CS24: Introduction to Computing Systems

Spring 2014

Lecture 8

LAST TIME

- Began examining explicit heap allocators
  - The program is responsible for releasing memory when it's no longer needed
- Allocator must deal with several challenges:
  - Avoiding or minimizing memory fragmentation
  - Coalescing adjacent blocks of free memory
  - Dealing with data alignment issues

00:	00	01	02	03	01:	04	05	06	07	02:	08	09	0A	0B	03:	0C	0D	0E	0F
04:	10	11	12	13	05:	14	15	16	17	06:	18	19	1A	1B	07:	1C	1D	1E	1F
08:	20	21	22	23	09:	24	25	26	27	0A:	28	29	2A	2B	0B:	2C	2D	2E	2F

Heap Allocator Interface

- Common interface exposed by explicit allocators:

  `void * malloc(size_t size)`

  - Allocates a block of memory of [at least] **`size`** bytes
  - Returns a pointer to start of the block, or **`NULL`** if **`size`** bytes are not available

  `void free(void *ptr)`

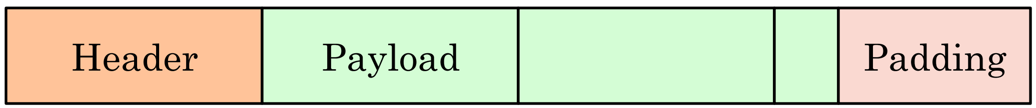
  - Releases a block of memory back to the heap
- Sometimes other operations as well:

  `void * realloc(void *ptr, size_t size)`

  - Attempts to change the size of an existing allocation
  - If reallocation succeeds, original contents are copied to the new region, and original allocation is freed
  - If reallocation fails, returns **`NULL`** and original allocation is left unchanged

Representing Memory Blocks

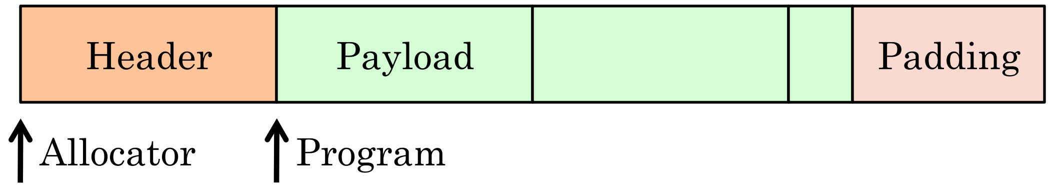
- Common way to represent blocks of memory on heap:



- Header specifies:
  - Total block size in bytes, including all parts
  - “Allocated” / “free” flag
  - Since memory block is usually word-aligned, the block-size value won’t actually use all bits
    - e.g. for aligning blocks on 8 bytes, bottom 3 bits of size will be 0
    - Can use bottom-most bit(s) to store allocated/free flag
- Payload: area that the program gets to use
  - Payload’s size is what the program requested
- Padding: any space necessary to make the block word-aligned (if necessary or desirable for platform)

Representing Memory Blocks (2)

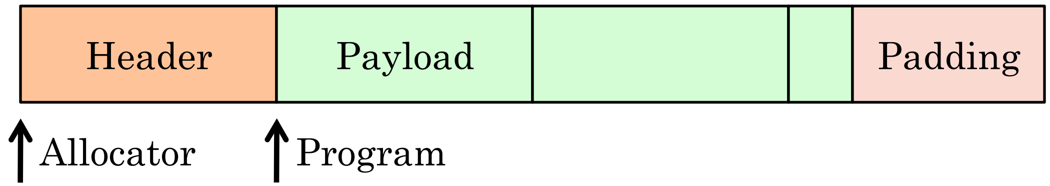
- Heap memory blocks:



- When a program requests memory:
  - Allocator works with block structure, creates/updates a block header to satisfy request
  - Allocator returns a pointer to start of the *payload*, not start of the *header*
- Abstraction:
  - Caller *doesn't care* how allocator manages the heap
  - Just wants some memory to use for their program!

Representing Memory Blocks (3)

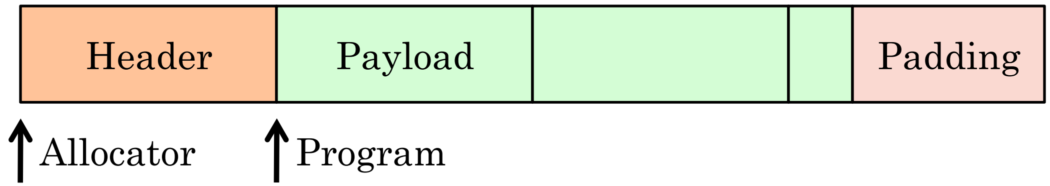
- Heap memory blocks:



- What can happen if program accidentally writes outside of the payload region?
  - e.g. due to a bug, the program writes past the end of its payload, or perhaps before the start
- In these cases, the heap can become corrupted
  - Can no longer keep track of heap's allocation state
- These bugs usually manifest at the *next* allocation or deallocation operation
  - This is when the allocator tries to use its state…

Representing Memory Blocks (4)

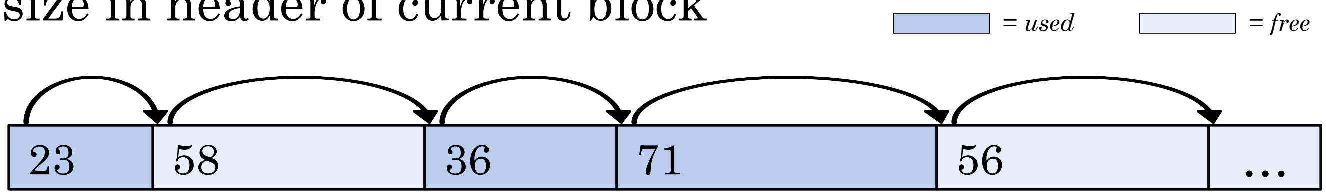
- Heap memory blocks:



- When program frees memory:
  - Program passes its pointer to start of the payload back to the allocator
  - Allocator must adjust pointer to gain access to the header

Implicit Free List

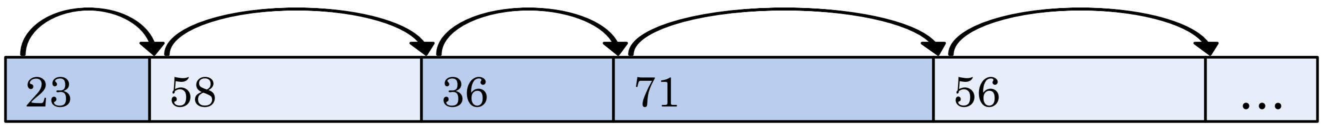
- Heap memory blocks form an implicit free list
  - Can determine start of next memory block by looking at size in header of current block



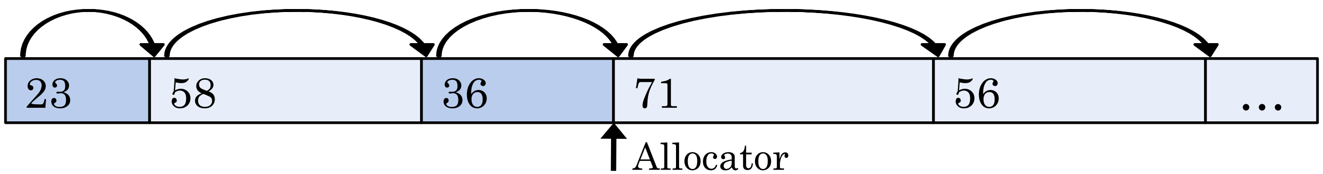
  - When allocation request is made, allocator searches thru list of blocks to find a free block that can satisfy the request
- Several strategies for finding a suitable free block
  - <u>First fit</u> – start at beginning of list, stop when first suitable free block is found
  - <u>Next fit</u> – similar to first-fit, but remember where the last suitable free block was found, and start next search there
  - <u>Best fit</u> – check *all* free blocks; choose the smallest free block that can satisfy the request

Coalescing Blocks

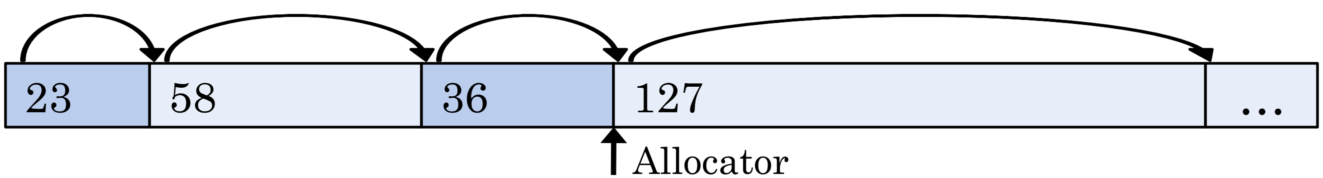
- When a block is freed, may need to coalesce it with adjacent free blocks



- Example: 4th block is freed by application

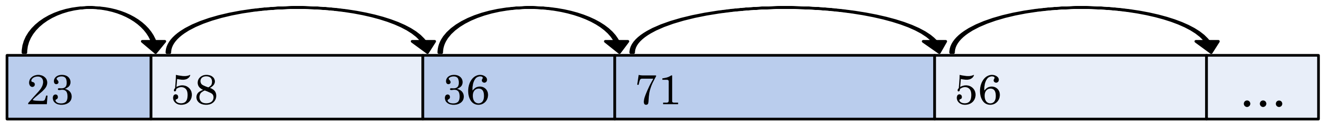


  - Allocator checks if next block is also free
  - If so, coalesce into a larger free block

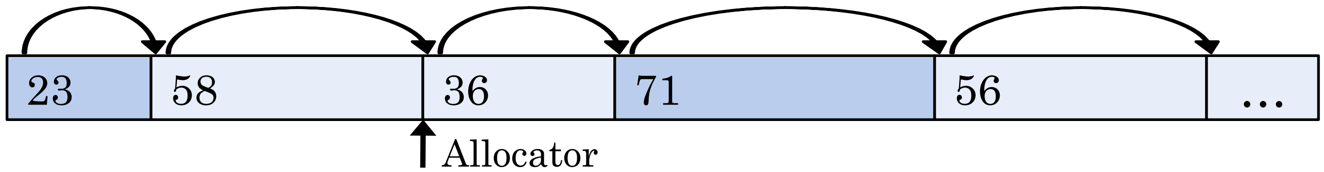


Coalescing Previous Blocks

- This is only fast when coalescing with *next* block



- Example: 3rd block is freed by application



  - How can we efficiently find the *previous* free block?!
- A simple solution: boundary tags (Knuth)
  - Give each block a footer as well as a header
    - (Footer is identical to header)
  - Can easily find prev. block size, and whether it's free

Explicit Free Lists

- Implicit free list approach is simple, but not fast
  - Allocation requires a scan of *all* blocks, whether allocated or free
  - No point in looking at allocated blocks!
- Two observations:
  - Really only need to keep the *free* blocks in a list, since these are the ones we check during allocation
  - Since free blocks aren't used by the program, could even store free-list pointers within the block payloads
- For example:
  - Use a linked list to chain together free blocks
- Called an explicit free list approach
  - Explicitly arranging free blocks into a data structure

Organizing Free Blocks

- Many strategies for organizing/selecting free blocks
- Example 1: Maintain free-list in *LIFO order*
  - Newly freed blocks always go at front of free-list
  - Use first-fit policy for assigning new blocks
  - Even though free-list is in LIFO order, can use boundary tags on memory blocks to support constant-time coalescing with neighboring blocks
  - Very fast approach, but more susceptible to memory fragmentation
- Example 2: Maintain free-list in *address order*
  - Keep free blocks sorted by increasing address in free-list
  - Use first-fit policy for assigning new blocks
  - Slower due to linear-time insertion sort when freeing
  - Better memory utilization than LIFO-order free list

Organizing Free Blocks (2)

- Other strategies maintain multiple free-lists
- Example: Segregated fits strategy
  - Each free-list is assigned a size class
    - e.g. {0-1024}, {1025-2048}, {2049-3072}, etc.
    - Each list contains free blocks of that size class
  - When allocating memory:
    - First-fit strategy, starting with appropriate size class
    - If no block is found, go to next larger size class
    - If still no block found, request more memory from OS
    - Once available block is found, may optionally split block and put free part into free-list for corresponding size class
  - When freeing memory:
    - Coalesce with adjacent free blocks, then put result into free list of appropriate size class
- This approach frequently used by standard allocators
  - Fast: allocation searches target blocks of appropriate size
  - Efficient: approaches memory usage of best-fit strategies

Dynamic Memory Allocation

- Like branching instructions, a heap facility greatly expands the programs we can write
- Heap management is a hard problem to solve!
  - Don't force every program to solve it separately…
  - Provide a run-time facility that implements this capability
  - Programs can leverage this facility as needed
- Understanding how heaps are implemented will help you use them more effectively. For example:
  - Code that performs allocations and deallocations of varying sizes can suffer from memory fragmentation issues
  - Code that always allocates/deallocates blocks of the same size won't suffer from fragmentation
  - Allocating many small objects will tend to waste lots of time and memory, due to bookkeeping overhead

Stack vs. Heap: Comparison

- Stacks are much faster and simpler than heaps
  - Bookkeeping complexity/overhead of heaps is why we don't allocate each local variable on the heap!
  - Much faster to keep local variables and other temporaries on stack
- Heaps provide different capabilities than stacks
  - Enables different usage patterns, but also has additional costs

- Important to understand these distinctions so you use the right tool for the job

Important Optimization Principles

- Two <u>major</u> optimization principles in systems:

- **Make the common case fast.**
  - Determine the most common behaviors of programs, and optimize hardware to execute these cases <u>fast</u>.
  - (Will explore this principle in 2^{nd} half of the term!)

- **Make the fast case common.**
  - If hardware is good at certain cases, arrange your computations to make them as frequent as possible!

Data Alignment and Optimization

- See this very clearly with data alignment:
  - "Make the fast case common."
  - Compiler and assembler adjust the layout of your code to align it with word boundaries for the CPU
- Accumulator code from lecture 5:

```
int value;

int accum(int n) {
    value += n;
    return value;
}

int reset() {
    int old = value;
    value = 0;
    return old;
}
...
```

```
        .file   "amain.c"
        .text
        .p2align 4,,15
.globl accum
        .type   accum, @function
accum:
        ... # code for accum()
        ret
        .size   accum, .-accum
        .p2align 4,,15
.globl reset
        .type   reset, @function
reset:
        ... # code for reset()
```

- Same thing also happens with data structures

Heterogeneous Data Structures in C

- C represents heterogeneous data structures with **`struct`** declarations
  - **`struct`** members can be different data types, if desired
  - Each member has its own name
  - Members can be primitive types, pointers, arrays, other structs, etc.
- Example: a linked-list node

```
struct node {
  int number;
  char *string;
  struct node *next;
};
```

- Using our struct:

```
struct node n;
n.number = 42;   /* Refer to members of struct. */
n.string = "answer";
n.next = NULL;
```

C Structures in IA32

- Compiler computes important details for each struct:
  - Relative offset of each member from start of struct
  - Size of each member's data type

Member	Offset	Size
`int number`	0	4 bytes
`char *string`	4	4 bytes
`node *next`	8	4 bytes

- When C code accesses struct members:
  - Compiler adds computed offsets to starting address of struct to access specific members

```
void init(struct node *n) {
    n->number = 42;
    n->string = "answer";
    n->next = NULL;
}
```

```
init:
  pushl %ebp
  movl  %esp, %ebp
  movl  8(%ebp), %eax   # eax = n
  movl  $42,     (%eax) # n->number
  movl  $.LC0, 4(%eax)  # n->string
  movl  $0,     8(%eax) # n->next
  popl  %ebp
  ret
```

Structures and Data Alignment

- Previous example had all 4-byte members
  - No data alignment issues
- Can use members that are less than a word

```
struct s1 {
    int i;
    char ch;
    int j;
};
```

00	01	02	03	04	05	06	07	08
i				ch	j			

  - If compiler uses only one byte for **ch**, can't properly align **j** with word boundaries
- Compiler *pads* the struct to properly align all members

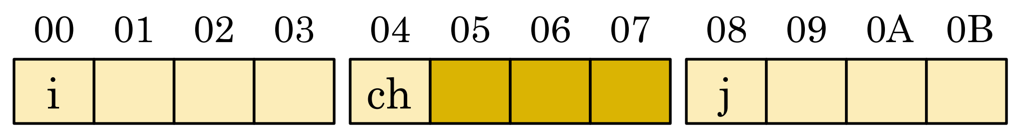
00	01	02	03	04	05	06	07	08	09	0A	0B
i				ch				j			

  - **ch** is followed by 3 unused bytes, to properly align **j**

Structures and Data Alignment (2)

- In this example:

```
struct s1 {
     int i;
     char ch;
     int j;
};
```



- Accesses to **`ch`** use byte-width operations

```
void init(struct s1 *rec) {
    rec->i = 1234;
    rec->ch = 'a';
    rec->j = 5678;
}
```

```
init:
  pushl %ebp
  movl  %esp, %ebp
  movl  8(%ebp), %eax   # eax = rec
  movl  $1234,  (%eax) # rec->i
  movb  $97,    4(%eax) # rec->ch
  movl  $5678, 8(%eax) # rec->j
  popl  %ebp
  ret
```

- **`sizeof(s1)`** reports 12 bytes, not 9 bytes

Structures and Data Alignment (3)

- Some compilers support packing these structures into minimal space necessary (non-standard!)
  - e.g. **gcc** supports a **`__packed__`** attribute

```
struct s1 {
    int i;
    char ch;
    int j;
} __attribute__((__packed__)) ;
```

00	01	02	03	04	05	06	07	08
i				ch	j			

```
void init(struct s1 *rec) {
    rec->i = 1234;
    rec->ch = 'a';
    rec->j = 5678;
}
```

```
init:
  pushl %ebp
  movl  %esp, %ebp
  movl  8(%ebp), %eax  # eax = rec
  movl  $1234,  (%eax) # rec->i
  movb  $97,   4(%eax) # rec->ch
  movl  $5678, 5(%eax) # rec->j
  popl  %ebp
  ret
```

- **`sizeof(s1)`** also reports 9 bytes now

STRUCTURES AND DATA ALIGNMENT (4)

- Packed version of structure will be significantly slower to work with in memory
  - e.g. need multiple reads to access value of **`j`**

00	01	02	03	04	05	06	07	08
i				ch	j			

  - If working with an array of **`s1`** values, *many* member accesses will not be word-aligned!

00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11
i				ch	j				i				ch	j			

- Primarily useful for easily breaking apart packed values – e.g. network packets, disk/file structures
  - In memory, work with properly word-aligned struct
  - For IO, use packed version of struct to easily convert to/from a byte-sequence

Structures and Data Alignment (5)

- Even if small member is at end of structure, still need to pad to word boundary

```
struct s2 {
    int i;
    int j;
    char ch;
};
```

00	01	02	03	04	05	06	07	08	09	0A	0B
i				j				ch			

  - With local variables, don't want to affect other variables' word alignment

```
void foo(int i, int j) {
    struct s2;          /* -12(%ebp) */
    int d;              /* -16(%ebp) */
    ...
}
```

  - If using in an array, want every element to be word-aligned

```
struct s2 values[100];
```

    - **Note:** 25% of space is wasted on padding to word boundaries! Important to consider in data structure design.

Optimization and Data Alignment

- Optimization: "Make the fast case common."
  - Computers are fastest when accessing data aligned on word boundaries…
  - …so the compiler lays out instructions and data to be aligned on word boundaries.
- Wastes a certain amount of space, but yields a substantial performance improvement

C Unions

- C also provides unions:
  - Unlike structs, all union members occupy the same memory location

```
union value {
    int int_val;
    char *str_val;
    float float_val;
};
```

Member	Offset	Size
`int int_val`	0	4 bytes
`char *str_val`	0	4 bytes
`float float_val`	0	4 bytes

  - All members are assigned same offset by the compiler
  - Size of union is size of largest member
- Very useful for representing different, mutually-exclusive kinds of values in a single structure
- Also can be bug-prone, since C type-system can't keep you from misinterpreting a value!

C UNIONS (2)

- Unions normally used in concert with a tag field
  - Unlike structs, union members all occupy the same memory location

```
struct value {
    enum ValType type;
    union {
        int int_val;
        char *str_val;
        float float_val;
    };
};
```

  - Code can use **type** to determine what union-member to access

```
if (v->type == IntValue)
    set_result(a * v->int_val);
```

C Language Run-Time Facilities

- C is a relatively low-level language
  - Virtually all C abstractions translate easily to assembly language
    - Primitive data types, procedures, arrays and composite data types, flow-control statements
  - Not many run-time facilities for C programmers
- Example: array bounds checking
  - C does not stop you from indexing past an array's bounds!

```
int a;
int r[4];
int b;
...
r[4] = 12345;       /* Compiles!      */
r[-1] = 67890;      /* Also compiles! */
```

  - May affect **a** and/or **b**, depending on relative placement of the variables by the compiler

C Programs and Array Bounds

- Lack of array bounds-checking can cause problems
- Buffer overflows:
  - Program includes a **char** buffer for receiving input data

```
/* Buffer for reading in lines of input data. */
char buf[100];
...
gets(buf);    /* Standard C function in stdio.h */
```

  - An example implementation of **gets()**:

```
char * gets(char *s) {
    int i = 0;
    int ch = getchar(); /* Get char from console */
    while (ch != EOF && ch != '\n') {
        s[i] = ch; ch = getchar(); i++;
    }
    s[i] = 0;   /* Zero-terminate the string. */
    return (ch == EOF && !feof(stdin)) ? NULL : s;
}
```

C Programs and Array Bounds (2)

- **`gets()`** has no way of knowing how large its buffer is!

```
char * gets(char *s) {
    int i = 0;
    int ch = getchar();
    while (ch != EOF && ch != '\n') {
        s[i] = ch; ch = getchar(); i++;
    }
    s[i] = 0;   /* Zero-terminate the string. */
    return (ch == EOF && !feof(stdin)) ? NULL : s;
}
```

- **`gets()`** is a common source of buffer overflows!
  - NEVER EVER use **`gets()`** in your C programs!
  - Much better to use:

```
char * fgets(char *s, int size, FILE *stream)
```

  - Takes the buffer's size as an argument
  - Function makes sure to stay within the buffer

Buffer Overflow Exploits

- Buffer overflows don't just cause your program to crash!
- They can frequently be leveraged to compromise system security

- Denial of service:
  - Cause the server program to crash unexpectedly when fed bad input
  - Example: Ping of Death
    - IP packets > 64KB in size are illegal
    - Many OSes had a packet buffer size of exactly 64KB
    - When sent a packet larger than 64KB, target machine would crash, hang, or otherwise freak out

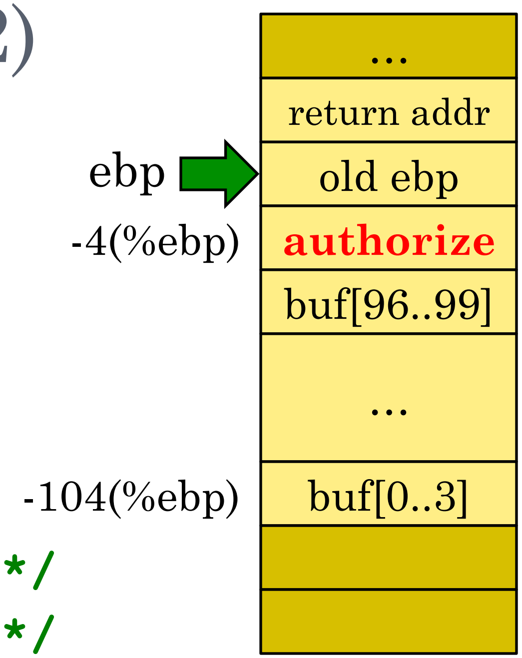
Buffer Overflow Exploits (2)

- Modify program state:
  - Example server code:

```
void handle_request() {
  int authorize = 0;
  char buf[100];

  ...  /* Read request into buffer */
  ...  /* Verify user, etc.        */

  if (authorize) {
     ... /* Request is allowed!  Do it. */
  }
}
```



- By overflowing the input buffer, can modify value of other local variables.

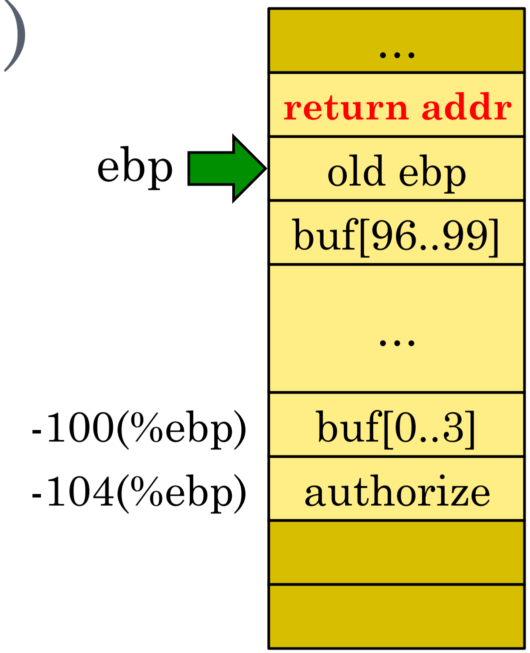
Buffer Overflow Exploits (3)

- Executing arbitrary code:

```
void handle_request() {
  char buf[100];
  int authorize = 0;

  ...  /* Read request into buffer */
  ...  /* Verify user, etc.        */

  if (authorize) {
     ... /* Request is allowed!  Do it. */
  }
}
```



- Instead of modifying state, exploit aims to change the actual *return address* on stack
  - Input includes malicious code loaded into the input buffer
  - Set return-address to jump into the buffer

Buffer Overflow Exploits (4)

- These examples store the buffer on the stack
- Heap-based buffer overflows are just as feasible
  - Overwrite jump-tables, function pointers, other code
- Details of these attacks depend very much on system details! For example:
  - Which direction the stack grows
  - Stack layout of function(s) being exploited
    - What local variables are present, and what effect they have
    - Where the return-address is stored
  - Heap layout of program being exploited
  - …many other details of server being compromised…
- Nonetheless, a very large percentage of exploits use buffer overflows to compromise the system.

Avoiding Buffer Overflows

- A simple solution to buffer overflows?
  - Build array bounds-checking into your language
  - All array indexes are verified before access occurs
  - Invalid indexes are flagged with some kind of error
- To support this, need to store more information about arrays
  - Add metadata to the array representation
- Example:

```
struct array_t {
    int length;  /* Number of elements */
    struct value_t values[];
};
```

  - Arrays include length information in their run-time representation

Array Bounds-Checking

- New array representation:

```
struct array_t {
    int length;  /* Number of elements */
    struct value_t values[];
};
```

  - Last member of a struct can be an array with no size
  - Supports variable-size arrays in structs

```
array_t *a = (array_t *)
    malloc(sizeof(array_t) + n * sizeof(value_t));
a->length = n;
```

  - **`values`** is a pointer to start of variable-size array
- Memory layout:

a	length	values[0]	values[1]	…	values[n-1]

Array Bounds-Checking (2)

- Can expose array length as a member for programs to reference

```
for (int i = 0; i < a.length; i++) {
    compute(a[i]);
}
```

- In language implementation, <u>all</u> array-indexing operations are bounds-checked against length
- Still need to further constrain our language!
- Close off other potential holes:
  - No more pointer manipulation and pointer arithmetic
  - Present a simplified, opaque "reference" abstraction for programmers to use
  - Need a clean way to report errors when they occur

Memory Leaks

- Another common issue: memory leaks
- Explicit allocators require program to inform allocator when memory is no longer used
- If the program fails to do this properly:
  - Program no longer has a pointer to allocated memory
  - But, allocator thinks memory block is still in use!
  - Cannot reclaim memory until program terminates
- *Implicit allocators* assume the responsibility for reclaiming unused memory
  - <u>Garbage collection</u>: reclaiming unused heap storage
  - Program no longer has to free memory itself
  - Memory-leak issues largely disappear

Looking Forward

- Languages like C don't make it very easy to write correct and bug-free programs!
  - Much easier than assembly language, but can easily have buffer overflows, memory leaks, exploits…
- Take another step up the abstraction hierarchy:
  - Programming languages that provide more powerful abstractions can mitigate <u>many</u> of these problems
- Will focus on three very common features:
  - Memory management: references, implicit allocators
  - Object-oriented programming and polymorphism
  - Another flow-control mechanism: exception handling