**INDIVIDUAL ASSIGNMENT**

**ADVANCED SOFTWARE SECURITY**

**HAND OUT DATE : 12 MARCH 2024**

**HAND IN DATE : 17 APRIL 2024**

**NAME : DANISY EISYRAF BIN YUSNI AFINDI**

**Table of Contents**

[1.0 Introduction 4](#_Toc164271692)

[2.0 Software Development 5](#_Toc164271693)

[2.1 Introduction of Named Pipes in Win32 API (Windows Based Machine) 5](#_Toc164271694)

[2.2 Overview of User-End CLI 7](#_Toc164271695)

[2.2.1 Successful execution of all unit 7](#_Toc164271696)

[2.2.2 Error execution: Invalid Credentials 9](#_Toc164271697)

[2.2.3 Error execution: Failed to update status to pipe 12](#_Toc164271698)

[3.0 Software Vulnerabilities 13](#_Toc164271699)

[3.1 Race Condition Vulnerability 13](#_Toc164271700)

[3.1.1 Race Condition in IPC 13](#_Toc164271701)

[3.1.2 Exploiting IPC Race Condition for Eavesdropping 14](#_Toc164271702)

[3.2 Resource Exhaustion Vulnerability 18](#_Toc164271703)

[3.2.1 Pipe Starvation to cause DOS 19](#_Toc164271704)

[3.2.1 Pipe Flooding DOS 22](#_Toc164271705)

[3.3 SQL Injection Vulnerability 25](#_Toc164271706)

[3.3.1 Example of SQL Injection 25](#_Toc164271707)

[3.3.2 SQLi to Bypass Authentication 26](#_Toc164271708)

[4.0 Secure Coding Implementation 28](#_Toc164271709)

[4.1 Multi-Threading Coding Concept 28](#_Toc164271710)

[4.1.1 Implementing Multi-Threading in IPC 29](#_Toc164271711)

[4.1.2 POC: Mitigating Race Condition Vulnerability 33](#_Toc164271712)

[4.2 Process Authentication via Security Descriptor by Leveraging ACL 34](#_Toc164271713)

[4.2.1 POC: Authorized Users Only 35](#_Toc164271714)

[4.3 Parameterized Queries to Mitigate SQLi 38](#_Toc164271715)

[4.3.1 POC: Parsing Values Through Placeholders 38](#_Toc164271716)

[5.0 Conclusion 41](#_Toc164271717)

[6.0 References 42](#_Toc164271718)

# 1.0 Introduction

Software security is a critical factor in software development to protect the confidentiality, integrity, and availability of the software, data, and services that it processes (Julien Delange, 2022). Confidentiality ensures that sensitive information remains accessible only to authorized individuals or systems, preventing unauthorized disclosure or exposure of data. Integrity safeguards the accuracy and consistency of data, ensuring that it remains unaltered and trustworthy throughout its lifecycle. Availability guarantees that information and resources are accessible and usable whenever needed, minimizing downtime and ensuring uninterrupted operations.

The CIA triad holds significant importance within the information security domain and finds application in widely recognized standards such as ISO 27001, which serves as a global benchmark for information security management (Luke Irwin, 2023). By adhering to the principles of confidentiality, integrity, and availability, organizations can establish robust security measures to protect against various threats and vulnerabilities, including malicious attacks, data breaches, and system failures. Implementing security practices such as encryption, access controls, and regular audits helps reinforce the CIA triad and enhances the overall security posture of software systems, ensuring the protection of sensitive information and the continuity of business operations.

# 2.0 Software Development

For this module assignment context, several assumptions have been made in order to create a detailed context on where the vulnerability would practically exist and exploited. These assumptions include that the host running the code and services is in a Windows environment. In addition, on the back-end side assumptions, this assignment assumes that a simple SQL database have been set up named as “ASC\_Database” as well as a server connection to the database, all configured locally. The full source code for the vulnerable system, malicious exploits, as well as the patched system has been uploaded to <https://github.com/DanisyEisyraf/Vulnerable-IPC.git> for a detailed reference.

2.1 Introduction of Named Pipes in Win32 API (Windows Based Machine)  
For this module assignment, the system developed is a command line (CLI) based interface as it is a proof of concept (POC) of an actual unsecure coding that can be implemented realistically regardless of whether it is embedded behind a GUI or CLI. The developed unsecure coding mainly revolves around Windows API (Win32 API) Inter Process Communication (IPC), namely *ConnectNamedPipe*. Win32 API is the primary APIs for developing 32-bit windows applications that are responsible for multiple functions, that includes:

* Administration and Management
* Diagnostics
* Graphics and Multimedia
* Networking
* Security
* System Services
* Windows User Interface

Inter-process communication (IPC) is an essential concept in OSes that allows processes to exchange data and synchronize activities. It allows different processes to work together, share resources, coordinate operations to achieve the developer’s goals (Vishal Sharma, 2023). One popular implementation of IPCs in Windows based host is ConnectNamedPipe function inside the Win32 API.

ConnectNamedPipe (documented as Named Pipes) is a multi-way pipe communication between pipe server and one or more pipe clients (Microsoft, 2024). It is a versatile means of IPC where multiple instances can coexist, sharing common pipe names while maintaining individual buffers and handles. This architecture enables concurrent usage of the same named pipe by multiple pipe clients.

Subject to security level, with named pipes, any process can engage in communication, enabling efficient data exchange among related or unrelated processes. Furthermore, the inherent flexibility allows any process to assume roles as both server and client, fostering peer-to-peer communication. In this context, a pipe server, responsible for creating a named pipe, employs functions such as CreateNamedPipe, while a pipe client, initiating connections, utilizes functions like CreateFile, WriteFile, and ReadFile.

## 2.2 Overview of User-End CLI

### 2.2.1 Successful execution of all unit

To execute the software, users can either run it via an IDE such as Visual Studio environment, or directly compile the software and run the respective executable file. Upon executing, the user is prompted with the following windows box and the user should enter their credentials.

A screen shot of a computer

Description automatically generated

A screen shot of a computer

Description automatically generatedUpon entering the required information, the credentials will be sent to a local pipe that is expecting a client handler connection to accept the data.

A screenshot of a computer

Description automatically generatedTo simulate a real world scenario, another code will be executed namely *client\_handler.py* to act as a pipe client to handle all incoming and outgoing data from the main process itself. Upon running the client handler, the following information will be prompted (assuming the credentials exist):

A screenshot of a computer screen

Description automatically generatedWhereas the main pipe server code will return a prompt stating the connection to the pipe has been established, credentials and status are sent through the pipe.

Back the client handler process, a constant information about the pipe server is prompted every 2 seconds. The information includes the TP number the user entered, device name, the process ID (PID), the date and time that the information was sent, as well as the status of the server.

A screenshot of a computer

Description automatically generated

The information will be constantly updated until the user manually ends the program by keyboard interrupt (CTRL + C).

### 2.2.2 Error execution: Invalid Credentials

A screen shot of a computer

Description automatically generatedAt the first stage, if the user submits an invalid TP number ID and password whereby the credential does not match with any inside the ASC\_Database database table of “Users”, the main pipe server will inevitably submit the credentials to the client handler. However, the client handler upon checking the credentials will return whether the user exist or not. This will prompt the main pipe server that the user does not exist.

The program then will be aborted. To re-enter credentials, simply re-run the same file again with the right existing credentials. For the user’s accessibility, the credentials can be found inside the ASC\_Database under the Users table which can be directly accessed under SQL Server Management Studio (SSMS).

#### A screenshot of a computer Description automatically generatedC*hecking available credentials in the database*

Open the SSMS and connect to the database via the SQL Server. The server name, database names, authentication, all are subjected to how the user configured their SQL Server and database.

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generatedSelect “Edit Top 200 Rows” so that the user can both view the credentials as well as edit the TP numbers and passwords.

Users can either utilize the credentials to login back in the main pipe server or edit the credentials for ease of access.

### 2.2.3 Error execution: Failed to update status to pipe

This error occurred when the client handler is not listening or reading the pipe server. To mitigate this issue, re-run the entire main pipe server, followed by the client handler as explained in 2.3.1.

# 3.0 Software Vulnerabilities

## 3.1 Race Condition Vulnerability

Race conditions are a common vulnerability often intertwined with business logic flaws. It occurs when a program processes multiple requests at once without proper precautions. This situation leads to multiple threads accessing the same data or resources simultaneously, causing an unexpected application behaviour. Malicious actors exploit these vulnerabilities by timing the requests carefully and precisely, to cause deliberate collisions for malicious purposes.

The period which these collisions can occur, are known as “race window” (PortSwigger, 2024) which can be as small as milliseconds between interactions with the database. The impact of race condition is varied depending on the application and the specific feature involved.

### 3.1.1 Race Condition in IPC

As explained in section 2.1, one of the essentials of local windows pipes is to allow multiple processes utilizing shared resources. However, it is critical to ensure that only authorized processes should be able to access these pipes including reading and writing. Otherwise, malicious programs including malwares and spywares can cause abnormal behaviour to the original software or even disrupt the service, causing a denial of service (DOS).

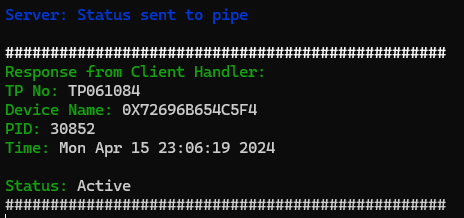
A race condition in IPCs can occur when two or more processes are reading or writing one source of shared data, and the final results depend on who runs faster and obtain the first access, therefore “wins” the race. The exact determination on which process runs fasters depends on the precise instruction sequence of the processes, which is determined by the OS CPU scheduler (Robert Chun, n.d).

CVE-2022-0847 (also known as DirtyPipe) is an example of how a misconfigured OS pipe can cause a high level severity vulnerability in a UNIX based system. Dirty pipe is a local privilege escalation vulnerability in the Linux kernel that allows a local attacker to bypass any file permissions, and write arbitrary data to any file under certain conditions. The magnitude of the exploit can extent where an attacker might gain root privileges by exploiting a race condition in the handling pipes.

In this case (referring to CVE-2022-0847 or Dirty Pipe), the race condition occurred due to improper synchronization between data updates and page marking within the kernel's page cache. By exploiting this race condition, an attacker could overwrite read-only files and potentially critical system files, leading to privilege escalation and potentially full control over the affected system.

### 3.1.2 Exploiting IPC Race Condition for Eavesdropping

The implemented name pipe is vulnerable to a race condition exploitation due to the fact that it does not have any validation or security check on the processes that perform requests form or to the pipe server. This means, any process can access the resource such as reading and writing into the pipe at any CPU schedule, compromising all 3 aspects of the CIA triad at once.

An attacker can simply inject a malicious software to sniff the packets that are being sent through the pipe by discretely running a separate instance, waiting for a race window to win over the pipe and read the file.

Under a normal and expected circumstances, the client handler should be to read the data feed by the server to the pipes as shown above. However, an attacker might discretely run a sniffing process without the user knowing.

The following code demonstrates how a malicious python code “*malicious\_sniffer.py*” will sniff the pipe:

A screen shot of a computer program

Description automatically generated

A screenshot of a computer program

Description automatically generatedIn a normal and expected behaviour, the response from client handler occur every 2 seconds. However, due to an existing race window, some of the response are missing from the actual client because it has eavesdropped by the malicious code.

This demonstrates that a malicious software can definitely be put in place to sniff data that are exchange through the pipes if they are not configured properly. This sniffing method directly compromise the confidentiality aspect of the CIA triad. In a real world scenario, this can be embedded in spywares to steal information that legitimate softwares exchanges amongst other processes.

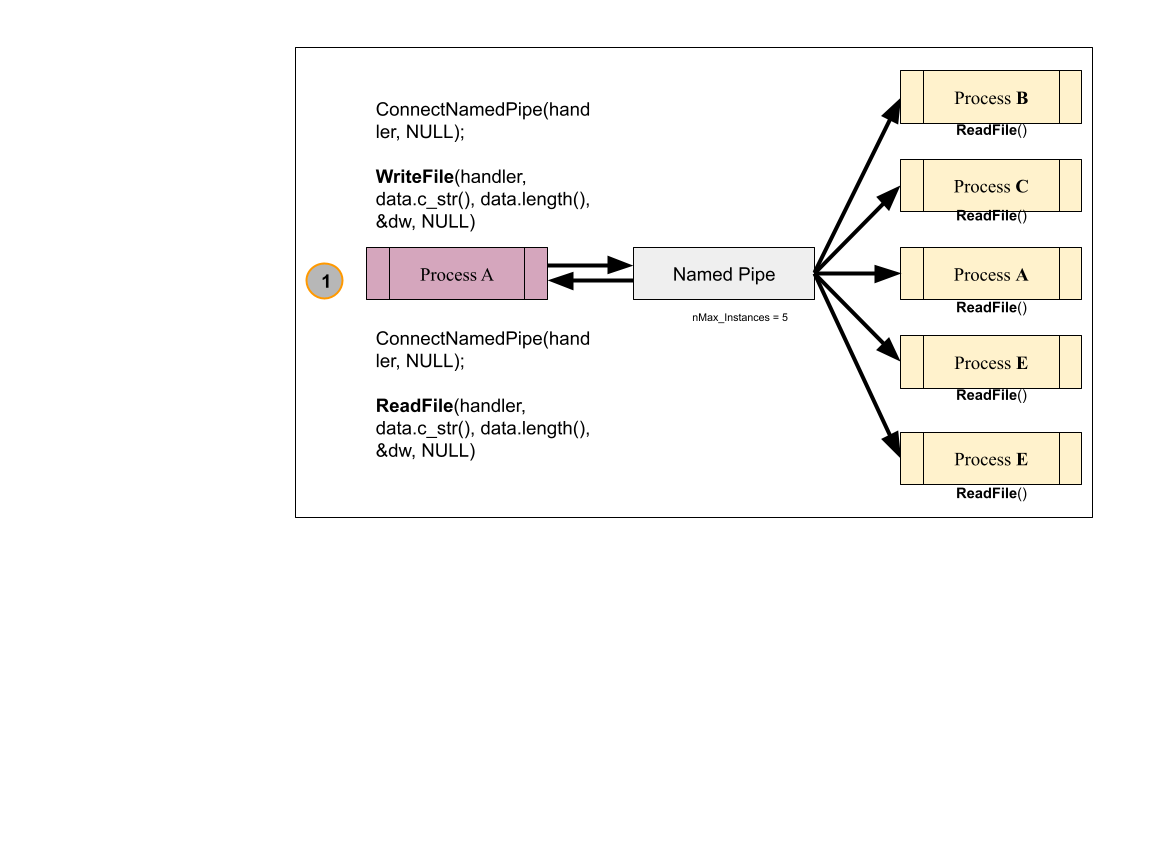
## 3.2 Resource Exhaustion Vulnerability

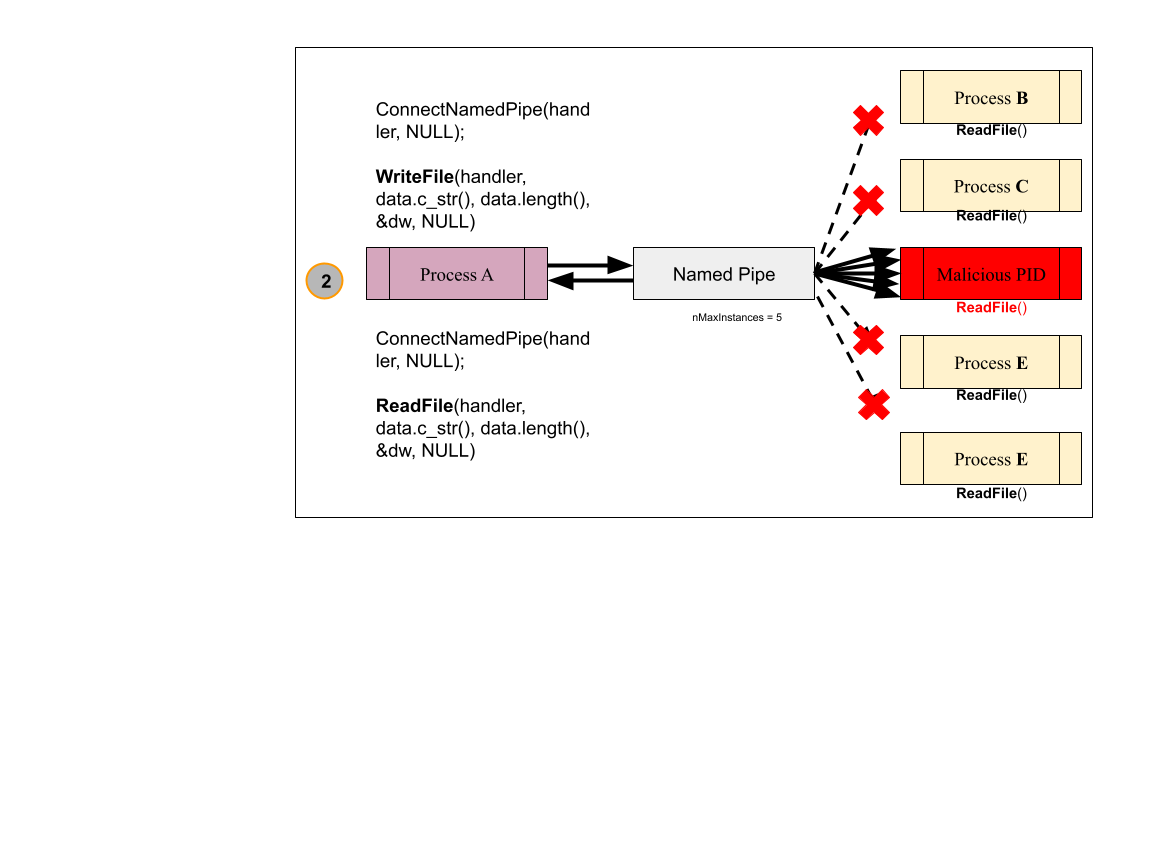
A resource exhaustion vulnerability is a type of software fault that causes the depletion of some resource, such as CPU, memory, or network bandwidth, by consuming or allocating it in an undefined or unnecessary way, or by failing to release it when no longer needed (Joao Antunes, n.d).

As described earlier, an IPC pipe is also a limited resource in a system given that it is a shared memory for processes to use with limited instances, buffers, as well as accessability. These open pipes occupy slots within the system's designated pool for named pipes. If an application keeps creating pipes without closing them, or flooding data through a duplex pipe without leaving any space or time gaps for other processes to access the pipe, it can cause an abnormal disruption to the working environment and eventually leading to a pipe DOS.

Attackers can exploit this vulnerability to cause a pipe DOS and block any other processes from accessing the pipe server. This can be done via starvation of read file API request where the malicious script constanly requests for file reading to the pipe without leaving any space and time for other process to access the pipe, blocking the entire client-side use of the pipe, similar to a DHCP starvation. Another approach which is the opposite of starvation is flooding the duplex pipe with write file API where the malicious script constantly writes into the pipe, eventually overflowing every client buffer listening to the pipe. Therefore, other pipes will unable to utilize that pipe.

### 3.2.1 Pipe Starvation to cause DOS

Named pipes can handle single or multiple requests concurrently, subject to configuration and resource availability. By default, the max instance that a pipe allow in the Win 32 API is up to 255 pipe instances. This can be done by directly input the value, or set the MAX\_UNLIMITED\_INSTANCES for the nMaximumInstance parameter. Besides instances, threads can also be starved. In this case, 1 thread is assigned to a pipe to handle multiple client sequentially depending on the scheduler. Under normal circumstances, each client should have enough time slice to retrieve the data from the pipe.

However, an attacker can run a malcious script that when executed, floods the pipe server with read requests, performing a starvation attack to that single thread blocking other process from reading that pipe.

The following code ***thread\_starved.py*** is a slight modified version of the *malicious\_sniffer.py* where the read requests does not have any time sleep at all, causing an infinite and very fast execution of the request instruction in the loop. This causes other process have insufficient time to perform a request instruction.

A screen shot of a computer program

Description automatically generated

A screenshot of a black screen

Description automatically generatedWhen the malicious thread\_stavation.py code is running, we can notice that the original client\_handler.py cannot retrieve any information for a long period (may retrieve if it wins a race window).

This disrupt the normal flow of the actual intended operation of the software since the client handler cannot retrieve the data. In the real world, if such starvation occurs, users might notice lagging on their side since the back end operation is slowed due to the inconsistentcy of data flow.

This attack compromises the availability factor from the CIA triad as it can cause the service to be temporarily unavailable over a period of time.

A screenshot of a computer program

Description automatically generated

The diagram above shows the output of thread\_starvation.py code when running. Once in a while, a successful read can occur . However, for the majority of the time, it will deny the service as seen from the actual client\_handler.py output.

### 3.2.1 Pipe Flooding DOS

Contrary to starvation, flooding the pipe with write file requests can also cause a DOS to the named pipe. In addition, if the buffer size for reading pipe of the server (Process A) was not being pre-defined beforehand, the server is also susceptible to a buffer overflow attack, causing all data inside the buffer being flooded out of the memory.

In this context shown in diagram, the malicious process constantly flood the named pipe with write requests with significantly higher frequency and lower time slice to maximize the requests, and winning more race windows, eventually blocking other processes from reading and writing into the file. In addition, if another legitimate process tries to read the pipe and in a very rare case, wins the race window, the legitimate process will be able to read the data that is overflowing the pipe.

The following sample script Flooding\_Dos.py is a slight modified version of the thread\_starve.py whereby the request made is WriteFile instead, and the API sends the DOS\_FLAG which is “FFFFFFF” multiplied by 9999 with 0 time sleep to increase reduce time slice and request as many as possible at a time. This will flood the pipe with the writefile request and potentially blocking every other process from reading the pipe.

A computer screen shot of a code

Description automatically generatedA screenshot of a computer screen

Description automatically generated

A screenshot of a computer screen

Description automatically generatedThe diagram above demonstrates when the malicious process is executed, will occupy the pipe instance as much as it could. When the client\_handler.py is running, we can see that the actual client cannot retrieve any data at all from the pipe.

In addition, even when a malicious sniffer is running, it also cannot access to the pipe and constanly returning an error as shown below:

A screen shot of a computer

Description automatically generated

## 3.3 SQL Injection Vulnerability

SQL injection (SQLi) is a cyberattack that injects malicious SQL code into an application (Bart Lenaerts-Bergmans, 2022) that allows threat actors to view, modify a database, or abrupt the normal logical flow of an application. It is considered as the third most serious web application security risk as reported by OWASP in 2021.

SQL injection (SQLi) typically occurs when web applications fail to properly sanitize user input before constructing SQL queries. This vulnerability allows attackers to manipulate the structure of SQL queries executed by the application, potentially leading to unauthorized access or manipulation of the underlying database.

It's important to note that exploiting an SQL vulnerability doesn't always entail dumping the entire database. Attackers can also leverage SQL injection to bypass authentication mechanisms, gaining unauthorized access to sensitive data or services. This underscores the versatility and potential severity of SQL injection attacks in compromising the security of web applications.

### 3.3.1 Example of SQL Injection

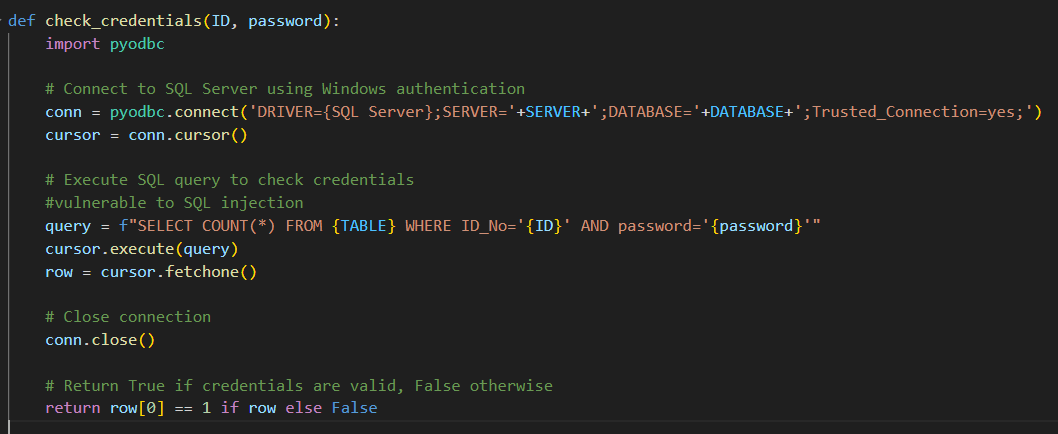
Given an application with a login feature that queries the database to authenticate users. Assume that the SQL query used is vulnerable to SQLi due to insufficient input validation. An attacker inputs a string into the username field as *“' OR '1'='1”.* Upon submission, the application will construct the SQL query together with the injected input and the end product will look as follows:

*SELECT \* FROM users WHERE username='' OR '1'='1' AND password='[password]'*

The injected SQL code ' OR '1'='1' alters the query's logic to always evaluate to true, bypassing the password authentication check. Consequently, the attacker gains unauthorized access to the application.

### 3.3.2 SQLi to Bypass Authentication

The developed software embedded an insecure code of SQL sanitization that is used to check with a local database. The vulnerable snippet can be found the the client\_handler.py file in the following code:

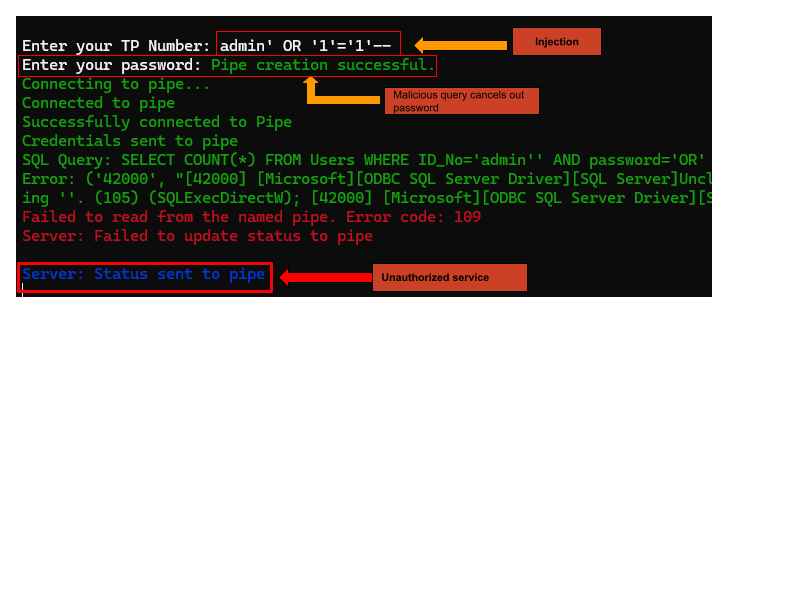


Both ID and password queries were parsed directly as a string via inherited parameter by the function without proper sanitization or checking. An attacker can exploit this vulnerability by inserting malicious query into the ID and password that were retrieved from the pipe.

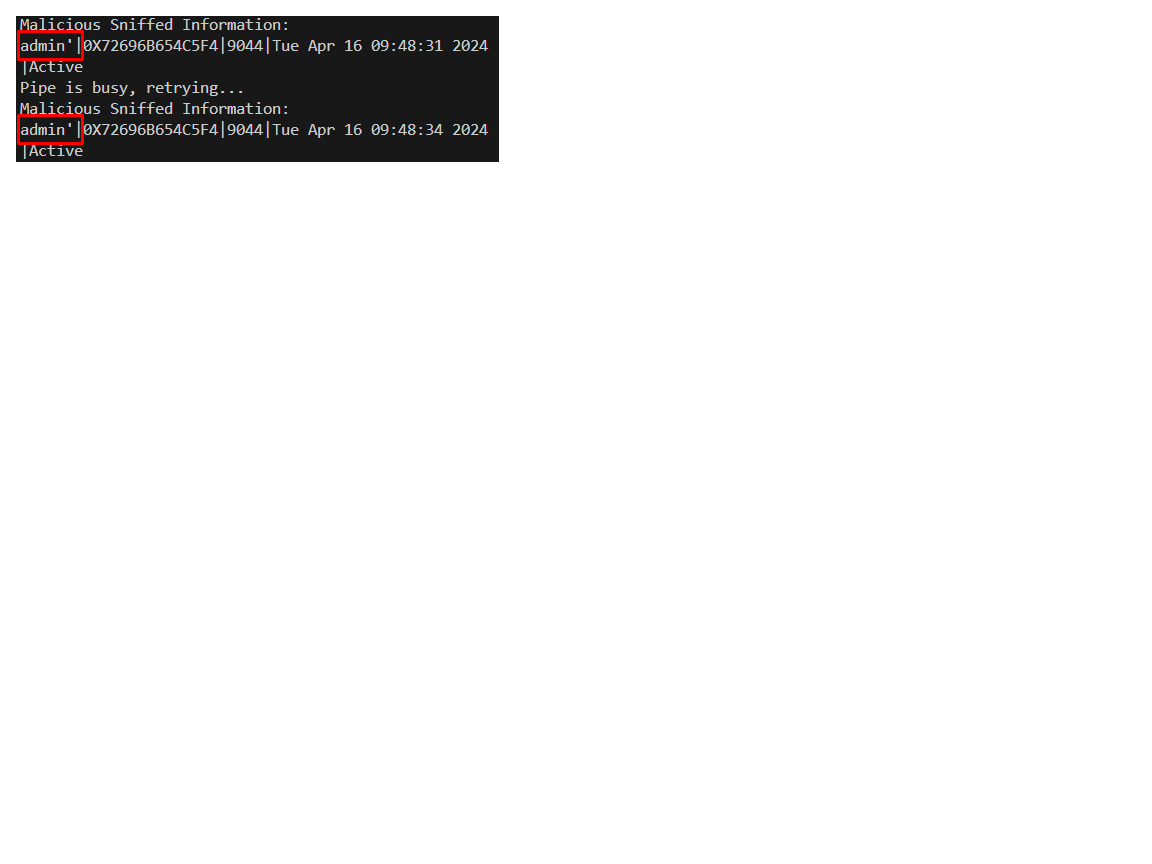
A screen shot of a computer program

Description automatically generated

Therefore, the attacker should insert the malicious query in the input during the execution of the pipe server.



The above snippet demonstrates how an attacker may bypass the login authentication and perform the get\_deviceinfo() method. Allowing them to obtain information about the device and process without authentication.

To facilitate this attack, an attacker might even take a step further and place a malicious sniffer to read what was sent out from the unauthorized status sent to the pipe. To simulate the exploitation, run the same *malicious\_sniffer.py* to sniff the information being sent out.

From here, we can see that the ID that was sent out was *admin’*. Notice that this user does not even exist in the ASC\_Database, yet the attacker managed to execute the service and obtain the device and process information.

# 4.0 Secure Coding Implementation

3 main vulnerabilities were highlighted in the previous section which includes race condition, resource exhaustion, and SQL injection. Each vulnerability already has its mitigation implemented in the real world. However, the secure coding mitigations might vary depending on how other parts of the code works and implemented such as having a new different functions, network aspects, resource availability, customer needs, and many more. However, there are several common secure coding concepts that can be implemented to prevent these vulnerabilities being exploited in general conditions.

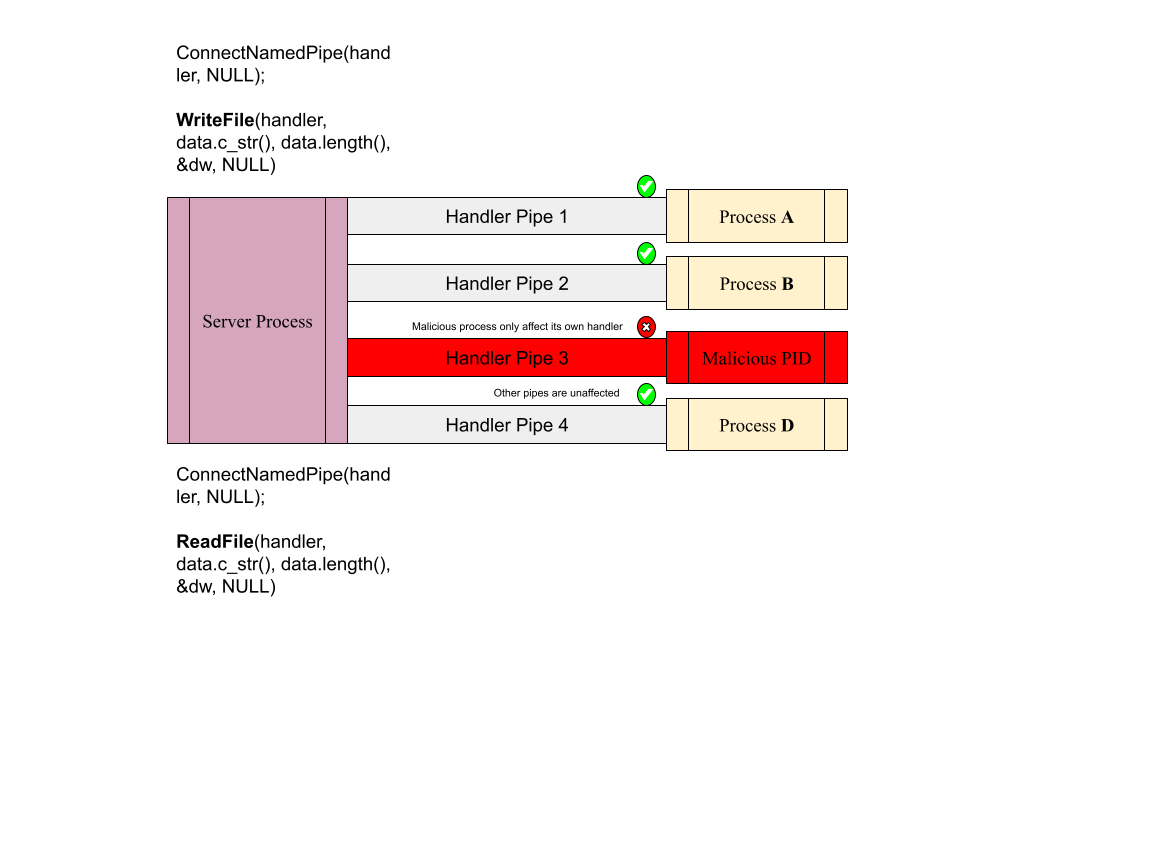
## 4.1 Multi-Threading Coding Concept

Multi-threading refers to a program or OS’ capacity to facilitate concurrent execution of multiple tasks, allowing more than one user to interact with it simultaneously. This functionality eliminates the need for running multiple instances of the program on the computer. Additionally, multithreading enables the system to manage and process numerous requests from a single user concurrently.

Race conditions vulnerabilities exist when functions are not designed to be thread-safe which means that if multiple threads of execution are running in parallel, they have the possibility of interfering with each other (Poston, 2021).

A high-level concept mitigation of race condition can be described by designing the functions so that multiple instances of the function can be executed concurrently in parallel while maintain synchronization and without any collisions or disruption to one another’s resource. This can be achieved by allocating each requesting process its own thread. Concurrently, granting a new and different thread to another process that is requesting to use the same resource. Therefore, 2 or more threads can run concurrently and in parallel without disrupting each other as they are both in their own instances.

### 4.1.1 Implementing Multi-Threading in IPC

Implementing Multi-Threading in Inter-Process Communication (IPC) involves utilizing threading libraries such as pthreads in C++ or the threading module in Python. By spawning multiple threads within each process, communication channels like Named Pipes can handle concurrent requests efficiently. For instance, in C/C++, the pthread library enables the creation of threads with functions like pthread\_create(), allowing each process to execute tasks concurrently.

In the context of Named Pipes, multithreading can be leveraged to mitigate Race Conditions by dedicating separate threads and pipe handlers for accessing the pipe. For example, in a client-server architecture using Named Pipes for communication, multiple client threads can send requests simultaneously to the server. By employing mutex locks or semaphores, critical sections of code accessing shared resources like the Named Pipe buffer can be synchronized, preventing data corruption or inconsistencies. Thus, each thread operates independently, ensuring seamless communication and avoiding race conditions within the Named Pipe system.



The snippet above is a redacted of the original server pipe code for ease of reading and understanding. It highlights on which section of the multi-threading can be implemented to mitigate the race condition vulnerability.

A screen shot of a computer program

Description automatically generatedAs demonstrated in 3.1.2 where the race condition occurs during the cWriteFile that sends out deviceinfo.c\_string() to the pipe, the following coding concept will be implemented here to mitigate the vulnerability to ensure the code is more secure.

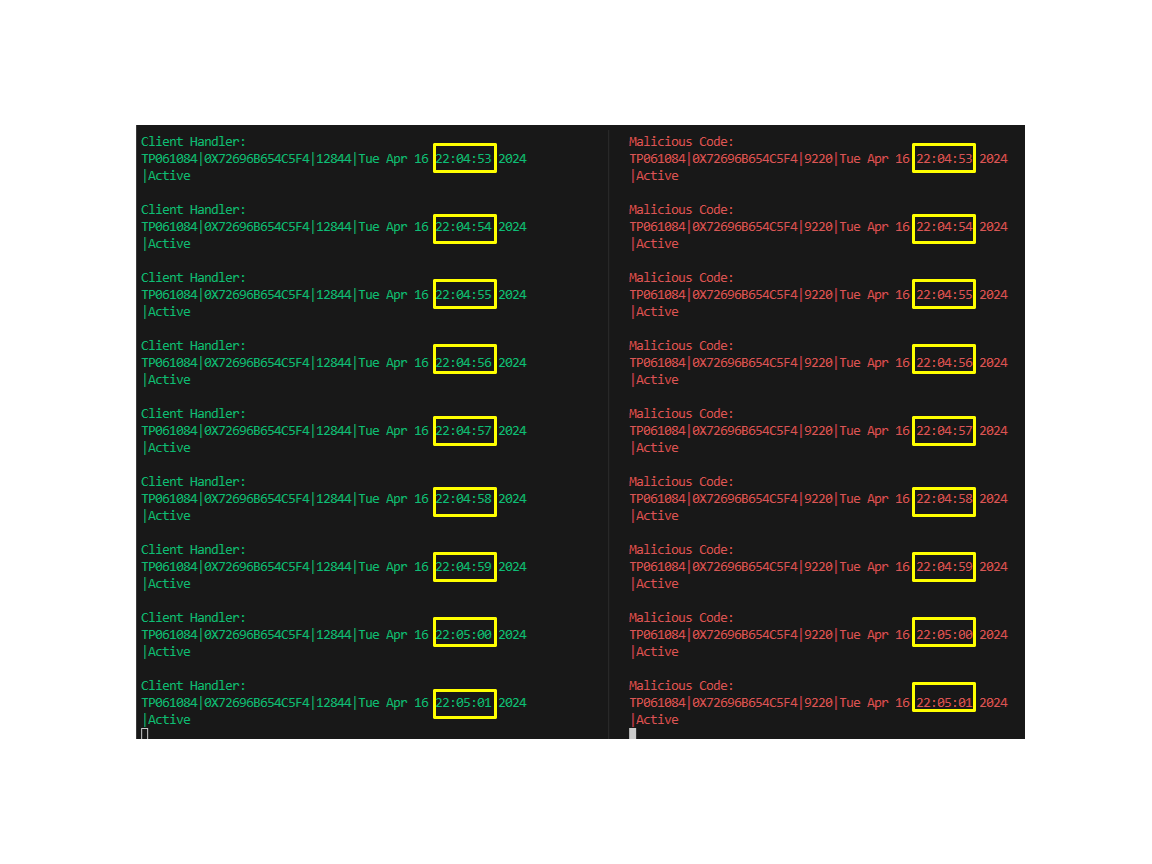
A server thread is initialized by passing the PipeServerThread function as an argument to std::thread constructor, enabling concurrent execution of server-side tasks while allowing the main thread to proceed. Concurrently, multiple client threads are created within a loop, each attempting to connect to the named pipe using the CreateFile function.

By employing std::vector to manage client threads, the program dynamically creates and manages multiple threads, enhancing the parallelism of client operations within the IPC framework.

To mitigate race conditions, the new code ensures that each client thread operates independently and does not interfere with the operations of other threads or the server. Since each client thread attempts to connect to the named pipe independently, there is no shared state or resource accessed concurrently by multiple threads, thereby mitigating race conditions. Additionally, the use of separate handles for each client thread's pipe connection ensures thread safety, as each thread operates on its unique instance of the pipe without interfering with others.

By joining client threads sequentially after their creation and ensuring proper resource management and cleanup that facilitate synchronized execution, eliminating the possibility of race conditions in the multithreaded IPC environment.

### 4.1.2 POC: Mitigating Race Condition Vulnerability

In the new secure code, if an attacker attempts to cause a starvation or flooding to the pipe, it will do so only to its very own instance and thread. Thus, not affecting other processes, in this case the client\_handler.py. This ensures that other processes remain protected in their own instance thread and not affected by the malicious requests.

The diagram above demonstrates the actual client handler being run concurrently with the malicious script that was designed for the pipe DOS via starvation. The diagram highlights the time that the log files were written, came from the same line of code in the pipe server. With the new secure coding concept, the client handler is not affected by the malicious script. This ensures the availability factor from the CIA triad.

Therefore, there are no race windows between processes as each process are dedicated with their own instance and thread. In addition, starvation and flooding attack will only affect the malicious process’ own environment and not DOSing other intended processes.

## 4.2 Process Authentication via Security Descriptor by Leveraging ACL

The implementation of multi-threading mitigates the race condition vulnerability. However, the program is still susceptible eavesdropping or sniffing by other processes. To mitigate this risk, it is important to ensure that only authorized process or users can utilize the pipe. One method of achieving the objective, is by leveraging the Security Descriptor feature alongside Microsoft’s Access Control List.

In the Win32 API, a Security Descriptor (SD) is a data structure that contains security information associated with a securable object, such as a file, directory, or named pipe. It comprises four main components: the owner SID, group SID, discretionary access control list (DACL), and system access control list (SACL). The owner SID identifies the security principal that owns the object, while the group SID specifies the security group to which the object belongs. The DACL contains a list of access control entries (ACEs), each specifying a security principal and the permissions granted or denied to that principal. Meanwhile, the SACL defines auditing settings for the object, enabling the monitoring of attempted access.

Within the context of IPC security, leveraging Security Descriptors and Access Control Lists (ACLs) is crucial to ensuring only authorized users can access a named pipe. By associating a Security Descriptor with the named pipe during its creation using functions like CreateNamedPipe, administrators can define the access rights granted or denied to various users or groups. This is achieved by populating the DACL with ACEs that specify the permissions for each security principal. For instance, administrators can grant read or write access to specific users or groups while denying access to others. Additionally, by including appropriate audit settings in the SACL, administrators can monitor and log attempts to access the named pipe, enhancing security and accountability.

To implement Security Descriptors effectively in IPC security involves specifying the appropriate security principals (e.g., user accounts, security groups) and defining their corresponding access rights (e.g., read, write, execute). By effectively leveraging Security Descriptors and ACLs, organizations can enforce access control measures in IPC environments, and enforce the confidentiality factor from the CIA triad, safeguarding sensitive data communication in IPCs and resources from unauthorized access and potential security threats.

### 4.2.1 POC: Authorized Users Only

In the new secure code, security features have been integrated into the named pipe creation process using Security Descriptors and Access Control Lists (ACLs) to ensure that only authorized users, specifically administrators, can access the pipe. This is achieved by defining an explicit access control entry (ACE) that grants all permissions (GENERIC\_ALL) to the built-in administrators group (SID: S-1-5-32-544). The ACE is then added to an Access Control List (ACL) using the SetEntriesInAcl function, and the resulting ACL is associated with the named pipe's Security Descriptor.

By setting the Security Descriptor's discretionary access control list (DACL) to allow only administrators to connect to the named pipe, the code effectively restricts access to authorized users. This prevents unauthorized users from eavesdropping or sniffing on the pipe's communication, as attempts to connect from non-administrative accounts will be denied by the security mechanisms enforced by the operating system. Additionally, the use of ACLs enables fine-grained control over access permissions, allowing administrators to define specific security policies tailored to their organization's requirements. This enhances the overall security posture of the IPC environment, mitigating the risk of unauthorized access and potential security breaches.

To demonstrate the security feature, we first need to gain an admin privilege to activate and create the pipe. Therefore, we need to execute the pipe server as an administrator.

A screenshot of a computer

Description automatically generated

From the above snippet, the data has been sent to pipe. We can now attempt to sniff the data as a normal user using the malicious\_sniffer.py to check whether we can eavesdrop the data. Thus, we can first run the malicious\_sniffer.py as a normal user.

A computer screen with green text

Description automatically generated

Upon running the malicious\_sniffer.py, we returned an error message stating that access is denied. However, if we try to run a different python file with the same functionality as an admin, we can now see the data.

A computer screen with text and numbers

Description automatically generated

The diagram above demonstrates that only an admin can access the pipe including reading. This ensures that only authorized users and processes can access the pipes and server resources. Therefore, the security descriptor paired with ACL ensures the confidentiality of the CIA triad.

## 4.3 Parameterized Queries to Mitigate SQLi

To safeguard code from SQL Injection, a prevalent approach involves encapsulating and parameterizing SQL commands. Parameterized queries serve to disentangle the SQL query from user input values. Through this technique, user input values are transmitted as parameters, preventing them from containing executable code. Instead, parameters are treated purely as literal values and subjected to checks for type and length, thereby ensuring the security and integrity of the system.

Implementing parameterized queries is a fundamental strategy to mitigate SQL injection (SQLi) vulnerabilities in software applications. Parameterized queries separate SQL commands from user-supplied input values by using placeholders for parameters in SQL statements. These parameters are then bound to specific values before the query is executed, ensuring that user input is treated purely as data and not as executable SQL code. By enforcing this separation between SQL commands and user input, parameterized queries prevent attackers from injecting malicious SQL code into queries, thus significantly reducing the risk of SQLi attacks.

### 4.3.1 POC: Parsing Values Through Placeholders

To implement parameterized queries, developers replace hardcoded values in SQL queries with placeholders, typically represented by question marks (?) or named placeholders. When executing the query, the actual values for these parameters are provided separately, ensuring that they are treated as data and not as executable code. This approach effectively mitigates SQLi vulnerabilities by eliminating the possibility of attackers manipulating SQL queries through user input.

A screen shot of a computer

Description automatically generated

In the new secure coding implemented, instead of directly inserting user-supplied input into the SQL query string, placeholders represented by question marks (?) are used in the SQL query to represent parameters. These placeholders serve as markers for where user input values will be bound to the query. By separating the SQL query from user input values, parameterized queries ensure that user input is treated purely as data and not as executable SQL code, effectively preventing attackers from injecting malicious SQL code into queries.

When executing the SQL query, the actual values for the parameters are provided separately as a tuple (ID, password), ensuring that they are securely bound to the query. This approach mitigates the risk of SQLi attacks by enforcing a clear separation between SQL commands and user input, thus preventing attackers from manipulating SQL queries through user-supplied input.

A screenshot of a computer program

Description automatically generated

The diagram above demonstrates that the new secure coding implementation mitigates the SQLi vulnerability and prevent unauthorized and unregistered users from accessing the service inside the software. In addition, it prevents any malicious manipulation towards the query sent to the database, ensuring the integrity factor of the CIA triad.

# 5.0 Conclusion

To conclude this module assignment, software vulnerabilities can also occur at a lower level part of software, particularly at the OS level. This assignment demonstrates that exploiting a low level vulnerability can also disrupt the entire software and compromise the CIA triad.

In addition, it is critical for software developers to perform code reviews or audits to ensure that there are no know vulnerabilities and minimize any risks on the code side. It is also important to enforce a secure coding practice to ensure the workability, accessibility, integrity, availability, and confidentiality of a software.

# 6.0 References

Irwin, L. (2023, September 13). *Demystifying the CIA Triad: Why It’s Crucial for Cyber Security*. IT Governance UK Blog. <https://itgovernance.co.uk/blog/what-is-the-cia-triad-and-why-is-it-important>

Computer Hope. (2019, January 31). *Win32*. <https://www.computerhope.com/jargon/w/win32.htm>

Sharma, V. (2023, March 7). Inter-Process Communication in Operating Systems: A Comprehensive Guide with Real-life Examples and Code. *Medium*. <https://medium.com/@the_daft_introvert/inter-process-communication-in-operating-systems-a-comprehensive-guide-with-real-life-examples-and-c508cf3bfb1a>

PortSwigger. *Race conditions | Web Security Academy*. (n.d.). https://portswigger.net/web-security/race-conditions

Robert, C. (n.d). *Race Conditions, Critical Sections and Semaphores.* [Day-8---Race-Semaphores.pdf (sjsu.edu)](https://www.sjsu.edu/people/robert.chun/courses/cs159/s0/Day-8---Race-Semaphores.pdf)

CrowdStrike. (2023, November 8). *What is a SQL Injection Attack? - CrowdStrike*. crowdstrike.com. <https://www.crowdstrike.com/cybersecurity-101/sql-injection/>

Joao, A., Nuno, F.N., Paulo, V., (n.d). *Detection and Prediction of Resource-Exhaustion Vulnerabilities).* University of Lisboa. [ISSRE08.pdf (ul.pt)](https://www.di.fc.ul.pt/~nuno/PAPERS/ISSRE08.pdf)

Poston, H. (2024, March 4). *How to mitigate Race Conditions vulnerabilities*. Infosec. <https://www.infosecinstitute.com/resources/secure-coding/how-to-mitigate-race-conditions-vulnerabilities/>

Kirvan, P. (2022, May 26). *multithreading*. WhatIs. <https://www.techtarget.com/whatis/definition/multithreading>