intro

- This will obviously be specific to each command, each will have its own function to do this
- but good thing is that we can use the same map, where the value of the key is the name of the command, to map each to its validator

As we said before, each, or at least most, commands will not just say for example "shellcode", it will also have specific arguments

- So for example now if we see it's shellcode, then we also need to see does it have the arguments required for shellcode? are these arguments valid?

validCommands

- RIght now it has name of command as key, and empty struct as value
- But we can now add a function type to the empty struct
- In Go, functions can be types (explain this mini-lesson)
- So let's first define this Validator type, which you can thing as "the function signature for all commands argument validation"
- Add to same command_api.go file

```go
// CommandValidator validates command-specific arguments
type CommandValidator func(json.RawMessage) error
```

Great now we can add this to `validCommands`

```go
// Registry of valid commands with their validators and processors
var validCommands = map[string]struct {
    Validator CommandValidator
}{
    "shellcode": {
        Validator: validateShellcodeCommand,
    },
}
```

So we have this function (which we will create next) called validateShellcodeCommand as the value mapped to `Validator`, we have not created it yet so obvs it errors out.

Great so we can see now here we also map the specific function `validateShellcodeCommand` which we now of course need to create

validateShellcodeCommand

let's create a new file for all logic specific to shellcode command, still in control package create `shellcode.go`

```go
// validateShellcodeCommand validates "shellcode" command arguments from
client
func validateShellcodeCommand(rawArgs json.RawMessage) error {
	if len(rawArgs) == 0 {
		return fmt.Errorf("load command requires arguments")
	}

	var args models.ShellcodeArgsClient

	if err := json.Unmarshal(rawArgs, &args); err != nil {
		return fmt.Errorf("invalid argument format: %w", err)
	}

	if args.FilePath == "" {
		return fmt.Errorf("file_path is required")
	}

	if args.ExportName == "" {
		return fmt.Errorf("export_name is required")
	}

	// Check if file exists
	if _, err := os.Stat(args.FilePath); os.IsNotExist(err) {
		return fmt.Errorf("file does not exist: %s", args.FilePath)
	}

	log.Printf("Validation passed: file_path=%s, export_name=%s",
args.FilePath, args.ExportName)

    return nil
}
```

shellcode will receive 2 arguments

- FilePath
- ExportName
- FilePath is the path (on the server) to the DLL containing the shellcode we want to run
- ExportName is the name of the exported function we want to call, we are doing this since this allows us multiple exported functions per DLL, greater degree of flexibility

FilePath

- Make sure argument is not empty
- Make sure file exists

ExportName

- Make sure argument is not empty
- In processing - if the exported name does not exist, it will default to DllMain

handleCommand

Now finally back in our command handler, we need to use it

```
func commandHandler(w http.ResponseWriter, r *http.Request) {

    // Instantiate custom type to receive command from client
    var cmdClient models.CommandClient

    // The first thing we need to do is unmarshall the request body into
the custom type
    if err := json.NewDecoder(r.Body).Decode(&cmdClient); err != nil {
        log.Printf("ERROR: Failed to decode JSON: %v", err)
        w.WriteHeader(http.StatusBadRequest)
        json.NewEncoder(w).Encode("error decoding JSON")
        return
    }

    // Visually confirm we get the command we expected
    var commandReceived = fmt.Sprintf("Received command: %s",
cmdClient.Command)
    log.Printf(commandReceived)

    // Check if command exists
    cmdConfig, exists := validCommands[cmdClient.Command] // Replace _
with cmdConfig
    if !exists {
        var commandInvalid = fmt.Sprintf("ERROR: Unknown command: %s",
cmdClient.Command)
        log.Printf(commandInvalid)
        w.WriteHeader(http.StatusBadRequest)
        json.NewEncoder(w).Encode(commandInvalid)
        return
    }

    // Validate arguments
    if err := cmdConfig.Validator(cmdClient.Arguments); err != nil {
        var commandInvalid = fmt.Sprintf("ERROR: Validation failed for
'%s': %v", cmdClient.Command, err)
        log.Printf(commandInvalid)
        w.WriteHeader(http.StatusBadRequest)
        json.NewEncoder(w).Encode(commandInvalid)
        return
    }

    // Confirm on the client side command was received
    w.WriteHeader(http.StatusOK)
    json.NewEncoder(w).Encode(commandReceived)

}
```

WE can see on line 19 we replaced `_` with `cmdConfig`

New code on 28 - 35
Explain it

test

- Great so let's again test by calling a file that does not exists, we expect to fail, and then using correct arguments
- Run the server

```
❯ curl -X POST http://localhost:8080/command \
  -d '{
    "command": "shellcode",
    "data": {
      "file_path": "./payloads/derp.dll",
      "export_name": "LaunchCalc"
    }
  }'
"ERROR: Validation failed for 'shellcode': file does not exist:
./payloads/derp.dll"

❯ curl -X POST http://localhost:8080/command \
  -d '{
    "command": "shellcode",
    "data": {
      "file_path": "./payloads/calc.dll",
      "export_name": "LaunchCalc"
    }
  }'
"Received command: shellcode"
```

```
❯ go run ./cmd/server
2025/11/06 14:07:29 Starting Control API on :8080
2025/11/06 14:07:29 Starting  server on 0.0.0.0:8443
2025/11/06 14:07:45 Received command: shellcode
2025/11/06 14:07:45 ERROR: Validation failed for 'shellcode': file does
not exist: ./payloads/derp.dll
2025/11/06 14:07:55 Received command: shellcode
2025/11/06 14:07:55 Validation passed: file_path=./payloads/calc.dll,
export_name=LaunchCalc
```