Custom Memory Allocator Project

System Requirements

Target Platform

Attribute	Requirement
Architecture	x86-64
Operating System	Linux (kernel ≥ 5.4)
Distribution	Ubuntu 22.04 LTS or later
Compiler	GCC ≥ 11.0, Clang ≥ 14.0
C Standard	C17 (ISO/IEC 9899:2018)
Build System	GNU Make or CMake ≥ 3.20

Hardware Assumptions

Attribute	Assumption
Page size	4096 bytes (standard pages)
Huge page size	2 MiB (optional support)
Pointer size	8 bytes
Cache line size	64 bytes
`max_align_t` alignment	16 bytes

Development Environment

Required tools:

- GDB ≥ 10.0
- Valgrind ≥ 3.18
- AddressSanitizer (via compiler)
- perf (Linux profiling)

Recommended tools:

- pwndbg or GEF (GDB extensions)
- heaptrack (heap profiler)
- Compiler Explorer (godbolt.org)

Interface Specification

Public Interface

All implementations must expose a single registration structure. No creative interpretation allowed.

Header File

```
// allocator.h
#ifndef ALLOCATOR_H
#define ALLOCATOR_H

#include <stddef.h>
#include <stdbool.h>
#include <stdint.h>
```

```c
#ifdef __cplusplus
extern "C" {
#endif

typedef struct allocator_features {
    bool thread_safe;
    bool per_thread_cache;
    bool huge_page_support;
    bool guard_pages;
    bool canaries;
    bool quarantine;
    bool zero_on_free;
    size_t min_alignment;
    size_t max_alignment;
} allocator_features_t;

typedef struct allocator_stats {
    size_t bytes_allocated;
    size_t bytes_in_use;
    size_t bytes_metadata;
    size_t bytes_mapped;
    uint64_t alloc_count;
    uint64_t free_count;
    uint64_t realloc_count;
    uint64_t mmap_count;
    uint64_t munmap_count;
} allocator_stats_t;

typedef struct allocator {
    void* (*malloc)(size_t size);
    void (*free)(void* ptr);
    void* (*realloc)(void* ptr, size_t size);
    void* (*calloc)(size_t nmemb, size_t size);
    void* (*memalign)(size_t alignment, size_t size);
    void* (*aligned_alloc)(size_t alignment, size_t size);
    size_t (*usable_size)(void* ptr);
    void (*free_sized)(void* ptr, size_t size);
    void* (*realloc_array)(void* ptr, size_t nmemb, size_t size);
    void (*bulk_free)(void** ptrs, size_t count);
    void (*print_stats)(void);
    bool (*validate_heap)(void);
    bool (*get_stats)(allocator_stats_t* stats);
    int (*init)(void);
    void (*teardown)(void);

    const char* name;
    const char* author;
    const char* version;
    const char* description;
    const char* memory_backend;

    allocator_features_t features;
} allocator_t;

#define ALLOC_HAS(a, fn) ((a)->fn != NULL)

#define ALLOC_VERSION_MAJOR 1
#define ALLOC_VERSION_MINOR 0
#define ALLOC_VERSION_PATCH 0

#ifdef __cplusplus
}
#endif

#endif
```

```
#endif
```

Registration

Each implementation should export a single global `allocator_t` instance with external linkage.

```
// example_allocator.c
allocator_t myalloc_allocator = {
    .malloc         = myalloc_malloc,
    .free           = myalloc_free,
    .realloc        = myalloc_realloc,
    .calloc         = myalloc_calloc,
    .memalign       = NULL,
    .aligned_alloc  = NULL,
    .usable_size    = myalloc_usable_size,
    .free_sized     = NULL,
    .realloc_array  = NULL,
    .bulk_free      = NULL,
    .print_stats    = myalloc_print_stats,
    .validate_heap  = myalloc_validate,
    .get_stats      = myalloc_get_stats,
    .init           = myalloc_init,
    .teardown       = myalloc_teardown,
    .name           = "dihalloc",
    .author         = "tanush mumdani",
    .version        = "6.7.6.7",
    .description    = "something that makes you sound smart",
    .memory_backend = "mmap-only",
    .features = {
        .thread_safe        = false,
        .per_thread_cache   = false,
        .huge_page_support  = false,
        .guard_pages        = false,
        .canaries           = false,
        .quarantine         = false,
        .zero_on_free       = false,
        .min_alignment      = 16,
        .max_alignment      = 4096,
    },
};
```

Functional Requirements

malloc

ID	Requirement
FR-MALLOC-001	`malloc(size)` returns a pointer to at least `size` bytes
FR-MALLOC-002	`malloc(size)` returns NULL on failure
FR-MALLOC-003	`malloc(size)` does not initialize returned memory
FR-MALLOC-004	`malloc(0)` returns NULL or a unique freeable pointer (document your choice)
FR-MALLOC-005	Returned pointer is aligned to at least 16 bytes
FR-MALLOC-006	Successive calls don't return overlapping regions unless freed

free

ID	Requirement
FR-FREE-001	`free(ptr)` releases memory from malloc/calloc/realloc
FR-FREE-002	`free(NULL)` is a no-op
FR-FREE-003	`free(ptr)` on invalid pointer is undefined behavior
FR-FREE-004	`free(ptr)` on already-freed pointer is undefined behavior
FR-FREE-005	Freed memory may be returned to OS or retained

realloc

ID	Requirement
FR-REALLOC-001	`realloc(ptr, size)` resizes allocation to at least `size` bytes
FR-REALLOC-002	`realloc(ptr, size)` preserves contents up to `min(old_size, size)`
FR-REALLOC-003	`realloc(NULL, size)` is equivalent to `malloc(size)`
FR-REALLOC-004	`realloc(ptr, 0)` is equivalent to `free(ptr)`, returns NULL
FR-REALLOC-005	`realloc` may return same pointer or different pointer
FR-REALLOC-006	If realloc returns different pointer, original is freed
FR-REALLOC-007	If realloc fails, original allocation stays valid

calloc

ID	Requirement
FR-CALLOC-001	`calloc(nmemb, size)` allocates `nmemb * size` bytes
FR-CALLOC-002	`calloc(nmemb, size)` returns zero-initialized memory
FR-CALLOC-003	`calloc(nmemb, size)` returns NULL on overflow
FR-CALLOC-004	`calloc(nmemb, size)` returns NULL on allocation failure
FR-CALLOC-005	Overflow detection uses safe multiplication

memalign (Optional)

ID	Requirement
FR-MEMALIGN-001	`memalign(alignment, size)` returns memory aligned to `alignment`
FR-MEMALIGN-002	`alignment` must be a power of two
FR-MEMALIGN-003	Invalid alignment behavior is implementation-defined
FR-MEMALIGN-004	Returned pointer is freeable via `free()`

aligned_alloc (Optional)

ID	Requirement

ID	Requirement
FR-ALIGNED-001	`aligned_alloc(alignment, size)` conforms to C11 semantics
FR-ALIGNED-002	`size` must be a multiple of `alignment`
FR-ALIGNED-003	If `size` isn't a multiple of `alignment`, behavior is undefined

usable_size (Optional)

ID	Requirement
FR-USABLE-001	`usable_size(ptr)` returns actual usable size
FR-USABLE-002	Returned value is ≥ originally requested size
FR-USABLE-003	Writing up to `usable_size(ptr)` bytes is valid

Lifecycle

ID	Requirement
FR-INIT-001	`init()` is called exactly once before any allocation
FR-INIT-002	`init()` returns 0 on success, non-zero on failure
FR-INIT-003	After failed `init()`, no other functions may be called
FR-TEARDOWN-001	`teardown()` releases all OS resources
FR-TEARDOWN-002	`teardown()` should only be called after all allocations are freed
FR-TEARDOWN-003	After `teardown()`, no other functions may be called

Size Constraints

ID	Requirement
FR-SIZE-001	Your allocator should support 1 byte to at least 2 GiB
FR-SIZE-002	Your allocator should support up to available virtual memory
FR-SIZE-003	Allocations near SIZE_MAX return NULL

Non-Functional Requirements

Performance

ID	Requirement	Target
NFR-PERF-001	Small allocation (≤256B) latency	p99 < 1μs
NFR-PERF-002	Small allocation throughput	> 1M ops/sec
NFR-PERF-003	Large allocation (≥1MB) latency	p99 < 100μs
NFR-PERF-004	Memory overhead ratio	< 1.25x requested
NFR-PERF-005	Metadata overhead	< 5% of allocated

ID	Requirement	memory Target

Reliability

ID	Requirement
NFR-REL-001	Your allocator doesn't crash on valid inputs
NFR-REL-002	Your allocator doesn't leak memory during normal operation
NFR-REL-003	Your allocator handles allocation failure gracefully
NFR-REL-004	`validate_heap()` (if implemented) detects corruption

Portability

ID	Requirement
NFR-PORT-001	Code compiles with both GCC and Clang
NFR-PORT-002	Code doesn't use compiler-specific extensions without fallbacks
NFR-PORT-003	Code doesn't use inline assembly
NFR-PORT-004	Code assumes only POSIX-compliant system interfaces

Maintainability

ID	Requirement
NFR-MAINT-001	Code compiles with `-Wall -Wextra -Wpedantic` without warnings
NFR-MAINT-002	Functions don't exceed 100 lines
NFR-MAINT-003	All public structures and functions are documented
NFR-MAINT-004	Magic numbers are defined as named constants

Implementation Constraints

Allowed System Interfaces

Memory Management

Function	Header	Usage
`mmap`	`<sys/mman.h>`	Obtain memory from OS
`munmap`	`<sys/mman.h>`	Return memory to OS
`mremap`	`<sys/mman.h>`	Resize mappings (optional)
`mprotect`	`<sys/mman.h>`	Guard pages (optional)
`madvise`	`<sys/mman.h>`	Memory hints (optional)
`sbrk`	`<unistd.h>`	Program break (optional)

Function	Header	Usage
`brk`	`<unistd.h>`	Program break (optional)

Allowed mmap Flags

Flag	Usage
`MAP_PRIVATE`	Required
`MAP_ANONYMOUS`	Required
`MAP_HUGETLB`	Optional (huge pages)
`MAP_NORESERVE`	Optional (overcommit)
`MAP_POPULATE`	Optional (prefault)

Utility Functions

Function	Header	Usage
`memcpy`	`<string.h>`	Copy memory
`memset`	`<string.h>`	Fill memory
`memmove`	`<string.h>`	Copy overlapping memory
`sysconf`	`<unistd.h>`	Query page size
`getpagesize`	`<unistd.h>`	Query page size

Threading (Optional)

Function	Header	Usage
`pthread_mutex_*`	`<pthread.h>`	Mutual exclusion
`pthread_key_*`	`<pthread.h>`	Thread-local storage
Atomics	`<stdatomic.h>`	Lock-free operations

Prohibited Interfaces

Prohibition	Rationale
`malloc`, `free`, `realloc`, `calloc`	Defeats project purpose
`posix_memalign`, `aligned_alloc`	Libc allocation
`strdup`, `strndup`	Implicit malloc
`asprintf`, `vasprintf`	Implicit malloc
External allocator libraries	Must be original work
Inline assembly	Portability requirement

Compilation Requirements

Required Flags

```
CFLAGS := -std=c17 -Wall -Wextra -Wpedantic -Werror
CFLAGS += -fno-strict-aliasing
CFLAGS += -D_GNU_SOURCE
```

Debug Build

```
CFLAGS_DEBUG := $(CFLAGS) -O0 -g3 -fsanitize=address,undefined
```

Release Build

```
CFLAGS_RELEASE := $(CFLAGS) -O2 -DNDEBUG
```

Benchmark Build

```
CFLAGS_BENCH := $(CFLAGS) -O3 -march=native -DNDEBUG
```

Testing Requirements

Test Categories

Category	Purpose	Pass Criteria
Correctness	Verify functional requirements	All assertions pass
Stress	Verify reliability under load	No crashes, no leaks
Edge Cases	Verify boundary conditions	Graceful handling
Fragmentation	Verify memory efficiency	Allocations succeed
Regression	Verify bug fixes	Previously failed tests pass

Correctness Tests

Basic Operations

Test ID	Description	Validates
TC-BASIC-001	malloc single allocation	FR-MALLOC-001
TC-BASIC-002	free single allocation	FR-FREE-001
TC-BASIC-003	malloc/free cycle	FR-MALLOC-001, FR-FREE-001
TC-BASIC-004	free(NULL)	FR-FREE-002
TC-BASIC-005	malloc(0) behavior	FR-MALLOC-004

Size Handling

Test ID	Description	Validates
TC-SIZE-001	Allocation sizes 1-256 bytes	FR-MALLOC-001
TC-SIZE-002	Allocation sizes 256B-64KB	FR-MALLOC-001
TC-SIZE-003	Allocation sizes 64KB-16MB	FR-MALLOC-001

Test ID	Description	Validates
TC-SIZE-004	Allocation sizes 16MB-256MB	FR-SIZE-001
TC-SIZE-005	Allocation near SIZE_MAX	FR-SIZE-003

Alignment

Test ID	Description	Validates
TC-ALIGN-001	16-byte alignment for all sizes	FR-MALLOC-005
TC-ALIGN-002	memalign power-of-2 alignments	FR-MEMALIGN-001
TC-ALIGN-003	memalign large alignments (4KB)	FR-MEMALIGN-001

realloc

Test ID	Description	Validates
TC-REALLOC-001	Grow allocation	FR-REALLOC-001, FR-REALLOC-002
TC-REALLOC-002	Shrink allocation	FR-REALLOC-001, FR-REALLOC-002
TC-REALLOC-003	realloc(NULL, n)	FR-REALLOC-003
TC-REALLOC-004	realloc(p, 0)	FR-REALLOC-004
TC-REALLOC-005	realloc same size	FR-REALLOC-001
TC-REALLOC-006	Contents preserved on grow	FR-REALLOC-002
TC-REALLOC-007	Contents preserved on shrink	FR-REALLOC-002

calloc

Test ID	Description	Validates
TC-CALLOC-001	Zero initialization	FR-CALLOC-002
TC-CALLOC-002	Overflow: SIZE_MAX × 2	FR-CALLOC-003
TC-CALLOC-003	Overflow: (SIZE_MAX/2+2) × 2	FR-CALLOC-003
TC-CALLOC-004	Large array allocation	FR-CALLOC-001

usable_size

Test ID	Description	Validates
TC-USABLE-001	usable_size ≥ requested	FR-USABLE-002
TC-USABLE-002	Write to full usable size	FR-USABLE-003

Stress Tests

Test ID	Description	Operations

Test ID	Description	Operations
TC-STRESS-001	Random malloc/free	1,000,000
TC-STRESS-002	LIFO pattern	1,000,000
TC-STRESS-003	FIFO pattern	1,000,000
TC-STRESS-004	Realloc chains	100,000
TC-STRESS-005	Peak memory cycling	100 cycles
TC-STRESS-006	Many simultaneous allocations	100,000 live

Edge Case Tests

Test ID	Description	Expected Behavior
TC-EDGE-001	malloc(SIZE_MAX)	Return NULL
TC-EDGE-002	malloc(SIZE_MAX - 4096)	Return NULL
TC-EDGE-003	100,000 × 1-byte allocations	All succeed
TC-EDGE-004	Allocations near page boundaries	Correct alignment
TC-EDGE-005	Rapid init/teardown cycles	No leaks

Fragmentation Tests

Test ID	Description	Success Criteria
TC-FRAG-001	Swiss cheese pattern	Large allocation succeeds
TC-FRAG-002	Sawtooth pattern	Memory returns to baseline
TC-FRAG-003	Size class thrashing	No excessive RSS growth
TC-FRAG-004	Long-running simulation	RSS bounded

Test Infrastructure

Will be determined at a later time, will post on discord notifying testing suite.

Benchmarking Requirements

Metrics

Metric	Unit	Collection Method
Throughput	ops/sec	Total operations / elapsed time
Latency (p50)	ns	50th percentile of samples
Latency (p99)	ns	99th percentile of samples
Latency (p999)	ns	99.9th percentile of samples
Latency (max)	ns	Maximum observed

Metric	Unit	Collection Method
RSS	bytes	`/proc/self/statm`
Overhead ratio	ratio	RSS / Σ(requested bytes)
Fragmentation	ratio	(RSS - in_use) / RSS

Synthetic Workloads

Workload ID	Description	Size Distribution	Pattern	Iterations
WL-SYN-001	Small fixed	64B	Immediate free	10M
WL-SYN-002	Small random	16-256B uniform	Immediate free	10M
WL-SYN-003	Medium fixed	4KB	Immediate free	1M
WL-SYN-004	Medium random	1-64KB uniform	Immediate free	1M
WL-SYN-005	Large fixed	1MB	Immediate free	100K
WL-SYN-006	Large random	64KB-4MB uniform	Immediate free	100K
WL-SYN-007	Mixed sizes	Power-law	Batch free (1000)	10M
WL-SYN-008	Realloc grow	16B → 4KB	Chain	1M
WL-SYN-009	Realloc shrink	4KB → 16B	Chain	1M
WL-SYN-010	Calloc	16-4KB	Immediate free	1M

Realistic Workloads

Workload ID	Source	Description
WL-REAL-001	Redis	YCSB workload trace
WL-REAL-002	SQLite	TPC-C trace
WL-REAL-003	Firefox	Page load trace
WL-REAL-004	Custom	Application-specific

Measurement Protocol

Environment Setup

```
# Pin to CPU 0
taskset -c 0 ./benchmark

# Disable ASLR
setarch $(uname -m) -R ./benchmark

# Disable turbo boost (optional)
echo 1 | sudo tee /sys/devices/system/cpu/intel_pstate/no_turbo

# Drop caches
sync && echo 3 | sudo tee /proc/sys/vm/drop_caches
```

Timing

- Use `clock_gettime(CLOCK_MONOTONIC)` for wall-clock time

- Warm up with 10,000 operations before measurement
- Sample latency every N operations to avoid overhead

Statistical Requirements

- Run each workload 10 times
- Report: min, max, median, mean, standard deviation
- Discard first run as warmup
- Use median for comparisons

Reporting Format

Summary Table

Allocator	Workload	Throughput	p50	p99	p999	RSS
myalloc	WL-SYN-001	5.2M	42ns	180ns	1.2μs	48MB
...	...	...	...	...	...	...

Required Graphs

1. Throughput vs allocation size (log-log)
2. Latency distribution (CDF)
3. RSS over time
4. Scalability (throughput vs thread count, if applicable)

Benchmarking Infrastructure

Will be determined at a later time, will post on discord notifying benchmarking suite.

Security Analysis Requirements

Objective

Break allocator and prove you are very smart

Exploit Primitives

Primitive	Severity	Description
Arbitrary Write	Critical	Write anywhere, do anything
Arbitrary Read	High	Read the forbidden memory
Code Execution	Critical	Be root basically
Overlapping Chunks	High	Schrödinger's allocation (two pointers, one chunk)
Double Allocation	High	malloc() said "sure, twice"
Use-After-Free (R)	Medium	Reading ghost data
Use-After-Free (W)	Medium	Haunting freed memory
Heap Overflow	Medium	Coloring outside the lines
Information Leak	Low	bypass ASLR

Vulnerability Classes

Metadata Corruption

(Lying to the allocator about its own bookkeeping)

- Size field corruption (gaslighting the chunk)
- Free list pointer corruption (redirecting traffic)

- Chunk header/footer corruption (identity theft)
- Bitmap corruption (flipping bits, flipping tables)

Unlink Attacks

(Classic hits from the 2000s)

- Classic unlink (fd/bk manipulation)
- Fastbin/tcache poisoning variants (modern)
- House of * techniques (the extended universe)

Implementation Bugs

- Integer overflow in size calculations (math is hard)
- Off-by-one errors (fencepost)
- Missing bounds checks
- Race conditions (if thread-safe)

Exploit Requirements

Code

- Self-contained exploit in `exploits/<target>_<author>/exploit.c`
- Must compile with project Makefile (no "works on my machine")
- Must run deterministically (no yolo heap spray)
- Must demonstrate primitive clearly (show your work)

Writeup

Each exploit needs a writeup so we know you didn't just get lucky:

1. **Vulnerability**: What broken
2. **Root Cause**: Why broken (skill issue or design flaw?)
3. **Exploitation**: How you abused it (step-by-step)
4. **Heap Layout**
5. **Primitive**: What can do
6. **Constraints**: Why you can't RCE
7. **Mitigations**: How the victim should've defended themselves

Rules

Rule	Description
No self-exploitation	Can't pwn yourself
Deterministic	Works every time or it doesn't count
Compiled as-is	Must survive `-O2`
Allocator bugs only	Don't exploit the test harness, that's cheating
Coordinated disclosure	Tell them before you embarrass them

Bonus Objectives

(For the overachievers)

Objective	Description
Primitive chaining	Combo your exploits
Hardening bypass	Report niche ways to bypass their security features

Deliverables

Week 3: Implementation Complete

Deliverable	Location	Requirements
Source code	`allocators/<author>/`	Compiles without warnings
README	`allocators/<author>/README.md`	Design documentation
Tests passing	CI/local	All correctness tests

README Contents

1. Architecture overview with diagrams
2. Memory sourcing strategy
3. Size class design (if applicable)
4. Free list / allocation strategy
5. Metadata layout with struct definitions
6. Key algorithms (allocate, free, coalesce, split)
7. Design tradeoffs and rationale
8. Known limitations
9. References to inspiring allocators

Week 4: Testing Complete

Deliverable	Location	Requirements
Stress tests	CI/local	All pass without leaks
Benchmark results	`docs/BENCHMARKS.md`	All workloads
Comparative analysis	`docs/BENCHMARKS.md`	Cross-allocator comparison

Benchmark Report Contents

1. Methodology description
2. Summary table (all allocators × all workloads)
3. Throughput vs size graph
4. Latency distribution graph
5. Memory efficiency comparison
6. Analysis and conclusions

Week 5: Project Complete

Deliverable	Location	Requirements
Exploit code	`exploits/<target>_<author>/exploit.c`	Compiles and runs
Exploit writeup	`exploits/<target>_<author>/writeup.md`	Complete documentation
Final presentation	Slides	10 minutes

Presentation Contents

1. Design overview (2 min)
2. Key implementation decisions (2 min)
3. Performance results (2 min)
4. Exploit walkthrough (3 min)
5. Lessons learned (1 min)

Project Timeline

Week	Phase	Milestones
1	Design	Architecture finalized, basic malloc/free working

Week	Phase	Milestones
2	Core	Size classes, realloc/calloc, free list management
3	Polish	All features complete, passes tests, documentation
4	Testing	Stress tests, benchmarks, comparative analysis
5	Security	Exploit development, writeups, final presentation

Detailed Week 1-3 Schedule

Day	Week 1	Week 2	Week 3
1-2	Research, draft architecture	Size class implementation	Optional features
3-4	mmap wrapper, basic malloc	realloc implementation	Performance tuning
5-6	Basic free, simple tests	calloc, edge cases	Final testing
7	Review, iterate	Run correctness tests	Documentation

Evaluation Criteria

Scoring Overview

Category	Base Points	Bonus Points	Penalty Points
Correctness	20	+5	-10
Performance	20	+10	—
Design	20	+5	—
Optional Features	10	+15	—
Security Analysis	20	+10	-15
Presentation	10	+5	—
Base Total	**100**	**+50**	**-25**

Final score can have bonus overflow.

Correctness (20 base)

Base Scoring

Points	Criteria
20	All required tests pass
15	≥95% tests pass, no critical failures
10	≥80% tests pass, core functions work
5	≥50% tests pass
0	<50% tests pass or doesn't compile

Points	Criteria

Bonus (+5 max)

Bonus	Criteria
+2	Zero memory leaks under Valgrind (all tests)
+2	Zero ASan/UBSan findings
+1	Passes additional self-authored edge case tests (submitted to shared suite)

Penalties (-10 max)

Penalty	Criteria
-5	Crash on valid input discovered during benchmarking or exploit phase
-5	Silent memory corruption discovered (not crash, but wrong results)

Performance (20 base)

Base Scoring

Allocators ranked on composite score across all workloads. Composite = weighted average of normalized metrics.

Metric	Weight
Throughput (ops/sec)	40%
Latency p99	30%
Memory overhead	30%

Normalization: best = 1.0, worst = 0.0, linear interpolation.

Points	Rank
20	1st place
17	2nd place
14	3rd place
11	4th place
8	5th place
5	6th place

Bonus (+10 max)

Bonus	Criteria
+3	Best throughput on WL-SYN-001 (small allocs)
+3	Best throughput on WL-SYN-007 (mixed workload)
+2	Lowest memory overhead across all workloads
+2	Lowest p99 latency across all workloads

Bonus	Criteria

Multiple allocators can earn the same bonus if tied within 5%.

Design (20 base)

Base Scoring

Points	Criteria
20	Clean architecture, comprehensive documentation, clear rationale
15	Good design, adequate documentation
10	Functional design, minimal documentation
5	Disorganized but functional
0	No coherent design or documentation

Evaluation Criteria

Aspect	Weight	Evaluation
Code organization	35%	File structure, modularity, separation of concerns
Readability	15%	Naming, formatting, comments
Documentation	5%	README completeness, diagrams, algorithm explanations
Rationale	45%	Justified tradeoffs, comparison to alternatives

Bonus (+5 max)

Bonus	Criteria
+2	Peer-voted "cleanest code" (anonymous vote)
+2	Peer-voted "best documentation"
+1	Includes benchmarks comparing design alternatives (e.g., different size class schemes)

Optional Features (10 base + 15 bonus)

Base Scoring

Points	Features Implemented
10	3+ optional features, all passing tests
7	2 optional features passing
4	1 optional feature passing
0	No optional features

Feature Point Values

Feature	Points	Test Requirement
`memalign`	2	TC-ALIGN-002, TC-ALIGN-003
`aligned_alloc`	1	C11 compliance test
`usable_size`	1	TC-USABLE-001, TC-USABLE-002
`free_sized`	1	Performance benchmark shows improvement
`realloc_array`	1	Overflow test
`bulk_free`	2	Performance benchmark shows improvement
`print_stats`	1	Produces valid output
`validate_heap`	2	Detects injected corruption
`get_stats`	1	Returns accurate statistics
Thread safety	5	Passes TC-THREAD-* tests
Huge page support	3	Benchmark shows RSS uses huge pages
Guard pages	3	Detects overflow in test
Canaries	2	Detects corruption in test
Quarantine	2	UAF test shows delayed reuse
Zero-on-free	1	Memory zeroed after free

Base points capped at 10. Additional points count as bonus.

Bonus Calculation

```
bonus = max(0, total_feature_points - 10)
```

Capped at +15.

Security Analysis (20 base)

Offensive Scoring (Your Exploits)

Points	Criteria
12	Working exploit demonstrating critical primitive
9	Working exploit demonstrating high primitive
6	Working exploit demonstrating medium primitive
3	Identified vulnerability, no working exploit
0	No security analysis

Primitive severity:

Severity	Primitives
Critical	Arbitrary write, code execution
High	Arbitrary read, overlapping chunks, double

Severity	allocation Primitives
Medium	UAF read/write, heap overflow
Low	Information leak

Writeup Quality

Points	Criteria
8	Comprehensive writeup with diagrams, mitigations, and clear explanation
6	Good writeup covering all required sections
4	Adequate writeup, missing some sections
2	Minimal writeup
0	No writeup

Offensive Bonus (+8 max)

Bonus	Criteria
+3	Multiple exploits against different allocators
+3	Exploit bypasses implemented hardening
+2	Exploit achieves code execution (not just arbitrary write)

Defensive Penalties (Exploits Found Against You)

Penalty	Criteria
-5	Critical primitive achieved against your allocator
-3	High primitive achieved against your allocator
-2	Medium primitive achieved against your allocator
-1	Low primitive achieved (info leak only)

Penalties stack but cap at -15.

Defensive Bonus (Hardening)

Bonus	Criteria
+2	No exploits found against your allocator
+1	Exploit found but required hardening bypass

Presentation (10 base)

Base Scoring

Points	Criteria
10	Clear, well-organized, confident delivery, handles Q&A

Points	Criteria
	well
7	Good presentation, minor issues
4	Adequate presentation, significant issues
0	Did not present

Evaluation Rubric

Aspect	Points	Criteria
Clarity	3	Audience understands design and implementation
Organization	2	Logical flow, good time management
Visuals	2	Effective diagrams, readable slides
Technical depth	2	Demonstrates understanding of tradeoffs
Q&A	1	Answers questions accurately

Bonus (+5 max)

Bonus	Criteria
+2	Live demo of exploit
+2	Peer-voted "best presentation"
+1	Includes performance comparison visualization

Final Rankings

Individual Ranking

Final score = base + bonus - penalties (capped at 0 minimum).

Rank by final score. Tiebreakers in order:

1. Security analysis score
2. Performance rank
3. Correctness score
4. Alphabetical by name

Allocator Rankings

Separate rankings published for:

Ranking	Metric
Overall	Final project score
Performance	Benchmark composite score
Efficiency	Memory overhead (lower is better)
Security	Defensive score (fewest/least severe exploits)
Features	Optional feature points

Awards

Award	Criteria
Best Overall	Highest final score
Best Speed	Best benchmark composite (Blazingly Fast AI B2B SaaS)
Least Memory Utilized	Lowest memory overhead
Best Security	Most secure (fewest exploits, or required hardest bypass)
Best Exploits	Most/best exploits against others
Clean Code	Peer-voted best design/documentation
Best Presentation	Peer-voted
Most Schizo	Most Novel Ideas

Score Calculation Example

```
Participant: Alice
Allocator: alice_alloc

Correctness:
  Base: 20 (all tests pass)
  Bonus: +2 (zero Valgrind leaks), +2 (zero ASan)
  Penalty: 0
  Subtotal: 24

Performance:
  Rank: 2nd → Base: 17
  Bonus: +3 (best mixed workload)
  Subtotal: 20

Design:
  Base: 18
  Bonus: +2 (peer-voted cleanest code)
  Subtotal: 20

Optional Features:
  memalign (2) + usable_size (1) + validate_heap (2) + canaries (2) = 7
  Base: 7
  Bonus: 0
  Subtotal: 7

Security Analysis:
  Offensive: 12 (arbitrary write exploit)
  Writeup: 8 (comprehensive)
  Offensive bonus: +2 (code execution)
  Defensive penalty: -3 (high primitive found against alice_alloc)
  Subtotal: 19

Presentation:
  Base: 9
  Bonus: +2 (live demo)
  Subtotal: 11

TOTAL: 24 + 20 + 20 + 7 + 19 + 11 = 101
```

Exploitation Resources

(For those who haven't seen heap internals)

Resource	URL
how2heap	github.com/shellphish/how2heap (start here if heap is black magic to you)
Heap Exploitation	heap-exploitation.dhavalkapil.com (actual explanations that make sense)
glibc malloc internals	sourceware.org/glibc/wiki/MallocInternals (do not)

Sidenote

AI is encouraged if it helps you learn. But let's be real: if you ask it to write your allocator, you're going to spend week 5 watching someone pop a shell on your "secure" implementation. AI is great at writing code that looks right. Allocators need code that actually is right.

Addendum: Non-C Implementations

If you decide to write your allocator in a language other than C (Rust, Zig, Carbon, etc.), no matter what you're responsible for the following:

1. **FFI Compatibility**: Your allocator must expose a C-compatible interface that conforms to the `allocator_t` struct. No exceptions. Your `unsafe` blocks are your problem. Also, if you can't figure out how to export a function pointer, maybe stick with C.
2. **Test Backporting**: You must backport the entire test suite to work with your implementation. The shared test harness won't be modified to accommodate your language choice. That's a you problem.
3. **Test Verification**: Your backported tests must be reviewed and approved to confirm they aren't fake or watered-down versions of the originals. No, you may not fake tests because some obscure language feature is blocking your function.
4. **Benchmark Fairness**: Your allocator will be benchmarked using the same harness as everyone else. If FFI overhead tanks your performance, that's on you. "But it's zero-cost abstractions" does not matter here.
5. **Exploit Compatibility**: Your allocator must be exploitable using standard C tooling. If your language's safety features make exploitation impossible through normal means, you forfeit defensive bonus points. Borrow checker doesn't get a trophy.
6. **Documentation**: Your README must include a section explaining why you chose a different language and how your FFI layer works. I do not want reasons for why you decided to code in Python to be "it was funny."
7. **No Excuses**: If something doesn't work because of language interop issues, that's a bug in your implementation, not a problem with the spec. Cope.