Leaving No Input Unsanitized

Gabriel Aubut-Lussier

Problem description

What

- Processing untrusted input is what most applications do
- Input validation failures may lead to security vulnerabilities
- It's hard to get ~~right~~ perfect

Why

- Had plans to implement a REST API in C++
- Knew beforehand there would be multiple inputs
- Knew beforehand I would screw up somewhere sometime

How

1. Deny direct access to the input
2. Enumerate inputs in a declarative style
3. Provide a generic request handler

Usage example

```
using Shipping  = InputDesc<CustomerInfo,
                            HeaderParam<"client">,
                            JSON>;

using Billing = InputDesc<CustomerInfo,
                          BodyParam,
                          QueryString>;
```

```
using Shipping  = InputDesc<CustomerInfo,
                            HeaderParam<"client">,
                            JSON>;

using Billing = InputDesc<CustomerInfo,
                          BodyParam,
                          QueryString>;
```

```
using Shipping  = InputDesc<CustomerInfo,
                            HeaderParam<"client">,
                            JSON>;

using Billing = InputDesc<CustomerInfo,
                          BodyParam,
                          QueryString>;
```

Hopefully, C++20's NTTP

```
using Shipping  = InputDesc<CustomerInfo,
                            HeaderParam<"client">,
                            JSON>;

using Billing = InputDesc<CustomerInfo,
                          BodyParam,
                          QueryString>;
```

Hopefully, C++20's NTTP

```
using Shipping  = InputDesc<CustomerInfo,
                            HeaderParam<"client">,
                            JSON>;

using Billing = InputDesc<CustomerInfo,
                          BodyParam,
                          QueryString>;
```

```
using Shipping  = InputDesc<CustomerInfo,
                            HeaderParam<"client">,
                            JSON>;

using Billing = InputDesc<CustomerInfo,
                          BodyParam,
                          QueryString>;
```

```
using Shipping  = InputDesc<CustomerInfo,
                            HeaderParam<"client">,
                            JSON>;

using Billing = InputDesc<CustomerInfo,
                          BodyParam,
                          QueryString>;
```

```
using Shipping  = InputDesc<CustomerInfo,
                            HeaderParam<"client">,
                            JSON>;

using Billing = InputDesc<CustomerInfo,
                          BodyParam,
                          QueryString>;
```

Simplified to serve as slide code

```
RequestHandler<Shipping, Billing> handler {
  [](auto shippingInfo, auto billingInfo) {
    cout << shippingInfo << '\n'
         << billingInfo  << '\n';
  }
};
```

Simplified to serve as slide code

```
RequestHandler<Shipping, Billing> handler {
  [](auto shippingInfo, auto billingInfo) {
    cout << shippingInfo << '\n'
         << billingInfo  << '\n';
  }
};
```

Simplified to serve as slide code

```
RequestHandler<Shipping, Billing> handler {
  [](auto shippingInfo, auto billingInfo) {
    cout << shippingInfo << '\n'
         << billingInfo  << '\n';
  }
};
```

Simplified to serve as slide code

```
RequestHandler<Shipping, Billing> handler {
  [](auto shippingInfo, auto billingInfo) {
    cout << shippingInfo << '\n'
         << billingInfo  << '\n';
  }
};
```

Generic
request handler

Generic request handler

```
template <typename InputDescription,
          typename Callable>
Response HandleRequest(Request req,
                       Callable process)
{
  return GetRawInput<InputDescription>(req)
    .and_then(ValidateInputs)
    .and_then(process)
    .or_else(RespondBadRequest);
}
```

Simplified to serve as slide code

Generic request handler

```
template <typename InputDescription,
          typename Callable>
Response HandleRequest(Request req,
                       Callable process)
{
  return GetRawInput<InputDescription>(req)
    .and_then(ValidateInputs)
    .and_then(process)
    .or_else(RespondBadRequest);
}
```

Simplified to serve as slide code

Generic request handler

```
template <typename InputDescription,
          typename Callable>
Response HandleRequest(Request req,
                       Callable process)
{
  return GetRawInput<InputDescription>(req)
    .and_then(ValidateInputs)
    .and_then(process)
    .or_else(RespondBadRequest);
}
```

Simplified to serve as slide code

Generic request handler

```
template <typename InputDescription,
          typename Callable>
Response HandleRequest(Request req,
                       Callable process)
{
  return GetRawInput<InputDescription>(req)
    .and_then(ValidateInputs)
    .and_then(process)
    .or_else(RespondBadRequest);
}
```

Simplified to serve as slide code

Generic request handler

```
template <typename InputDescription,
          typename Callable>
Response HandleRequest(Request req,
                       Callable process)
{
  return GetRawInput<InputDescription>(req)
    .and_then(ValidateInputs)
    .and_then(process)
    .or_else(RespondBadRequest);
}
```

Simplified to serve as slide code

Generic validation

Generic validation

- Strong semantic type validation handled by this template

```
template <typename T>
std::optional<T> Validate(std::string_view rawInput);
```

Generic validation

- Strong semantic type validation handled by this template

```
template <typename T>
std::optional<T> Validate(std::string_view rawInput);

// Requires a specialization for user-defined types
template <>
std::optional<CustomerInfo>
Validate<CustomerInfo>(std::string_view rawInput) {
    // Validation code here
    if (success) return CustomerInfo{/* […] */};
    return std::nullopt;
}
```

Let's generalize

- **argc**, **argv** and **envp**(arguments to **main**)
- **read** (syscall)
- Etc.

Conclusion

- Meet me in the hallways to discuss this idea
- Try it : https://github.com/Dalzhim/SecureRequestHandler