# Secure Software Programming and Vulnerability Analysis

Christopher Kruegel chris@auto.tuwien.ac.at
http://www.auto.tuwien.ac.at/~chris



# Heap Buffer Overflows and Format String Vulnerabilities

# Overview



- Security issues at various stages of application life-cycle
  - mistakes, vulnerabilities, and exploits
  - avoidance, detection, and defense
- Architecture
  - security considerations when designing the application
- Implementation
  - security considerations when writing the application
- Operation
  - security considerations when the application is in production



# Buffer Overflow



- Vulnerable buffer can be located
  - on the stack
  - on the heap
  - in static data areas
- Redirect execution flow by modifying
  - stack frames
  - longjump buffers
  - function pointers

➢ what can be done when overflowing a buffer on the heap?

# Heap Buffer Overflow



- Overflowing dynamically allocated memory

- Dynamically allocated memory
  - managed by a heap manager

- Heap manager
  - handles memory requested by user programs during run-time
  - sbrk() system call is very simple
  - library between user program and `sbrk()` system call
  - standardized `malloc` interface
  - different implementations for different operating systems



# Heap Management



- Goals
  - maximize portability / compatibility
    - alignment (8 byte hardwired), addressing rules
  - maximize locality
    - allocate chunks that are used together near each other
    - avoid fragmentation
  - maximize error detection
    - debug hooks, deactivated by default

  - minimize used space
    - as little management information as possible
  - minimize time for (de)allocation

# Heap Management



- Implementations

| Algorithm | Operating System |
|---|---|
| Doug Lea's `dlmalloc` | GNU LibC (Linux) |
| System V (AT&T) | Solaris, IRIX |
| BSD phk, BSD kingsley | *BSD, AIX |
| RtlHeap | Microsoft Windows |

- `dlmalloc`
  - keeps tags around allocated memory for book-keeping
  - overflow may modify these tags
  - functions `malloc`, `realloc`, `free`, `calloc` might be tricked into executing arbitrary code



# `dlmalloc`



- Memory layout
  - heap is divided into contiguous chunks of memory
  - no two free chunks may be physically adjacent

**Heap** low addresses → high addresses


```
U          ... used chunk
F          ... free chunk
Wilderness ... topmost free chunk
```

- Wilderness chunk
  - only chunk that may be increased (with system call `sbrk`)
  - treated as bigger than all other chunks

# dlmalloc



- Memory chunk
  - contiguous region of heap memory
  - can be allocated, freed, split, coalesced (two free chunks)

- Public and *Internal* routines

```
malloc(size_t n)
calloc(size_t unit, size_t quantity)
   → chunk_alloc()
realloc(void* ptr, size_t n)
   → chunk_alloc() / chunk_free()
free(void *ptr)
   → chunk_free()
```



# dlmalloc



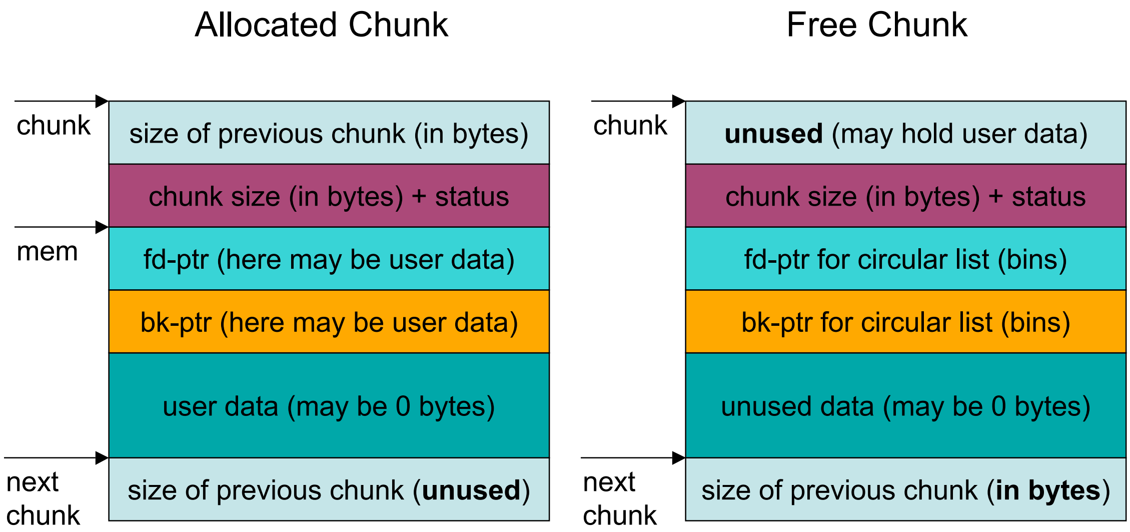
- Boundary tag
  - holds chunk management information
  - stored in front of each chunk
  - 16 bytes large → minimum allocated size

```
struct malloc_chunk {
   size_t prev_size;         // only used when previous chunk is free
   size_t size;              // size of chunk in bytes + 2 status-bits
   struct malloc_chunk *fd;  // only used for free chunks
   struct malloc_chunk *bk;  // only used for free chunks
};
```

- pointer returned by malloc (for user) starts at `fd`
  - usually 8 bytes overhead for allocated chunks

# dlmalloc








# dlmalloc



- Boundary tag – `prev_size` field
  - only used when previous chunk is free
  - to reduce memory wastage, field can hold user-data of previous chunk
- Boundary tag – `size` field
  - holds chunk size in bytes, but size is always a multiple of 8
  - ```
    chunk size = requested memory (by user via malloc)
               + 8 bytes (overhead)
               - 4 bytes (prev_field of next chunk)
               rounded up to next multiple of 8
    ```
  - 3 least significant bits are always 0, two of them are used as status bits
    - `PREV_INUSE (0x01)` – 1 if previous chunk is in use
    - `IS_MMAPED (0x02)` – 1 if chunk is memory mapped
    ```

# dlmalloc



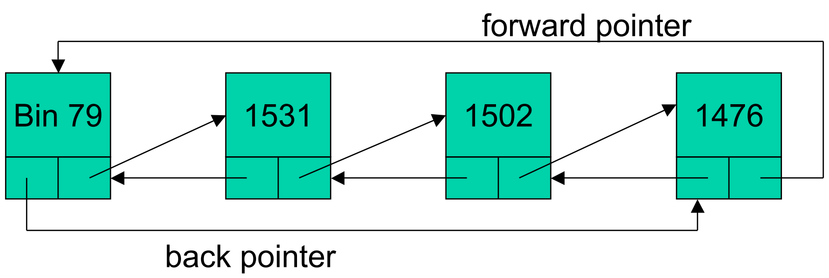
- Bin Management
  - available chunks are maintained in bins
  - depending on the size of the chunk, the corresponding bin is chosen
  - remainder of most-recently split (non-top) chunk and top (wilderness) chunk are never in any bin
  - chunks with a size of less than 512 bytes are called *small*
  - 128 available bins
  - 62 small bins (for small chunks of size 16 – 504 byte) only hold chunks of a certain size
  - regular bins hold chunks of a certain size range



# dlmalloc



- chunks are stored in bins on a circular doubly-linked list
- the bin itself consists of two pointers (forward/back) and acts as the corresponding list head
- each bin is initially empty
- chunks are maintained in decreasing sorted order by size
  → best fit algorithm



# `dlmalloc`



- Memory allocation

1. List of corresponding bin is scanned (starting backwards)
   - when chunk of exactly correct size (chunk size is equal or bigger by not more than 16 bytes than the requested size) is found, return it
2. Most-recent remainder of split is used (when large enough)
   - split it when it is too big, return it when size is exact
3. Other bins are scanned in increasing order
   - return chunk of exact size, split one that is too big
4. Split memory from wilderness chunk (when big enough)
5. Extend wilderness chunk (with `sbrk()`), when this fails, return `NULL`



# `dlmalloc`



- Memory de-allocation (free operation)

1. When the chunk to be freed borders the wilderness chunk, it is consolidated into it
2. If the chunk before the one to be freed is unallocated, it is consolidated into a single large chunk
3. If the chunk after the one to be freed is unallocated, it is consolidated into a single large chunk

# dlmalloc



- When chunks are handled, their entries have to be taken off or inserted into the corresponding lists

- Macro `unlink()`
  - used to take off entry P with its pointers FD and BK

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

- Macro `frontlink()`
  - used to insert P (size S, pointers FD, BK) into bin IDX



# dlmalloc



```
#define frontlink(A, P, S, IDX, BK, FD)      {
   IDX = bin_index(S);                       \
   BK = bin_at(A, IDX);                      \
   FD = BK->fd;                              \
   if (FD == BK) {                           \
        mark_binblock(A, IDX);               \
   } else {                                  \
        while (FD != BK && S < chunksize(FD) \
               FD = FD->fd;                  \
        }                                    \
        BK = FD->bk;                         \
   }                                         \
   P->bk = BK;                               \
   P->fd = FD;                               \
   FD->bk = BK->fd = P;                      }
```

# dlmalloc



- Exploiting the `unlink()` macro
  - overwrite an arbitrary memory position with arbitrary integer
  - overwrite address stored in `FD` + 12 (offset of `bk`) with `BK`

```
BK = P->bk;
FD = P->fd;
FD->bk = BK;
```

  - overwrite a function pointer (e.g. stored in GOT – global offset table) with address of the shell code
  - when function is later invoked, shell code is executed instead
  - used against `netscape`, `traceroute` and `slocate`



# dlmalloc



- Exploiting the `frontlink()` macro
  - overwrite an arbitrary memory position with address of modified chunk
  - overwrite address stored in BK + 8 (offset of fd) with address of chunk P

```
while (FD != BK && S < chunksize(FD)
        FD = FD->fd;
BK = FD->bk;
FD->bk = BK->fd = P;
```

  - beginning of chunk (`prev_size` field) has to contain executable code (e.g. jump to shell code)
  - same approach as `unlink()` macro
  - no known exploit in the wild, but `sudo` example in Phrack 57-8

# Heap Overflow

Automation Systems Group

- Heap overflow requires modification of boundary tags
  - in-band management information
  - task is to fake these tags to trick `dlmalloc` into overwriting addresses of attackers choice

- Different techniques for other memory managers
  - System V (Solaris, IRIX) - self-adjusting binary trees
  - Phrack 57-9 (Once upon a free())



# Format String Vulnerability

Automation Systems Group

- Problem of user supplied input that is used with `*printf()`
  - `printf("Hello world\n"); // is ok`
  - `printf(user_input);       // vulnerable`

- `*printf()`
  - function with variable number of arguments
    `int printf(const char *format, ...)`
  - as usual, arguments are fetched from the stack

- `const char *format` is called format string
  - used to specify type of arguments
    - `%d` or `%x` for numbers
    - `%s` for strings

# Format String Vulnerability



```
#include <stdio.h>

int main(int argc, char **argv){
   char buf[128];
   int x = 1;

   snprintf(buf, sizeof(buf), argv[1]);
   buf[sizeof buf - 1] = '\0';

   printf("buffer (%d): %s\n", strlen(buf), buf);
   printf("x is %d/%#x (@ %p)\n", x, x, &x);
   return 0;
}
```



# Format String Vulnerability



```
chris@euler:~/test > ./vul "%x %x %x %x"
buffer (28): 40017000 1 bffff680 4000a32c
x is 1/0x1 (@ 0xbffff638)

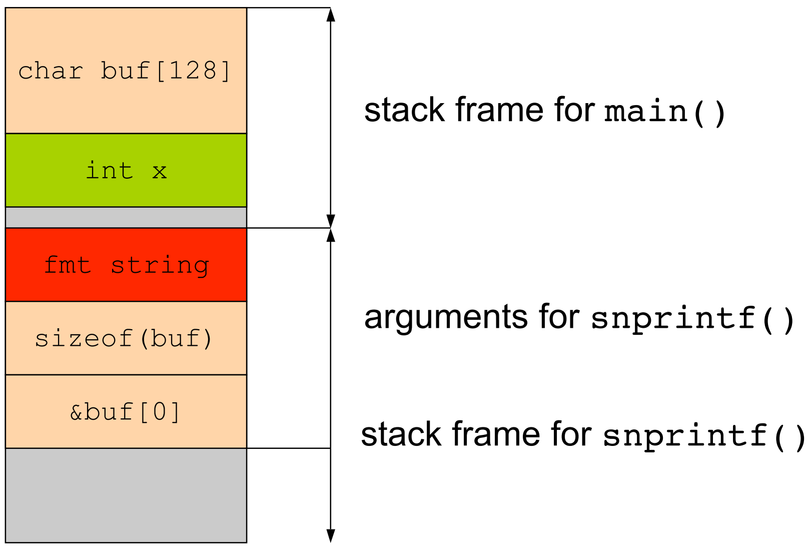
chris@euler:~/test > ./vul "AAAA %x %x %x %x %x"
buffer (35): AAAA 40017000 1 bffff680 4000a32c 1
x is 1/0x1 (@ 0xbffff638)

chris@euler:~/test > ./vul "AAAA %x %x %x %x %x %x"
buffer (44): AAAA 40017000 1 bffff680 4000a32c 1 41414141
x is 1/0x1 (@ 0xbffff638)
```

# Format String Vulnerability



Stack Layout






# Format String Vulnerability



```
chris@euler:~/test > perl -e 'system "./vul", "\x38\xf6\xff\xbf
    %x %x %x %x %x %x"'
buffer (44): 8öÿ¿ 40017000 1 bffff680 4000a32c 1 bffff638
x is 1/0x1 (@ 0xbffff638)

chris@euler:~/test > perl -e 'system "./vul", "\x38\xf6\xff\xbf
    %x %x %x %x %x%n"'
buffer (35): 8öÿ¿ 40017000 1 bffff680 4000a32c 1
x is 35/0x2f (@ 0xbffff638)
```

- One can use width modifier to write arbitrary values
  - for example, `%.500d`
  - even in case of truncation, the values that would have been written are used for `%n`