CX4067 2017-2018 Semester 2

Question 1

a) Meta-characters: characters with special meanings
‘: string terminator
#: comment

SQL injection:
SELECT * FROM users WHERE username=’’ OR 1=1;#’ AND password=’anything’;
(underlined is attacker input)

b) Escape characters are meta-characters that change the meaning of subsequent characters.

Usually used when meta-characters must be treated normally, eg if you wanted to store a string with a quote, UPDATE users SET description=’can\’t see me’;

SQL uses a backslash as the escape character.

See Lec7, Slide 20 (Rev 17/2/22)

c) Filters or sanitization functions looking for specific characters may only detect the UTF-8 character and miss the alternative, longer, Unicode encoding of the same character. Inputs can be canonicalized before processing to ensure that characters are of the expected form.
See Lec7, Slide 34 (Rev 17/2/22)

Question 2

a) Data Execution Prevention: mark memory that a process has written to as non-executable (See Lec3, Slide 43 (Rev 26/1/22))

   Address Space Layout Randomization: randomise position in memory of various regions like stack, heap, libraries and process code (See Lec3, Slide 44 (Rev 26/1/22))

   DEP has no impact on return-to-libc attacks. (*DEP only prevents ancient attacks like writing shellcode into memory, eg through a buffer overflow*)

   ASLR makes it harder to perform a return-to-libc attack because the attacker must first leak the position at which libc was loaded into memory, but does not fully protect against such an attack.

b) Return-oriented-programming abuses the presence of gadgets (whether in process code or other library regions), which are a series of instructions followed by a `ret` (*assembly, somewhat equivalent to pop eip*). This allows an attacker to write and execute arbitrary shellcode (*akin to BOF and writing in shell code, except not hindered by DEP/NX since the attacker only writes data into the stack*) because the attacker only writes addresses and data, not actual code.

   See Lec3, Slide 48 (Rev 26/1/22)

   Practical examples like https://codearcana.com/posts/2013/05/28/introduction-to-return-oriented-programming-rop.html might be easier to understand

c) Jump-oriented-programming relies on an attacker-controlled dispatch table somewhere in memory, and a few gadgets that advance a “pointer” into the dispatch table and jumps accordingly.

   JOP is harder to defend against because it is more flexible than ROP. The list of instructions can reside anywhere in memory, rather than only in the stack. Any register can be used to as a “program counter” rather than only ESP.

   See Tut3, Slide 15 (Rev 31/1/22)

Question 3

a) Type safe languages guarantee the absence of untrapped errors, ie no issues like stack overflow or out of bounds reading of arrays. However, the issue is not magically resolved but has to be handled gracefully by the programmers instead.

   See Lec12, Slide 3 (Rev 7/4/21)

b) Race condition: multiple simultaneous computations that access the same shared memory such that the end result depends on the order of access. (Lec 6, Slide 15 (Rev 17/2/2022))

   Type safety cannot detect or flag race conditions. It only guarantees that operations performed on an object are compatible with its type, but race conditions can still occur because of execution order.

   For example, Python is a high level, type safe language, but race conditions can still occur eg time-of-check-time-of-use:

   - Checks ownership of a file on the file-system
   - Modifies the file's permissions

   If the file is swapped or the ownership changes between step 1 and 2, unintended side effects (race condition) could occur.

   More concise way of justifying?

c) Direct modification of types can occur through multiple means:
   1) Buffer overflow allowing overwriting of underlying structure
   2) Modification of compiled representations eg modifying of Java bytecode file to redefine array types (See Lec12, Slide 27 (Rev 7/14/21))

Question 4

a)

1. Grade Tampering (Tampering with Data)
    1.1. Attacker obtains valid administrative credentials
        1.1.1. Reuse of credentials from compromised sites
        1.1.2. Phishing
        1.1.3. Brute force
            1.1.3.1. Login page has no rate limiting
    1.2. Attacker tampers with legitimate traffic
        1.2.1. HTTP traffic is unprotected

b)

Test Case	uid	pwd	cid	format
1	1	1	1	1
2	1	2	2	2
3	1	3	3	3
4	2	1	1->2	1->3
5	2	2	2->3	2->1
6	2	3	3->1	3->2
7	3	1	1->3	1->2
8	3	2	2->1	2->3
9	3	3	3->2	3->1

Column 1 and 2 were filled out so that all pairs were present (1-1, 1-2, 1-3, … 3-2, 3-3)

Column 3 was then filled out in running order (1, 2, 3, 1, 2, 3, 1, 2, 3). This forms all pairs between Column 1 and 3, but we see that not all pairs are formed between Column 2 and 3. As such, test case 4-6 were shifted by one place, while test case 7-9 were shifted by two places, forming all pairs between Column 1 and 3, and 2 and 3.

Same logic applied when filling out Column 4.

Question 5

a) Line 11: XPath injection
Line 22: XSS
Line 26: Leaking of verbose stack trace on error (See Lec8, Slide 49 (Rev 17/3/21))

b) http://www.abc.com/enter_grade?cid=a&pwd=0 or @cid='CS4061'&suid=John11&grade=A

Possible ambiguity due to operator precedence if query manipulated into
@cid=CS4061 and @lecturer_pwd='' or 1=1
which would select multiple courses, which is why I chose to insert yet another condition

c)

1. Parameterised queries
2. Sanitization/escaping
3. Validation

Example: ensuring that lecturer_pwd matches regular expression ^[0-9]{6}$