Computer Science 4271
Fall 2023
Midterm exam 2 (solutions)
November 16th, 2023
Time Limit: 75 minutes, 4:00pm-5:15pm

- Before starting the exam, you can fill out your name and other information of this page, but don't open the exam until you are directed to start. Don't put any of your answers on this page.
- This exam contains 7 pages (including this cover page) and 4 questions. Once we tell you to start, please check that no pages are missing.
- You may use any textbooks, notes, or printouts you wish during the exam, but you may not use any electronic devices: no calculators, smart phones, laptops, etc.
- You may ask clarifying questions of the instructor or TAs, but no communication with other students is allowed during the exam.
- Please read each question carefully before answering it. Remember that we can only grade what you write on the exam, so it's in your interest to show your work and explain your thinking within the available time and space.
- By signing below you certify that you agree to follow the rules of the exam, and that the answers on this exam are your own work only.

The exam will end promptly at 5:15pm. Good luck!

Your name (print): ______________________________

Your UMN email/X.500: ______________________________`@umn.edu`

Number of rows ahead of you: __________ Number of seats to your left, to an aisle: __________

Sign and date: ______________________________

Question	Points	Score
1	38	
2	20	
3	12	
4	30	
Total:	100	

1. (38 points) SQL injection

   Each of the following scenarios presents Java code that constructs a SQL database statement by using a string `input` in an unsafe way. Your task is to choose a malicious value for the `input` variable that will achieve an attacker's goal via SQL injection. In each case the SQL query is constructed by concatenating constant strings (Java strings use similar syntax as C) and variables using the `+` operator. Some reminders about SQL syntax are at the end.

   (a) One query is constructed as follows:

   ```
   String query = "SELECT name, roomnum FROM Guests " +
                  "WHERE name = '" + input + "';";
   ```

   This query retrieves information about a guest staying in a hotel. But there is another table in the same database, named `Evidence`, which contains information about your bad behavior. Choose a value for `input` that will modify the SQL to remove that table using SQL's `DROP TABLE` command.

   This is the only attack that needs to introduce a separate statement, since dropping a table is very different from a SELECT. The injection needs to have a single quote near the beginning to close the string that was started by the single quote in the constant. We use semicolons to terminate both the SELECT statement and the DROP TABLE statement. You also have to do something about the single quote that will be added after the injection. Probably the easiest thing here is to comment it out with `--`.

   ```
   x'; DROP TABLE Evidence; --
   ```

   (b) Here the vulnerable query is the same as in the previous part, but your goal is different. Instead you'd like to get information about all the guests (i.e., all rows of the table). Choose a value for `input` that will achieve that.

   The question was intended to set up the most classic kind of injected tautology to turn the WHERE clause into something that is always true. This more elegant, and will work in more circumstances, than adding an additional SELECT statement, though we did give credit for that too. You could terminate the injection with a comment as in the previous part, but something else you can do here is use the final single quote as part of your tautology.

   ```
   x' OR '1' = '1
   ```

(c) This query is more complex, with three concatenated variables, but assume that the other two variables are properly sanitized:

```
acct = sanitizeAcctNum(acct);
myname = sanitizeName(myname);
String query = "UPDATE PayableInfo SET acctnum = " + acct +
               " WHERE name = '" + myname +
               "' AND ssn = " + input + ";";
```

This query is used in a corporate HR system to update the information used for direct deposit of employees' salaries. In normal operation, you would be allowed to update the bank account number used for your pay, as long as you showed that you knew your Social Security number (`ssn`, stored as an integer). For your attack, your goal is to change the information for your co-worker Bob (his `name` in the database is just `Bob`) so that his paychecks are sent to your account instead. In your attack, the `acct` and `myname` variables will still hold your account number and your name, respectively; you can't change them. Instead, you must achieve your attack by changing only the `input` variable in this query.

Our preferred solution here is to broaden the WHERE clause by adding Bob's name as a disjuction. This is similar in structure to the previous attack, but you want to add just Bob to the matches and not everyone. In the context of this example it's not important to keep the original user from matching, because we are updating with the attacker's account information which should already be there. So for the Social Security number you could use the attacker's own SSN, or a value that shouldn't be a valid SSN. Note that just using a nested query to find Bob's SSN wouldn't be a complete solution, since the combination of `myname` *and Bob's SSN won't match any rows.*

```
-1 OR name = 'Bob'
```

(d) This query uses features to limit the size of the results that are returned:

```
String query = "SELECT username from Users " +
               "WHERE (uid = " + input + ") LIMIT 1;"
ResultSet results = stmt.executeQuery(query);
if (results.length() > 1) { throw new InvalidQuery(); }
```

This query converts a numeric user ID `uid` into a string username. Because user IDs are supposed to be unique, this query should yield at most one result row. The `LIMIT 1` clause enforces this so that even if the query matches multiple rows, only the first will be returned by the database. Furthermore, it is also checked by the Java code, which will raise an exception (and not print any data) if it gets more than one row from the database.

It turns out that this database also contains Social Security numbers, 9-digit numbers stored as integers in a column named `ssn`. You are a budding identity thief, and you have the partial information that the SSN of one user of the system is of the form `47213xxxx`, where you know the first five digits but not the last four. You goal is to use SQL injection to change this query into one that returns the username of the user who matches the information you know about their SSN. You can assume only one user has an SSN of that form, but you don't know their user ID either.

In contrast to the previous part, here it is important that the SELECT matches only one row, lest the row the attacker is interested in be dropped by the LIMIT 1. Our preferred way of doing this is to turn the existing part of the condition into an always false condition by ANDing it with a false value, which is dual to making something always true with OR and a tautology. AND has higher precedence than OR, so additional parentheses aren't needed to get the right grouping, but observe that as another instance of injection attacks not respecting grammar, the pair of parentheses in the constant parts of the query don't have to match each other. You could also try to guess a user ID value that is not used by any current user, but you don't have much context as to what values would work here. Note that if these are Unix UIDs, the value of 0 is valid.

We said that the SSN column was an integer because we had in mind doing the matching with arithmetic, such as comparing the bounds of a range; you could also use division. We also gave credit for using SQL string wildcards with LIKE and `____` *or* `%`.

```
1 AND 1 = 2) OR (ssn >= 472130000 AND ssn <= 472139999
```

We are following the convention of using `ALLCAPS` for SQL keywords, `CamelCase` for the names of tables, and `lowercase` for the names of columns. Constant strings in SQL are delimited by single quote marks. SQL can contain comments that are enclosed between `/*` and `*/`, as in C, or that start with `--` and go until the end of the line. SQL statements should properly be terminated with semicolons, as in C, though the server used here won't complain if a semicolon is missing at the very end of a query. The comparison operators in SQL are `=`, `<>`, `<`, `>`, `<=`, and `>=`. They can all be used on numbers, and the first two can also be used on strings. The logical operators in SQL, in order of decreasing precedence, are named `NOT`, `AND`, and `OR`.

2. (20 points) Matching Unix files and permissions

On the left are 8 sets of permissions for Unix filesystem objects, with the permission bits shown in octal, followed by the owner (always `root` in these examples), and the group owner. On the right are 10 pathnames and descriptions of Unix filesystem objects (e.g., files and directories). Match the objects on the right with their appropriate permissions from the left by writing the corresponding uppercase letter in each blank. In our model solution, which is based on pretty normal Ubuntu 22.04 Linux machine, all of the choices on the left are used at least once, and two are used twice.

A. `0440 root root`
B. `0644 root root`
C. `0660 root disk`
D. `0666 root root`
E. `0755 root root`
F. `1777 root root`
G. `2755 root games`
H. `4755 root root`

(a) __**C (A)**__ `/dev/sda`, representing direct access to a hard disk

(b) __**F**__ `/tmp`, a directory for anyone's temporary files

(c) __**E**__ `/bin/cp`, a utility program

(d) __**G (E)**__ `/usr/games/robots`, a game with a high-score file

(e) __**B**__ `/var/log/dpkg.log`, a log of package installations

(f) __**H**__ `/bin/su`, a setuid utility program

(g) __**A (C)**__ `/etc/sudoers`, a security-sensitive configuration file

(h) __**B (D)**__ `/etc/timezone`, a non-private configuration file

(i) __**E (H)**__ `/usr/bin`, a directory of system programs

(j) __**D (F)**__ `/dev/null`, writes to which are discarded

We gave full credit only for the answers that matched the real machine we used in developing the question, shown without parentheses above. A suggested process of elimination to narrow down to that mapping is to start by separating the non-executable (even digits) and executable/directory (odd digits) permissions. Then match the two permissions with distinct group names, and the ones with the special sticky, setgid, and setuid bits. Most of these system files and directories are not globally writable: the exceptions are `/dev/null` *(where writing needs to be allowed but has no effect) and* `/tmp` *(where it's managed using the sticky bit). The designers of the* `sudo` *utility suggest that its configuration file* `/etc/sudoers` *should be kept secret, probably because on a multi-user system it could tell an attacker which users are most valuable to attack. The global timezone setting not being modifiable by a normal user also reflects a multi-user system: more like CSE Labs workstations than your laptop. (You can override it for your own programs with an environment variable.)*

We gave 1/2 partial credit for what we felt were the most defensible additional answers, shown in parentheses.

3. (12 points) Multiple choice. Each question has only one correct answer: circle its letter.

   (a) Which of these vulnerability types is a kind of race condition?

   A. Double free

   B. Stack buffer overflow

   C. Response splitting

   D. TOCTTOU

   E. Insecure file permissions

   We can specifically describe time-of-check-to-time-of-use vulnerabilities as race conditions in which the attacker races to make a modification to a shared object in between the separated check and use.

   (b) The "MIFARE Classic" is a model of chip that was widely used in public transit smart cards since its introduction in the 1990s, but whose core cryptographic primitive was shown to be insecure in research published in 2008 and 2009. What was this primitive?

   A. Triple DES

   B. AES-128

   C. A proprietary shift-register stream cipher

   D. DES

   E. XORing each byte with `0x3c`

   Though I didn't mention it by name in lecture, this smart card family is one of the most high-profile recent examples of a shift-register-based stream cipher that turned out to be insecure. Among the other encryption primitives mentioned, none of the other have changed their security status recently. XORing with a fixed byte is a poor idea for the same reasons as a Caesar cipher. DES and Triple DES have no notable cryptanalytic vulnerabilities, though they are considered obsolete because of their small key sizes. AES-128 is the most commonly used block cipher and has no known practical weaknesses. In fact, DES and AES were used by the manufacture of the MIFARE Classic in successor smart-card chips that are now recommended.

   (c) Which of these vulnerability types is a kind of injection?

   A. Insecure use of temporary files

   B. XSS

   C. History stealing

   D. Uninitialized use

   E. Use after free

   Cross-site scripting is essentially injection of HTML and/or JavaScript.

   (d) Which of these is **not** a way to virtualize a program's interface with the operating system?

   A. VMware B. `chroot(2)` C. Solaris Zones D. FreeBSD jails **E. JavaScript**

   Zones and jails are brand names for kinds of OS containers; like a virtual machine like VMware, they give software the illusion that it has the operating system or its operating system in complete control of a computer, when in reality the computer is shared. The `chroot` *system call does something similar but just for the filesystem: programs running in a* `chroot` *have a virtual root directory that is actually just a subdirectory as seen outside the* `chroot`*. By contrast, JavaScript as used in web browsers just does not provide OS interfaces, since JavaScript is intended to run entirely inside the browser.*

4. (30 points) Matching definitions and concepts. Fill in each blank with the letter of the corresponding answer. Each answer is used exactly once.

   (a) __**L**__ Standard user name for UID 0

   (b) __**J**__ When a results depends on which of two programs executes first

   (c) __**T**__ A character encoding with synonyms for ASCII symbols

   (d) __**N**__ Transforming data to prevent its being useful for an attack

   (e) __**D**__ Tricking a user so their UI actions have an unintended effect

   (f) __**S**__ A file referring to another file system object by path

   (g) __**G**__ Multiple directory entries referring to the same inode

   (h) __**I**__ XORing a plaintext with fresh random bits

   (i) __**A**__ Used for escaping in C strings

   (j) __**B**__ E.g., trying all possible encryption keys one by one

   (k) __**H**__ A virtual machine monitor running under a normal OS

   (l) __**Q**__ Makes permissions on `/tmp` work differently

   (m) __**M**__ A modern version of the Caesar cipher

   (n) __**F**__ A command-line utility that shows extended permissions information

   (o) __**E**__ A declarative formatting language used with HTML

   (p) __**R**__ CPU flag set when an OS kernel executes

   (q) __**K**__ An idealized model of a cryptographic primitive

   (r) __**P**__ A system call to get metadata about a file

   (s) __**O**__ A command-line utility that shows the results of `stat(2)`

   (t) __**C**__ A command-line utility to change file permissions

A. \ (backslash) B. brute-force C. `chmod` D. clickjacking E. CSS F. `getfacl`
G. hard link H. hypervisor I. one-time pad J. race condition K. random oracle
L. `root` M. ROT-13 N. sanitization O. `stat(1)` P. `stat(2)` Q. sticky bit
R. supervisor bit S. symbolic link T. UTF-7

Since there were 20 parts and 30 points, each part should mathematically be 1.5 points. Depending on who graded your exam, you may see fractions listed in some places, but we always rounded the score up (i.e., in your favor) to a whole number before computing the final score. We also have half credit for some of the most plausible close-but-not-quite answers.