HexSafe: Towards Precisely
Detecting Memory Corruption Vulnerabilities

Mathias Payer

March 26, 2017

Despite extensive research into memory safety techniques (see [4] for a survey), exploits of memory corruptions remain omnipresent [5]. Systems languages like C/C++ require the programmer to manually guarantee spatial safety (bounds checks) and temporal safety (lifetime checks). As the past decade of security vulnerabilities shows, these checks are frequently inadequate. Such vulnerabilities in browsers and servers allow attackers to illicitly gain arbitrary code execution capabilities on remote systems.

Existing work [2, 1, 3] does not provide full spatial and temporal checks for all program data: heap, stack, and globals. Further, they do not protect critical system libraries such as libc. The libc is particularly prone to memory errors, notably the mem* and str* family of functions (e.g., memcpy or strcpy). A sanitizer that plugs these holes is needed so developers can find and fix all memory corruptions triggered by their test inputs.

We propose to develop a sanitizer (a tool used during software development, testing, and debugging) to locate and report memory safety vulnerabilities in the source code. Existing tools lack in precision and completeness.

Research Questions To provide spatial and temporal checks, sanitizers must maintain metadata about all memory objects (e.g., size, location, validity). Scaling to all allocations in a program requires a large amount of additional metadata to protect these extra allocations, particularly those on the stack. No known metadata scheme scales to support all program data, including data in libraries. Further, to protect long lived programs such as servers, the metadata scheme must support an infinite number of allocations and deallocations.

Scaling the metadata scheme to support large numbers of allocations is not sufficient, the metadata scheme must also be efficient. Existing sanitizers used by software developers during testing, such as AddressSanitizer [3] from Google, report up to 3x overhead on SPEC CPU2006, the standard compiler benchmarks. Any new tool must be at least this efficient, despite supporting more checks.

Approach We propose HexSafe, a compiler pass and runtime library, which provides precise, complete spatial memory safety and stochastic temporal memory safety by protecting all program data, including libc (and all other libraries). Safety is enforced, for all program data, by dynamically maintaining information about all
objects that are vulnerable to memory safety errors. Our instrumentation ensures that each reference must point to a valid object before it is dereferenced. This information is recorded through our novel metadata scheme. A compiler-based instrumentation pass is used to add code that records and checks metadata at runtime. HexSafe is intended to be used in a testing environment to help developers identify and fix bugs.

Our metadata scheme embeds a capability ID in a pointer without changing its bit width. A capability ID ties a pointer to the metadata for its underlying object. This metadata contains exact bounds for every object. Full temporal safety requires having a unique metadata entry for each capability ID. As an implementation trade-off we opt for stochastic memory safety by reusing ID’s probabilistically to scale to long running programs (otherwise the storage required for metadata would grow indefinitely for long running programs).

Our compiler pass inserts instrumentation at allocation sites, and at pointer dereferences. At allocation sites, we assign the object a capability ID and record its bounds information. Bounds checks are added to dereferences, and will almost always fail if the object has been free’d and the capability ID reused. For this to scale, our compiler pass performs static analysis to limit the set of stack allocations that must be instrumented. Further, we leverage features of the x86.64 architecture to optimize our bounds check.

**Preliminary Results**  We have developed a simplified, preliminary prototype to test the feasibility of our proposed metadata scheme on top of the LLVM compiler framework. Our preliminary prototype supports C programs on x86.64. Based on the NIST Juliet benchmark suite, we have no false positives (i.e., no spurious memory safety errors are reported) and 0.1% false negatives (i.e., a memory safety error remains uncaught). Related tools have both (i) some false positives, and (ii) significant false negatives (25% for SoftBound and 8% for AddressSanitizer). Consequently, we conclude that our approach is feasible. Further research is needed to enable efficient metadata management and to reduce the performance overhead.

**Contributions**  HexSafe will be open sourced upon publication – available to the entire community to help find and fix memory safety bugs. Further, we plan to use it in combination with fuzzing tools to find and patch vulnerabilities in common libraries that are susceptible to these types of bugs, such as libjpeg and openssl.

We will also release our suite of internal benchmarks created during development. These test a sanitizer’s support for common C idioms, and reflect our experience with coding styles that are difficult to support. Our benchmarks contain both memory safe programs, and programs with memory corruptions.

**Budget and Personnel**  We request support for 1 graduate RA student at 0.5 FTE AY for this project and $2,500 for travel support (flight, transfer, hotel, and conference registration) to present HexSafe, the corresponding research paper, and the developed prototype at an academic conference. We have identified Nathan Burow as the lead graduate student for this project.
References


