SYNACKTIV

Hooking Windows Named Pipes

Pass The Salt 2025

03/07/2025

Whoami

Thomas Borot

Pentester

thomas.borot@synacktiv.com

Synacktiv

- French offensive security company
- 180 security experts
- 4 departments :
  - Pentest / Redteam
  - Reverse Engineering / Vulnerability Research
  - Development
  - Incident Response
- Hexacon

Overview

- Windows Named Pipes presentation and APIs
- Common attacks to intercept and modify data
- Common mitigations against MitM attacks
- How to bypass mitigations
- Demo
- Injecting data into a named pipe

Windows Named Pipes

Bidirectional channel between a **client** and a **server**.

```
PS > .\pipelist64.exe

Pipe Name                                    Instances      Max Instances
---------                                    ---------      -------------
InitShutdown                                      3               -1
lsass                                             9               -1
ntsvcs                                            3               -1
scerpc                                            3               -1
Winsock2\CatalogChangeListener-2ec-0              1                1
Winsock2\CatalogChangeListener-3e0-0              1                1
epmapper                                          3               -1
Winsock2\CatalogChangeListener-254-0              1                1
LSM_API_service                                   3               -1
Winsock2\CatalogChangeListener-1d8-0              1                1
atsvc                                             3               -1
```

Windows Named Pipes APIs

Server:

`handle = CreateNamePipe("\\.\pipe\example_pipe")` -> listen on "example_pipe"

Client:

`handle = CreateFile("\\.\pipe\example_pipe")` -> connects to "example_pipe"

Both:

`WriteFile(handle, "hello world!")` -> sends "hello world!" to the server

`data = ReadFile(handle)` -> reads data from the pipe

Other Windows APIs can be used to perform asynchronous read and writes

Note: *Some* named pipes are accessible through the network

Example

```
PS > .\pipe.exe -mode sync -servermode -pipename "example_pipe"
[INFO] CreateNamedPipeW("\\.\pipe\example_pipe", ...) -> 308
[INFO] ConnectNamedPipe(308, 0) -> 1
[INFO]   New client connected
[INFO] ReadFile(308, readBuffer, 2048, pNbBytesRead, 0) -> 1
[INFO]   Got data (22 bytes): "Client says tutJxQNpew"
[INFO] WriteFile(308, "Server says FSrHdjnLcr", 22, pNbBytesWritten, 0) -> 1
[INFO]   Wrote 22 bytes
```

```
PS > .\pipe.exe -mode sync -pipename "example_pipe"
[INFO] CreateFileW("\\.\pipe\example_pipe", ...) -> 332
[INFO]   Connected to existing pipe
[INFO] WriteFile(332, "Client says tutJxQNpew", 22, pNbBytesWritten, 0) -> 1
[INFO]   Wrote 22 bytes
[INFO] ReadFile(332, readBuffer, 2048, pNbBytesRead, 0) -> 1
[INFO]   Got data (22 bytes): "Server says FSrHdjnLcr"
```

ACLs

Named pipes are securable objects, their DACL can be set at creation time

```
PS > .\accesschk64.exe \\.\pipe\ntsvcs

\\.\pipe\ntsvcs
  RW Everybody
  RW AUTORITE NT\ANONYMOUS LOGON
  RW BUILTIN\Administrators
```

ACLs

By default when running as administrator:

```
PS > .\printsddl.exe "example_pipe"
D:(A;;FA;;;SY)(A;;FA;;;BA)(A;;FA;;;BA)(A;;FR;;;WD)(A;;FR;;;AN)
RW NT AUTHORITY\System
RW BUILTIN\Administrators
R  Everybody
R  NT AUTHORITY\ANONYMOUS LOGON
```

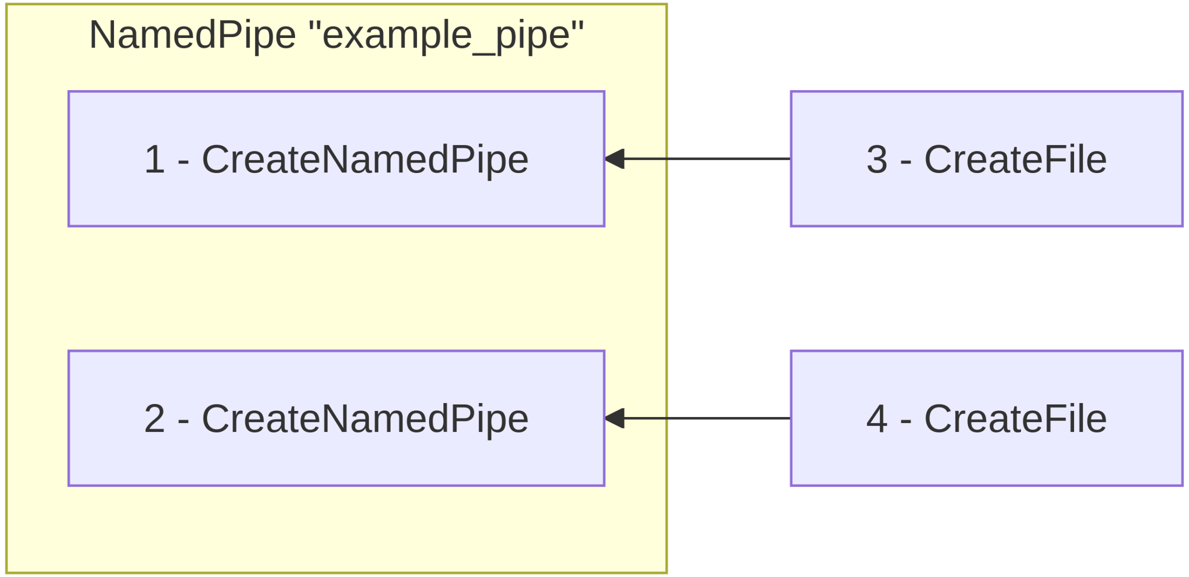
When running the server as non-administrator

```
PS > .\printsddl.exe "example_pipe"
D:(A;;FA;;;SY)(A;;FA;;;BA)(A;;FA;;;S-1-5-21-1687563665-1533190766-2569360332-1002)(A;;FR;;;WD)(A;;FR;;;AN)
RW NT AUTHORITY\System
RW BUILTIN\Administrators
RW DESKTOP-4NC0BMW\user
R  Everybody
R  NT AUTHORITY\ANONYMOUS LOGON
```

Listen for several clients

Listening to several clients implies calling CreateNamedPipe several times.

Instances are queued in a FIFO, each call to CreateFile dequeues one instance of the pipe.



Listen for several clients

```
PS > .\pipelist64.exe
Pipe Name                                    Instances       Max Instances
---------                                    ---------       -------------
ntsvcs                                            4               -1
```

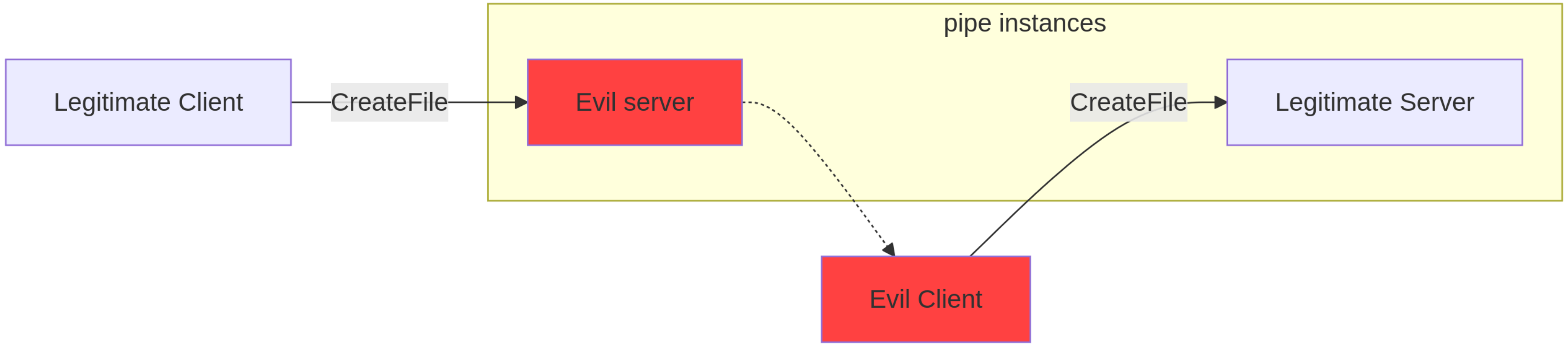
```
PS > .\pipe.exe -mode sync -pipename "ntsvcs"
[INFO] CreateNamedPipeW("\\.\pipe\ntsvcs", ...) -> 340
[INFO] ConnectNamedPipe(308, 0)
```

```
PS > .\pipelist64.exe
Pipe Name                                    Instances       Max Instances
---------                                    ---------       -------------
ntsvcs                                            5               -1
```

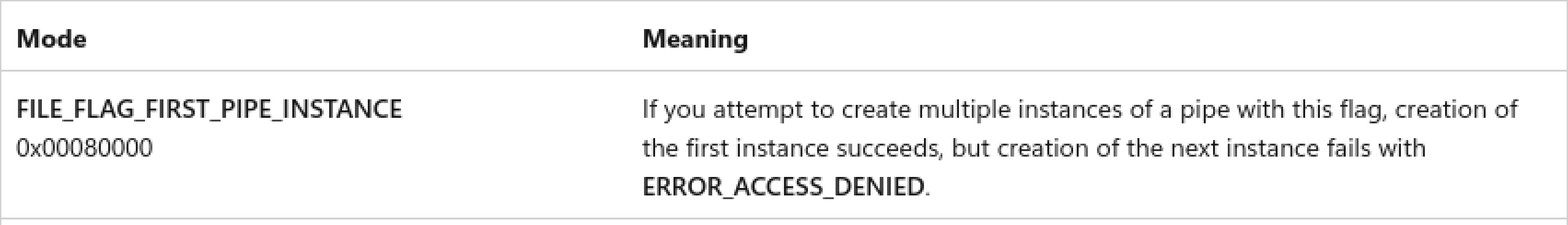
We can listen on top of an existing pipe instances, provieded we have the appropriate permissions (FILE_CREATE_PIPE_INSTANCE or FILE_APPEND_DATA or GENERIC_WRITE)

Common attacks

The Access rights of the pipes are the access rights of the first caller to CreateNamedPipe



Mitigations



Mode	Meaning
FILE_FLAG_FIRST_PIPE_INSTANCE 0x00080000	If you attempt to create multiple instances of a pipe with this flag, creation of the first instance succeeds, but creation of the next instance fails with ERROR_ACCESS_DENIED.

```
dwOpenMode = dwOpenMode | windows.FILE_FLAG_FIRST_PIPE_INSTANCE
handle, err := windows.CreateNamedPipe(pipename, dwOpenMode, pipeMode, windows.PIPE_UNLIMITED_INSTANCES, 65536, 65536, 0, nil)
```

Result:

```
[INFO] Running server in mode "waitforsingleobject"
[INFO] CreateNamedPipe("\\.\pipe\thats_no_pipe_test", ...)
[INFO] CreateNamedPipe -> 352
[INFO] Running server in mode "waitforsingleobject"
[INFO] CreateNamedPipe("\\.\pipe\thats_no_pipe_test", ...)
[INFO] CreateNamedPipe -> 18446744073709551615, Access denied.
```

Mitigations

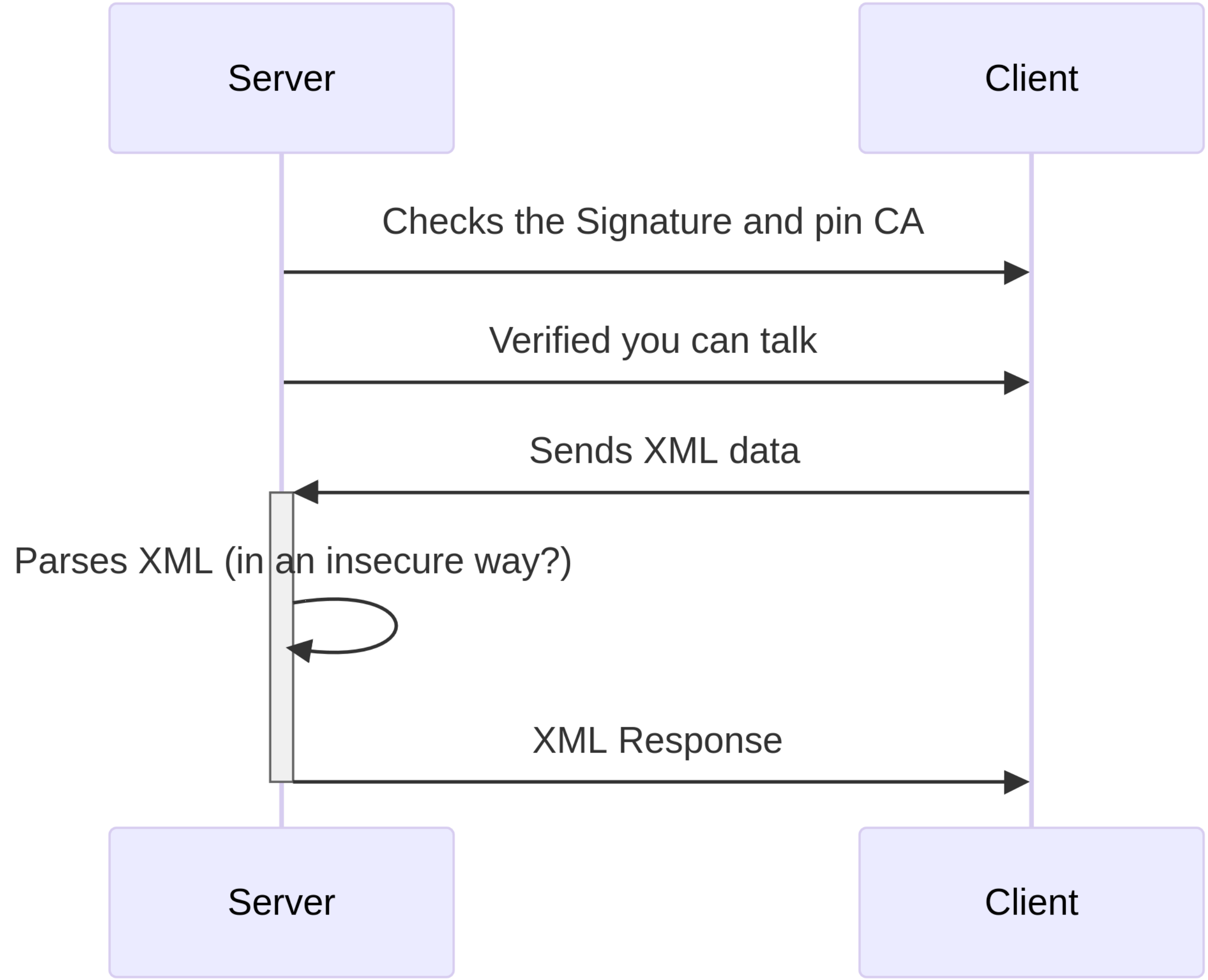
ACLs won't be enough if:

- The client process has to run in the context of the user (e.g. chrome, MSTSC)

The server could check that:

- The connecting process has a PID in an allow-list
- The exe of the connecting process is signed by a specific Certificate Authority

Example


Server

Client

Checks the Signature and pin CA

Verified you can talk

Sends XML data

Parses XML (in an insecure way?)

XML Response

Server

Client

Bypassing mitigations

- Injecting into a legitimate process at run-time (Frida)
- Changing the behavior of `NtReadFile` and `NtWriteFile` (Interceptor.attach)
- Use an HTTP Proxy to expose data to the security researcher (e.g. Burpsuite)

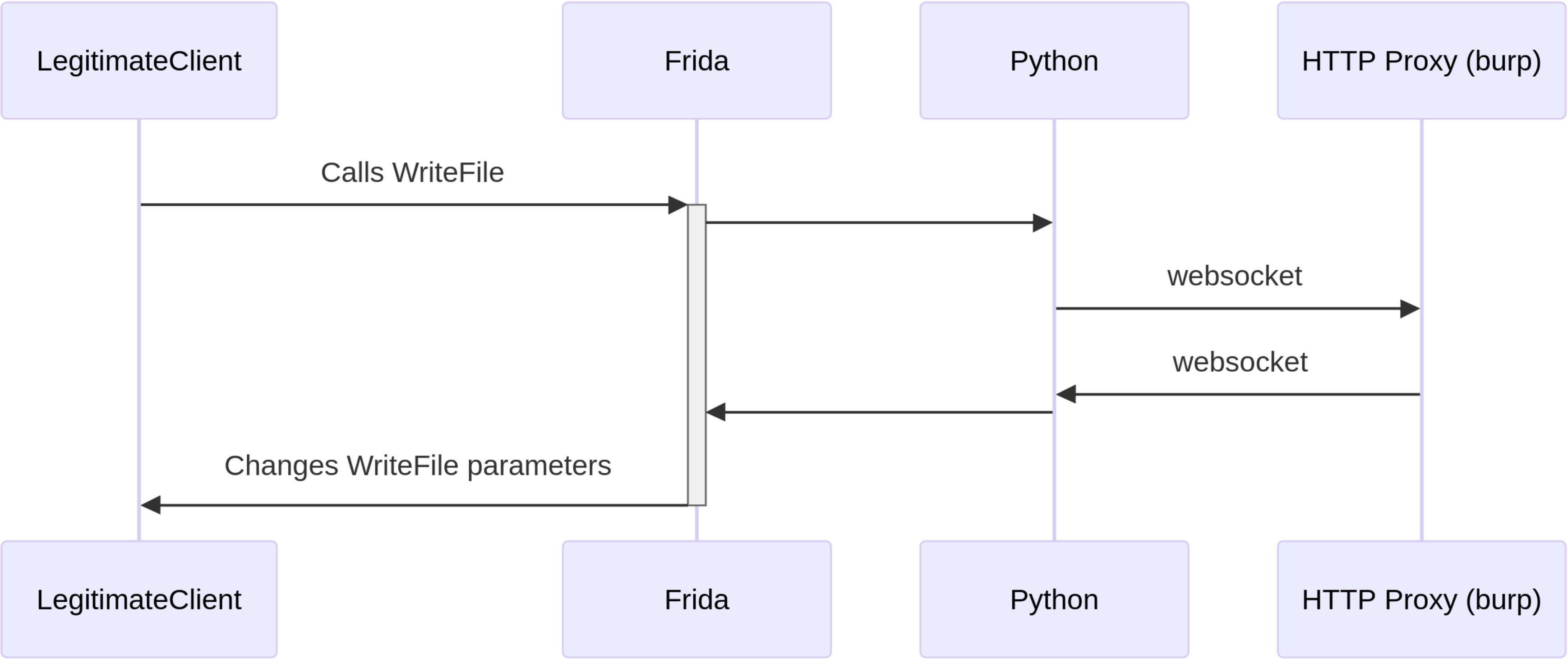
Frida 101

Process intrumentation tool. Using it in JavaScript looks like:

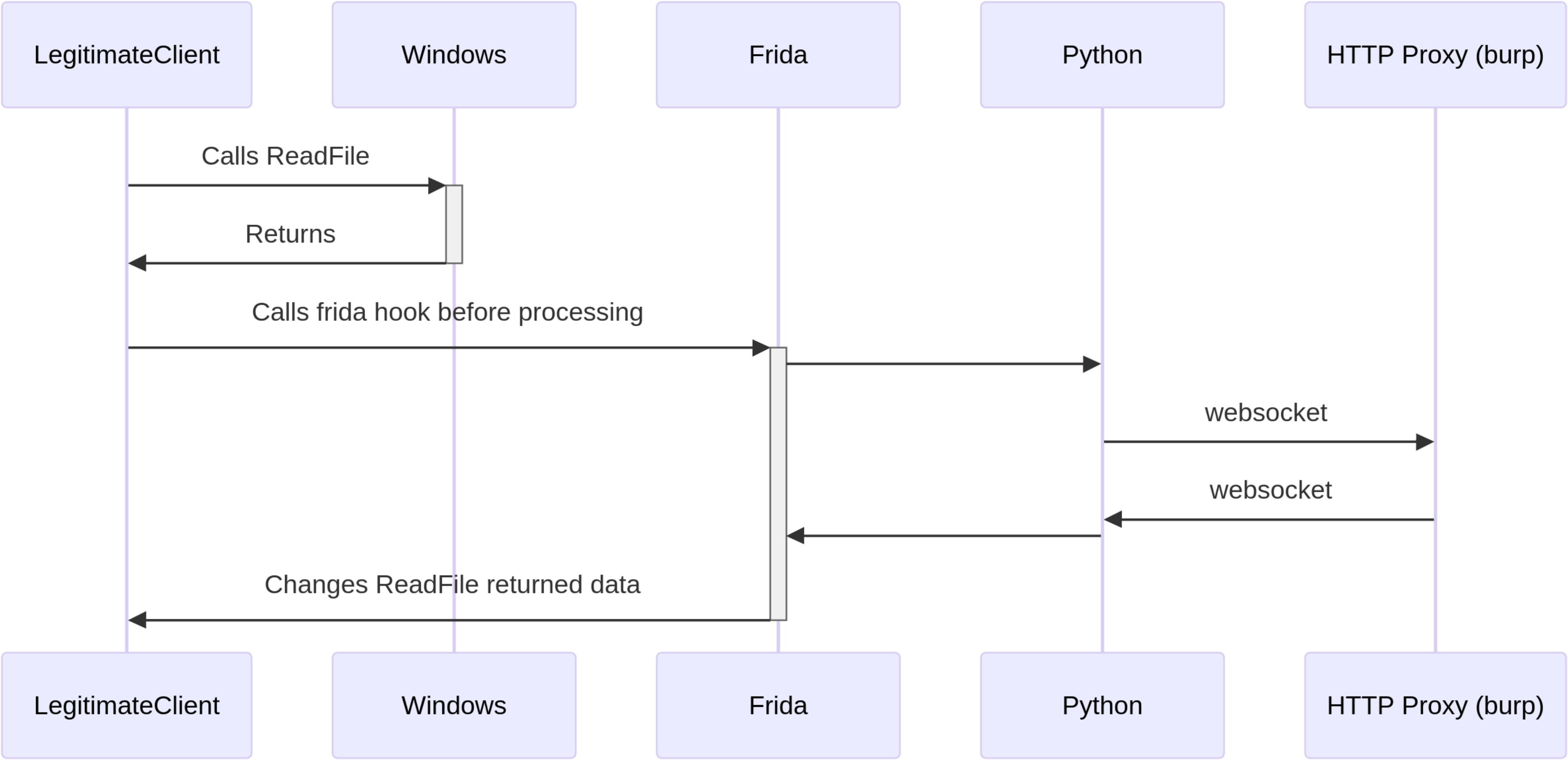
```
Interceptor.attach(Module.getExportByName(null, "NtWriteFile"), {
  onEnter: (args) => {
    const FileHandle = args[0];
    console.log(FileHandle.toInt32());
    args[0] = ptr(0x10); // Changing the Handle before the call to NtWriteFile
  },
  onLeave: (result) => {
    const NtStatus = result;
    result = ptr(0x0); // Ensure the NtWriteFile function returns STATUS_SUCCESS
  }
})
```

You can load this javascript snippet using Python

WriteFile flow



ReadFile flow



Catch 1: asynchronous reads

```
BOOL ReadFile(
  [in]                HANDLE       hFile,
  [out]               LPVOID       lpBuffer,
  [in]                DWORD        nNumberOfBytesToRead,
  [out, optional]     LPDWORD      lpNumberOfBytesRead,
  [in, out, optional] LPOVERLAPPED lpOverlapped
);

BOOL ReadFileEx(
  [in]            HANDLE                          hFile,
  [out, optional] LPVOID                          lpBuffer,
  [in]            DWORD                           nNumberOfBytesToRead,
  [in, out]       LPOVERLAPPED                    lpOverlapped,
  [in]            LPOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine
);
```

When lpOverlapped is not NULL, the syscall returns immediately. The program has to call another function to know when the data has been read.

Catch 1: asynchronous reads

```
typedef struct _OVERLAPPED {
  ULONG_PTR Internal;
  ULONG_PTR InternalHigh;
  union {
    struct {
      DWORD Offset;
      DWORD OffsetHigh;
    } DUMMYSTRUCTNAME;
    PVOID Pointer;
  } DUMMYUNIONNAME;
  HANDLE    hEvent;
} OVERLAPPED, *LPOVERLAPPED;
```

The hEvent is a Synchronization object used to signal the process that something happenned.

Catch 1: asynchronous reads

Developers tends to use one of these functions:

- WaitForSingleObject (Ex)
- WaitForMultipleObject (Ex)
- GetOverlappedResult (Ex)
- GetQueuedCompletionStatus (Ex)

We can maintain a list of overlapped operations that are pending for the process, when one of these functions dequeues an overlapped operation, we intercept it.

Catch 1: asynchronous reads

```
Interceptor.attach(NtReadFileAddr, {
onEnter: function(this: NtReadFileInvocationContext, args: InvocationArguments) {
  this.FileHandle = args[0]; // [in] HANDLE
  this.Event = args[1]; // [in, optional] HANDLE
  this.ApcRoutine = args[2]; // [in, optional] PIO_APC_ROUTINE
  this.ApcContext = args[3]; // [in, optional] PVOID
  this.IoStatusBlock = args[4]; // [out] PIO_STATUS_BLOCK
  this.Buffer = args[5]; // [out] PVOID
  this.Length = args[6]; // [in] ULONG
  this.ByteOffset = args[7] // [in, optional] PLARGE_INTEGER
  this.Key = args[8] // [in, optional] PULONG

  // Check if the Handle is a NamedPipe, and if we should intercept it
  this.handlePath = getPathByHandle(this.FileHandle)
  if (!isTargetHandlePath(this.handlePath)) { this.doIntercept = false; return }

  if (this.Event.toInt32() !== 0) {
    // This is an overlapped/asynchronous operation
    // register the overlapped operation for further use in getOverlappedResult
    pushOverlappedOperation({
      pOverlapped: this.IoStatusBlock,
      pBuffer: this.Buffer,
      bufferLength: this.Length.toInt32(),
      hEvent: this.Event.toInt32(),
      handleId: this.FileHandle.toInt32(),
      handlePath: this.handlePath,
    })
  }
```

Catch 1: asynchronous reads

```
Interceptor.attach(getOverlappedResultAddr, {
onEnter: function(args) {
  const handle = args[0];
  const lpOverlapped = args[1];
  const nbBytesTransferred = args[2];
  const bWait = args[3];

  const handlePath = getPathByHandle(handle);

  if (!isTargetHandlePath(handlePath)) {
    // Not something we monitor, exit
    return
  }
  const overlappedOperation = popOverlappedOperationByOverlapped(lpOverlapped)

  this.lpOverlapped = lpOverlapped
  this.nbBytesTransferred = nbBytesTransferred
  this.handlePath = handlePath
  this.handleId = handle.toInt32()

  if (overlappedOperation === undefined) { return }

  // Save all context data so that we can access them after the syscall
  this.doIntercept = true
  this.lpOverlapped = lpOverlapped
  this.buffer = overlappedOperation.pBuffer
  this.nbBytesTransferred = nbBytesTransferred
  this.handlePath = handlePath
  this.handleId = handle.toInt32()
},
```

Catch 1: asynchronous reads

```
    onLeave: function(result) {
      if (!this.doIntercept) { return }
      if (result.toInt32() === 0) { return }

      // Restore the context
      const lpOverlapped: NativePointer = this.lpOverlapped;
      const buffer: NativePointer = this.buffer;
      const bufferLength = (this.nbBytesTransferred as NativePointer).readU32();
      const handlePath: string = this.handlePath;
      const handleId: number = this.handleId;

      const identifier = sendMsg({
        funcName: "GetOverlappedResult",
        message: buffer.readByteArray(bufferLength) ?? new ArrayBuffer(0),
        id: 'to_ReadOperations',
        handlePath,
        handleId
      })
      popReadOperation({
        handleId,
        bufferLength,
        callback: (status, payload) => {
          // Handle cases
          //    - Do nothing is payload is equal to initial data
          //    - Overwrite buffer if payload is small enough
          //    - Simulate BUFFER_TOO_SMALL errors
        }
      })
    }
```

Catch 2: completion routines

```
BOOL ReadFileEx(
  [in]                HANDLE                          hFile,
  [out, optional] LPVOID                          lpBuffer,
  [in]                DWORD                           nNumberOfBytesToRead,
  [in, out]       LPOVERLAPPED                    lpOverlapped,
  [in]                LPOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine
);

NTSTATUS NtReadFile(
  _In_      HANDLE           FileHandle,
  _In_opt_ HANDLE           Event,
  _In_opt_ PIO_APC_ROUTINE  ApcRoutine,
  _In_opt_ PVOID            ApcContext,
  _Out_     PIO_STATUS_BLOCK IoStatusBlock,
  _Out_     PVOID            Buffer,
  _In_      ULONG            Length,
  _In_opt_ PLARGE_INTEGER   ByteOffset,
  _In_opt_ PULONG           Key
);
```

When ApcRoutine is non null, ApcContext contains a pointer to an IO_COMPLETION_ROUTINE

Catch 2: completion routines

We need to dynamically hook this function in NtWriteFile

```
    if (this.ApcContext.toInt32() != 0) {
      this.isOverlapped = true
      pushOverlappedOperation({
        pOverlapped: this.IoStatusBlock,
        pBuffer: this.Buffer,
        bufferLength: this.Length.toInt32(),
        hEvent: this.Event.toInt32(),
        handleId: this.FileHandle.toInt32(),
        handlePath: this.handlePath,
      })
      if (!isHooked(this.ApcContext)) {
        Interceptor.attach(this.ApcContext, {
          onEnter: completionRoutineOnEnter,
          onLeave: completionRoutineOnLeave,
        })
        attachedFunctions.push(this.ApcContext.toInt32())
      }
    }
```

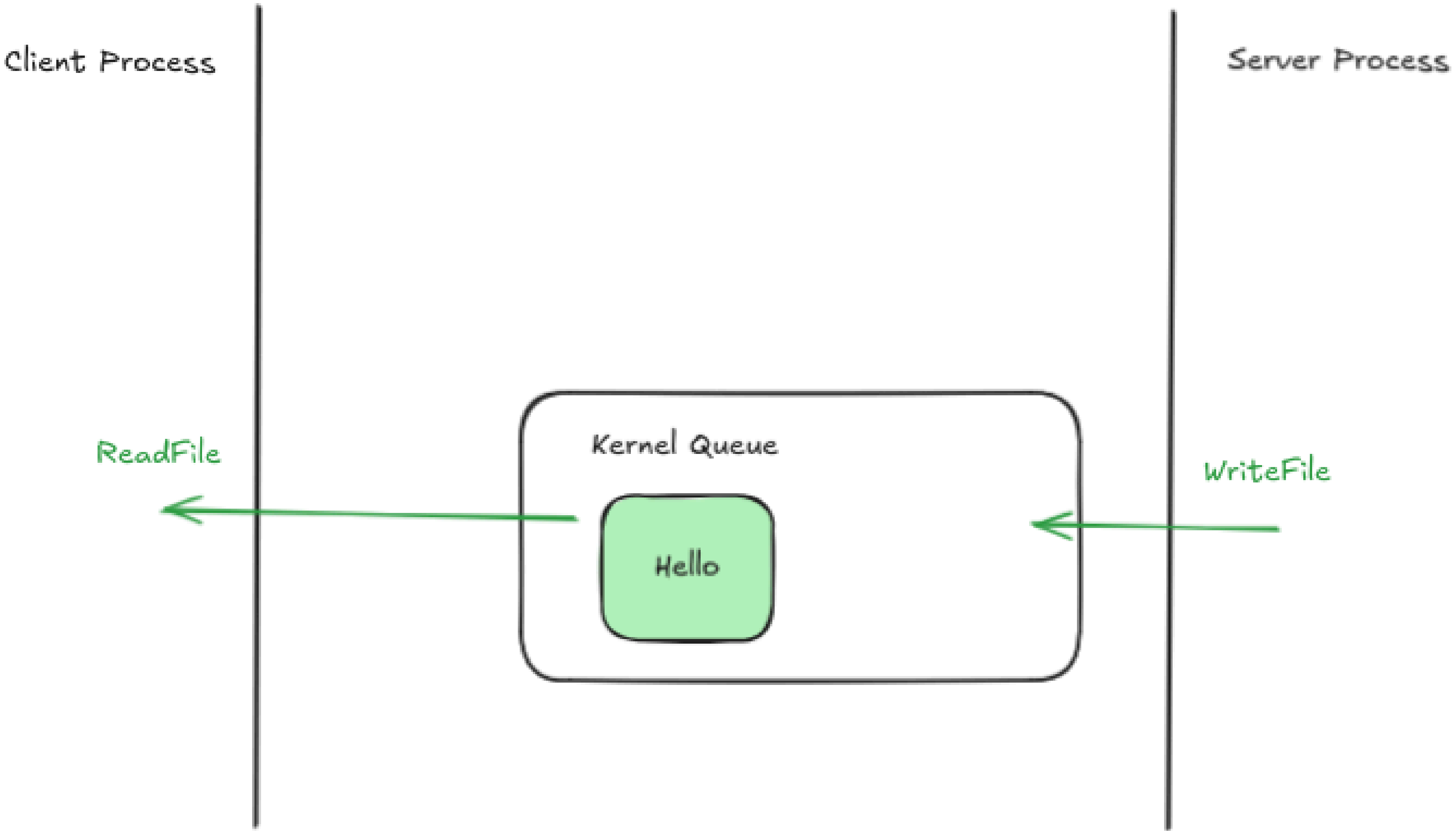
Demo time

Making the repeater work (WIP)

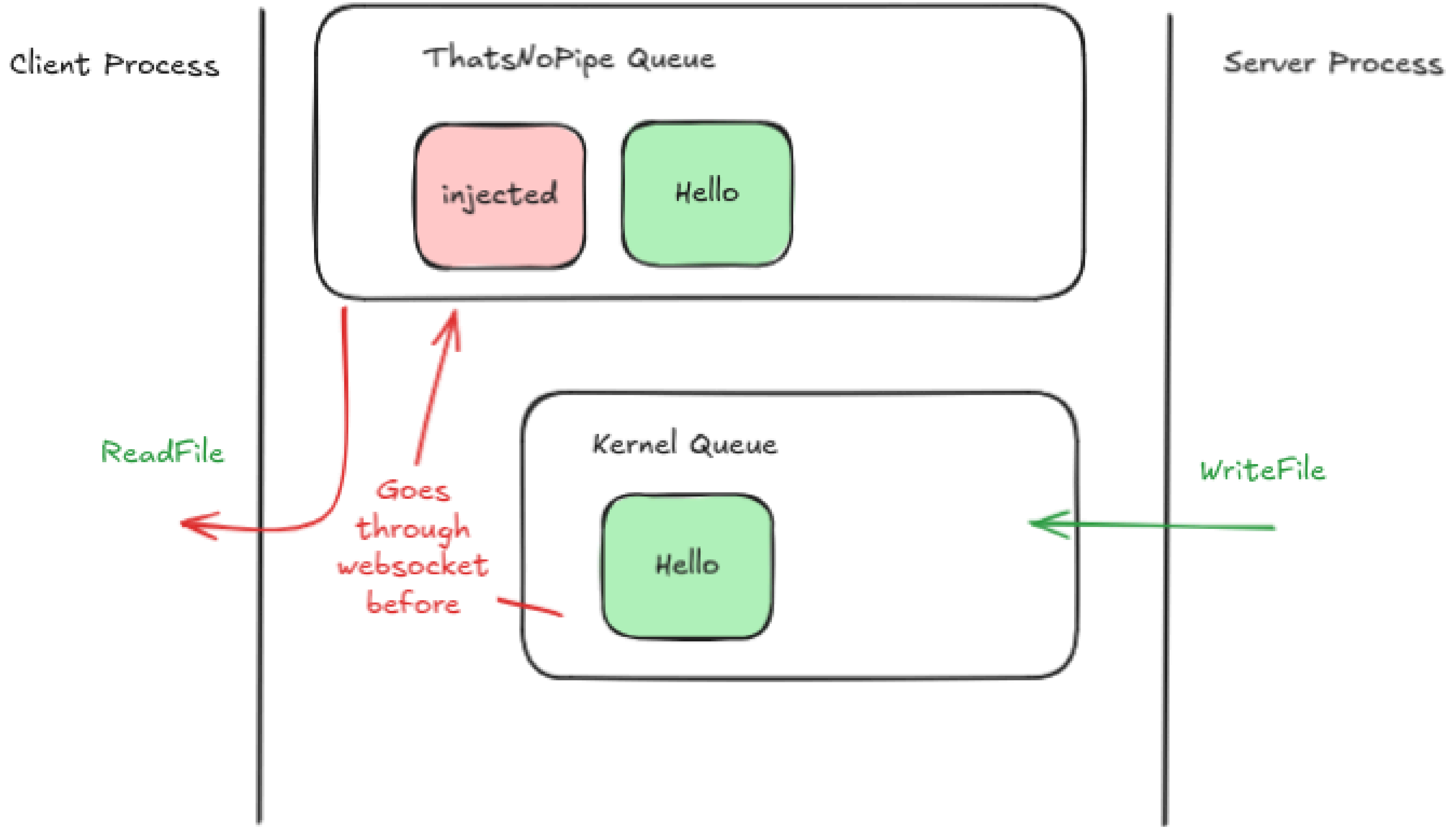
Sending a websocket message *to the server* corresponds to a **WriteFile** operation.

- Retrieve the handle (from the path of the websocket)
- Check if a WriteFile operation is pending (so that we do not block the process)
- If none are pending, call directly WriteFile from Frida
- (Check the data has been correctly written)

Making the repeater work (WIP)



Making the repeater work (WIP)



Making the repeater work (WIP)

Sending a websocket message *to the client* corresponds to a **ReadFile** operation.
This is more tricky because we need to wait for the legitimate process to call ReadFile.

- Maintain a queue of data to be read by the client
- When a ReadFile operation is dequeued by the legitimate process, intercept the buffer, then check for data in the queue corresponding to the named pipe handle
- When NtReadFile is called, check if there is already data in the queue. If yes, dequeues data and cancels the underlying syscall. Return immediately the dequeued data.

Conclusion

- Carefully review all CreateNamePipe options, especially ACLs and FILE_FLAG_FIRST_INSTANCE
- Send sensitive data to pipe clients only if you trust all processes in the client's context
- Consider data sent through named pipe as untrusted inputs, even after authentication of the client

SYNACKTIV

https://github.com/synacktiv/thats_no_pipe