Homework 4

CS161 Computer Security, Spring 2008

Assigned 4/14/08

Due 4/21/08

NOTE: Questions 1-3 are based on real-world systems, and present a fair amount of background material. Although the question statements are long, you should be able to answer them in a few sentences. Question 5 is a programming assignment with a short answer question. Please submit the code for number 5 electronically. The rest of the assignment should be handed in at the beginning of class as usual.

1. **TOCTTOU**

   You're working on a revolutionary online social networking game for GRizzly Unix Experts and treasure hunters. The game environment consists of an $8x8$ map, which contains gold, hunters, grues and walls. Gold can be used to personalize the game environment by erecting or destroying walls between squares. When two players occupy the same square, the system lets the two players chat with each other.

   (a) (2 points)

   You've decided to jump on the casual gaming bandwagon. Early user studies have found that casual gamers get frustrated when their characters bump into newly created walls. Also, none of them ever wants to initiate conversations with GRUEs, and would rather have their clients automatically avoid walking into squares occupied by GRUEs (preventing GRUEs from walking into a square that contains a hunter would be antisocial, so you don't need to prevent that).

   A beta version of `hunt` added a few extra functions to the client-server API in order to achieve these goals:

   - bool isThereAWallInFrontOfMe()

- bool isThereAGrueInFrontOfMe()

Coupled with the existing API:

- faceNorth()
- faceSouth()
- faceEast()
- faceWest()
- moveForward()
- pickUpGold()
- putDownGold()
- buildWall()

these functions initially prevented any hunter-initiated GRUE contact on your network. However, within minutes, the GRUE's used their experience with TOCTTOU bugs in the UNIX filesystem API to circumvent the solution.

Extend the API with locks to remove any race conditions while preserving as much concurrency as possible. Be sure to tell us which clients are prevented from taking action by a lock, and which actions the lock(s) prevent.

This sort of lock is like a mandatory file lock; not like a mutex in a multithreaded program. Clients must manually acquire locks if they wish to prevent the action the lock protects against. Furthermore, once a lock is granted, actions that the lock prevents are guaranteed not to occur until the lock is released.

(b) (3 points)

Soon after locking is deployed, the online game grinds to a complete halt; apparently the GRUEs have decided not to release the locks you added in part (a). Provide a new API with the same atomicity guarantees (and functionality) as before, but do not allow client programs to acquire locks.

(c) (5 points)

With your changes, the game becomes a wild success! As the 256th player signs up, you realize the 8x8 map is too small to accommodate future growth. You expand the map to cover a 256x256 grid, but divide it into 8x8 tiles. When a user moves from one tile to another, their session is moved onto the server that owns the new tile.

Rather than allocating an entire server to each tile, you keep the contents of unoccupied tiles in files on a file server. If a user moves into an unoccupied tile, the server reads the file from disk, and decides that it owns the tile.

Once a second, each server broadcasts the list of tiles that it owns to the other servers. If two servers both think they own the same tile, one server transfers the appropriate user state to the other server, and deletes its version of the tile.

You login to the game the next morning, and discover that a pair of treasure hunters are doubling their money every few seconds. They already have more gold than the game world is supposed to contain!

How can a pair of players cheat and double their gold?

(d) (5 points)

A concerned GRUE suggests that you use conflict resolution to fix the gold doubling bug. The idea is to remember what actions each player takes in each copy of the tile, and then to automaticlaly fix problems after they occur.

How long do the servers have to track such information? What should they check for? What should they do if they discover a problem?

2. **Serialization**

Java has a number of built-in data encapsulation / protection features. In this question you will learn how to use serialization to circumvent them. Consider the following class:

```
class Tree {
  private Tree left; private Tree right;
  private int value;
  public Integer search(int i) {
    if(i == value) {
       return i;
    } else if(i < value) {
       if(left)
          return left.search(i);
       else
          return NULL;
    } else {
       if(right)
```

```
            return right.search(i);
         else
            return NULL;
      }
    }
    public Integer insert(int i) {
      ...
    }
}
```

Tree's constructors produce valid trees, and insert() maintains the proper invariants. Java's runtime prevents other classes from modifying Tree's private state. Therefore, one might assume that all instances of Tree are valid binary search trees.

(a) (5 points)

Assume the Tree class has been modified to support a custom serialization and deserialization format. The file format is text based, and looks like this:

```
// Declare objects
Tree a; // Reference to first object is
        // returned by deserialization
Tree b;
Tree c;
Tree d;
// Fill in object fields
b = { NULL, NULL, 1 }; // { left, right, value }
a = { b, c, 2 };
d = { NULL, NULL, 3 };
c = { d, NULL, 4 };
```

Produce a file that will cause search() to crash. Your file should only contain Tree objects, just like the example. The set of objects returned by deserialization should type check; all Tree fields should point to Tree objects or be NULL, and all integer fields should contain integers.

(b) (5 points)

Modify the serialization format to prevent your attack.

3. **ObjectInputStream**

Read the documentation for Java 1.5's ObjectInputStream:

`http://java.sun.com/j2se/1.5.0/docs/api/java/io/ObjectInputStream.html`

(a) (5 points)

Assume that an attacker is able to install a malicious java class on a server, but is not able to invoke methods from the class directly.

However, the server uses ObjectInputStream to read data from unauthenticated network connections.

Use this fact to gain complete control over the server's JVM. Describe your malicious class (include relevant method names, when appropriate), and explain how you would generate input for ObjectInputStream that would trigger the vulnerability.

(b) (5 points)

You've decided to implement a secure version of ObjectInputStream by overriding `resolveClass()`, which takes a class descriptor (read from the input stream), and returns the local version of the class. The idea is to throw an exception instead of letting potentially malicious code execute. Explain how your version of `resolveClass()` will work.

4. **Reasoning about code (15 points)**

   The following (poorly written) function ROT-13 encodes and prints the string passed into it. Prove its correctness using preconditions, postconditions and loop invariants.

   Note that the program is not memory safe, and does not behave correctly on all inputs. Therefore, you will need to choose preconditions on the input in order to make your proof go through. Similarly, you should document any side-effects of the function using postconditions.

   You will also need to document the preconditions and postconditions of library functions.

```
void printRot13(char * c) {
  c = strdup(c);
  if(!c) { abort(); }
  size_t i = strlen(c) + 1;
  c+=i;
  while(i--) {
    c--;
    if(*c >= 'A' && *c <= 'Z') {
      *c = 'A' + (*c-'A'+13) % 26;
    }
  }
  printf(c);
  free(c);
}
```

5. **Buffer overflow exploit (25 points)**

   For this problem, you will write an exploit for a buffer overflow vulnerability. To get started, read over Aleph One's "Smashing the Stack for Fun and Profit".[1] (You don't have to read the section "Shell Code", since we provide you with shellcode, though you may find it interesting.)

   The vulnerable program is `/home/ff/cs161/hw4-s08/targets/target` on the instructional machines. Copy the directory `/home/ff/cs161/hw4-s08/exploits` to your working space; it contains skeleton code and a Makefile for your exploit program.

   Your task is to edit `exploit.c` so that it exploits the buffer overflow vulnerability in `target` to run a shell. We provide exploit code in

[1] `http://reactor-core.org/stack-smashing.html`

`shellcode.h`; you just have to cause it to be executed in `target`. If you are successful, you should see a '$' shell prompt:

```
bash-3.1$ ./exploit
$
```

The only file you should edit is `exploit.c`. Build it with `gmake`. The path to `target` is hard-coded in `exploit.c`; please do not change it.

Because buffer overflow exploits are highly machine-dependent, you are restricted to working on **sphere.cs**, **rhombus.cs**, or **pentagon.cs**. Your exploit must work on one of those machines (they are Solaris x86 boxes).

To start with, we recommend that you use `gdb` to explore the stack and memory layout of `target`. It will be different when called via `execve()`, so here is the best way to get set up (after running `gdb exploit`):

```
(gdb) run
Starting program: /home/ff/cs161/hw4/exploit/exploit

Program received signal SIGTRAP, Trace/breakpoint trap.
0xce7cd062 in ?? ()
(gdb) symbol-file /home/ff/cs161/hw4-s08/targets/target
Load new symbol table from "/home/ff/cs161/hw4-s08/targets/target"? (y or n) y
Reading symbols from /home/ff/cs161/hw4-s08/targets/target...done.
warning: rw_common (): unable to read at addr 0xce7ac660
warning: sol_thread_new_objfile: td_ta_new: Debugger service failed
(gdb) break main
Breakpoint 1 at 0x80507d7: file target.c, line 12.
(gdb) continue
Continuing.

Breakpoint 1, main (argc=2, argv=0x8047f34) at target.c:12
12          if (argc != 2) {
(gdb)
```

Running it this way makes it difficult to restart, however, so you may want to just run `gdb target` to explore initially and then switch to the `execve()` version when it's time to find the actual addresses for your exploit.

You will want to become familiar with the following `gdb` commands (use the 'help' command): `break`, `where`, `disassemble`, `next`, `nexti`, `x`, and `info`. Be sure to explore the display options for the `x` command.

You should not follow Aleph One's directions too closely. You may or may not want to execute the shellcode on the stack, and you can use `gdb` to figure out the exact address to jump to, so you don't have to use anything like `get_sp()` or NOP padding.

You must submit your code electronically. Go to the directory where `exploit.c` resides and type `submit hw4`. You should only submit exploit.c; you should not change the other files.

You must ensure that your code runs on one of the three servers listed above. We should be able to type `gmake` and then `./exploit` to run your exploit.

You should include in your homework writeup a brief description of how you tackled this problem, including how you determined which address to jump to. Please tell us which server you ran your code on (sphere, rhombus, or pentagon). The writeup for this question should be no more than 6 sentences.

Do not forget to list students you worked with for this homework. As always, you may discuss problems with other students but you may not share writing (including code).