CUP: Comprehensive User-Space Protection

Nathan Burow, Derrick McKee, Scott A. Carr, Mathias Payer

hexhive

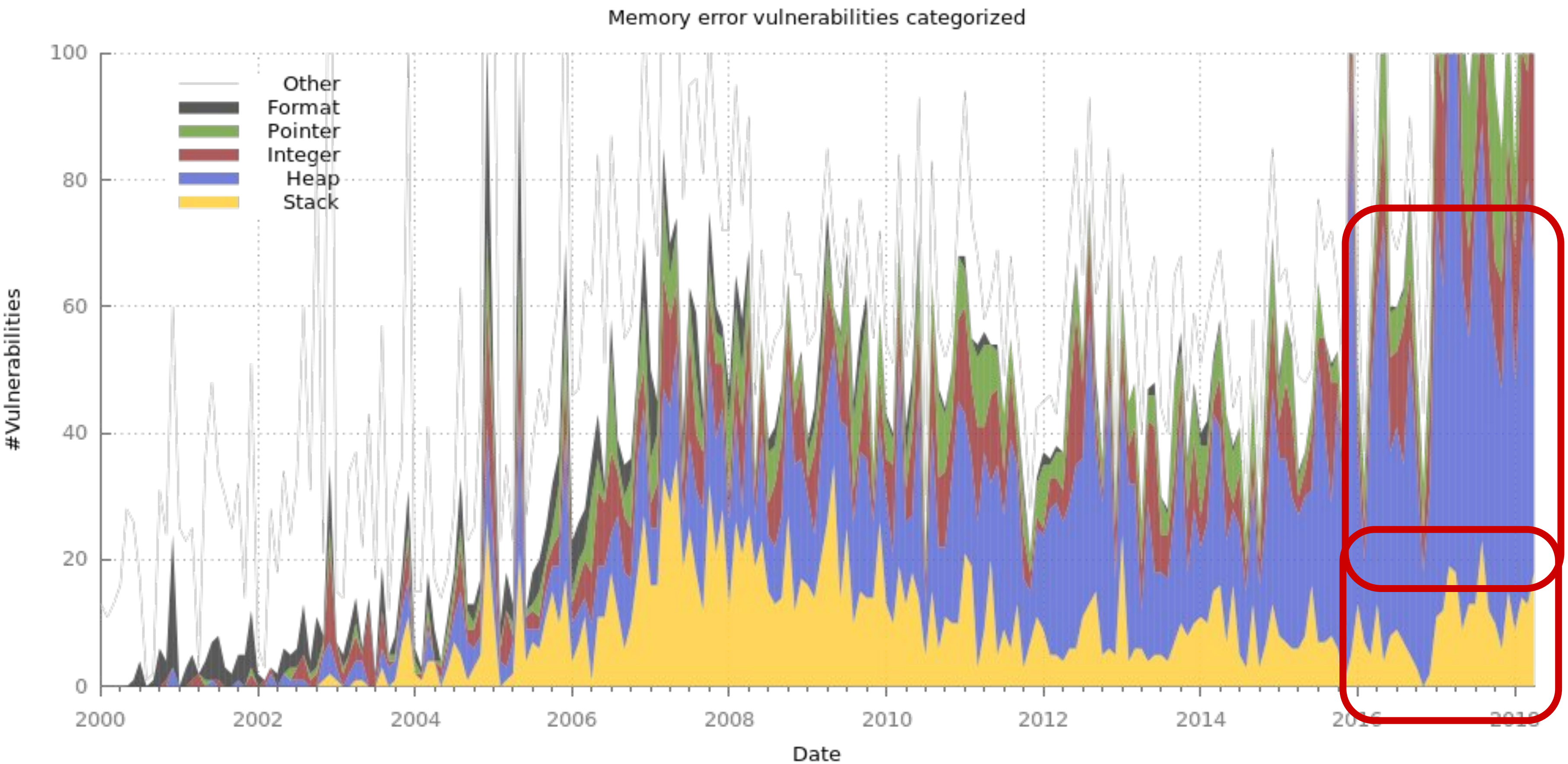
PURDUE UNIVERSITY

Memory Safety

- All software exploits rely on corrupting memory state
  - Control-flow hijacking: Code-pointers
  - Data only: Critical variables, program state
- C / C++ do **not** provide memory safety
- ~60 vulnerabilities and ~30 exploits per **month** [1]

[1] Victor Van der Veen, Lorenzo Cavallaro, and Herbert Bos. "Memory errors: The past, the present, and the future." RAID’12.

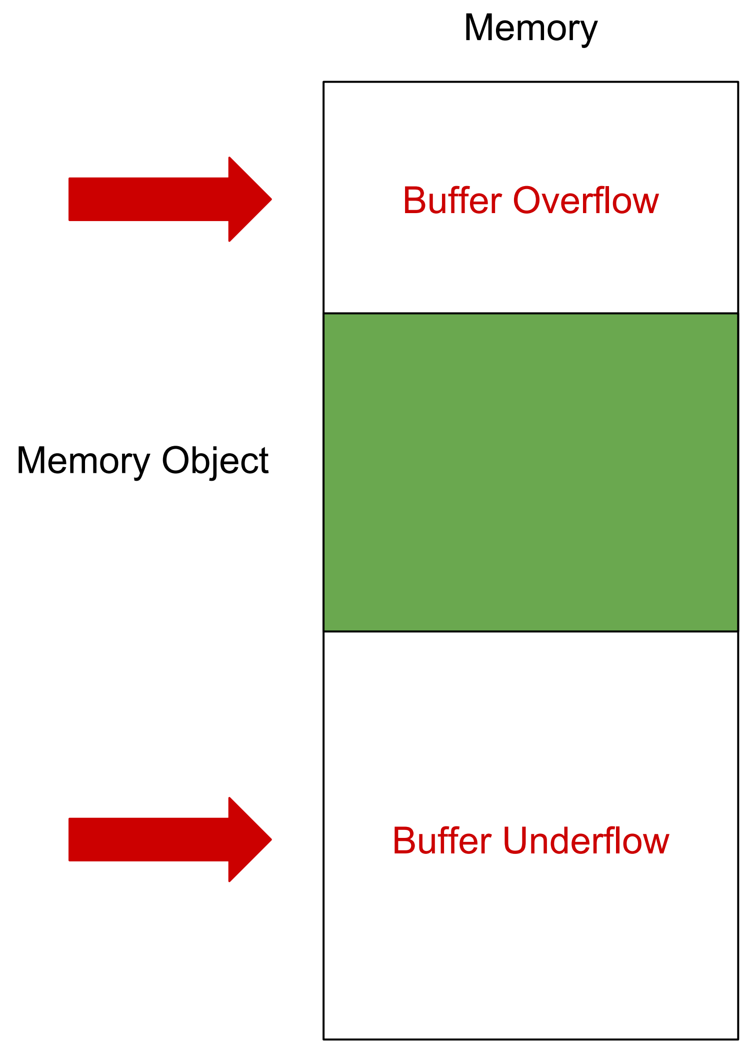
Memory Safety In The Wild



Updated data from: www.vvdveen.com/memory-errors

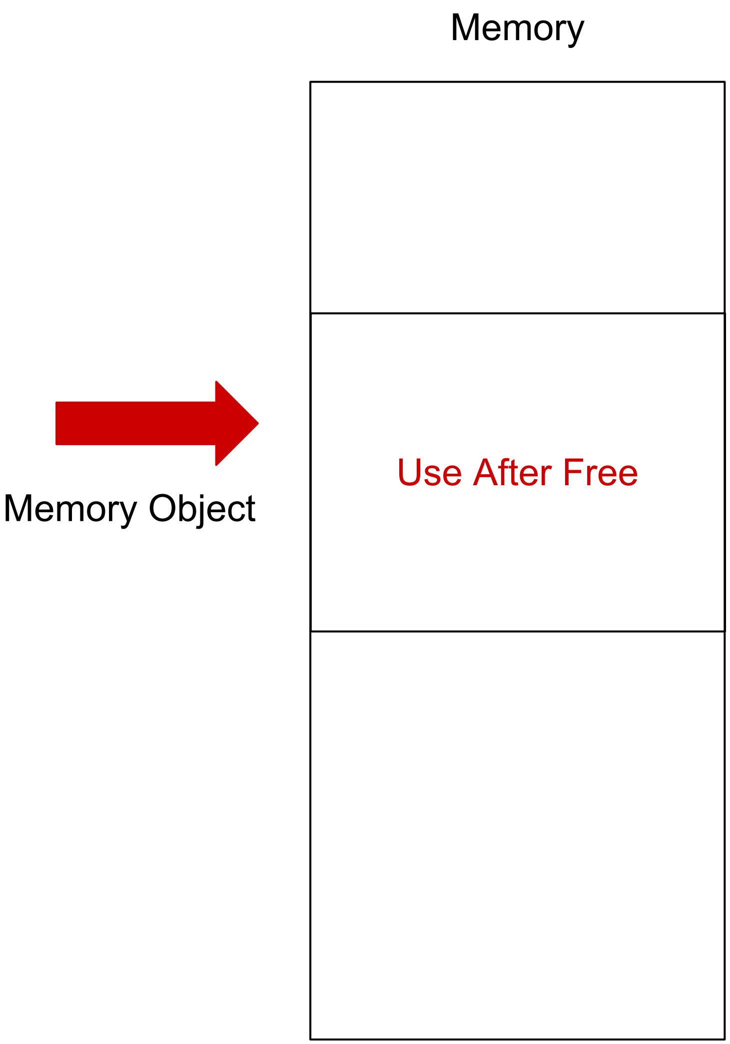
Memory Safety Definition

- Memory objects have capabilities:
  - **<u>Size</u>** -- base address and length
  - **<u>Allocation Status</u>** -- allocated, free
- Spatial violation
  - Violate the size capability
  - Buffer overflow
- Temporal violation
  - Violate the allocation status capability
  - Use-after-free



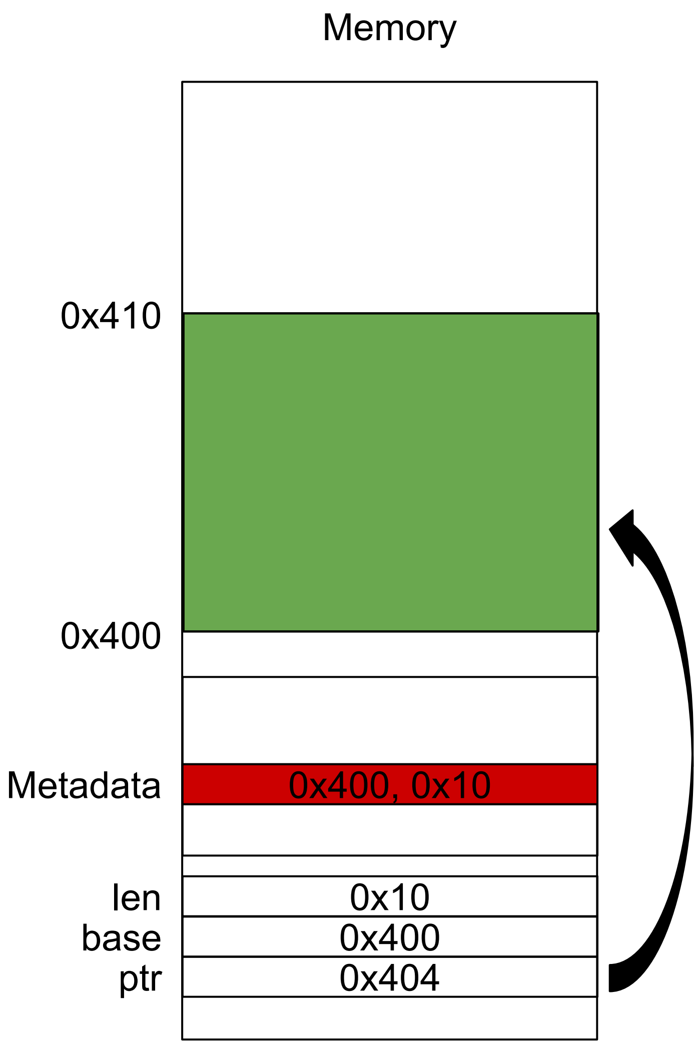
Memory Safety Definition

- Memory objects have capabilities:
    - **<u>Size</u>** -- base address and length
    - **<u>Allocation Status</u>** -- allocated, free
- Spatial violation
    - Violate the size capability
    - Buffer overflow
- Temporal violation
    - Violate the allocation status capability
    - Use-after-free



Related Work

- Spatial Safety
  - Fat Pointers -- inline metadata
  - SoftBound -- disjoint metadata
  - Low-Fat Pointers -- alignment based
- Temporal Safety
  - CETS -- persistent disjoint metadata
  - DangNull -- modify pointers on free



Limitations of Related Work

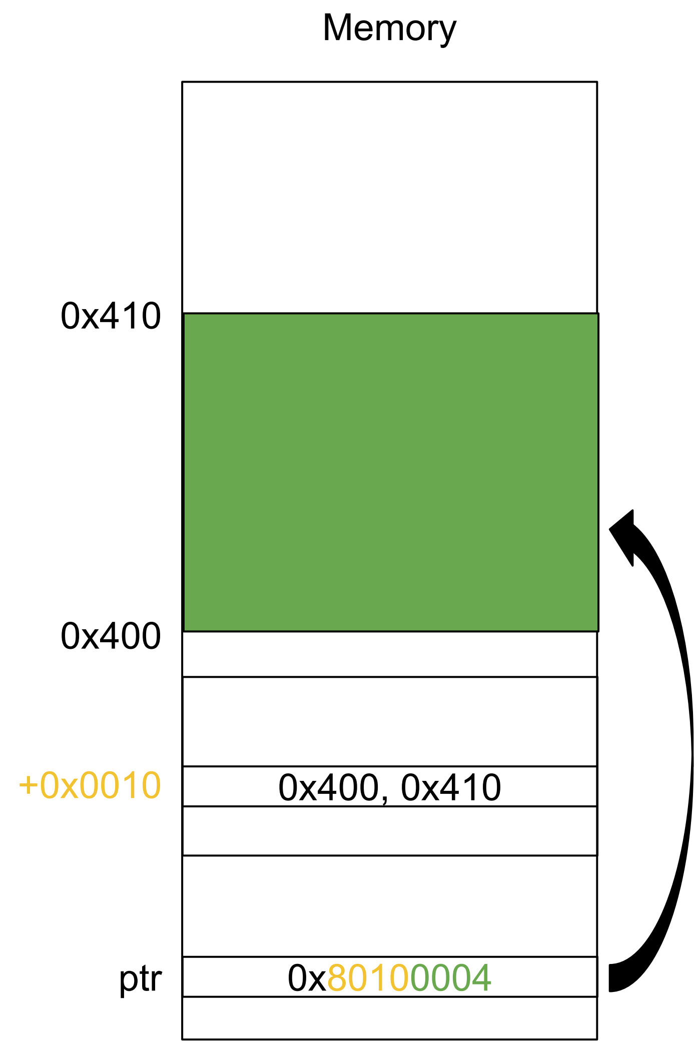
- Focus on compatibility instead of security
    - Do **<u>not</u>** modify pointers
    - Can silently fail to check a dereference
    - Validating correctness of implementation is difficult
- SoftBound+CETS
    - Two levels of indirection to look up metadata
    - Permanent storage of 8 bytes per object
- Do not scale to handle **<u>all</u>** memory allocations
    - SPEC CPU2006 benchmarks allocate up to 205 billion objects with pointers
    - Firefox allocates 1.4 billion objects with pointers to run the Kraken benchmark

Memory Safety Mechanism Requirements

- Precision
  - Must know exact size capability of every pointer
- Object Awareness
  - Must be able to track the allocation status capability of every pointer
- Comprehensive Coverage
  - Must protect all types of allocations: stack, heap, global
  - Must protect all allocations in user space
- Exactness
  - No false positives → Usable
  - No false negatives → Secure

Design

- Hybrid Metadata
  - Encodes capability ID in pointer
  - Fail Closed -- unchecked deref fail by default
  - Performant
    - IDs propagate naturally on assignment
    - Direct lookup of metadata
- Tradeoff: limited IDs
  - Reuse IDs → Probabilistic temporal guarantees
  - Full temporal safety until ID is reused
- Static analysis on Stack
  - Use local metadata for stack allocations
  - Saves capability IDs → Improves temporal guarantees



Validating Instrumentation Through Design

- Observation: finding memory allocations is easier than finding derefs
  - Can design guarantee that all pointers to instrumented allocations are checked?
  - If so, would only need to prove that all allocations are instrumented to validate implementation
- Enrich all pointers on allocation so that CPU faults if dereferenced
- Fails closed: enriched pointers **cannot** be dereferenced without check
  - Leads to no false negatives
  - Validates correctness of our implementation
- Improves over existing work which can silently miss a check

Implementation: Allocation

- Create metadata entry
  - Base is the first valid address
  - End is the last valid address
- Capability ID → index in metadata table
- Replace pointer with capability ID and offset
  - Set high order bit to 1
  - Next 31 bits are the ID -- metadata index
  - Low order 32 bits are offset in object
  - Offset is ptr - base, initially 0
- Hybrid metadata: pointer encodes ID

```
typedef struct {
  void *base;
  void *end;
} metadata_t;
```

```
typedef struct {
  unsigned int32 enriched : 1;
  unsigned int32 capbility_id : 31;
  unsigned int32 offset;
} enriched_t;
```

```
typedef union {
  void *native;
  enriched_t enriched;
} ptr_t;
```

Implementation: Dereference

- Reconstruct pointer: offset + base
- If pointer is in bounds:
    - Ptr - base >= 0
    - Upper - ptr >= 0
    - If fail, high order bit is 1 (negative number)
- Check computes these and puts high order bit in reconstructed pointer
- General purpose fault for out of bound dereferences

```
void *check_bounds(size_t ptr,
            size_t base, size_t upper)  {
  size_t valid = (ptr - base) | (upper - ptr);
  valid &=  0x8000000000000000;
  // valid is 0 if ptr >= base && ptr < upper
  return (void *)(ptr | valid);
}
```

Challenges for CUP: Temporal Safety

- On free, invalidate metadata
- Problem: eventually run out of capability IDs
  - Does not affect spatial safety, **only** temporal
- Solution is policy dependent:
  - Number of capability IDs in configurable -- tradeoff object size versus number of IDs
  - Reuse capability IDs
    - Free list
      - Memory usage: put IDs at front of free list
      - Security: randomize ID reuse
    - Garbage collect capability IDs
- Temporal safety depends on time to ID reuse
- If new capability does not overlap any previous capability → Secure

Comprehensive Coverage

- CUP recompiled and supports libc
- All user-space code **should** be recompiled with CUP
  - Compatibility mode exists to support incremental deployment
  - Significantly weakens security guarantees
- Kernel remains unprotected
  - Must instrument the syscall boundary between user and kernel space
  - Calls into kernel: unenrich pointers
  - Returns from kernel: enrich pointers

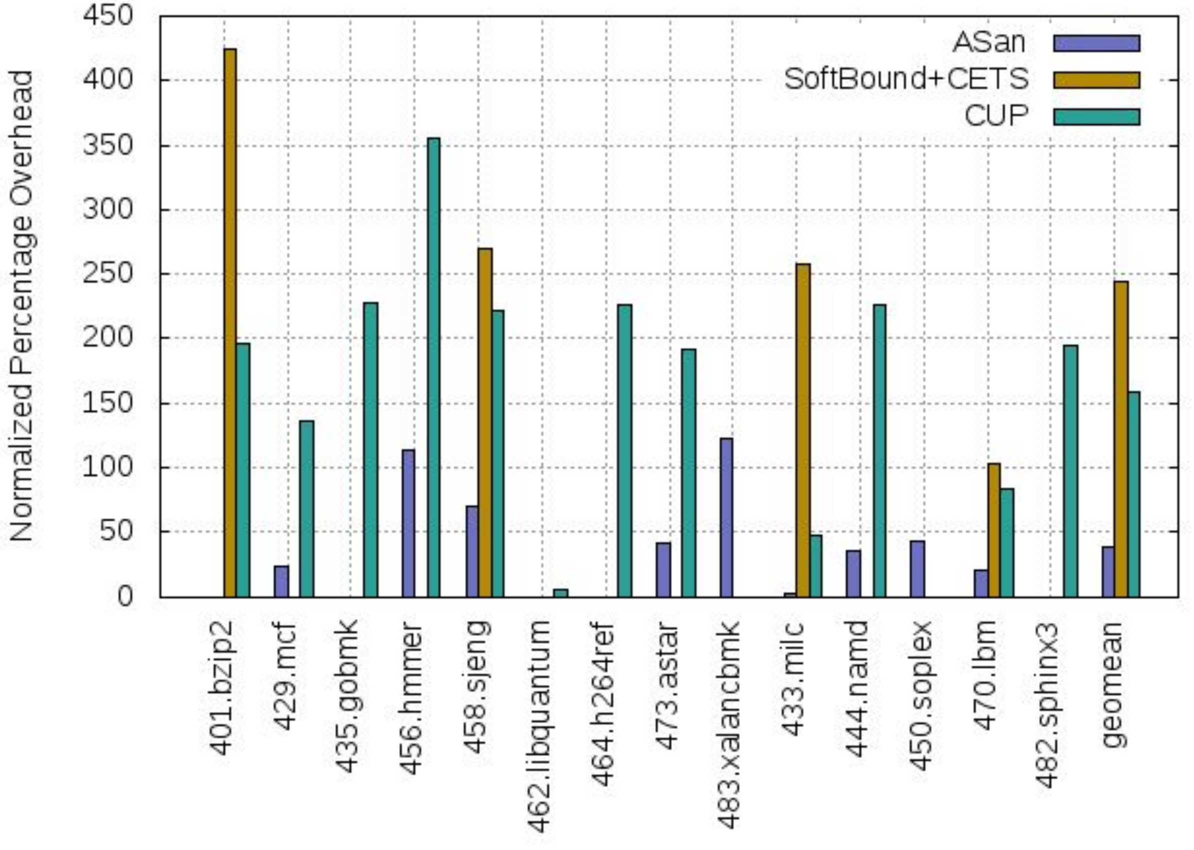
Evaluation: Security

- NIST provides a test suite of all CWEs called Juliet
  - Use to validate the CUP implementation
  - No false negatives or false positives
- False Positives
  - Implementation bug in SoftBound fails to handle **`alloca()`** calls correctly
- False Negatives
  - Primarily due to libc functions, e.g., **`strcpy`** or **`memcpy`** not being protected
  - Neither SoftBound nor AddressSanitizer fail closed
  - Cannot guarantee that all memory safety violations are caught

	False Negatives	False Positives
SoftBound+CETS	1032 (25%)	12 (0.3%)
AddressSanitizer	315 (8%)	0 (0%)
CUP	0 (0%)	0 (0%)

Evaluation: Performance on SPEC CPU2006

- 158% vs 38% for ASan
- 126% vs 245% for SoftBound on benchmarks where both run



Conclusion

- CUP presents Hybrid Metadata
  - Faster than SoftBound’s disjoint metadata
  - Supports temporal safety by allowing object aware metadata
- Fails Closed
  - No False Negatives on Juliet
  - Design validates implementation
- Performant Memory Safety remains a hard problem

https://github.com/HexHive/CUP

Questions?