers to be initialized before use
  - Augments varargs implementation
  - Adds garbage collection and region analysis to prevent spatial memory errors

Cyclone

- Several pointer kinds
  - Possibly-NULL (*-pointers)
  - Never-NULL (@-pointers)
  - Fat pointers (?-pointers)
- Possibly-NULL are like C pointers
  - NULL checks inserted before every dereference
- Never-NULL are optimizations of possibly-NULL
  - NULL checks inserted only when pointer changes

Cyclone

- Pointer arithmetic is only allowed on fat pointers
- Store the base address and size of the referent
- *size* field is programmer accessible, like *length* in Java
- NULL- and bounds-checked when dereferenced
- Arrays, strings automatically converted to ?-pointers

```
int strlen(const char *s) {
  int i = 0;
  if (!s) return 0;
  /* UNSAFE if S isn't
     NULL terminated */
  while (*s) i++;
  return i;
}
```

```
int strlen(const char ?s) {
  int i, n;
  if (!s) return 0;
  n = s.size;
  for (i=0; i<n; i++, s++)
    if (!*s) return i;
  return n;
}
```

Cyclone

- Safe C-like alternative to Java
  - Can’t mix with C code
  - Tool support is lacking
    - No lex, yacc, etc
  - Analysis can reject safe code
- Performance Overhead
  - With garbage collection: 0-185%
  - Without garbage collection: < 36%

CCured

- Program transformation tool
- Adds type-safety to C
- Solves many of the same problems as Cyclone
- Like Cyclone, introduces new pointer kinds
  - Three main kinds: SAFE, SEQ, WILD
  - Minor kinds: RTTI, FSEQ, others
- Unlike Cyclone, infers pointer kinds based on use
- Kind of pointer reflects how safely it is used

CCured

- SAFE - not used in pointer arithmetic
  - Just require NULL-check before dereference
- SEQ - used in pointer arithmetic, but no unsafe casts
  - Require bounds-check and NULL-check
- WILD - used in pointer arithmetic, unsafe casts
  - Bounds-, NULL-checks, and dynamic type checking
- At compile time, CCured suggests ways to make inferred WILD pointers into SAFE or SEQ pointers

CCured

- Real strength lies in its inference algorithm
  - < 1% WILD, < 10% SEQ
- Some changes to code are required
  - SEQ and WILD pointers are fat
  - External function wrappers to convert from fat to normal pointers
  - Some vararg functions like scanf require changes
  - sizeof should use a variable, not a type

CCured

- Performance
  - Runtime overhead: 3-891%
    - Removing bc, 3-87%, average: 30.8%
    - bc without garbage collection: less than 50%
- Sophisticated, sound analysis
- Supports lots of programming paradigms in C
- Works with non-instrumented code via wrappers
- Drop-in replacement for gcc

Current Research

Motivation

- Despite CCured's quality, has unacceptable overhead
- Security is only worth it if the cost is less than the price of being attacked
  - Highly trafficked commercial servers, online stores
- We want to leverage its analysis, but forgo overhead
- Simplest way to do this is to maintain two very similar versions of our software
  - CCured version for debugging and diagnosis
  - Release version

Methodology

- Highly pragmatic
- Development version D uses CCured
  - Used during coding, test, and bug diagnosis phases
- Release version is conventional C code
  - Possibly use address space randomization or IDS
- Vast majority of source is shared
  - D contains code in addition to or in place of R’s
  - Satisfy CCured’s compilation requirements

Methodology

- When a security bug is found in the release version
  - Glean information from logs, core dumps, etc
  - Attempt to replay attack on version D
- Currently exploring two analysis options
  - Lightweight - generate vulnerability signature
  - Heavyweight
    - Identify buffer, location of bug
    - Try to illuminate the cause via debugging tool

Lightweight

- Detect an attack and automate replay
  - Replace handlers for SIGSEGV, SIGBUS, SIGILL
- Augmented version of CCured can tell us
  - Which buffer was involved
  - Input size
  - Distribution of characters in the input
- Based on this, we can create a vulnerability signature
- Use this signature into an input filter

Heavyweight

- CCured aborts with arcane message upon error
- Must use debugging aids to find location of error
- For complex bugs, knowing *where* isn't always helpful
- We augment CCured with a tool that identifies the buffer that was accessed incorrectly
  - We then replay the attack to allow the user to watch for important memory related events
    - Allocation/reallocation, free, pointer arithmetic
  - Filter events by how directly they affect the buffer

Example

```
http.c:112: recv expected at least 1024 bytes of readable
data, but buffer has only 224.

Problematic buffer: pPostData (0x8157f10)
Perform further analysis? y
Degree of contribution? 1

Problematic buffer: pPostData (0x8157f10)
Allocated at http.c:100 using calloc
  Element count: 224 (conn[sid].dat->in_ContentLength+1024)
    conn[sid].dat->in_ContentLength: -800
...
  Element size: 1 (sizeof(char))
Assigned to pPostData at http.c:109
...
```

Questions?