Build a Reflective Shellcode Loader C2 in Golang!

WELCOME!

Glad you could join us!

1 - first

2 - second

3 - third

> whoami

- Researcher @ Active CM
- Instructor @ AntiSyphon
- Building @ aionsec.ai

> follow

home | faanross.com

linkedin | faan rossouw

x | @faanross

github | @faanross

youtube | @faanross

Let’s get a quick overview of today’s plan, so you know what to expect, and how to get the best value for your time.

First, 4 opening lectures

→ This Welcome

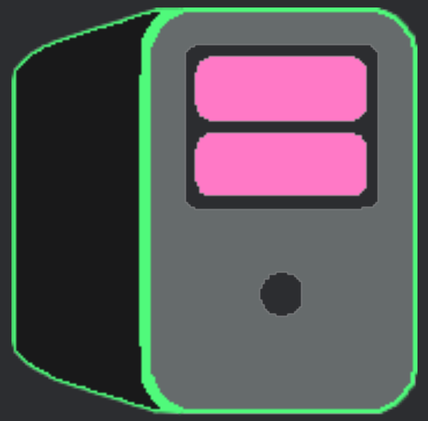
→ Setup

→ Review of Starting Code

→ What We'll Build

Then, jump into labs

Labs: Server-Side Logic



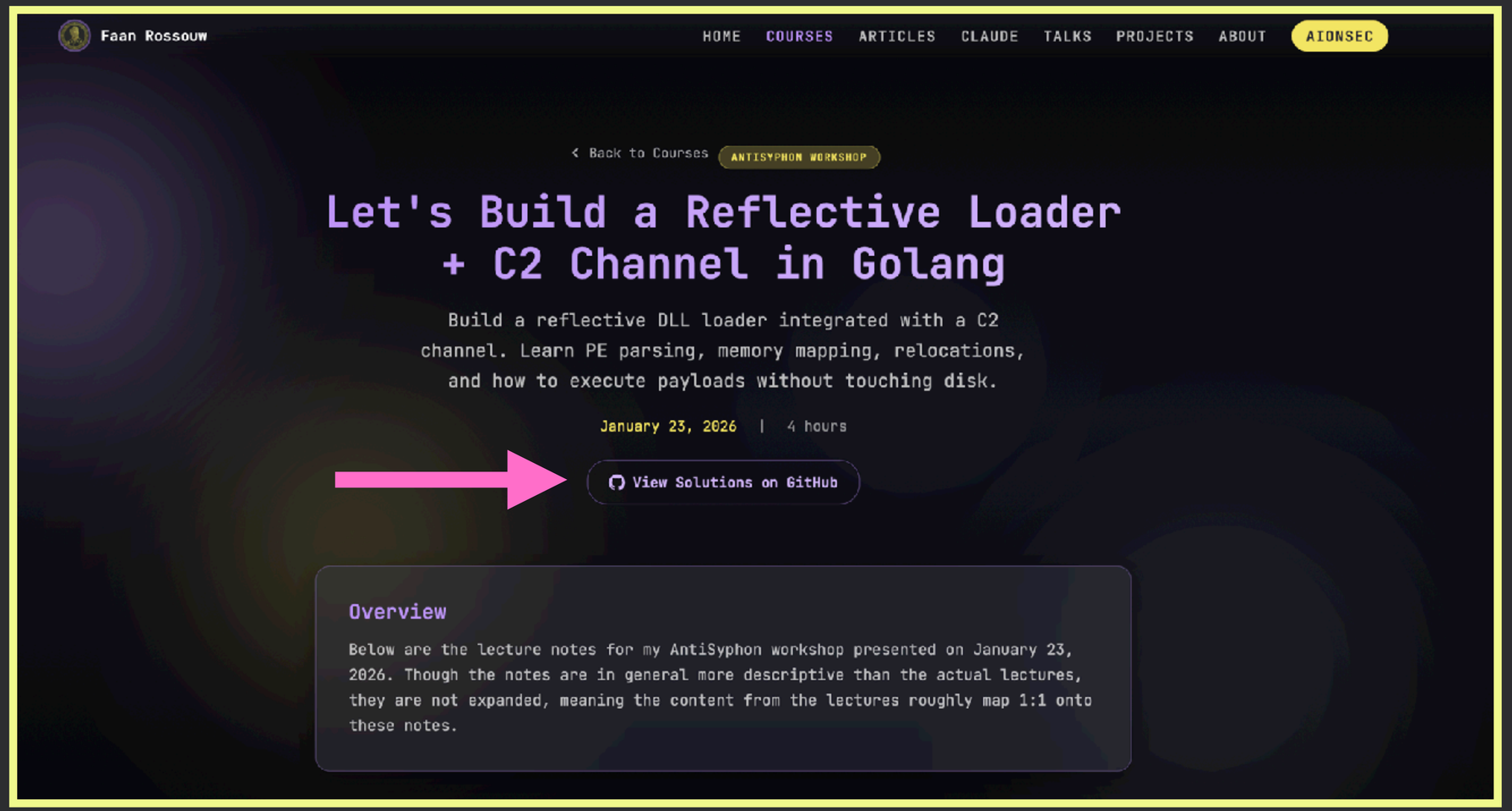
1. Implement Command Endpoint
2. Validate Command Exists
3. Validate Command Arguments
4. Process Command Arguments
5. Queue Commands
6. Dequeue and Send Commands to Agent

Labs: Agent-Side Logic

7. Create Agent Command Execution Framework
8. Implement Shellcode Orchestrator
9. Create Shellcode Doer Interface and Implementations
10. Implement Windows Shellcode Doer
11. Server Receives and Displays Results

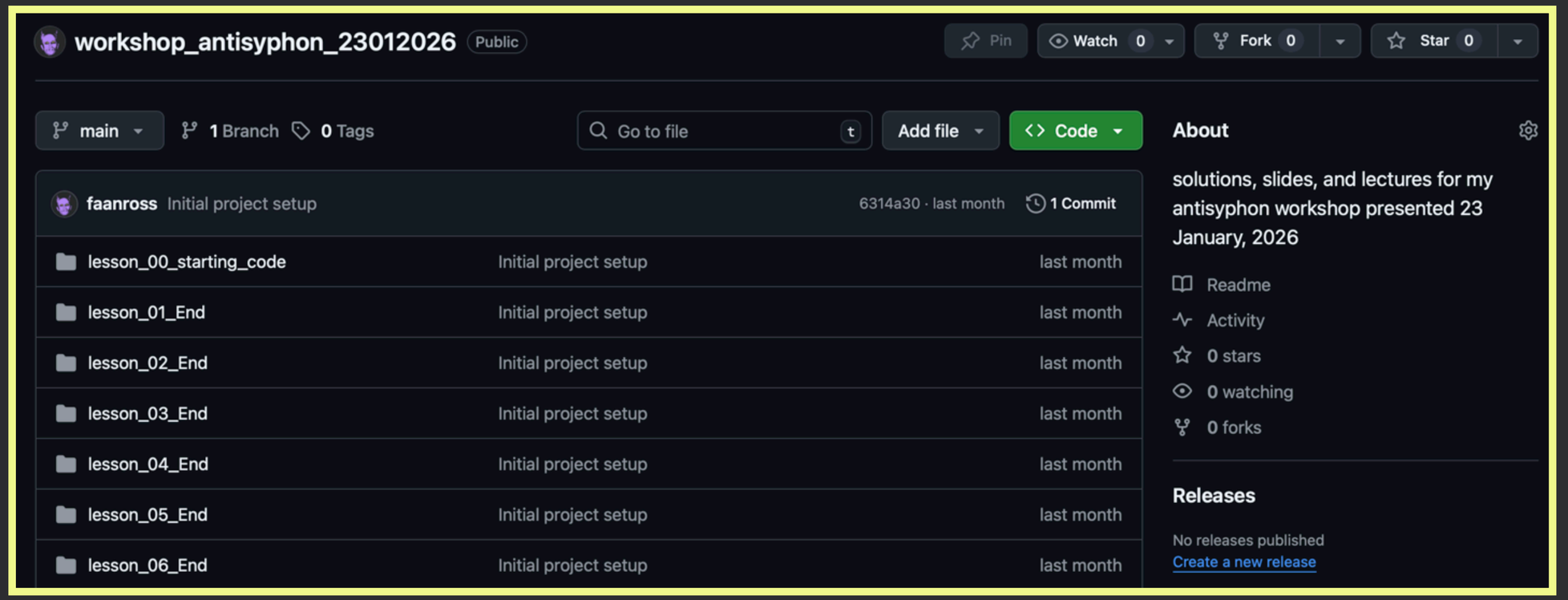
all the lectures can be found here

www.faanross.com/courses/workshop03/



all the lab code can be found here

github.com/faanross/workshop_antisyphon_23012026



let’s chat about that…

for each lab (11 total) we have:

Begin Code at the start of the lesson

→ “Skeleton code” - try completing yourself

End Code at the end of the lesson

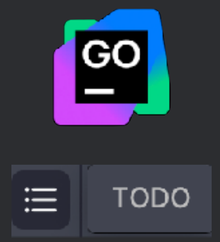
→ If you ever get lost, rejoin us next lesson

for each lab (11 total) we have:

Begin Code at the start of the lesson

→ “Skeleton code” - try completing yourself

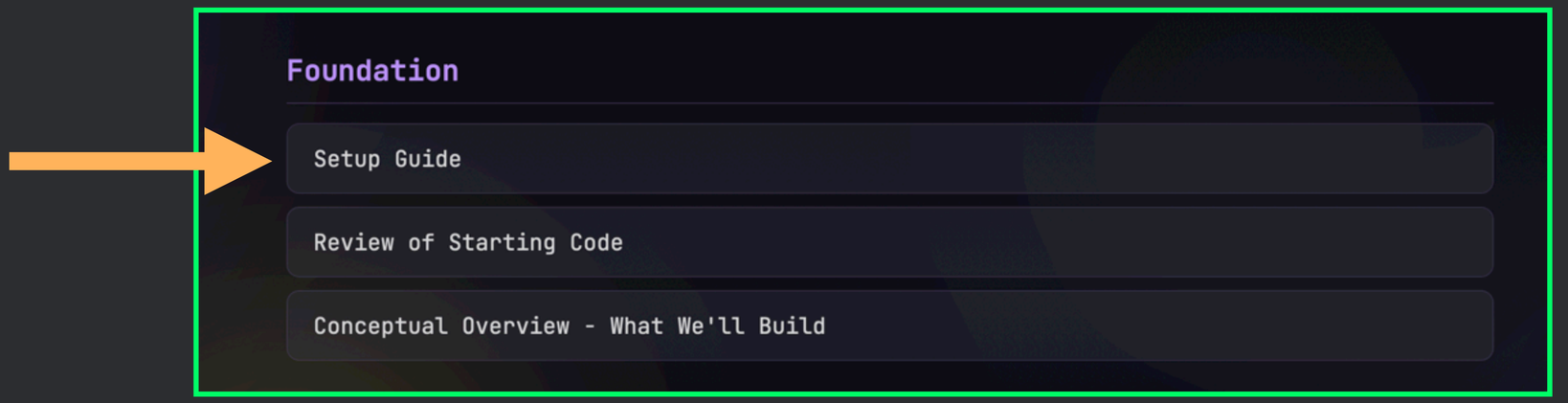
```
agentCfg := config.Config{
    // TODO Set protocol to HTTPS
}
```



use ctrl/cmd + F

SETUP

Please follow along!



You only need 4 things

→ Go (runtime environment)

→ Your IDE of Choice

→ Course Repo

→ Windows x64 “Victim” Host/VM (only for last step)

Go (runtime environment)

https://go.dev/dl/

IDE of Choice

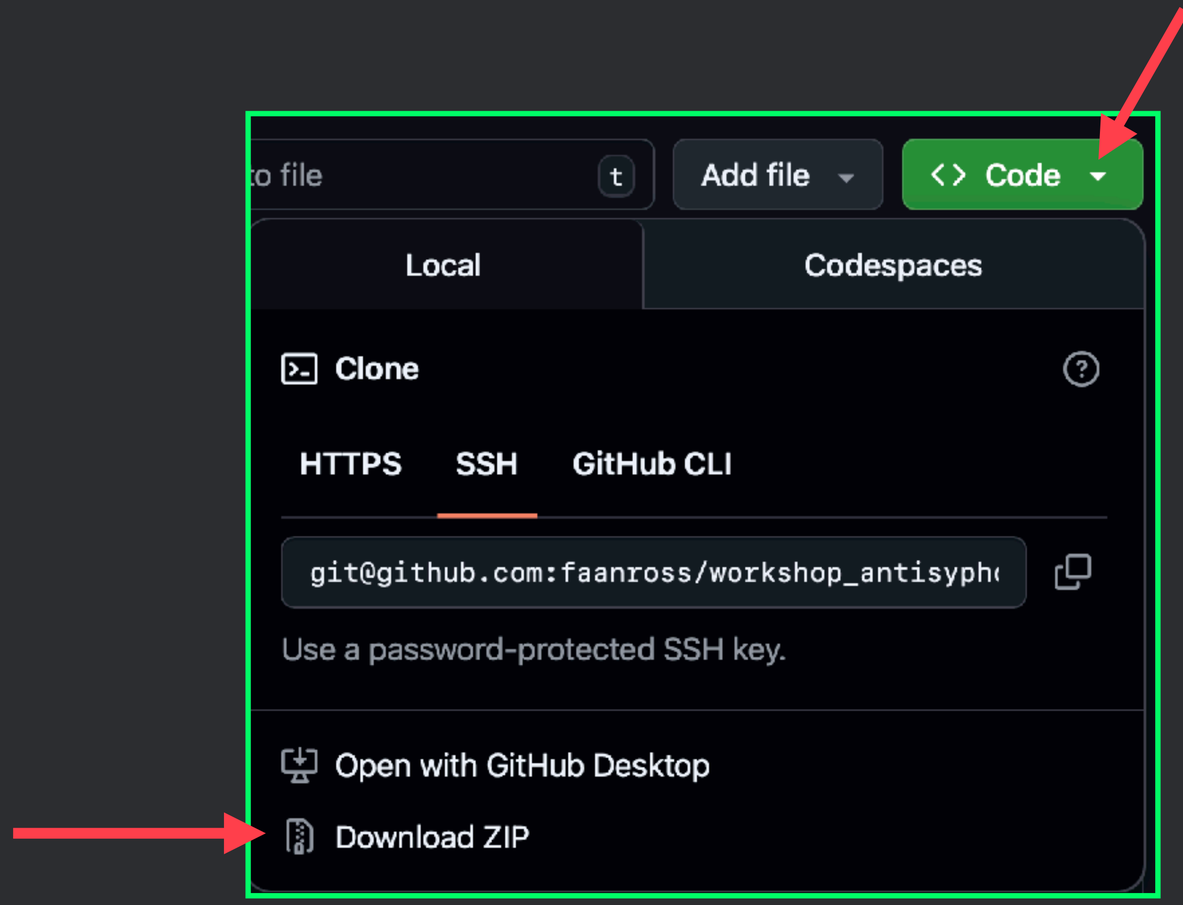
MAX_JETBRAINS

www.jetbrains.com/go/download

https://code.visualstudio.com/

Go by Google

Course Repo




```
git clone https://github.com/faanross/
workshop_antisyphon_23012026.git
```

Finally, we'll need a Windows victim

→ Extra host with Defender disabled

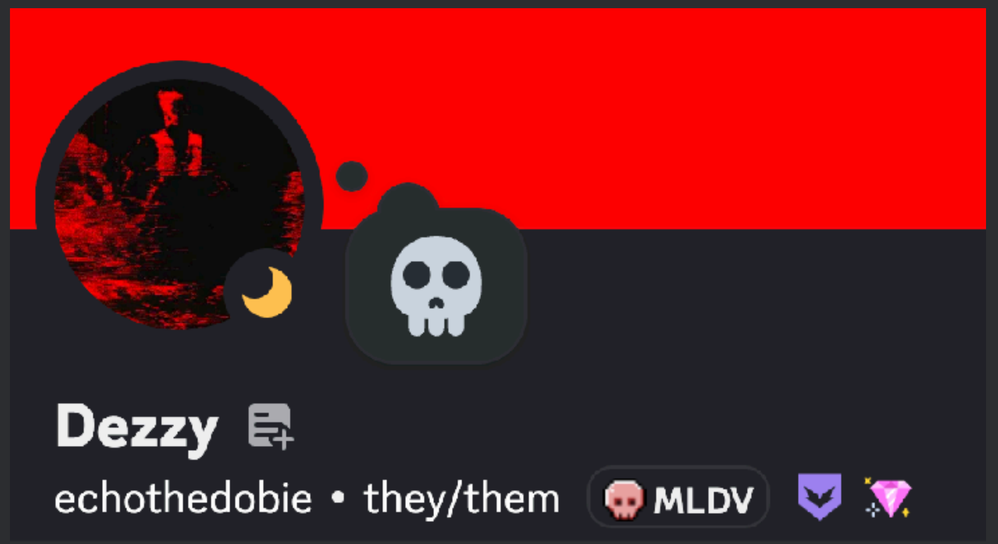
→ Has to me amd64 (no ARM)

→ If you did not prep for this, ALL GOOD

→ Only final step of final lesson requires

→ You can always go and do it later

There is still time, go get it!


Dezzy
echothedobie • they/them
MLDV

let’s review what we’re starting with

today’s workshop a bit unusual
in that we are not starting with
a blank slate, but have a basic
C2 outline in place so we can get
going on some of the more exciting
aspects of C2 - “doing stuff”

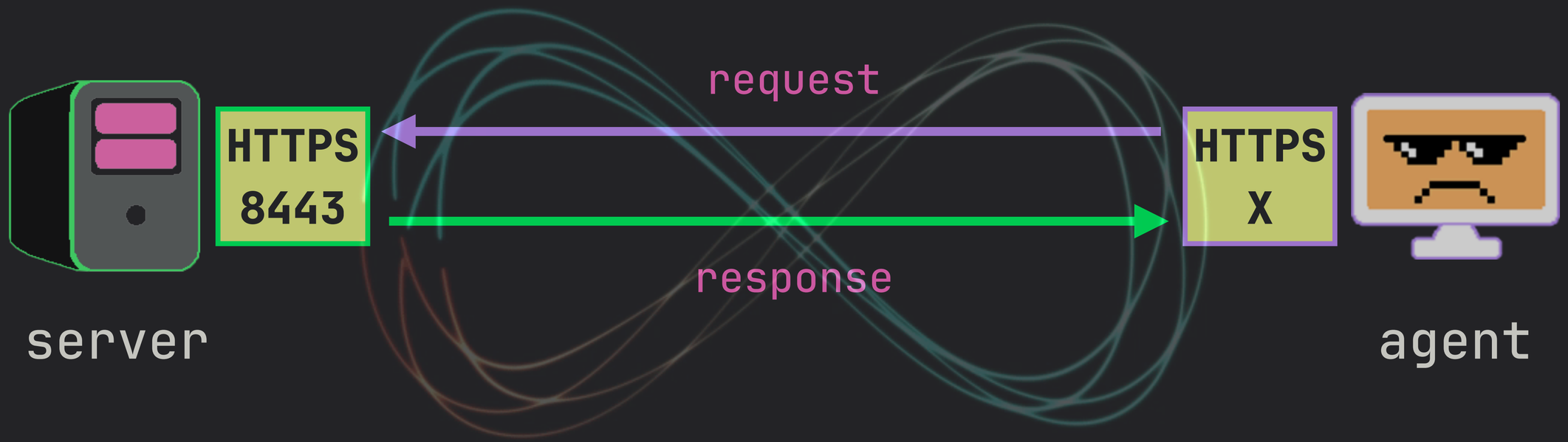
starting code

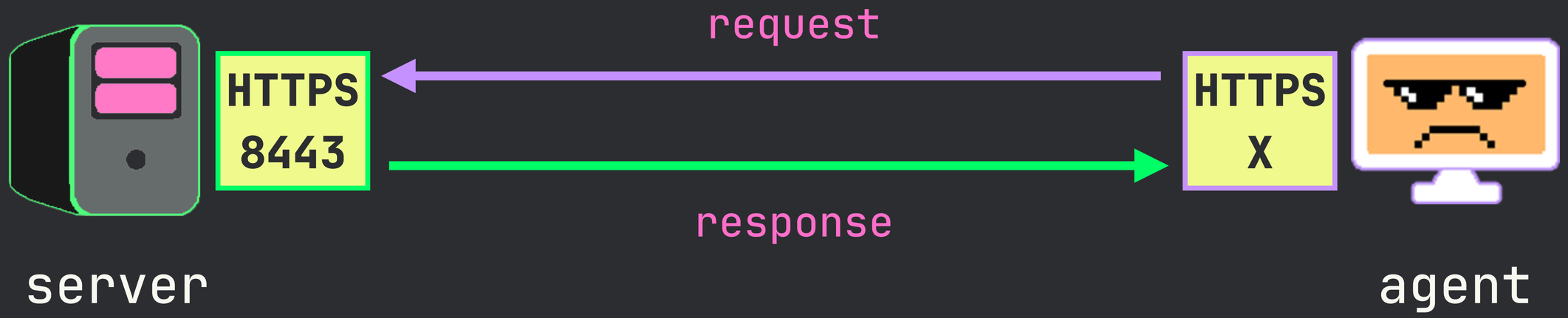
a simple server and agent

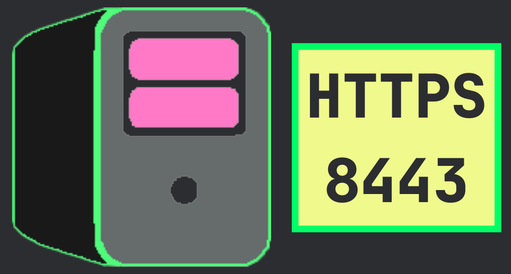
communicating over HTTPS (tcp 8443)

using request + response cycle

repeating in an infinite loop


request
HTTPS
8443
HTTPS
X
response
server
agent


request
HTTPS
8443
HTTPS
X
response
server
agent

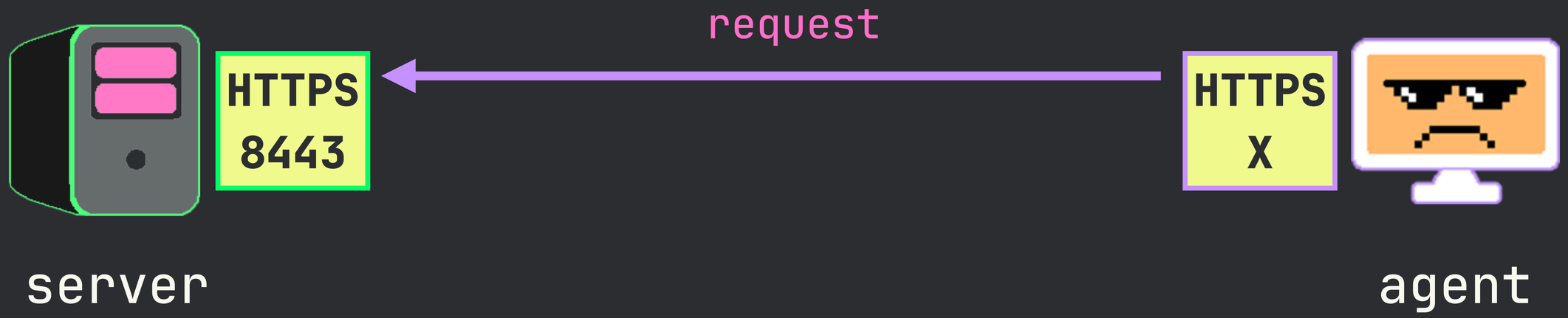


server

agent

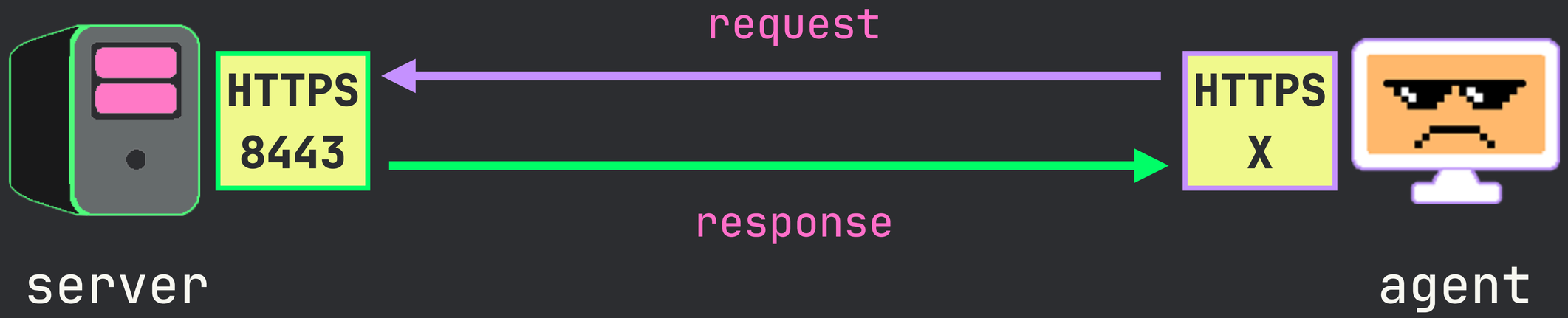
start server

```
❯ go run ./cmd/server
2025/12/02 17:18:18 Starting  server on 0.0.0.0:8443
```



start agent → connects

```
❯ go run ./cmd/agent
2025/12/02 17:18:57 Starting Agent Run Loop
2025/12/02 17:18:57 Delay: 5s, Jitter: 50%
2025/12/02 17:18:57 Response from server: "You have hit the server's root endpoint"
2025/12/02 17:18:57 Sleeping for 6.421640093s
2025/12/02 17:19:03 Response from server: "You have hit the server's root endpoint"
2025/12/02 17:19:03 Sleeping for 4.697667528s
```



server responds

```
❯ go run ./cmd/agent
2025/12/02 17:18:57 Starting Agent Run Loop
2025/12/02 17:18:57 Delay: 5s, Jitter: 50%
2025/12/02 17:18:57 Response from server: "You have hit the server's root endpoint"
2025/12/02 17:18:57 Sleeping for 6.421640093s
2025/12/02 17:19:03 Response from server: "You have hit the server's root endpoint"
2025/12/02 17:19:03 Sleeping for 4.697667528s
```

one more thing…

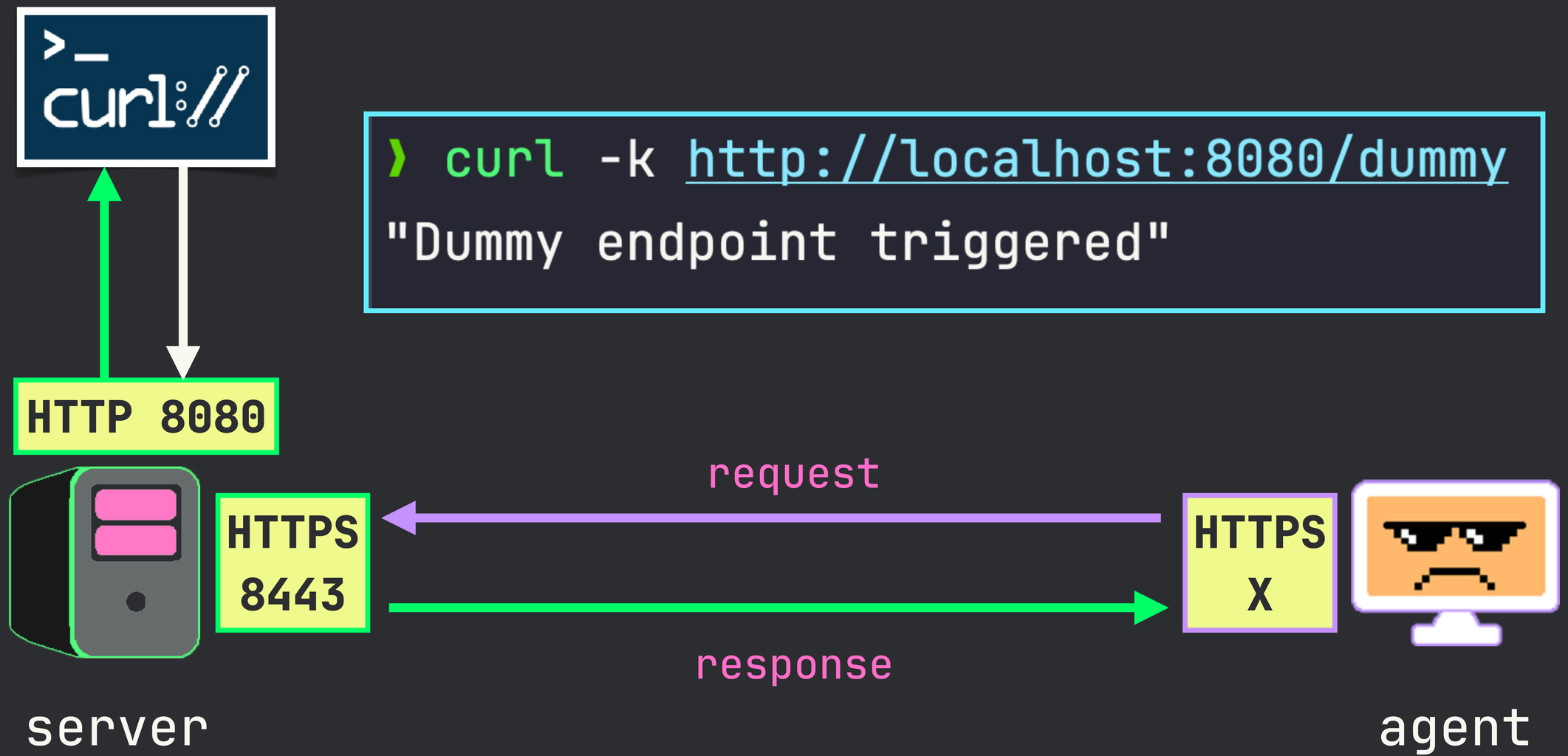
server has one API point

on tcp :8080

now just returns a message

this is foundation of what

will become our client

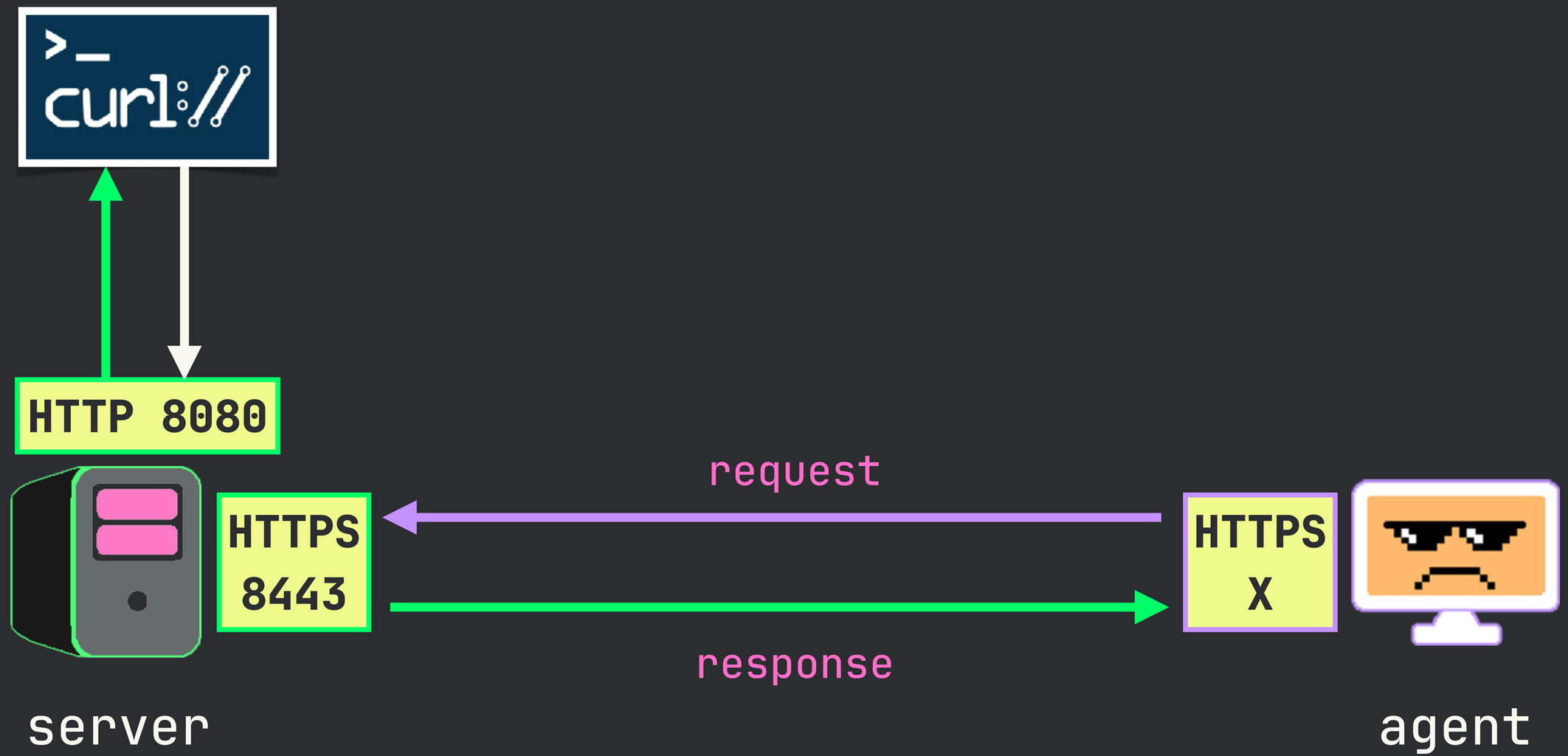

curl://
❯ curl -k http://localhost:8080/dummy
"Dummy endpoint triggered"
HTTP 8080
request
HTTPS
8443
HTTPS
X
response
server
agent

for now just want you to

have idea of what code does

what we'll build today

as we just saw, we have a server and agent that can communicate with one another, and an ability to trigger something via an API endpoint on the actual server.


>_
curl://
HTTP 8080
HTTPS
8443
request
response
HTTPS
X
server
agent

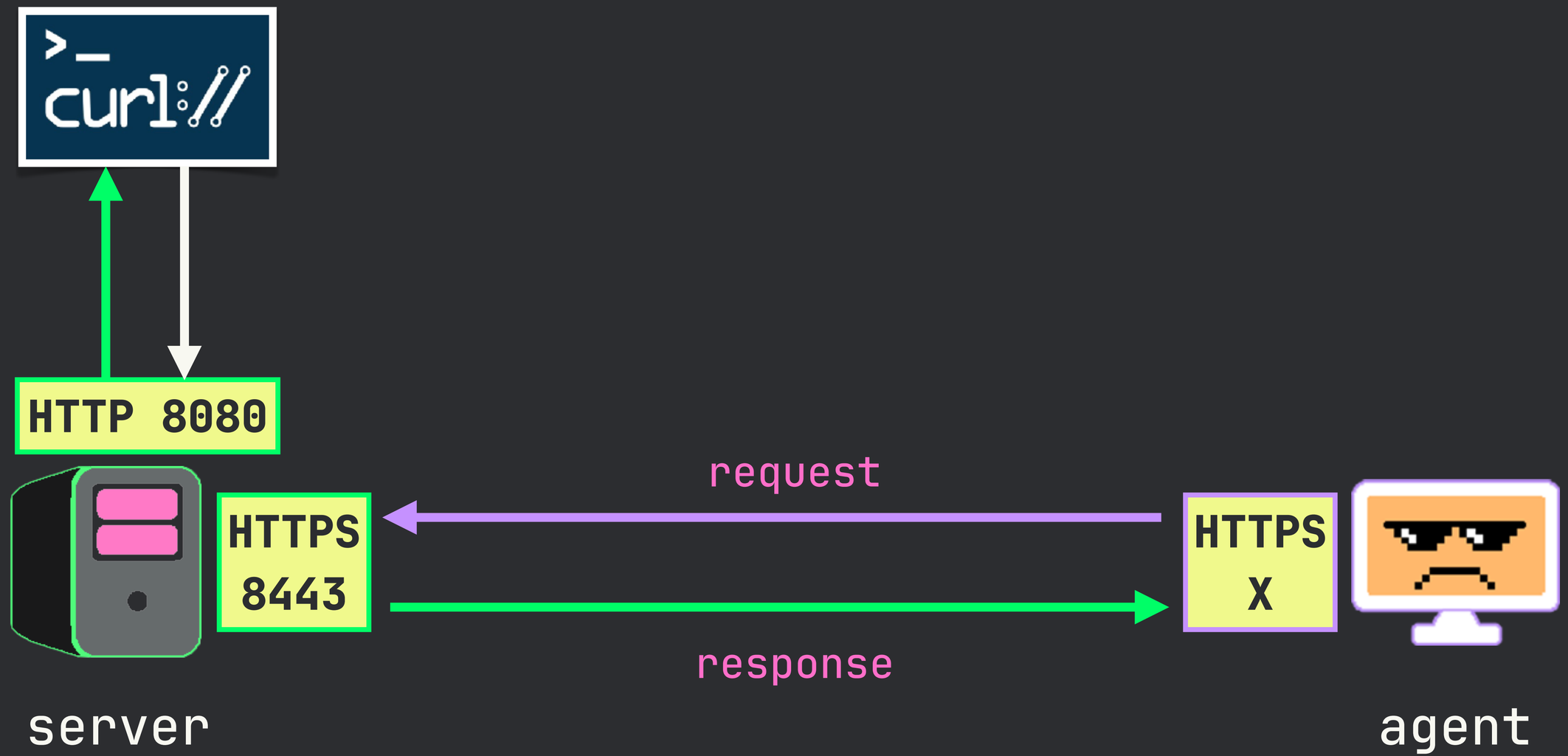
STEP 1: Implement Command Endpoint

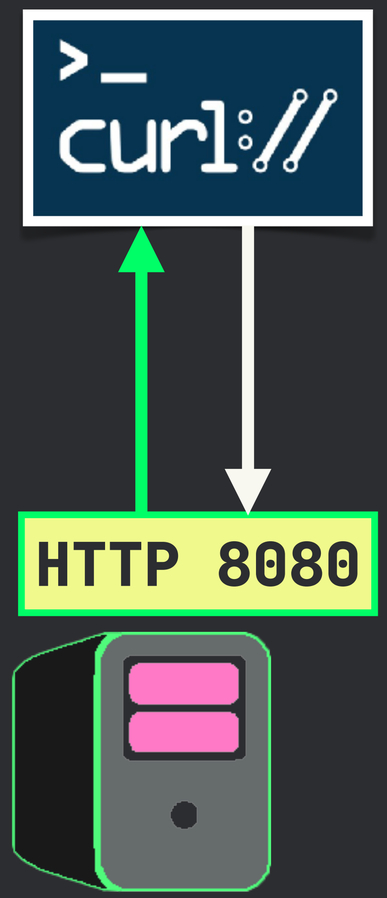
→ client hits a /command endpoint

→ client also sends POST body

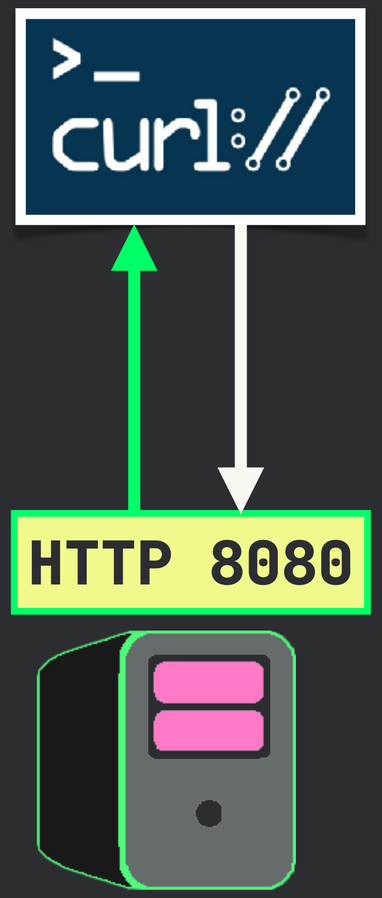
for ex: {"command": “shellcode”}

→ server handler is capable of parsing JSON

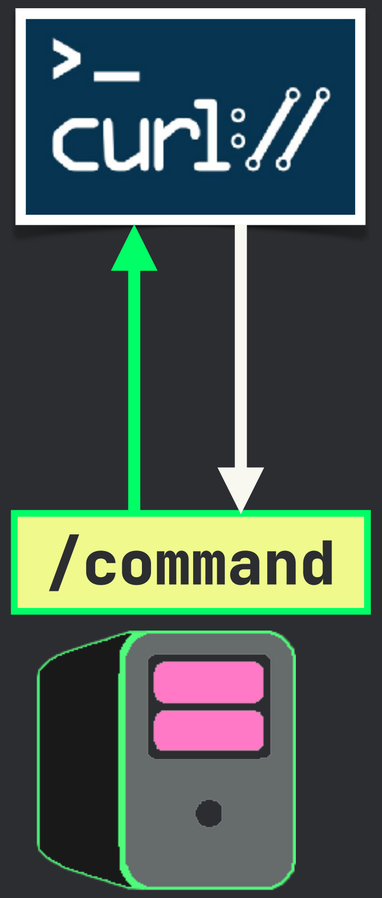

>_
curl://
HTTP 8080
HTTPS
8443
request
response
HTTPS
X
server
agent



server



server

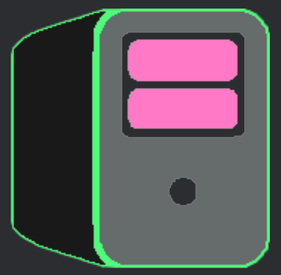


{"command": “shellcode”}

server

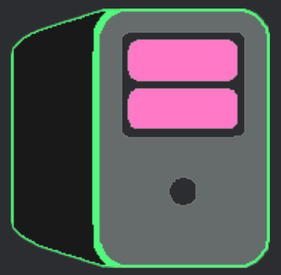
/command

{"command": “shellcode”}



server

/command “shellcode”



server

STEP 2: Validate Command Exists

→ server has parsed command

→ but we need to ensure this command is valid

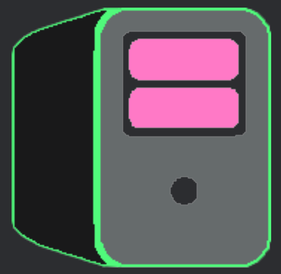
→ create registry (map) containing valid commands

→ compare parsed command to registry

→ exists: accept

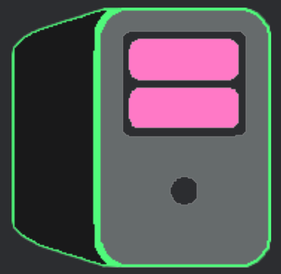
→ does not exist: reject

/command “shellcode”

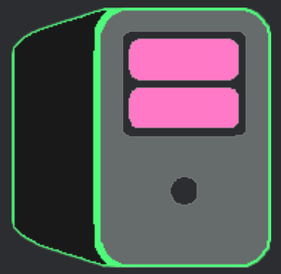


server

“shellcode”

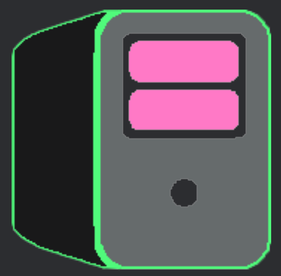


server



server

“shellcode”

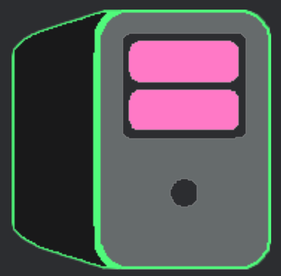


server

“shellcode”

REGISTRY

“shellcode”

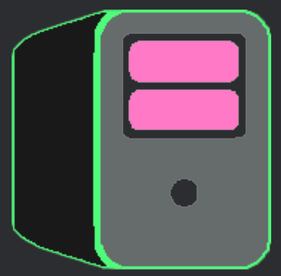


server

“shellcode”

REGISTRY

“shellcode”

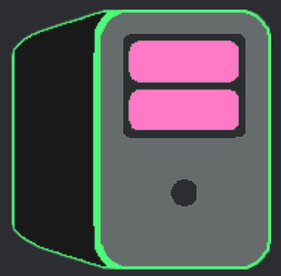


server

“shellcode”

REGISTRY

“shellcode”

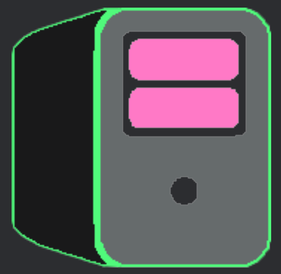


server

“shellcode”

REGISTRY

“shellcode”

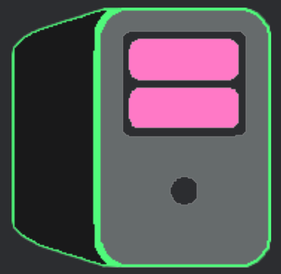


server

“chaliponga”

REGISTRY

“shellcode”

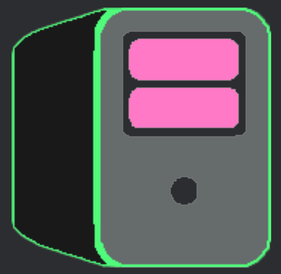


server

“chaliponga”

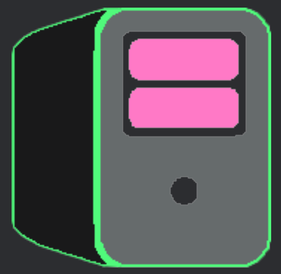
REGISTRY

“shellcode”



“chaliponga”

server



server

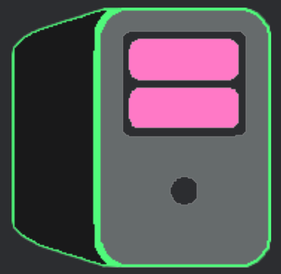
STEP 3: Validate Command Arguments

→ client does not actually just send command

→ also sent arguments

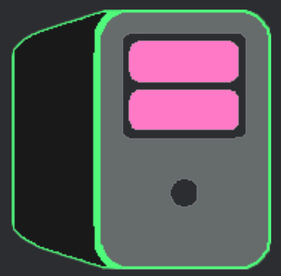
→ for shellcode: path (to DLL), function name

→ for accepted commands: validate arguments



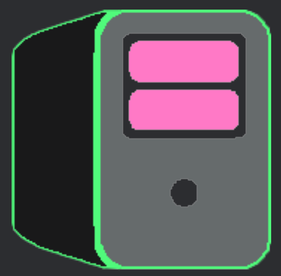
server

“shellcode”



server

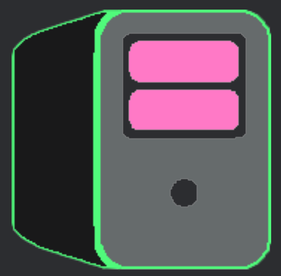
```
command: “shellcode”
path: ./payloads/calc.dll
function: LaunchCalc
```



server

Does the file exist?

```
command: “shellcode”
path: ./payloads/calc.dll
function: LaunchCalc
```

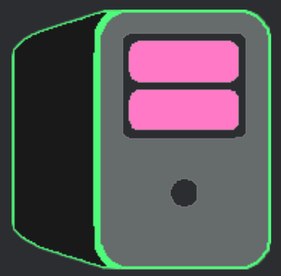


server

Does the function exist?

```
command: “shellcode”

path: ./payloads/calc.dll

function: LaunchCalc
```

Accept/Reject based on arguments



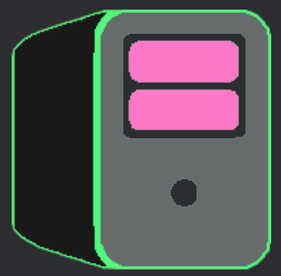
server

```
command: “shellcode”
path: ./payloads/calc.dll
function: LaunchCalc
```

STEP 4: Process Command Arguments

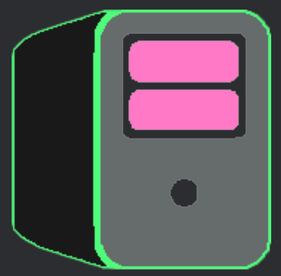
→ some arguments need to be transformed before being sent from the server to the agent

for ex: client sends the server the path to the DLL, but server should send the agent the actual (encoded) data from the DLL



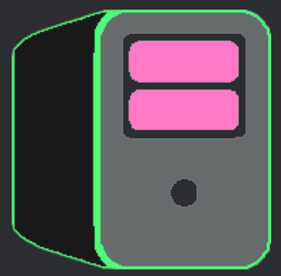
server

```
command: “shellcode”
path: ./payloads/calc.dll
function: LaunchCalc
```



server

```
command: “shellcode”
data: aGVsbG8gd29ybGQ…
function: LaunchCalc
```



server

```
command: “shellcode”
data: aGVsbG8gd29ybGQ…
function: LaunchCalc
```

STEP 5: Queue Commands

→ command + processed arguments now ready for agent

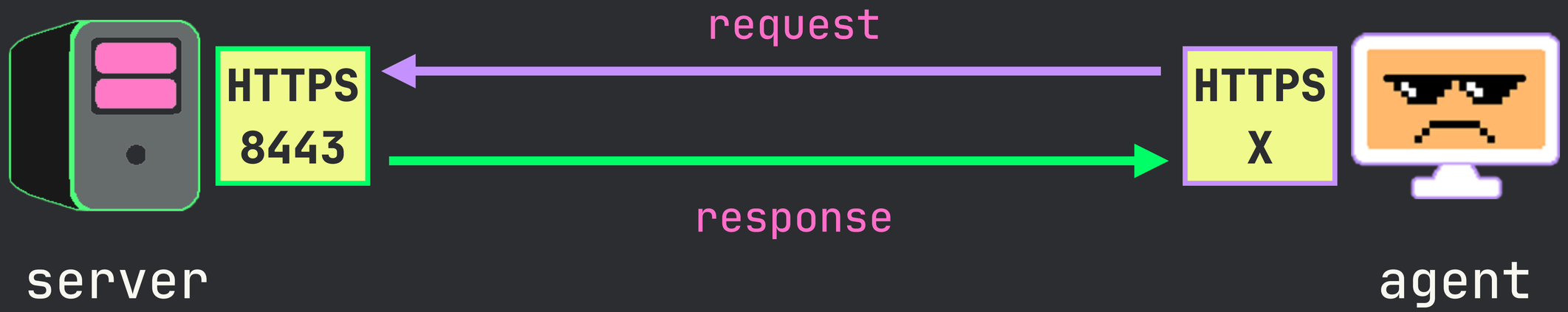
→ but remember: server cannot push to agent

→ agent HAS TO pull from server (request + response)

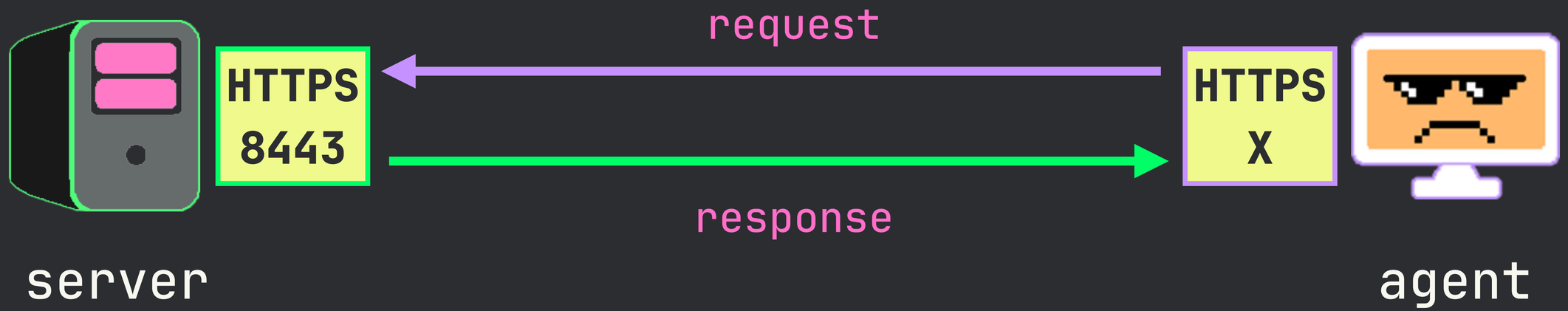
→ but it only does so periodically (delay + jitter)

→ command needs to be stored somewhere in meantime

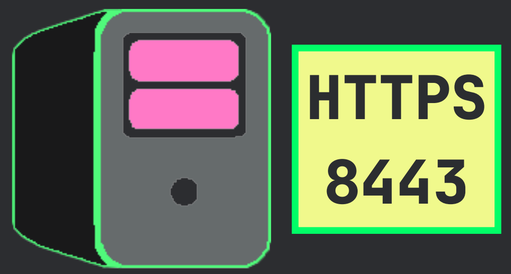
→ so then at this point we push it to a command queue


request
HTTPS
8443
HTTPS
X
response
server
agent

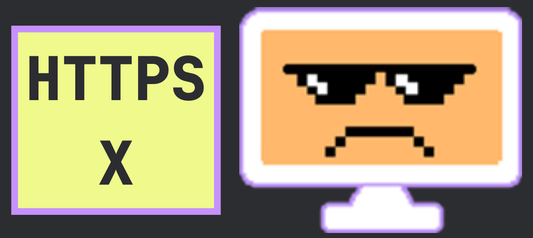
```
command: “shellcode”

data: aGVsbG8gd29ybGQ…

function: LaunchCalc
```



```
command: “shellcode”

data: aGVsbG8gd29ybGQ…

function: LaunchCalc
```

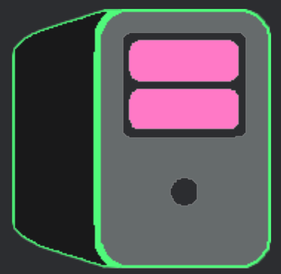


server



agent

```
command: “shellcode”

data: aGVsbG8gd29ybGQ…

function: LaunchCalc
```



HTTPS
8443

server

HTTPS
X

agent

QUEUE

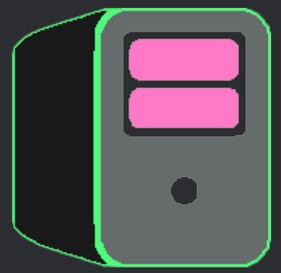
command: “shellcode”

data: aGVsbG8gd29ybGQ…

function: LaunchCalc

→ can store multiple

→ FIFO



HTTPS
8443

server

HTTPS
X

agent

STEP 6: Dequeue and Send to Agent

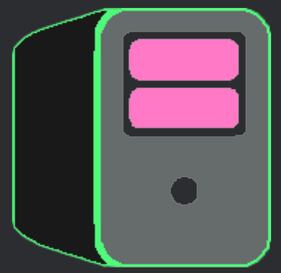
→ now when agent hits endpoint (request)

server checks → is there something at [0] in queue?

no → tells agent there's nothing

yes → removes command from queue, sends to agent

QUEUE

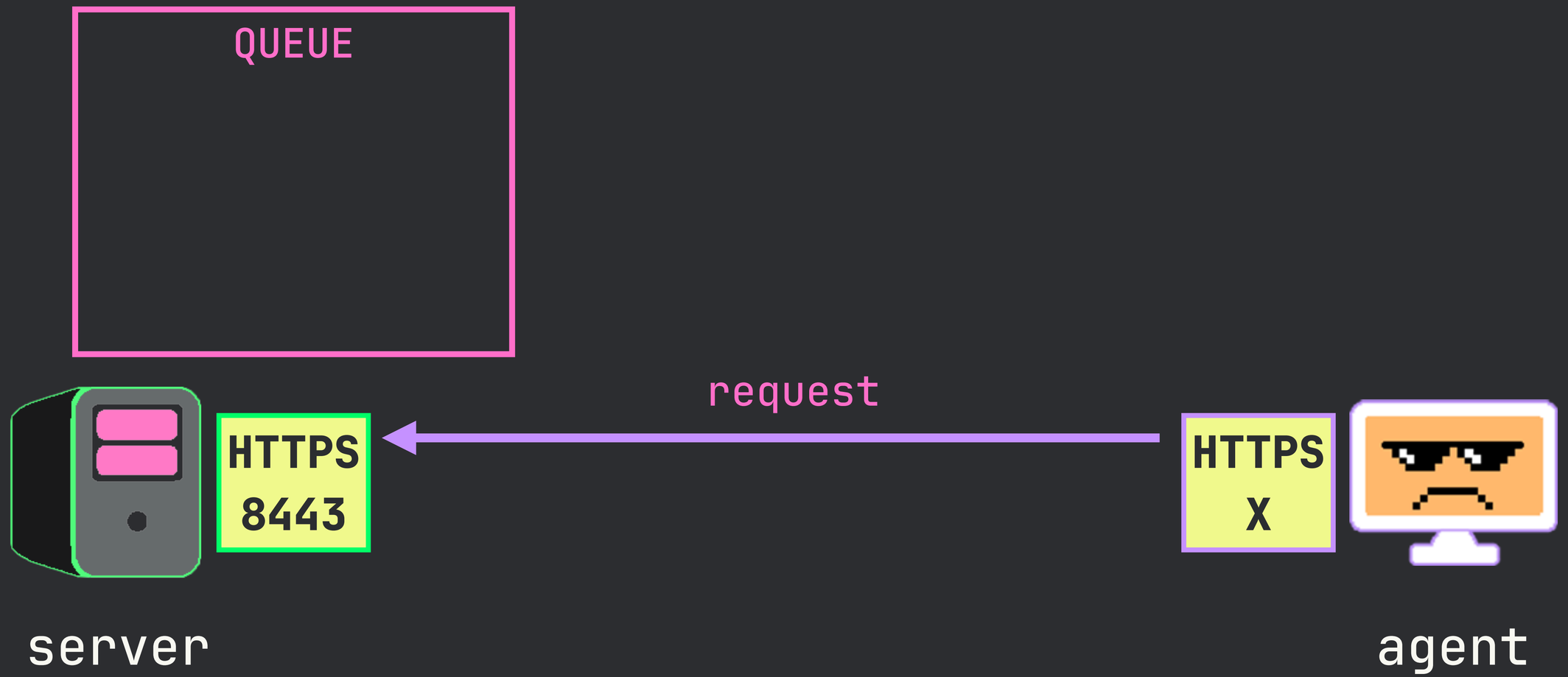


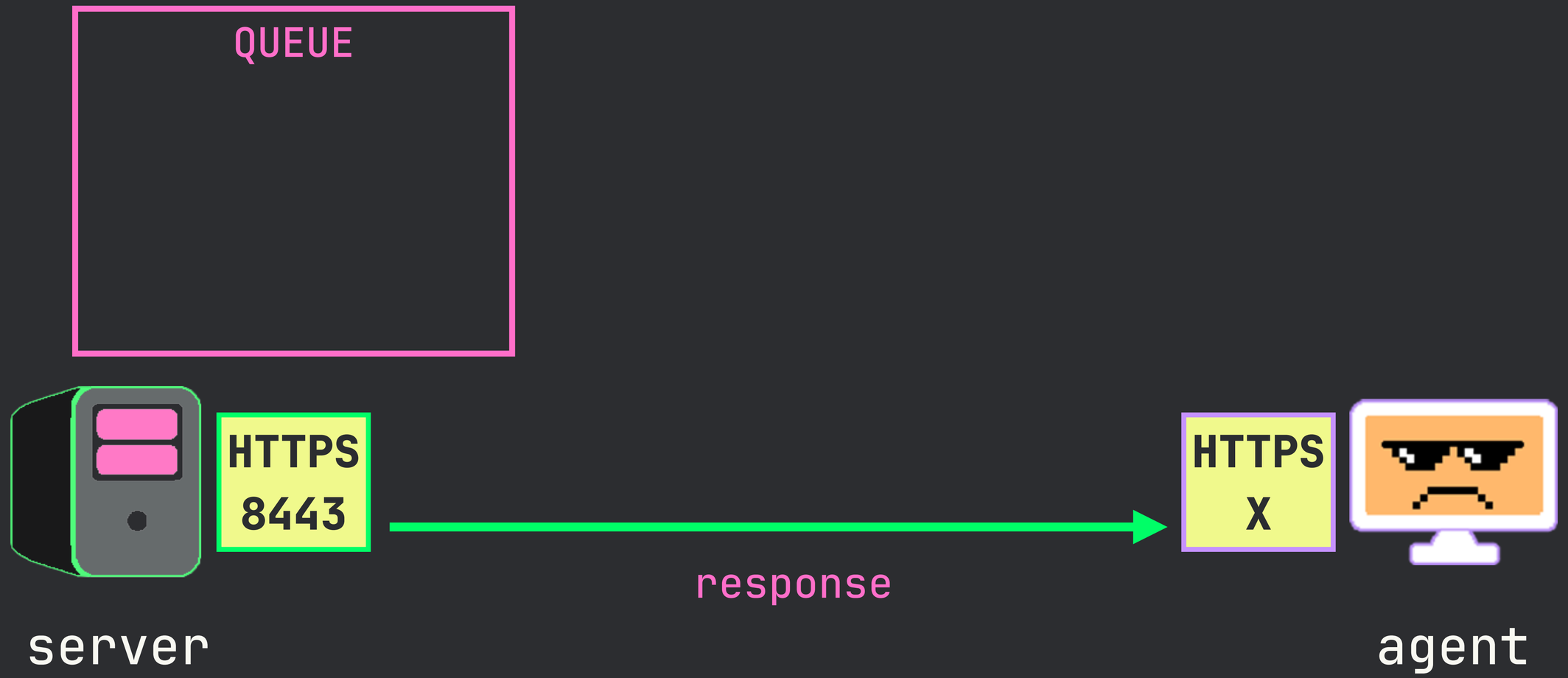
HTTPS
8443

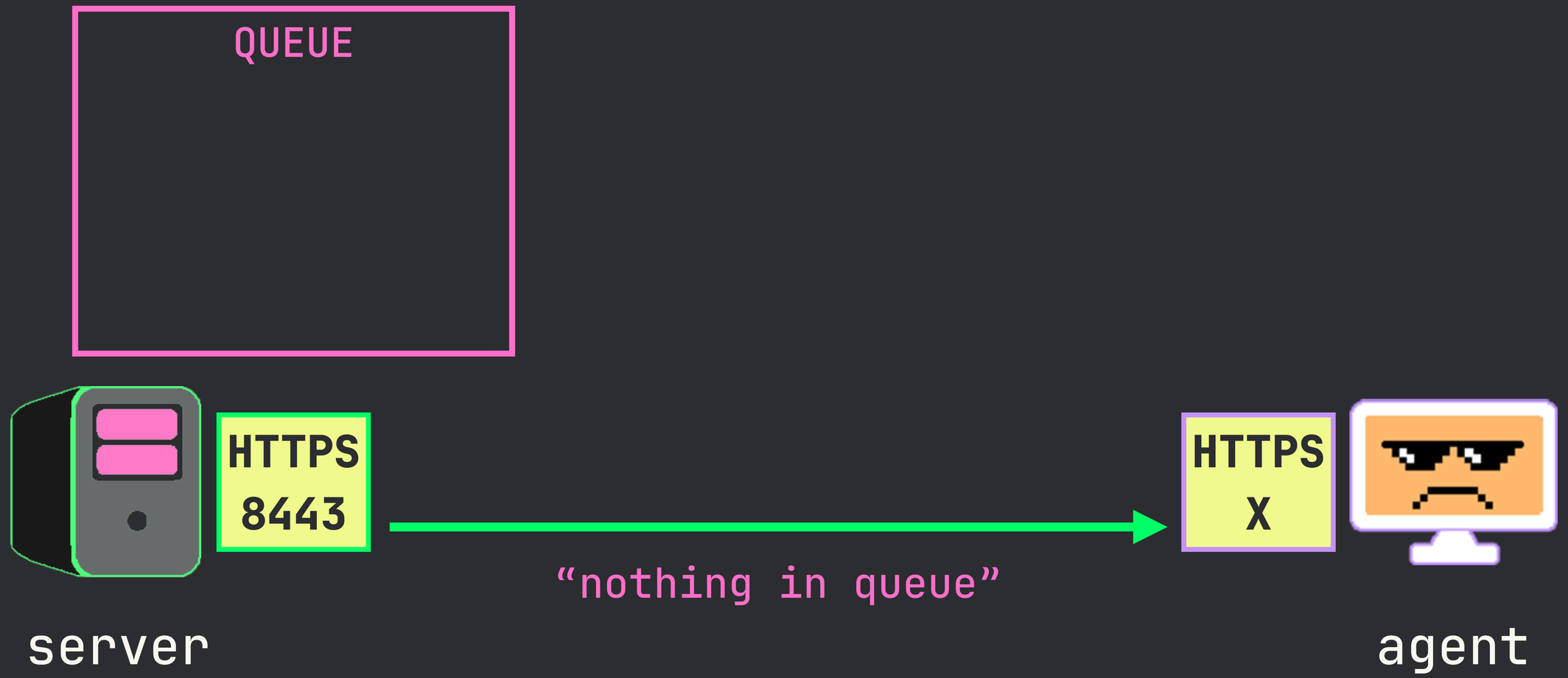
server

HTTPS
X

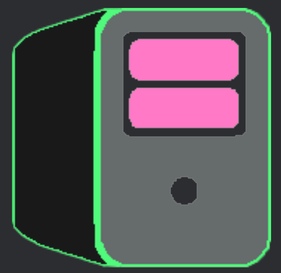
agent


QUEUE
request
HTTPS
8443
HTTPS
X
server
agent


QUEUE
HTTPS
8443
HTTPS
X
response
server
agent


QUEUE
HTTPS
8443
HTTPS
X
“nothing in queue”
server
agent

QUEUE



HTTPS
8443

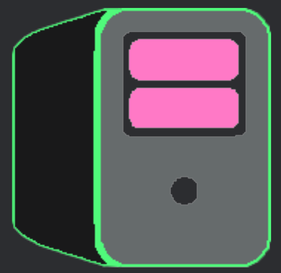
server

HTTPS
X

agent

QUEUE

```
command: “shellcode”

data: aGVsbG8gd29ybGQ…

function: LaunchCalc
```

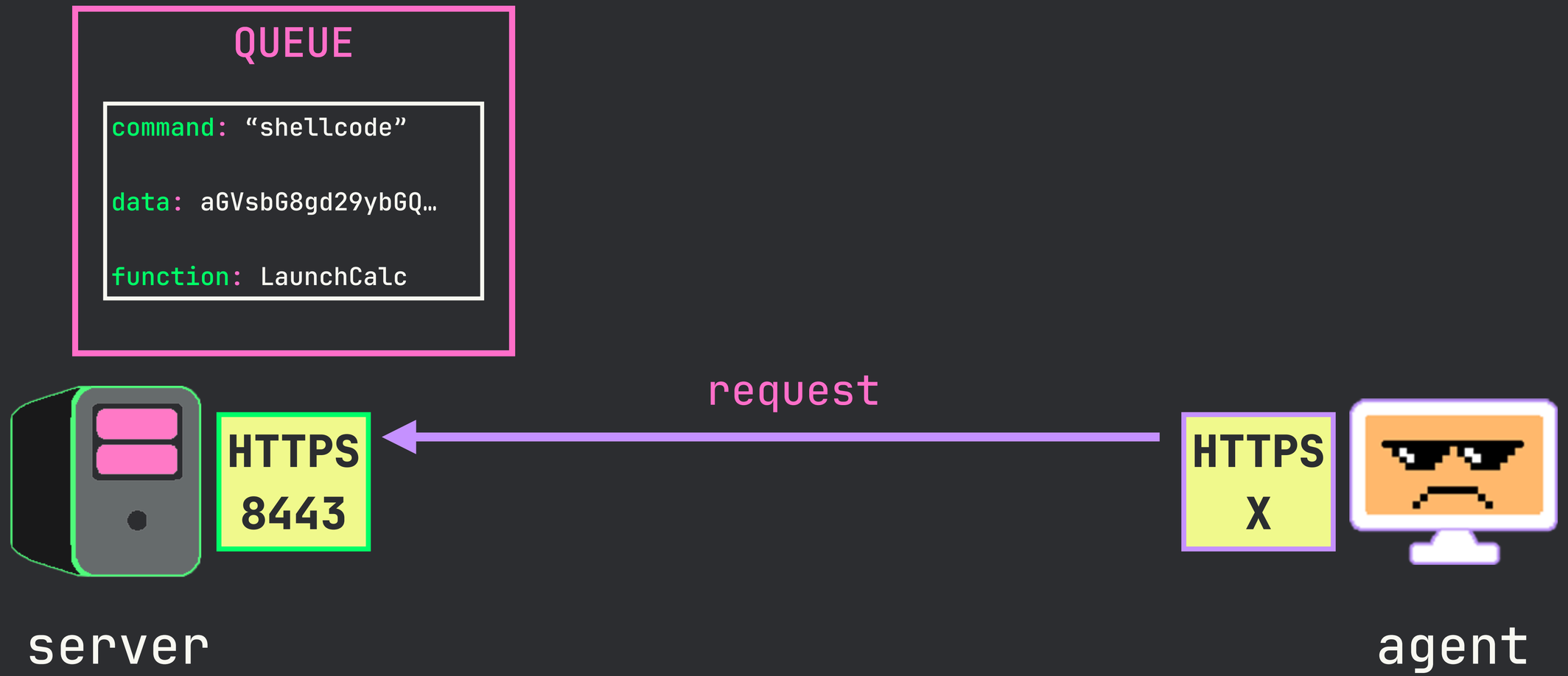


HTTPS
8443

server

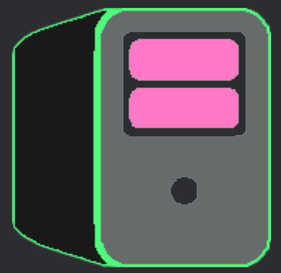
HTTPS
X

agent


QUEUE
command: “shellcode”
data: aGVsbG8gd29ybGQ…
function: LaunchCalc
request
HTTPS
8443
HTTPS
X
server
agent

QUEUE

```
command: “shellcode”

data: aGVsbG8gd29ybGQ…

function: LaunchCalc
```



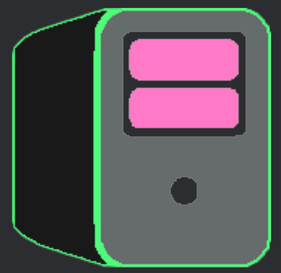
HTTPS
8443

server

HTTPS
X

agent

QUEUE



HTTPS
8443

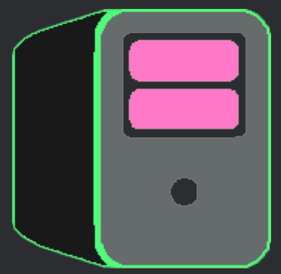
server

```
command: “shellcode”

data: aGVsbG8gd29ybGQ…

function: LaunchCalc
```

HTTPS
X

agent

QUEUE



HTTPS
8443

response

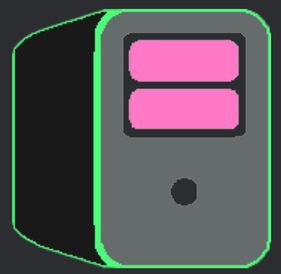
HTTPS
X

server

agent

```
command: “shellcode”

data: aGVsbG8gd29ybGQ…

function: LaunchCalc
```

QUEUE



HTTPS
8443

response

HTTPS
X

server

agent

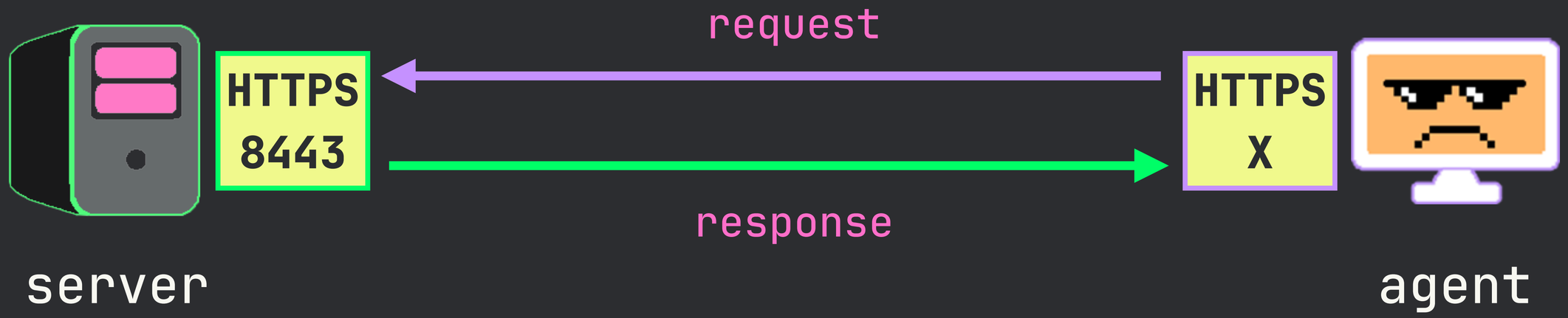
```
command: “shellcode”

data: aGVsbG8gd29ybGQ…

function: LaunchCalc
```

STEP 7: Create Agent Command Execution Framework

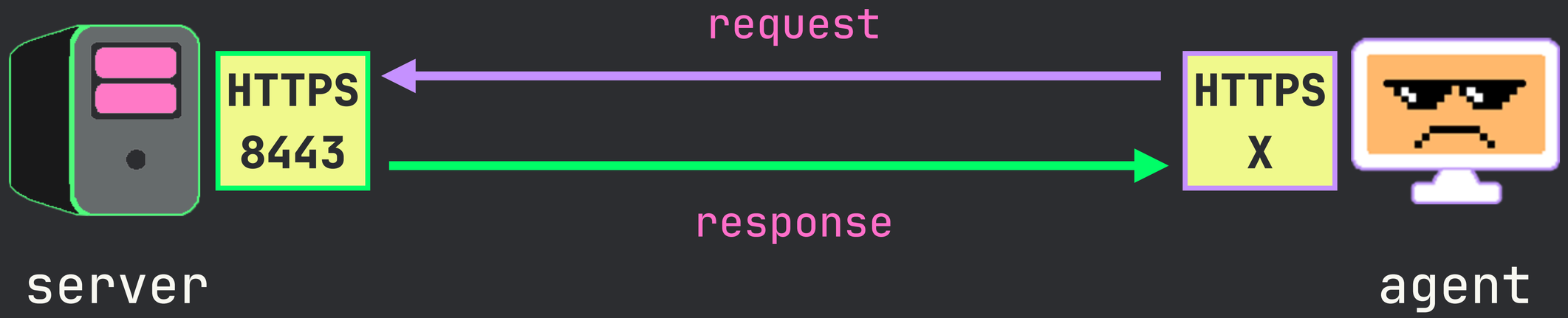
→ agent has now received cmd + args, but no idea what to do with it

→ the entire design of this flow, from the agent receiving cmd, to returning result, is what we are creating today

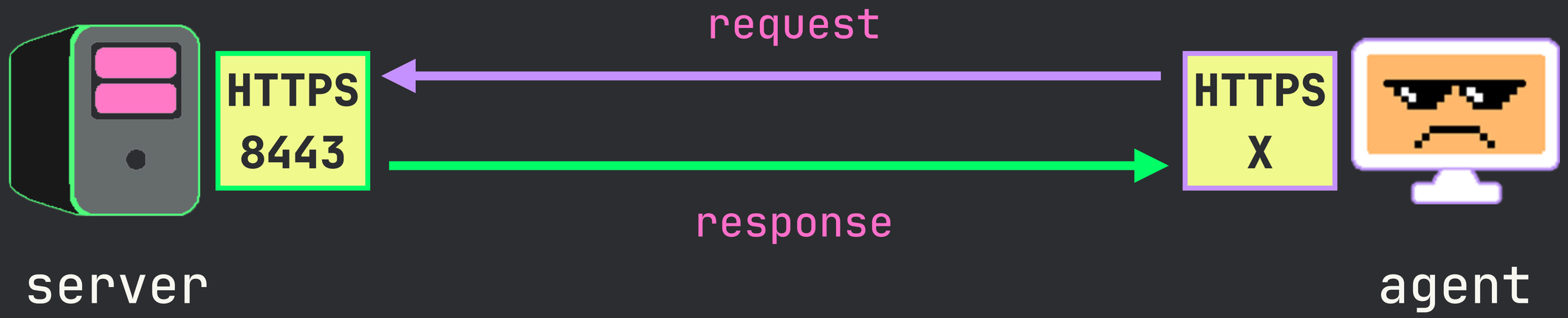
let’s explore this a bit more…



we know agent and server are communicating back and forth in a continuous loop with each other, but what is the logic responsible for this?



RunLoop() on the agent side



right now, VERY simple

```go
func RunLoop(agent *Agent, ctx context.Context, delay time.Duration, jitter int) error {

    for {
        // Check if context is cancelled
        select {
        case <-ctx.Done():
            log.Println("Run loop cancelled")
            return nil
        default:
        }

        response, err := agent.Send(ctx)
        if err != nil {
            log.Printf("Error sending request: %v", err)
            // Don't exit - just sleep and try again
            time.Sleep(delay)
            continue // Skip to next iteration
        }

        log.Printf("Response from server: %s", response)

        // Calculate sleep duration with jitter
        sleepDuration := CalculateSleepDuration(delay, jitter)

        log.Printf("Sleeping for %v", sleepDuration)

        // Sleep with cancellation support
        select {
        case <-time.After(sleepDuration):
            // Continue to next iteration
        case <-ctx.Done():

            log.Println("Run loop cancelled")
            return nil
        }
    }
}
```

infite for

send request

(hit server EP)

receive response

print response

sleep…

REPEAT…

```go
func RunLoop(agent *Agent, ctx context.Context, delay time.Duration, jitter int) error {

    for {
        // Check if context is cancelled
        select {
        case <-ctx.Done():
            log.Println("Run loop cancelled")
            return nil
        default:
        }

        response, err := agent.Send(ctx)
        if err != nil {
            log.Printf("Error sending request: %v", err)
            // Don't exit - just sleep and try again
            time.Sleep(delay)
            continue // Skip to next iteration
        }

        log.Printf("Response from server: %s", response)

        // Calculate sleep duration with jitter
        sleepDuration := CalculateSleepDuration(delay, jitter)

        log.Printf("Sleeping for %v", sleepDuration)

        // Sleep with cancellation support
        select {
        case <-time.After(sleepDuration):
            // Continue to next iteration
        case <-ctx.Done():

            log.Println("Run loop cancelled")
            return nil
        }
    }
}
```

print response

```go
func RunLoop(agent *Agent, ctx context.Context, delay time.Duration, jitter int) error {

    for {
        // Check if context is cancelled
        select {
        case <-ctx.Done():
            log.Println("Run loop cancelled")
            return nil
        default:
        }

        response, err := agent.Send(ctx)
        if err != nil {
            log.Printf("Error sending request: %v", err)
            // Don't exit - just sleep and try again
            time.Sleep(delay)
            continue // Skip to next iteration
        }

        log.Printf("Response from server: %s", response)

        // Calculate sleep duration with jitter
        sleepDuration := CalculateSleepDuration(delay, jitter)

        log.Printf("Sleeping for %v", sleepDuration)

        // Sleep with cancellation support
        select {
        case <-time.After(sleepDuration):
            // Continue to next iteration
        case <-ctx.Done():

            log.Println("Run loop cancelled")
            return nil
        }
    }
}
```

we don’t want to print,

we want to process

response!

```go
func RunLoop(agent *Agent, ctx context.Context, delay time.Duration, jitter int) error {

    for {
        // Check if context is cancelled
        select {
        case <-ctx.Done():
            log.Println("Run loop cancelled")
            return nil
        default:
        }

        response, err := agent.Send(ctx)
        if err != nil {
            log.Printf("Error sending request: %v", err)
            // Don't exit - just sleep and try again
            time.Sleep(delay)
            continue // Skip to next iteration
        }

        log.Printf("Response from server: %s", response)

        // Calculate sleep duration with jitter
        sleepDuration := CalculateSleepDuration(delay, jitter)

        log.Printf("Sleeping for %v", sleepDuration)

        // Sleep with cancellation support
        select {
        case <-time.After(sleepDuration):
            // Continue to next iteration
        case <-ctx.Done():

            log.Println("Run loop cancelled")
            return nil
        }
    }
}
```

if false (no command)

sleep and repeat

```go
func RunLoop(agent *Agent, ctx context.Context, delay time.Duration, jitter int) error {

    for {
        // Check if context is cancelled
        select {
        case <-ctx.Done():
            log.Println("Run loop cancelled")
            return nil
        default:
        }

        response, err := agent.Send(ctx)
        if err != nil {
            log.Printf("Error sending request: %v", err)
            // Don't exit - just sleep and try again
            time.Sleep(delay)
            continue // Skip to next iteration
        }

        log.Printf("Response from server: %s", response)

        // Calculate sleep duration with jitter
        sleepDuration := CalculateSleepDuration(delay, jitter)

        log.Printf("Sleeping for %v", sleepDuration)

        // Sleep with cancellation support
        select {
        case <-time.After(sleepDuration):
            // Continue to next iteration
        case <-ctx.Done():

            log.Println("Run loop cancelled")
            return nil
        }
    }
}
```

if true (command),

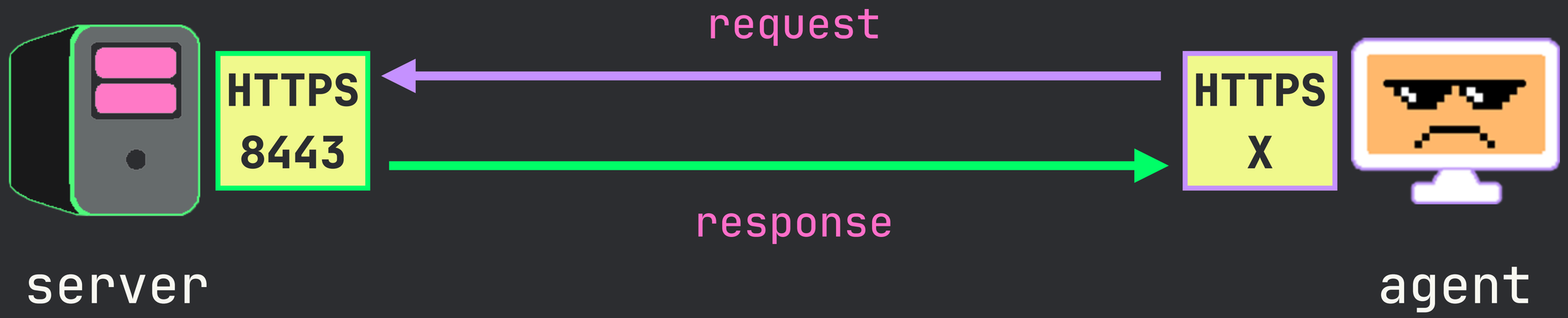
execute command,

return result,

sleep and repeat

```go
func RunLoop(agent *Agent, ctx context.Context, delay time.Duration, jitter int) error {

    for {
        // Check if context is cancelled
        select {
        case <-ctx.Done():
            log.Println("Run loop cancelled")
            return nil
        default:
        }

        response, err := agent.Send(ctx)
        if err != nil {
            log.Printf("Error sending request: %v", err)
            // Don't exit - just sleep and try again
            time.Sleep(delay)
            continue // Skip to next iteration
        }

        log.Printf("Response from server: %s", response)

        // Calculate sleep duration with jitter
        sleepDuration := CalculateSleepDuration(delay, jitter)

        log.Printf("Sleeping for %v", sleepDuration)

        // Sleep with cancellation support
        select {
        case <-time.After(sleepDuration):
            // Continue to next iteration
        case <-ctx.Done():

            log.Println("Run loop cancelled")
            return nil
        }
    }
}
```

but IMPORTANTLY, that logic will not go here, RunLoop() is not responsible for processing commands


request
HTTPS
8443
HTTPS
X
response
server
agent

PS

agent

RunLoop()

ExecuteTask()

Orchestrator()

Doer()

PS

There is a command, pass it to ExecuteTask()

agent

RunLoop()

ExecuteTask()

Orchestrator()

Doer()

```
command: “shellcode”

data: aGVsbG8gd29ybGQ…

function: LaunchCalc
```

PS

The command is “shellcode” - send it to that orchestrator

agent

RunLoop()

ExecuteTask()

Orchestrator()

Doer()

```
command: “shellcode”

data: aGVsbG8gd29ybGQ…

function: LaunchCalc
```

PS

agent

Validate Arguments,

Prepare Arguments for Doer,

Call the Doer.

RunLoop()

ExecuteTask()

Orchestrator()

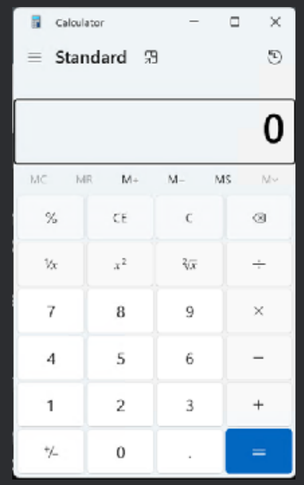
Doer()

```
command: “shellcode”

data: aGVsbG8gd29ybGQ…

function: LaunchCalc
```

PS

agent

Execute the actual command,
Capture result



RunLoop()

ExecuteTask()

Orchestrator()

Doer()

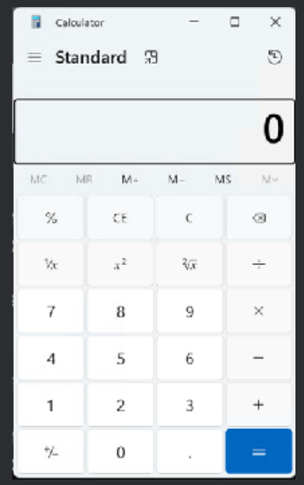
```
command: “shellcode”

data: aGVsbG8gd29ybGQ…

function: LaunchCalc
```

PS

agent

Execute the actual command,

Capture result



RunLoop()

ExecuteTask()

Orchestrator()

Doer()

result: “success”

PS

agent

Sends along to ExecuteTask()

RunLoop()

ExecuteTask()

Orchestrator()

Doer()

result: “success”

PS

Marshalls result and calls () to return it to server

agent

RunLoop()

ExecuteTask()

Orchestrator()

Doer()

result: “success”

PS

agent

Marshalls result and calls () to return it to server

RunLoop()

ExecuteTask()

Orchestrator()

Doer()

result: “success”

PS

Marshalls result and calls () to return it to server

agent

RunLoop()

ExecuteTask()

Orchestrator()

Doer()

PS

agent

Go to sleep, repeat process.

RunLoop()

ExecuteTask()

Orchestrator()

Doer()

RunLoop()

- receives initial command

ExecuteTask()

- decides which command to call
- returns final result to server

Orchestrator()

- prepares args for doer (+ interface)

Doer()

- actually executes the command

- Lesson 7

- Lesson 7

- Lesson 8

- Lesson 9 + 10

Lesson 11: Server Receives and Displays Results

→ Agent (ExecuteTask) sends results to server

→ Server display result

→ That’s it → we’ve closed the loop

That being the case, lets jump in and get going!