Homework 3

CS161 Computer Security, Fall 2008
Assigned 10/07/08
Due 10/13/08

For your solutions you should submit a hard copy; either hand written pages stapled together or a print out of a typeset document[1].

1 Program Correctness

1. *Reasoning about Code.* (8 points)

```
/*
* Function StringEncrypt
* @ input : A string input that should be encrypted.
* @ key : 8 byte key value to be used for encryption.
* @ Return Value : On failure, it returns NULL. On success, it returns a
* length-encoded encrypted value of the input string - the length of the
* actual encrypted string is encoded in the first 4 bytes, followed by the
* (not NULL-terminated) encrypted string.
*/

char *
StringEncrypt(char * input, char  *key)        {


size_t length = strlen (input);
unsigned int i;
char * encrypted = malloc (length + 4);


for (i = length; i != 0; i--)
{


*(encrypted + i + 4) = *(input + i) ^ (key[i%8]);  // XORing with key.

}

*(size_t *)(encrypted) = length;
return encrypted;
}
```

Figure 1: Code for Problem 1

The (poorly written) function `StringEncrypt`, shown in Figure 1, encrypts the input string and returns the encrypted string. The function is not memory safe, and does not behave correctly on all inputs.

[1] LaTeX is the most suitable tool for typesetting mathematical documents, but other use of other editors are perfectly acceptable

- (4 points) Spot all the bugs that could cause the program to execute an unsafe memory operation.
- (4 points) Introduce the additional necessary checks to ensure that all memory operations in the function execute safely, without changing the intended functionality. You may refer to the `man` pages for the documentation of the C library function interface. You should show the checks inlined in the code above at the right places where they should appear; you should annotate your inserted code as C comments to demarcate it from original code.

2 Memory Errors

1. *Heap overflows and double free bugs.* (16 points) Consider the contrived example of a web server program, shown in figure 2, which takes three command line arguments. It has several serious bugs which exploit implementations of dynamic memory management functions.

```
int main (int argc, char* argv[])
{
   char *p, *q, *r;

   p = malloc (100); // Returns address 0x8049000
   r = malloc (100); // Returns address 0x8049080

   strncpy (r, argv[3], 99); r[99] = NULL;
   printf ( "Your account number is %s\n", r);

   strncpy (p, argv[1], 99);  p[99] = NULL;
   printf ("Method is %s", p);
   free (p);
   free (r);

    if (!strncmp (&r[8], ''0'', 1)) {
    q = p;
    snprintf (q, 256, "%s Welcome to your account, %s\n", argv[1], argv[2]);
    printf ("Thanks for using online banking!   %s", q);
    }

    p = malloc (100);
    r = malloc (100);
    if (strcmp (q, "System Admin")) {
    free (q);
    } else {
    authnticate_admin (q)
    }
    free (p);
    free (r);
}
```

Figure 2: Code for the contrived web server in problem 2

The server is running on an x86 based Unix system. On the target Unix system, the OS and the C standard library provide functions to handle variable amounts of memory in a dynamic way. This allows programs to dynamically request memory blocks from the system. The operating system only provides a very low-level system call `brk` to change the size of a big memory region, which is known as the heap.

On top of this system call the C library's `malloc` interface is implemented, which provides a layer between the application and the system call. It can dynamically split the large single block into smaller 'chunks', free those chunks on request of the application and avoid fragmentation while doing so. One of the goals of an implementation of C library function is to be fast and space efficient. Here is the specific scheme used on the target system.

Each 'chunk' is a unit of memory allocation. Chunks are of different sizes; the library implementation always manages chunks of sizes that are a power of 2. For this problem, you can assume that each chunk has size of 128 bytes. Internally, a chunk has the following data type as shown in Figure 3.

```
struct metadata {
struct chunk* next; // Points to next chunk in free list, or is NULL.
struct chunk* prev; // Points to previous chunk in free list, or is NULL.
};

struct chunk {
union {
struct metadata mdata;
char data [128];
} content;
}
```

Figure 3: Type definition for a 'chunk'

All free chunks are stored on a *doubly-linked list*, called the "free list". A global `free_list_head` pointer is the head of the free list, and it always points to the first node on the free list. When an allocated chunk is in use, the data of the user is stored in the field `data` of the union `content`. The `mdata` field (free list prev and next pointers) are meaningful only when the chunk is on the free list. Therefore, to conserve space, the data and the associated metadata share locations in memory using a `union`, as shown in Figure 3.

The target implementation of the C library uses a first-fit algorithm, i.e it finds the first block on the free-list that is sufficient to serve an allocation request. The target C library implementation of `free` always adds the chunks being `free`'d as the second element of the free list (immediately after the first element). The relevant code for `malloc` and `free` is shown in figure 5 and figure 6 respectively. You must first look at these implementations to be able to solve this problem.

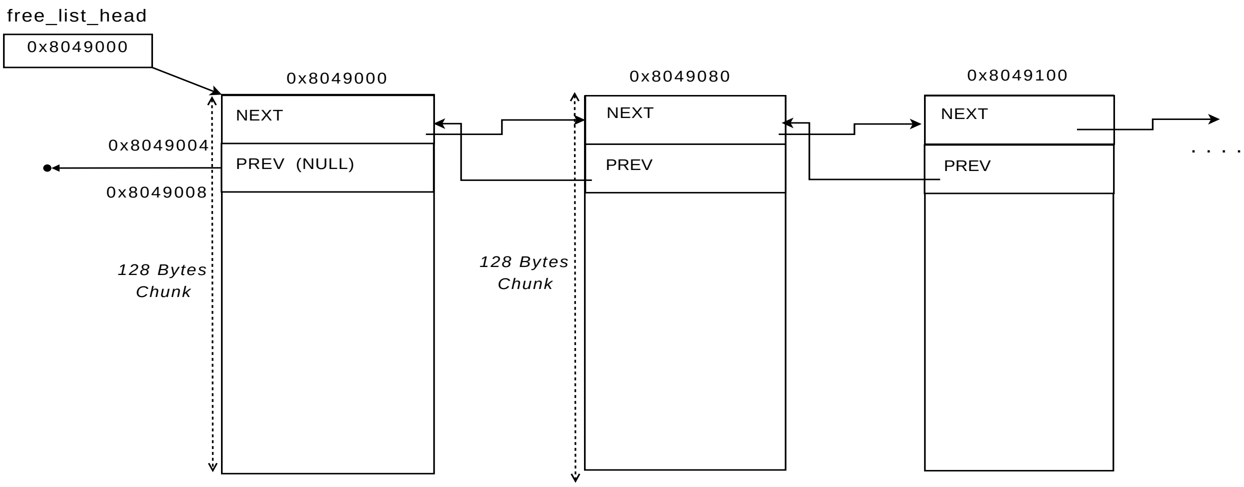

Figure 4: State of the free list at the start of `main`.

```
void * malloc (int size) {

/* find_first : It traverses the free list in the forward direction,
 * with the starting address of the first chunk provided as the first
 * argument. It returns the pointer to the first chunk that is large
 * enough to hold the request, or NULL if it fails.
 */
 void * p = find_first (free_list_head, size);

/* delete_from_list : It removes chunk pointed to by the first argument,
 * from the free list. If the argument is the address of the first
 * chunk on the free list, it also sets the free_list_head appropriately.
 */
 delete_from_list (p);
 return p;
}
```

Figure 5: Code for `malloc` in the target C library.

```
/* add_chunk_after_current : Adds chunk pointed to by NEWNODE,
 * immediately after CURRENT. If CURRENT is NULL, it adds NEWNODE
 * as the first node in free list.
 * @ current : Pointer to the chunk after which the new chunk
 *             has to be inserted
 * @ newnode : Pointer to the new chunk
 */

 void add_chunk_after_current (struct chunk* current, struct chunk* newnode) {
   if (current) {
     newnode->next = current->next;
     if (current->next) current->next->prev = newnode;
     current->next = newnode;
     newnode->prev = current;
   }
   else { ... // Add 'newnode' as the first node}

 }


int free (void * p) {

  /* Adds p as the second element on free list. */
  add_chunk_after_current (free_list_head, p);
  return 0;
}
```

Figure 6: Code for `free` in the target C library

Before the server makes its first allocation, the free list is as shown in Figure 4. The first free chunk of 128 bytes is at address `0x8049000`. The second free chunk is at 0x8049080. Notice that these two chunks happened to be placed adjescent to each other in memory (0x8049080 - 0x8049000 = 128).

- (2 points) Write psuedo-code for the `delete_from_list` operation (invoked in figure 5), to remove an element from the doubly-linked list. Your psuedo-code should have no more than 8 statements.

- (3 points) Draw the state of the free list at line number 18, using a diagram similar to figure 4. Show the base addresses of the chunks in the free list.

  Keep in mind that for this problem, all allocation requests are of size 100 and chunks are always 128 bytes – therefore, the function `find_first` in Figure 5 will always end up returning the current value of `free_list_head`.

  Hint : You will need to understand the implementations of `malloc` and `free` in figure 5 and figure 6.

- (4 points) Line 19 in Figure 2 has a buffer overflow that allows oveflows across chunks. If you look carefully, you can see that an attacker can exploit it to write a `NULL` value to *any* memory location of his choice. Show the contents of the input strings (passed as `argv[1]`), that exploits this bug to write a 4 byte value `0x00000000` at address `0xAAAAAAAA`.

  You should avoid showing irrelevant bytes in the input; for instance, you should use the notation "... 80 bytes ..." to denote 80 bytes of random values. But, you should show all the needed values for full credit. If you want to represent a hexadecimal byte value in ASCII, say `0xAA`, you should show the ASCII notation in your answer as `\xAA`.

  State the source line on Figure 2 where the exploit will be triggered. You may include an explaination, in no more than 8 sentences, for your approach.

  Hint : Look at the code you wrote in the first subpart, and identify an operation that gives the attacker a write-anything-anywhere capability, if she controls all values in chunk being deleted.

- (5 points) Suppose we fix the heap overflow on line 19 – we replace the line with :

  `snprintf(q, 99, "%s Welcome to your account, %s", argv[1], argv[2]);`.

  There is yet another bug – look carefully at the line 11, 18 and 26 in figure 2. What is the bug?

  An attacker can exploit it to write the value `0x8049000` to any memory location of his choice using this second bug. Show the contents of `argv[1]` necessary to write the value `0x8049000` to address `0xAAAAAAAA`.

  Hint : Analyze at the function `add_chunk_after_current` carefully, and find the statement that lets the attacker write a fixed value to an arbitrary location if she controls the contents of the two chunks passed in as parameters.

- (2 points) Exploiting the bug on line 26 requires the program control flow to execute statements on line 3, 4, 11, 12, 18, 19, 23, 24, and 26. State the constraints that the program inputs should satify, to cause the control flow to execute take the `true` branches at the conditional statements on line 17 and line 25. Note that, in general, these constraints could be very complex and extracting them manually may be hard in practice – this is why programmers are likely to miss such bugs during testing.

3 Defenses

1. *Canary Based Return Address Protection*. (3 points) Write the psuedo-code of a vulnerable function that is susceptible to return address corruption, even though the compiler uses a canary based return address protection scheme.

2. *Memory Layout Randomization*. (2 points) An OS vendor enforces a page based alignment for the heap, stack, and code sections of a program. On the 32-bit x86 machine, each page is 4096 bytes. The OS vendor employs memory layout randomization using only a runtime loader modification – each time a binary is loaded in the process address space, the loader selects a random page-aligned address for the code, data and heap segments.

   The attacker discovers an exploitable stack-based buffer overflow. The attacker decides to exploit it using a return-to-libc attack. He has to correctly guess the buffer start address, as well as the location of `execve` system call (which is the target of the control transfer in his attack). How many guesses are necessary, in the worst case, for the attacker to ensure a successful attack? Show how you arrive at your answer with an explaination at most 2 sentences long.