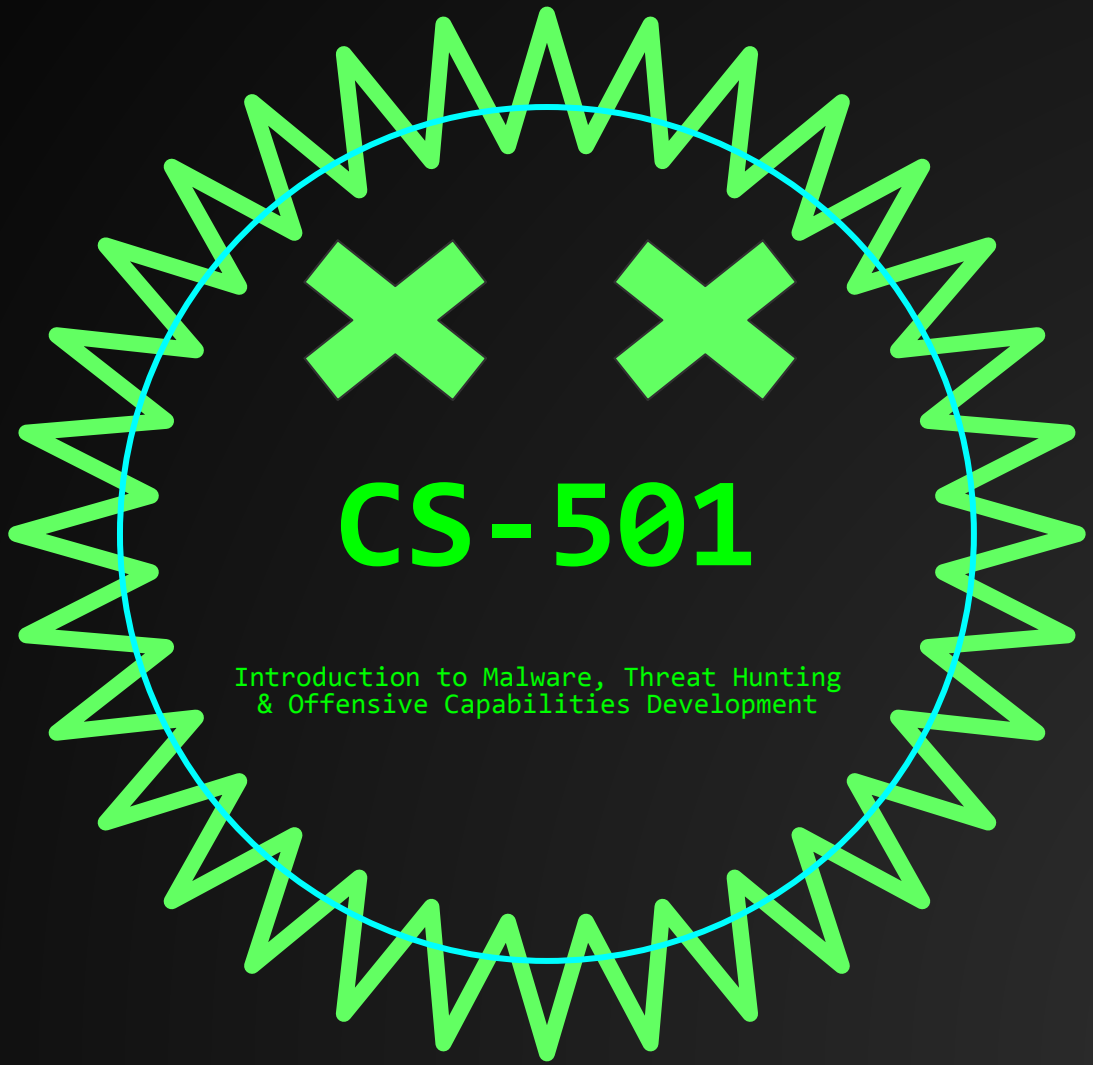


CS-501

Introduction to Malware, Threat Hunting
& Offensive Capabilities Development

Lecture 8
Consoles, Processes, Pipes

Windows Subsystems

Console Subsystem

Kernel Object: Processes

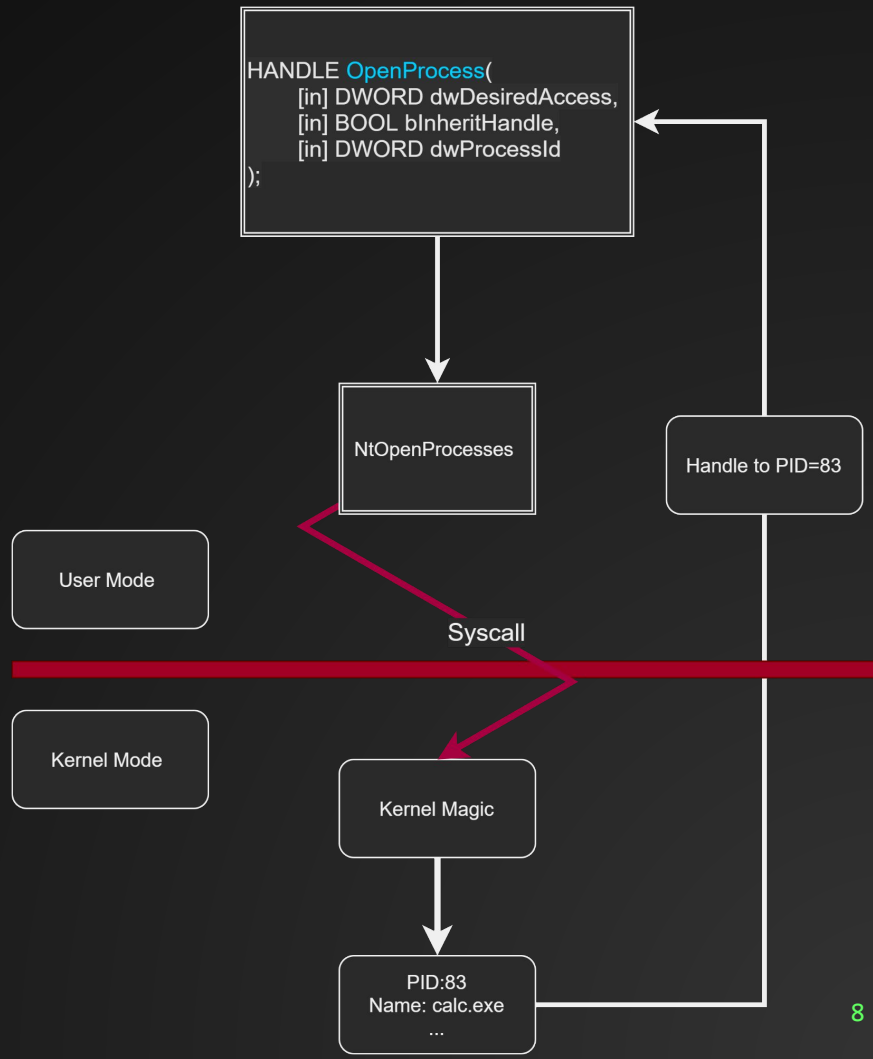
- Recall that a processes is an object managed by the kernel
- It has a *Private* Virtual Address Space
- Each processes contains multiple *modules*
- A *module* is an executable file or DLL that has been loaded into the Virtual Address space of a running process
  - Example: kernel32.dll gets loaded by every process
  - As a technical detail, a module is registered in a per-process data structure. If the module is purged from the data structure, or is loaded without using LoadLibrary, built in functions that rely on the handle being a handle to a module may fail!
- The Windows API gives us multiple functions to interact with processes and their associated resources
  - Most of these functions are contained in the PSAPI

Processes: Getting a Handle

- A process handle is an opaque identifier that allows userland processes to interact with process objects
- We can either spawn a new processes and get a handle, or we can request a handle to an existing processes.
  - A thread can even get a handle to the process that contains it!
- Recall that a processes is uniquely identified by its Processed ID (PID)
- This value is recycled once the processes terminates.
  - This can lead to some subtle race conditions
- To open a Handle to a processes that already exists, use OpenProcess(...)
  - If you need a handle to the current running process, use `GetCurrentProcess`
    - Technically speaking, the value returned from GetCurrent* is a *pseudo* handle. Same idea as a handle, except that it cannot be closed.

OpenProcess

- Used to create/get a handle to a running processes
- Requires the PID of a running processes
- Different operations on the processes object will require different accesses rights.
- These are set in dwDesiredAccess
  - For example, to delete (terminate) a processes, we must have the DELETE (0x00010000L) accesmask
  - To Query information (such as the processes Name) we need
- If we would like any child processes spawned by the current processes to inherit the newly acquired handle, we can set this in bInheritHandle



Querying Process Attributes

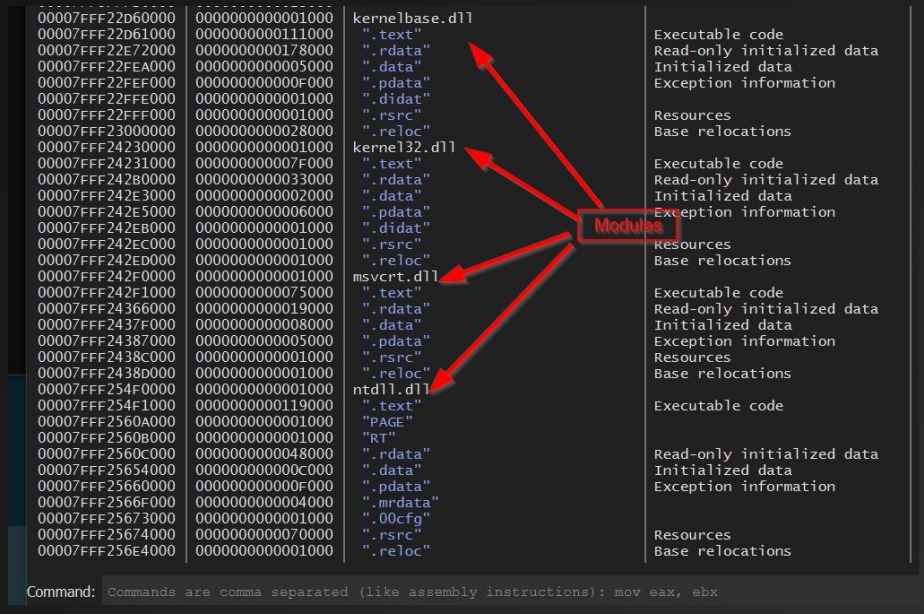
- Usually requires the following access mask: PROCESS_QUERY_INFORMATION | PROCESS_VM_READ
- Process Explorer displays just about everything you could want to know about a process
- Some highlights include:
  - Basic Process information (pids, name)
  - Loaded modules (dlls, PEs),
  - Open handles,
  - Environment variables,
  - In memory strings,
  - token information,
  - running threads

Querying Processes Image Name

- Recall that an executable image is the file on disk that contains the program that is eventually memory mapped and executed inside of a process
- In many situations, it is helpful to know where the running processes executable image is
- If we have a handle to a processes with the correct access mask, we can simply call `QueryFullProcessImageNameA`
- Note there: is some shenanigans you can pull to spoof this value
- Note further that you can’t (easily) delete a file on disk if it is an executable image that is loaded in a running process

Querying Modules

- Recall that a module is a memory mapped Portable Executable
- This includes DLLs required to run an executable, as well as the executable itself
- Modules can be loaded and unloaded at runtime
- A handle to a module in a process is the PE's base address in that processes virtual address space!
  - I.e., it is a 32/64bit unsigned integer depending on the arch.
  - Recall that the base address of a PE is the start of the



Demo
Enumerating Loaded Modules

Processes: Creation

- Win32/Native:
  - `ShellExecute`
  - `CreateProcess`
  - `CreateProcessAsUser`
  - `NtCreateProcessEx`
  - `CreateProcessInternal`
- CRT:
  - `popen`
  - `system`
  - `exec` family of functions

ShellExecute

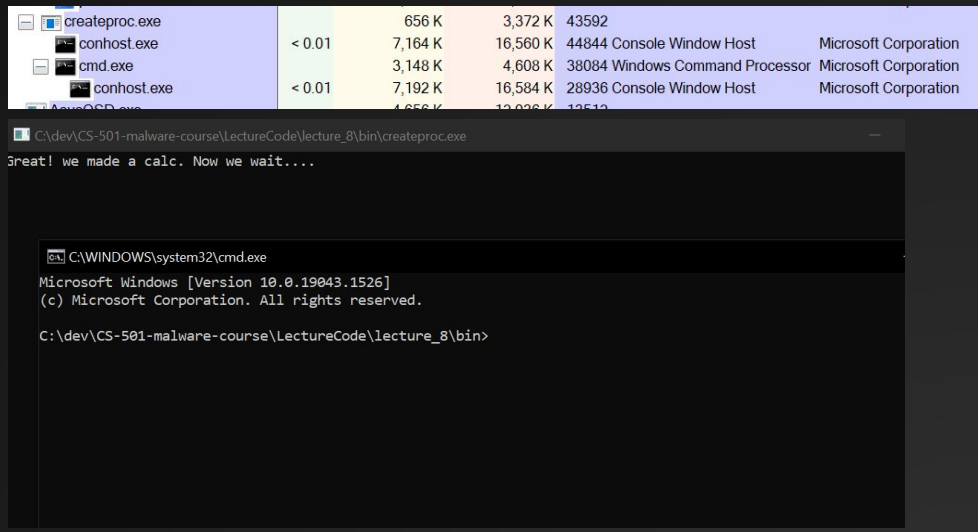
- Easy to use: runs a file with a registered shell extension
  - I.e., the PE loader doesn’t care that blah.exe is named `blah.exe`, but nothing will happen if you double click the same file but named `blah.txt`
- Can be useful for spawning elevates processes that trigger User Access Control (UAC) prompts
  - More on UAC later but for now, but for now, just know it is the mechanism that allows lower privileged processes to do Admin stuff :D
  - Like sudo but way more obtuse and error prone

ShellExecuteEx

- Allows us to get a handle to the created processes
- Fun fact, I have used ShellExecuteEx on an engagement before as a Hail Mary
- I kept asking them for UAC permission until they finally got annoyed and clicked yes

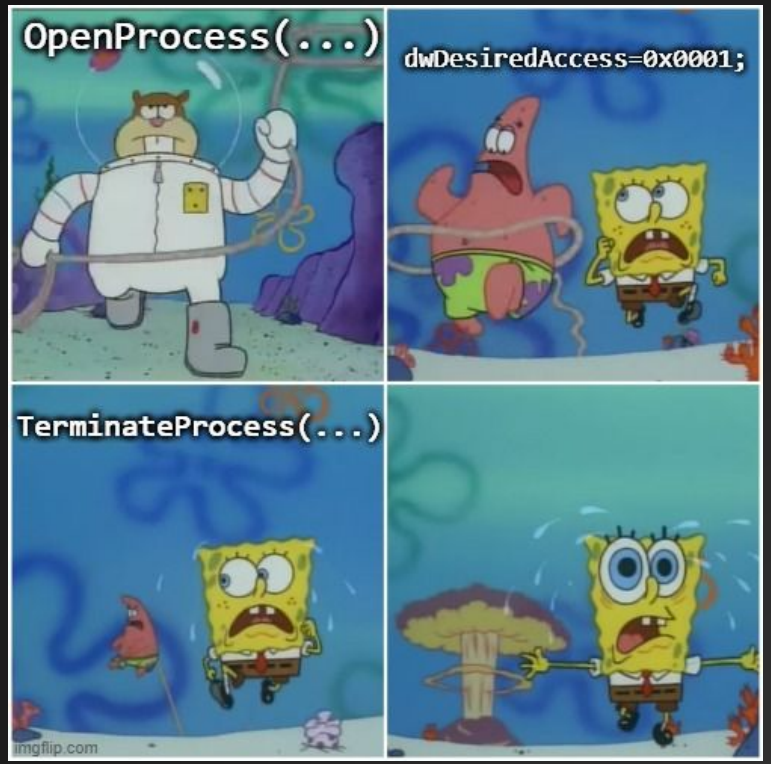
CreateProcessA

- Creates a Process :-)
- Provides more customization for *how* the process is created
- The Parent process that calls CreateProcess (can) get a handle to the child process
- We will spend more time on the arguments and structs at various points of this course!
- For now, let's look at a process that spawns cmd.exe, and waits for the child process to exit before continuing
- WaitForSingleObject: "Waits until the specified object is in the signaled state or the time-out interval elapses."
    - In our case, we wait for the process to exit. You can also do the same with a thread/fiber…etc



Terminating a Processes

- There are ways to terminate a process
- The most straightforward way is to use the Win API function TerminateProcess
- You can also crash the process, or force a call to `ExitProcess` :-)



Processes Enumeration

- `EnumProcesses`: Easy to use, but only returns the full list of PIDs. This is not helpful on its own, and it is inefficient to constantly call OpenProcess, nor can you get a handle to lower-numbered processes without System Privileges
- `CreateToolhelp32Snapshot`:
- `WTSEnumerateProcesses`: Leverages the *Windows Terminal Services* functions. Very powerful API, not covered in this lecture
- `NtQuerySystemInformation`: Uses the native API, and is by far the most useful of the options, but is not future proof!

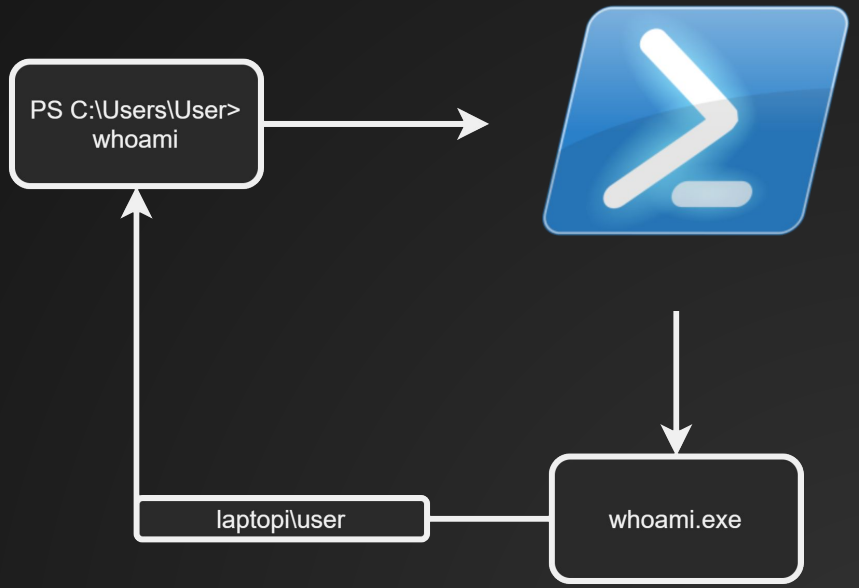
CreateToolhelp32Snapshot

- Doesn’t require Admin or special privileges
- Can be used to query not just PIDs, but names, Heaps, modules, and more!
- Is one of the (really annoying) cases where there are A/W versions, but for the A version, the A is omitted from the function name and for the W version the W is appended
  - If UNICODE is defined, it automatically expands…

Retrieving output from a Child Process

How do we create a process, and read the output?

- Example: running `whoami` in powershell actually spawns a new process (whoami.exe) and reads the output



Starting Processes and Getting Output

- Example: What if we want to run powershell or a CMD command?

Python Analogy

Directing output to a File

```
In [1]: import os

In [2]: os.system("whoami")
laptop-\user
Out[2]: 0

In [3]: os.system("whoami > a.txt")
Out[3]: 0

In [4]: with open("a.txt" ,"r") as f:
   ...:     result = f.read()
   ...:

In [5]: print(result)
laptop-\user
```

Directing Output to a Pipe

```
In [8]: import subprocess

In [9]: result = subprocess.Popen('whoami', stdout=subprocess.PIPE)

In [10]: print(result.stdout.read())
b'laptop-4rn9eipi\\user\r\n'
```

Anatomy of a Console

Time to read some Documentation!

A *console* is an application that provides I/O services to character-mode applications.

A console consists of an input buffer and one or more screen buffers. The *input buffer* contains a queue of input records, each of which contains information about an input event. The input queue always includes key-press and key-release events. It may also include mouse events (pointer movements and button presses and releases) and events during which user actions affect the size of the active screen buffer. A *screen buffer* is a two-dimensional array of character and color data for output in a console window. Any number of processes can share a console.

https://docs.microsoft.com/en-us/windows/console/consoles

Console Handles

A console process uses handles to access the input and screen buffers of its console. A process can use the GetStdHandle, CreateFile, or CreateConsoleScreenBuffer function to open one of these handles.

The GetStdHandle function provides a mechanism for retrieving the standard input (`STDIN`), standard output (`STDOUT`), and standard error (`STDERR`) handles associated with a process. During console creation, the system creates these handles. Initially, `STDIN` is a handle to the console's input buffer, and `STDOUT` and `STDERR` are handles of the console's active screen buffer. However, the SetStdHandle function can redirect the standard handles by changing the handle associated with `STDIN`, `STDOUT`, or `STDERR`. Because the parent's standard handles are inherited by any child process, subsequent calls to GetStdHandle return the redirected handle. A handle returned by GetStdHandle may, therefore, refer to something other than console I/O. For example, before creating a child process, a parent process can use SetStdHandle to set a pipe handle to be the `STDIN` handle that is inherited by the child proces

https://docs.microsoft.com/en-us/windows/console/console-handles

Creating a Console Application!

```
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpszCmdline, int nCmdShow)

...
```

Pipes

A pipe is a block of shared memory that processes can use for communication and data exchange. They behave in the same easy as a file, where multiple processes can read and write from the file, just (hopefully) not at the exact same time.

Pipes can either be anonymous or named. Anonymous pipes are only accessible from the processes they were created in and will get a random identifier

Named pipes exist in memory and can be accessed via //pipes//...

Named Pipes enables two unrelated processes to exchange data between themselves, even if the processes are located on two different networks.

A named pipe server can open a named pipe with some predefined name and then a named pipe client can connect to that pipe via the known name. Once the connection is established, data exchange can begin.

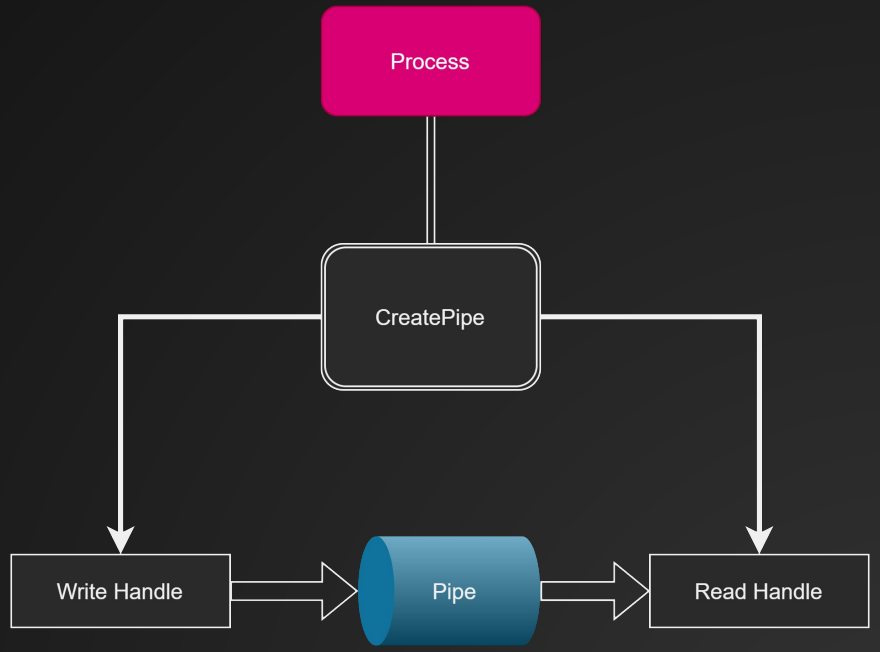
https://www.ired.team/offensive-security/privilege-escalation/windows-namedpipes-privilege-escalation#:~:text=Overview,located%20on%20two%20different%20networks.

Anonymous Pipes

- One-way inter process communication (IPC) mechanism
- Restricted to local machines
- Handles are created via CreatePipe
- Handles are inherited via Child/process relationships or duplicated and shared using a different IPC mechanism

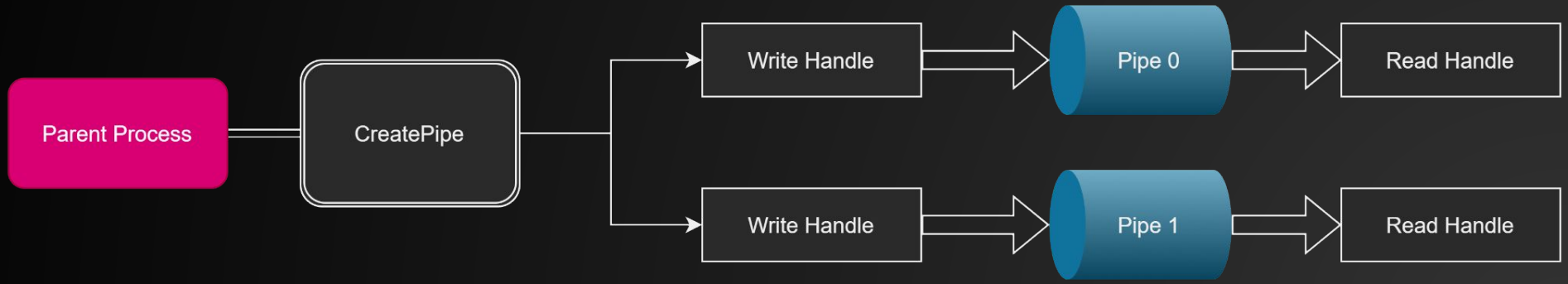
Using Anonymous Pipes for IO

- Anon Pipes are created by calling `CreatePipe`
- This will create two handles: one for reach end of the pipe
  - The Write handle which allows us to send data through the pipe
  - And the Read handle which allows us to read data that was sent to the pipe
- Anon Pipes are buffered, and only support synchronous IO
- This means Calls to (Read|Write)File are blocking!

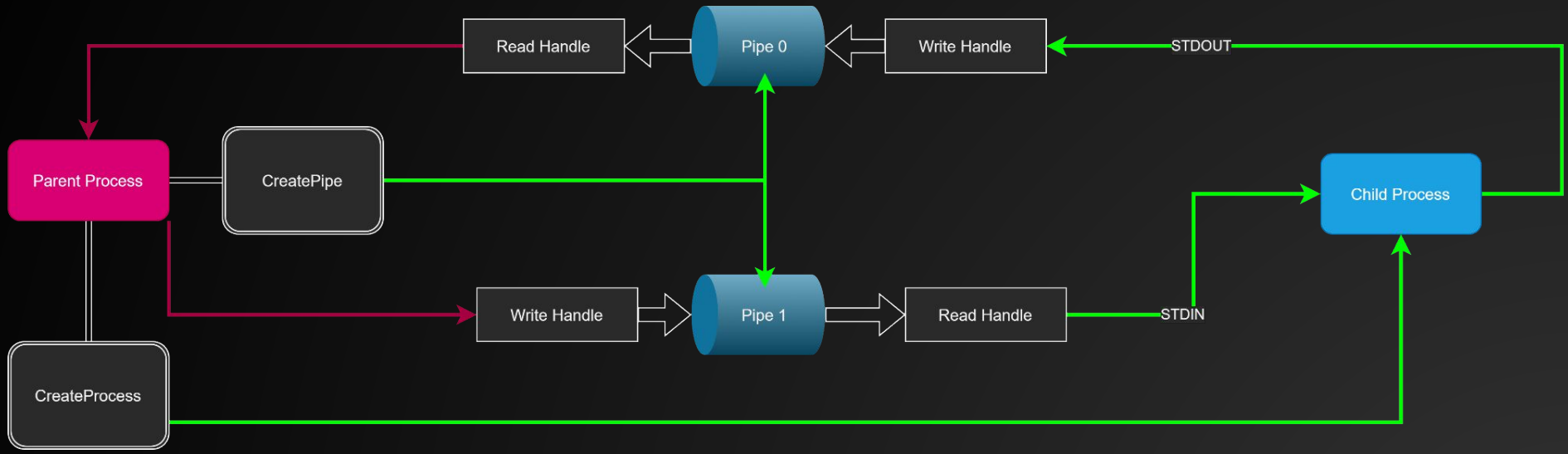


Anon Pipes to Redirect I/O

- We can use anonymous pipes to
  - send input from the parent process to the child process
  - Send data from the child process to the parent process
- This is accomplished by a parent process creating two pipes:
  - One for the parent process to send inputs to the child process (i.e., the pipe for the child process STDIN)
  - One for the child process to write its output, and for the parent to read from (Child process STDOUT)



Parent/child process I/O redirection



Handle Inheritance

In the special case where a processes is created by another process, the parent process can choose to share its handles with the child process

Once CreateProcess is called with the 5th argument set to TRUE, the child process will inherit all handles owned by the parent process that have their inheritance bit set.

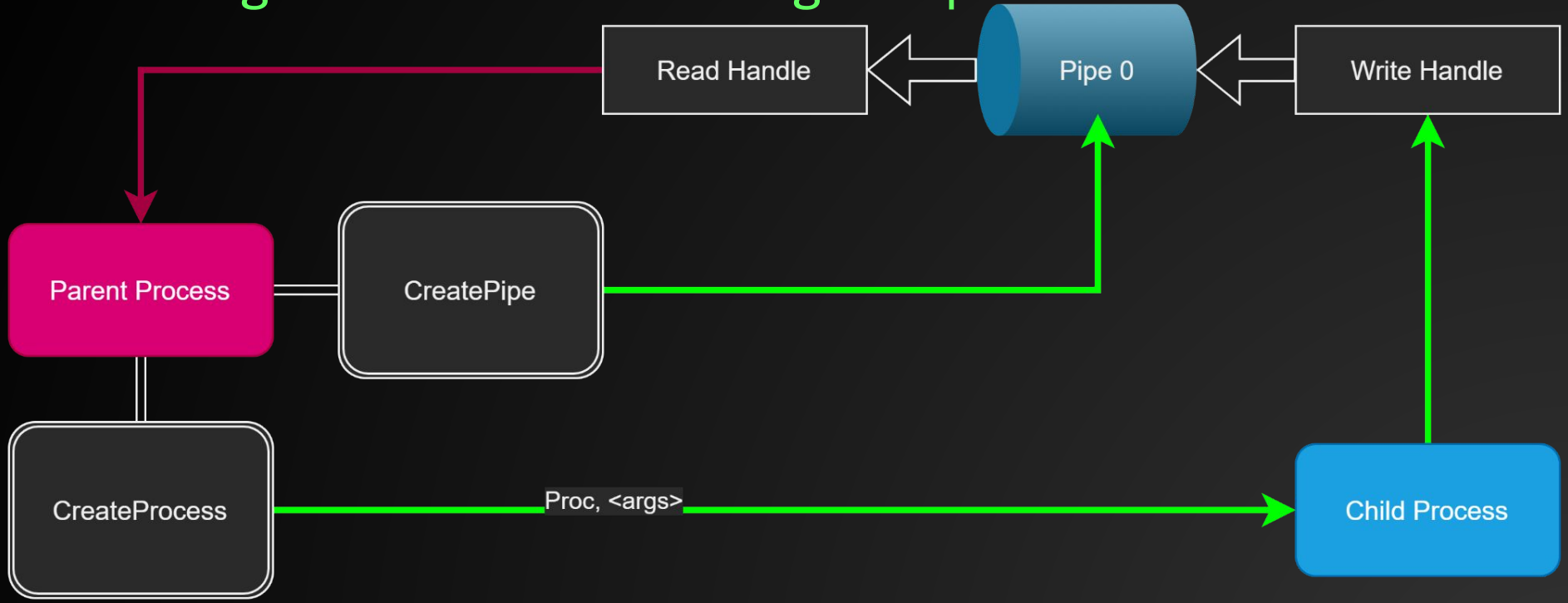
Inheritance actually works under the hood by duplicating the handles and passing them to the child processes

Note that the handle values are the same as in the parent process

Putting it all together

- Create two pipes
  - One for STDOUT/STDERR
  - One for STDIN (unless you only need to run the cmdline)
- We allow the child processes to write to these pipes, setting the correct attributes to enable handle inheritance
- From there, we wait for the process to terminate
- Once the process terminates, we use the Read handles for our pipes, and check the the contents.
- Alternatively, if we want to interactively send and receive data, we can keep sending/receiving data using the named pipes.
- Note if you try to read from a named pipe that has no data available, you will get an error!
  - To check if there is data available, we can use `PeekNamedPipe`
- Cleanup as needed

Running a Cmd and Reading output



Demo: CreateProcess w/ Pipe