COMPSCI 750 – Assignment 4
Gunnar Yonker

1.

(1) Yes, the provided Set-UID program has a race condition vulnerability, specifically a Time-of-Check to Time-of-Use vulnerability. This is because the access() checks if the real user has write permission on /etc/passwd and if the user does not have write permission, the program proceeds to open a file /tmp/X and write to it. The vulnerability arises due to the time gap between the check through using access and the use when using open(). An attacker can exploit this gap to perform malicious operations. An example of how this could be done is that an attacker could create a symbolic lin from /tmp/X to another file just after access() checks the permissions and just before open() open the file. If the Set-UID program is running with root permissions, then when the program writes to /tmp/X, it will instead write to the file that the link points to, possibly damaging the system or overwriting critical files.

(2) This version does not have the same race condition vulnerability as the previous code. The reason is that by the time we are checking the file descriptor with faccess(), we have already opened the file. The use of file descriptors provides a more concrete reference to the file, making it resistant to attacks that exploit the name/path. The vulnerability associated with the gap between checking and using the file is largely eliminated because we are no longer relying on the pathname, but rather the already-opened file descriptor. As such, there is no window for an attacker to swap out the file or modify its state based on its path after the check but before the action. The use of faccess() makes this program safer when compared to the previous program above.

2.

The issue with this program is that the race condition is introduced where the permissions or existence of the file could change between the call to access() and the call to open(). This is when a user could exploit this race condition to trick your program into writing to a file that it shouldn't be writing to. To fix this race condition we need to avoid the separate check-then-act logic and try to open the file directly. This way the window of time between checking the file permissions and acting on it are eliminated. This is in line with the principle of least privilege because you're only attempting the action you need, rather than checking permissions explicitly and then acting.

```
f = open("/tmp/XYZ", O_WRONLY);


if (f !=  -1) {
   write_to_file(f);
} else {
   // Handle the error
   fprintf(stderr, "Permission denied or other error occurred\n");
}
```

This modified code will try to open the file and if the file doesn't exist or if there are insufficient permissions, the open() call will fail, and we can handle the error if need be. There is no separate checking phase, which means there's no window for a race condition to occur.

3.

C:

Buffer Overflows: Always use bonded functions to prevent buffer overflows. Prefer library functions that manage memory safely.

Dynamic Memory Management: Always check for allocation failures and ensure to free all dynamically allocated memory to avoid memory leaks. Avoid using gets(), which is inherently unsafe.

Integer Overflows: Check for integer overflow/underflow before arithmetic operations.

Format String Vulnerabilities: Avoid using non-constant format strings in functions like printf().

Initialization: Always initialize variables. This avoids undefined behaviors.

Use Safe Libraries: Favor the use of libraries designed with security in mind.

Avoid Dangerous Functions: Some functions are historically problematic and should be replaced with safer alternatives.

Type Safety: Ensure that you don't make assumptions about data types, especially when reading data from external sources.

Input Validation: Validate all input from untrusted sources.

Compiler Warnings: Compile code with the highest warning level and fix warnings.

Java:

Object Immutability: Make classes and objects immutable whenever possible.

Avoid Exposing Internal Representations: Return copies of mutable objects from accessor methods.

Input Validation: Always validate input, especially if it's from an untrusted source.

Exception Handling: Don't disclose sensitive information in exceptions.

Deserialization: Be cautious with deserialization to prevent arbitrary code execution. Always whitelist allowed classes.

Secure Class Loading: Be aware of the source of classes and JARs using secure classloaders when needed.

Safe Synchronization: Use proper synchronization to prevent race conditions.

SQL Injection: Always use prepared statements or parameterized queries.

Avoid Storing Secrets in Code: Secrets or keys should not be hard-coded.

Secure Session Management: Ensure sessions are securely managed, and session IDs are unpredictable.

Python:

Use Latest Version: Always use the latest Python version to benefit from the latest security patches.

Avoid eval() and exec(): These can execute arbitrary code, making them a major security risk if not handled with care.

Input Validation: Sanitize and validate all inputs, especially if they come from external sources.

Use Parameterized Queries: Prevent SQL injection by using parameterized queries.

Safe File Operations: Be cautious with file operations, especially if the filename or path is derived from user input.

Permissions: Run your Python script with minimum necessary permissions.

Dependency Management: Ensure third-party libraries are up-to-date and from trusted sources.

Limit Information Disclosure: Handle exceptions so that they don't reveal sensitive system information.

Use Secure Channels: If transmitting sensitive data, use secure channels like HTTPS, SSL/TLS.

COMPSCI 750 – Assignment 4
Gunnar Yonker

4.

Test Case 149145

https://samate.nist.gov/SARD/test-cases/149145/versions/2.0.0

CWE-120 Buffer Copy Without Checking Size of Input (Classic Buffer Overflow)

Original Code:

```
#include <stdlib.h>

#include <string.h>


#define MAX_SIZE 10


int main(int argc, char *argv[])

{

	char str[MAX_SIZE];

	// Often Misused String Management:

	// Buffer overflow with strcpy function

	if (argc > 1)

		strcpy(str, argv[1]);                                   /* FLAW */

	return 0;

}
```

This test case is a classic example of a buffer overflow vulnerability due to misuse of the strcpy() function, especially when dealing with external input (in this case, command line arguments via argv). The program defines a buffer str of size MAX_SIZE which is 10 bytes. The program then takes an external input from the command line argv[1] without checking its length. It copies the input directly into the buffer using the strcpy() function, which doesn't perform any bounds checking. This means if the input string is longer than 9 characters (the 10^{th} being reserved for the null terminator), the buffer str will overflow.

Solution:

The solution is to use a function that performs bounds checking or to manually ensure that the input size does not exceed the buffer's capacity before copying. One potential solution is to use strncpy(), which copies up to a specified number of characters. If the source string is longer than the specified limit, then

it will not copy the remaining characters. However, note that if the limit is reached before the source string is fully copied, strncpy() will not null-terminate the destination string.

Modified Code:

```
#include <stdlib.h>

#include <string.h>


#define MAX_SIZE 10


int main(int argc, char *argv[])

{

	char str[MAX_SIZE];

	if (argc > 1)

	{

		strncpy(str, argv[1], MAX_SIZE-1);  // Copies up to MAX_SIZE-1 characters

		str[MAX_SIZE-1] = '\0';		// Ensure null-termination

	}

	return 0;

}
```

The test case we reviewed provides a reminder of the pervasive risks associated with handling strings in C, particularly when incorporating external inputs. There are a few important takeaways to keep in mind from this test case:

Always operate under the principle that all external inputs could be malicious. In this case, command line inputs, represented by argv[], were directly consumed without verification. Anytime we interface with external sources, such as user inputs, files or network packets, skepticism is our best defense.

The strcpy() function's inherent vulnerability stems from its lack of bounds checking. Whenever data is transferred between buffers, it's imperative to ensure that the source data does not exceed the capacity of the destination buffer. Failure to do so can result in overflows, potentially leading to unexpected behaviors, data corruption, or in some cases, an attacker gaining control over the execution of a program.

COMPSCI 750 – Assignment 4
Gunnar Yonker

While the C language provides a vast array of library functions, not all are safe to use without caution. Functions such as strcpy() have safer alternatives like strncpy(). However, it is also crucial to understand the quirks of these “safer” functions.

Buffer flows are exploitable. It’s not just about preventing a program crash. Buffer overflows can be, and frequently are, exploited by attackers to inject malicious code or alter program flow. By overwriting critical memory locations, such as the stack or function return addresses, attackers can gain undue influence over the program’s behavior, potentially leading to full system compromises.