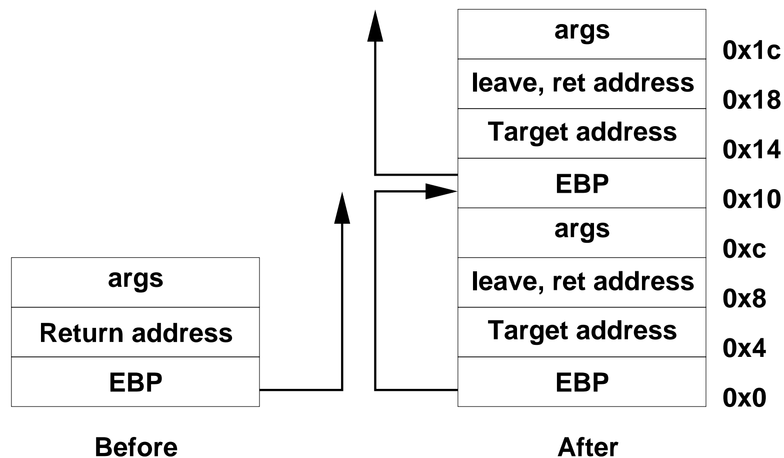
Overview

- We take a look at a couple of more advanced buffer overflow exploits
- We focus on two attack types:
  - Arc injection (or return-into-libc) attacks
  - Heap overflow attacks
- For more detailed information see:
  - *Vudo Malloc Tricks* by MaXX in Phrack, volume 57
  - *Secure Coding in C and C++* by Seacord
  - *Beyond Stack Smashing: Recent Advances in Exploiting Buffer Overruns*, IEEE Security and Privacy magazine (available electronically from the library)

Arc Injection

- Arc injection attacks involve the transfer of control to code already present in the process address space
- They are so-called because they introduce a new arc into a program's control-flow graph (cf. code injection which introduces a new node into the graph)
- The basic idea is to overflow a buffer in order to, as usual, overwrite some data on the stack
- However, unlike with a traditional buffer overflow, the aim is to not only rewrite the return address but to build new artificial frames on the stack (and so more complicated)

Arc Injection



Arc Injection

- Best explained with an example...
- Suppose we overflow a buffer local to procedure `foo` to rewrite the "before" stack to look like the "after" stack (see diagram)
- Before `foo` returns it executes the usual epilogue instructions

```
mov ebp, esp
pop ebp
ret
```

- This has the effect of popping into `ebp` the value at `0x0` so `ebp` now points to `0x10`

Arc Injection

- The target address is the address of the function we wish to execute but while executing it we want it to see a valid stack frame (as would normally be the case)
- This means we need it to see its arguments and a return address on the stack, we have placed them at `0xc` and `0x8` respectively
- The return address is popped and the target executes
- When the target returns `ebp` will again be restored to `0x10` and the return address popped
- This return address is the address of any function epilogue in the process address space

Arc Injection

- Executing this sequence restores the `ebp` at `0x10` and pops a return address: this is the address of our second target function
- The second target function's arguments and return address have once again been placed on the stack, at `0x18` and `0x1c` respectively
- By handcrafting such frames and chaining them together we can execute multiple functions
- Crucially, we have not inserted any code into the stack segment and thus this attack will defeat non-executable stack protection mechanisms

GNU C Library Memory Allocation

- Memory allocation in the GNU C library is based on Doug Lea's `malloc` implementation
- Heap management functions provided include `malloc`, `calloc`, `free` and `realloc`
- A good memory allocator should be fast, portable, tuneable and conserve space
- These goals are achieved in Doug Lea's implementation through the incorporation of boundary tags and bins to hold free chunks

Boundary Tags

- Memory chunks are bounded by size fields before and after each chunk
- This allows for the coalescing of two adjacent unused chunks into a larger usable chunk
- It also facilitates traversal of all chunks in forward or backward directions starting from any known chunk
- An attacker can sometimes rewrite boundary tags to cause the execution of arbitrary code

Binning

- Available chunks are maintained in 128 bins, grouped by size with contents sorted
- Thus a chunk of 200 bytes is stored in the bin that holds free chunks of 200 bytes
- A chunk of 16392 bytes is stored in the bin that holds free chunks where $16384 \leq size < 20480$
- Searches for available chunks are processed in smallest-first, best-fit order

Chunks of Memory

- The heap is divided by `dlmalloc` into contiguous chunks of memory
- Heap layout evolves as the following functions are used - `malloc` (request new chunk), `free` (release chunk), `realloc` (resize chunk, preserving data), `calloc` (`malloc` with returned chunk zeroed out)
- No free chunk ever borders another: two bordering free chunks are always coalesced into a single free one

Chunks of Memory

- When a new chunk is requested the effective amount of memory allocated is always greater than that requested
- This overhead is due to boundary tag requirements and an 8-byte alignment rule
- Chunks will always begin at an 8-byte-aligned address
- When `malloc(0)` is called 16 bytes are actually allocated on the heap, this is the size of a boundary tag
- And what does a boundary tag look like?

Boundary Tag Structure

- Stored in front of every chunk
- The way a boundary tag's fields are used depend on whether it is free or not...
- This makes things a little complicated but the basic idea is a simple one: if a chunk is free then use the now free data area to store chunk metadata i.e. data about that chunk

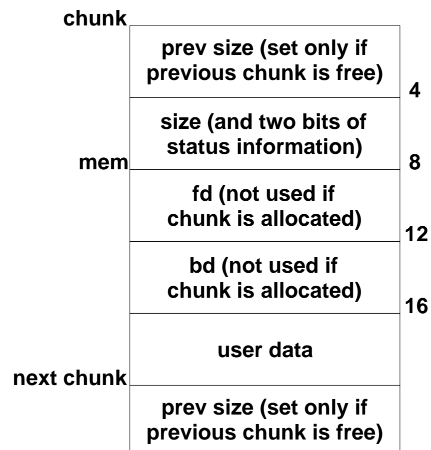
Allocated Chunks

- First 4 bytes hold size of previous chunk, if the previous chunk is free
- Next 4 bytes hold size of this chunk and some status information
- User data then begins (data can be stored in the previous size field of the following tag)
- Because the size is always a multiple of 8-bytes the 3 least significant bits of the size field are always zero
- One indicates whether the preceding chunk is free or not: `PREV_INUSE`

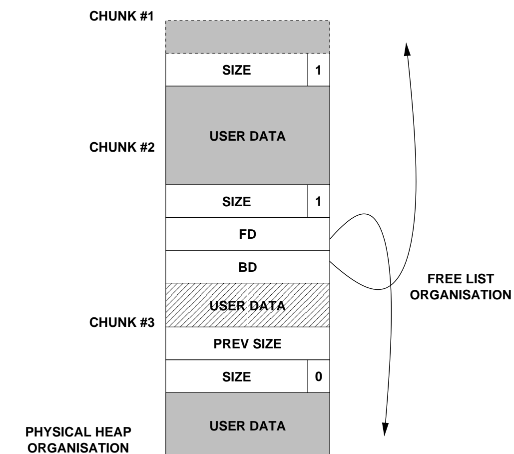
Free Chunks

- Free chunks are stored in doubly-linked lists
- Forward and backward pointers in each free chunk point to preceding and following free chunks (not to neighbouring physical chunks)
- If a chunk physically follows a free one the previous size field is set and thus the address of the free chunk can be calculated

Free Chunks



Free Chunks



Bin Management

- As stated, free chunks are stored in circular doubly-linked lists
- A bin has two pointers: a forward to the head and a backward pointer to the tail of the list
- The forward pointer points to the biggest chunk and so on until the forward pointer of the smallest chunk points back to the bin
- In order to take a chunk P off its list the back pointer of the following chunk must be set to the chunk preceding P

Bin Management

- In addition, the forward pointer of the chunk preceding P must be set to the chunk following P, here's the code:

```
#define unlink(P, BK, FD) {
  BK = P -> bk;
  FD = P ->fd;
  FD -> bk = BK;
  BK -> fd = FD;
}
```

- In order to place a new free chunk in its corresponding bin the `frontlink` macro is invoked

Bin Management

Effect of unlink (P, BK, FD)

Execution of Arbitrary Code

- If an attacker can trick `dlmalloc` into processing a carefully constructed fake memory chunk then any integer in memory can be overwritten with an arbitrary one
- If the overwritten memory is a function pointer then arbitrary code can be run
- If the address of the target function pointer is X and an attacker stores $X - 12$ in FD and the address of her shellcode in BK then what would happen on `unlink`?

Execution of Arbitrary Code

```
1. BK = P -> bk     BK points to shellcode
2. FD = P -> fd     FD points to X - 12
3. FD -> bk = BK    X points to shellcode!
4. BK -> fd = FD    Some shellcode is erased
```

Fake Chunk Construction

- Assume `malloc` has been called twice resulting in two adjacent chunks of memory
- If a buffer overflow vulnerability is present an attacker may overflow the first chunk to rewrite the boundary tag of the second one
- She can write arbitrary previous size, size, forward and backward pointers and `unlink` will do the rest
- However, for `unlink` to apply the corrupted chunk must be construed as free (since `unlink` applies only to free chunks)

Fake Chunk Construction

- The attacker needs to trick `dlmalloc` into thinking the corrupted chunk is free
- Information on whether a chunk is free or not is stored in the chunk following the free one
- Imagine we have three contiguous chunks A, B, C; when chunk A is freed, `dlmalloc` will check if B is free by looking at the `PREV_INUSE` field of C
- If it is free, `unlink` is invoked
- The location of C is found using the size field of B

Fake Chunk Construction

- This size field is under the attacker's control
- If she sets it to -4 `dlmalloc` will think the fictitious third chunk begins before the second chunk
- The previous size field of the second chunk will be interpreted as the size field of the third chunk
- Thus if the attacker turns off the `PREV_INUSE` bit in this "size field" she can cause `unlink` to be invoked on the corrupted chunk upon a `free` of the first chunk
- Message: buffer overflow vulnerabilities on the heap are exploitable