Montrehack | Heap Exploitation

September 16th 2020 - @alxbl_sec

Challenge Server

Challenge nc ctf.segfault.me 3000

Files http://ctf.segfault.me/

Template http://ctf.segfault.me/exploit.py

NOTE You will need libc.so for flag #3

For people who want a head start

Contents

- Heap Concepts
- Common Heap Bug Classes
- Heap Exploitation
- Hardening Techniques
- Hints :)

Heap Concepts

Concepts | Heap vs. Stack

The Heap

- One* heap per process
- Persistent
- No size limit (virtual memory)
- Many management strategies
- Good for long-lived data
- Good for resource sharing
- Bad* for fast allocations
- Bad* for temporary data

The Stack

- One per thread
- Volatile (thread / frame lifetime)
- Limited size
- More efficient
- Good for temporary data
- Good for small data
- Bad for long-lived objects
- Bad for resource sharing

* Actually depends on the allocator's algorithm & implementation

Concepts | Memory Management

- Allocator based (malloc / free, new / delete, new[], delete[])
- Usually handled by the common runtime libraries
  - GLIBC on most *NIX system
- Different allocators have different goals / benefits
  - High frequency, small size allocations
  - Low frequency, large size allocations
  - Optimized data locality for CPU bound processes (e.g. rendering)
  - ...
- Heap allocator is responsible for tracking allocated and free memory blocks
- Can be either fully contiguous or separated in several large chunks (growable heap)

Concepts | Memory Management (cont.)

Commonly Tracked Metadata

- **Free List** A linked list of blocks that can be allocated to
- **Allocation Size** This is necessary to know how much space to free
- **Allocator** Some heaps support custom allocators to help with debugging or otherwise

Concepts | Memory Fragmentation

- Similar to disk fragmentation
- Space between two allocations is too small to put anything useful there
- Leads to wasted space
- High fragmentation leads to high memory usage

Bug Classes

Bug Classes | Buffer Overflow

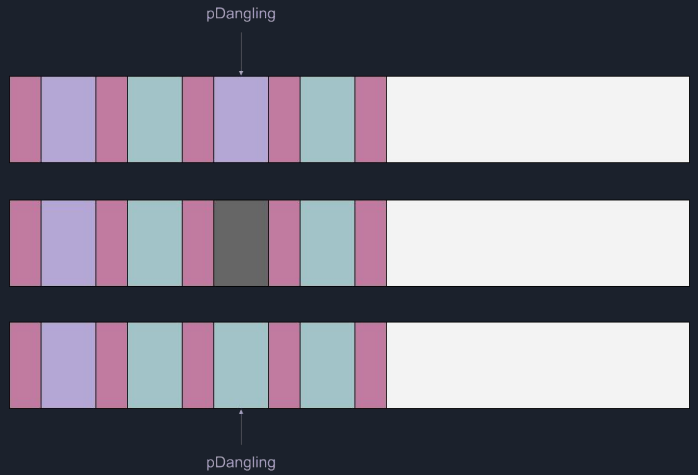
CAUSE: An unchecked memory copy operation that stores data past the available buffer space.

IMPACT: The heap metadata and allocation directly following the overflown buffer is corrupted.

Bug Classes | Use After Free

CAUSE: An object which is freed is later used by a different component that kept a (now invalid) pointer to the data.

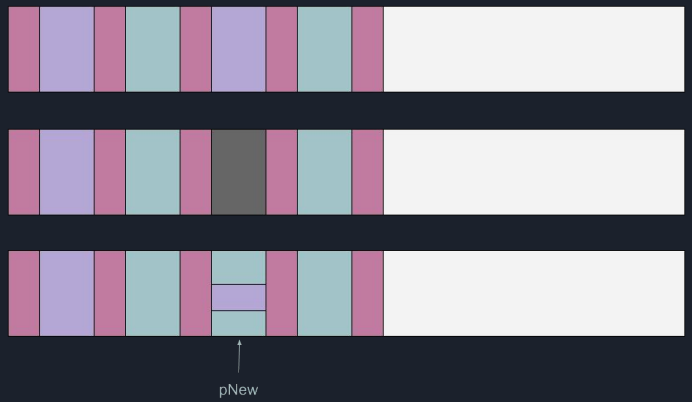
IMPACT: The data may have been overwritten by a different allocation and cause a crash or unexpected behavior.



Bug Classes | Uninitialized Memory

CAUSE: A newly allocated object in memory does not initialize one of its field before accessing it.

IMPACT: The value of the uninitialized field will be a leftover from whatever object was occupying the heap space prior to allocation.



Exploitation Techniques

Exploitation | General Flow

1. Identify a vulnerable heap object
2. Understand the allocator (Reverse Engineering, Code Review)
3. Identify an object that can be used as an arbitrary read primitive
4. Identify an object that can be used as an arbitrary write primitive (**Write-What-Where**)
5. Abuse the allocator to manipulate and predict the heap layout (**Heap Grooming**)
6. Corrupt the heap to achieve arbitrary read/write with objects from (3) and (4)
7. (Optional) Defeat ASLR and other mitigations if needed
8. Use your RW primitive to attack the binary (patch a function, PLT entry, etc.)
9. Submit flag and enjoy SOUCCESS

Exploitation | Arbitrary R/W Primitive

An object which has an API/fields that read or write memory. When corrupted, this object may be used to read/write out of bounds or at a specified location.

Usually any type of image/bitmap buffer header is an interesting candidate:

- obj->data
- obj->width
- obj->height
- get_pixel(w, h) = data[h*width + w]

Corrupt width and height => Arbitrary Read based at **data**

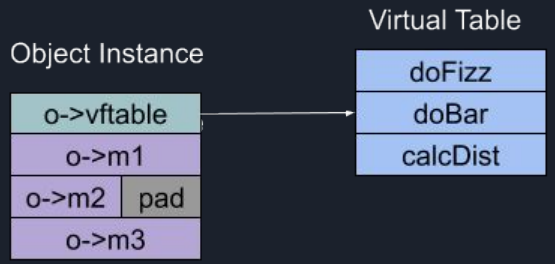
Exploitation | Heap Grooming

alloc and free objects of specific sizes in a specific sequence to predict with a high level of confidence the layout of the heap.

- Extremely heap/allocator-specific
- Object-specific (alloc sizes may matter depending on the scenario)
- Once predicted, it’s possible to line up vulnerable objects to exploit a bug class

Exploitation | Vtable Hijacking (C++)

- Object-Oriented Code uses Virtual Function Tables
- Type stores a pointer to a table with function addresses
- Corrupt Pointer => Fake vtable
- Can control instruction pointer when a function is called
- Non OOP => raw function pointers



Heap Hardening

Heap Hardening Techniques

- **Zeroing Allocator** memset new allocations with benign data
- **Heap Canaries** Validate that canaries are intact whenever performing bookkeeping
- **Guard Pages** Will cause a crash as soon as overflow is triggered (Memory heavy)
- **Allocation Randomization** Make the heap allocator non-deterministic*
- **Metadata Encoding** use a heap key to protect metadata from being corrupted

NOTE As with most mitigations, these can be bypassed with the right circumstances

Challenge Server

Challenge nc ctf.segfault.me 3000

Files http://ctf.segfault.me/

Template http://ctf.segfault.me/exploit.py

NOTE You will need libc.so for flag #3

For people who want a head start

Questions? Ask now or @alxbl_sec ;)

Code, slides and solutions will be posted on github.com/montrehack/challenges

Hints

- The vulnerability is in the **void create()** function

Hints

- The vulnerability is in the **void create()** function
- Overflow **canvas->title** to corrupt the FLAG 1 canvas's **private** field

Hints

- The vulnerability is in the **void create()** function
- Overflow **canvas->title** to corrupt the FLAG 1 canvas's **private** field
- (Challenge 2) You must build an **arbitrary read** primitive and defeat ASLR

Hints

- The vulnerability is in the **void create()** function
- Overflow **canvas->title** to corrupt the FLAG 1 canvas's **private** field
- (Challenge 2) You must build an **arbitrary read** primitive and defeat ASLR
- (Challenge 2) Corrupt an object to make it easier to corrupt other objects

Hints

- The vulnerability is in the **void create()** function
- Overflow **canvas->title** to corrupt the FLAG 1 canvas's **private** field
- (Challenge 2) You must build an **arbitrary read** primitive and defeat ASLR
- (Challenge 2) Corrupt an object to make it easier to corrupt other objects
- (Challenge 2) Use readelf -s chal | grep flag_two to get the RVA of FLAG 2

Hints

- The vulnerability is in the **void create()** function
- Overflow **canvas->title** to corrupt the FLAG 1 canvas's **private** field
- (Challenge 2) You must build an **arbitrary read** primitive and defeat ASLR
- (Challenge 2) Corrupt an object to make it easier to corrupt other objects
- (Challenge 2) Use readelf -s chal | grep flag_two to get the RVA of FLAG 2
- (Challenge 3) You must build an **arbitrary write** primitive (very similar to read)

Hints

- The vulnerability is in the **void create()** function
- Overflow **canvas->title** to corrupt the FLAG 1 canvas's **private** field
- (Challenge 2) You must build an **arbitrary read** primitive and defeat ASLR
- (Challenge 2) Corrupt an object to make it easier to corrupt other objects
- (Challenge 2) Use readelf -s chal | grep flag_two to get the RVA of FLAG 2
- (Challenge 3) You must build an **arbitrary write** primitive (very similar to read)
- (Challenge 3) Find GLIBC, **system** and how to trigger a call to it

Hints

- The vulnerability is in the **void create()** function
- Overflow **canvas->title** to corrupt the FLAG 1 canvas's **private** field
- (Challenge 2) You must build an **arbitrary read** primitive and defeat ASLR
- (Challenge 2) Corrupt an object to make it easier to corrupt other objects
- (Challenge 2) Use readelf -s chal | grep flag_two to get the RVA of FLAG 2
- (Challenge 3) You must build an **arbitrary write** primitive (very similar to read)
- (Challenge 3) Find GLIBC, **system** and how to trigger a call to it
- (Challenge 3) Remember that the **GOT is readonly, but reading it** can get you far.