Comprehensive Guide to GLIBC Heap Exploitation

This guide clarifies the fundamental structures and advanced exploitation techniques targeting the GNU C Library (GLIBC) `malloc` implementation. Understanding these internals is key to analyzing memory corruption vulnerabilities.

I. GLIBC Heap Architecture (Chunks and Arenas)

The `malloc_chunk` Structure

Every memory block on the heap is managed by a **`malloc_chunk`** header that precedes the user-accessible memory. When a chunk is **free**, its user data area is repurposed to hold the forward and backward pointers for the linked lists (bins).

Field	Offset (64-bit)	Purpose	Notes
`prev_size`	`0x00`	Size of the physically preceding chunk.	Only used if the preceding chunk is **free**.
`size`	`0x08`	Total size of the current chunk.	The lower three bits contain **flags** (P, M, A).
`fd` (Forward Pointer)	`0x10`	Points to the next free chunk in the bin.	Used only when the chunk is **free**.

Field	Offset (64-bit)	Purpose	Notes
`bk` (Backward Pointer)	`0x18`	Points to the previous free chunk in the bin.	Used in **doubly linked** lists (Small/Large Bins).

Size Flags

The **`size`** field flags are crucial for consolidation logic:

- **P (PREV_INUSE):** If set, the preceding chunk is in use. If clear, the preceding chunk is free and can be consolidated with the current one upon freeing.
- **M (IS_MMAPPED):** Indicates the chunk was allocated using `mmap`.
- **A (NON_MAIN_ARENA):** Indicates the chunk belongs to a thread-local arena.

Arenas (`malloc_state`) and Bins

The **Arena** (represented by the `malloc_state` structure) is the central metadata repository that manages the heap memory pool, especially in multi-threaded programs.

Key structures within the Arena include:

- **The Top Chunk:** The large, unallocated chunk at the end of the heap. New memory is carved from here when no suitable free chunk is found.
- **Bins:** Arrays of pointers that manage free chunks based on size.
    - **Tcache:** (GLIBC 2.26+) A **singly linked list** for small chunks, prioritized for speed. Most modern exploits target its single pointer.
    - **Fastbins:** Legacy **singly linked lists** for small chunks. Chunks in Fastbins are **not consolidated** on free, making them easy to exploit.
    - **Smallbins / Largebins / Unsorted Bin:** **Doubly linked lists** for larger chunks. These require two pointers (`fd` and `bk`) for integrity.

II. Classic Heap Exploitation Techniques

These techniques target vulnerabilities like buffer overflows or double frees to corrupt the integrity of the heap's internal linked lists, often leveraging the predictable nature of early GLIBC versions.

1. The Unlink Exploit (Smallbins/Unsorted Bin)

This attack targets the function responsible for removing a chunk $P$ from a **doubly linked list** (Unsorted or Smallbins). The goal is to gain an **arbitrary memory write**.

The Vulnerability:

An attacker uses a heap overflow or other corruption primitive to overwrite the `fd` and `bk` pointers of a free chunk $P$ just before the allocator attempts to unlink it.

The Exploitation Primitive:

The attacker forges the pointers to point near a desired target address, $\mathbf{Target\_Address}$, such that when the unlinking macro executes its second write, the target address is overwritten.

$$\text{unlink}(P) \implies \begin{cases} (P \to \mathbf{fd}) \to \mathbf{bk} = P \to \mathbf{bk} \\ (P \to \mathbf{bk}) \to \mathbf{fd} = P \to \mathbf{fd} \end{cases}$$

By setting $\mathbf{P \to bk} = \mathbf{Target\_Address - 16}$ and $\mathbf{P \to fd} = \mathbf{Target\_Address - 24}$, the second write results in:

$$\mathbf{Memory\ at\ Target\_Address} = \mathbf{Target\_Address} - \mathbf{24}$$

This is a powerful primitive used to overwrite critical pointers like Global Offset Table (GOT) entries or hook functions.

2. Double Free and Tcache Poisoning (Modern)

Double Free occurs when `free()` is called twice on the same allocated memory address. This is dangerous because it inserts the same chunk into a singly linked list (Fastbin or Tcache) twice, allowing an attacker to manipulate the list's order and gain control of the next allocated address.

The resulting list corruption is used for **Tcache Poisoning** (most common today). By corrupting the `fd` pointer of the second instance of the freed chunk, the attacker can force the next `malloc` call to return an arbitrary address T, effectively giving them control over a forged chunk at T.

3. Forging Chunks (House of Spirit)

The **House of Spirit** attack is used to allocate a chunk at an arbitrary, controlled address. It requires two main steps:

1. **Forge a Fake Chunk:** The attacker creates a fake `malloc_chunk` header in a data buffer they control (e.g., on the stack or in global memory). The only critical field is the **`size`**, which must correspond to an available Fastbin size and must have the **P** flag set (meaning the preceding chunk is marked "in use") to prevent consolidation errors.
2. **Free the Fake Chunk:** The attacker finds a way to call `free()` with a pointer to the user data section of their fake chunk. This places the arbitrary address into a Fastbin.
3. **Allocate the Fake Chunk:** A subsequent `malloc()` call for that size will return the arbitrary address, giving the attacker control over that memory location.

4. House of Lore (Smallbins)

This technique targets **Smallbins** by corrupting the `bk` pointer of a chunk that is about to be placed into a bin. The goal is to divert the new chunk insertion into a controlled area (e.g., the stack) to place an attacker-controlled address into the arena's list of bins.

5. House of Force (Top Chunk)

This exploit is performed by overflowing the buffer immediately before the **Top Chunk**. The attacker corrupts the Top Chunk's **`size`** field to an extremely large value (e.g., `0xFFFFFFFF...`).

When the attacker requests a new chunk of memory, `malloc()` attempts to carve it from the corrupted Top Chunk. The enormous size allows the allocator to calculate a huge offset, essentially giving the attacker the ability to move the Top Chunk pointer to an **arbitrary memory address** within the process's address space. The next allocation will then return a pointer to that precise, arbitrary location.

6. House of Einherjar (Consolidation)

This advanced technique relies on performing a **consolidation** with a chunk that is structurally adjacent but logically separated. By corrupting the header of a neighboring allocated chunk, the attacker can make `free()` believe a chunk is free and consolidate it. This leads to placing a large, corrupt chunk into the Unsorted Bin, which can then be used to gain control over subsequent allocations.

III. Modern Mitigations and Defenses

GLIBC developers have continuously introduced mitigations to combat these classic techniques, shifting the focus of exploitation to newer structures like the Tcache.

Security Checks (Unlink Checks)

Modern GLIBC versions include runtime checks within the unlinking process for doubly linked lists (Small/Large Bins) to prevent the **Unlink Exploit**. Before performing the two pointer writes, the allocator verifies:

$$P \to \mathbf{fd} \to \mathbf{bk} == P \quad \text{and} \quad P \to \mathbf{bk} \to \mathbf{fd} == P$$

If these checks fail, indicating corruption, the program will terminate via an `abort()` call.

Safe Linking (GLIBC 2.32+)

This is the primary defense against **Tcache Poisoning**. Safe Linking **encodes** the `fd` and `bk` pointers of free chunks using an XOR operation:

$$\mathbf{Encoded\_Ptr} = \mathbf{Pointer} \oplus (\mathbf{Base\_Address} \gg 12)$$

This ties the integrity of the pointer to the address where it is stored. An attacker must now not only know the target address but must also leak the heap's base address (to obtain the $\mathbf{Base\_Address}$) to calculate the correct XOR-encoded value. Failure to provide a correctly encoded pointer results in a crash, acting as a canary for the heap's linked list integrity.

IV. Secure Coding Guidelines

Adhering to strict secure coding principles is the most effective defense against all heap-based attacks:

1. **Respect Boundaries:** Use only the exact amount of memory requested via `malloc`. **Never cross memory boundaries** (prevent buffer overflows).
2. **Strict Freeing:** **Free dynamically allocated memory exactly once** to prevent Double Free vulnerabilities.
3. **No Use-After-Free:** **Never access freed memory**. After calling `free()`, re-assign all associated pointers to `NULL` immediately.
4. **Validate Allocations:** Always check the return value of `malloc` for `NULL`.
5. **Zero Out Sensitive Data:** Before freeing memory that held sensitive data (e.g., passwords or encryption keys), use `memset` to **zero it out**.
6. **Avoid Assumptions:** Do not make assumptions about the relative positioning or layout of addresses returned by `malloc`, as the allocator's behavior is highly dynamic.