Be Less Primitive To Be More Secure

OWASP NZ DAY 2025

5 SEPTEMBER 2025

BY JAMES COOPER

Thank You to Our Sponsors and Hosts!

OWASP NEW ZEALAND owasp.org.nz

Waipapa Taumata Rau University of Auckland

AppSec NZ appsec.org.nz

BASTION SECURITY GROUP

SECURE CODE WARRIOR

exabeam

safe advisory.

CyberCX

plexure

SECDIM

Without them, this conference couldn't happen.

And you are?

- Security-interested software developer
- Application Developer at 2degrees
- Ph.D. in Computer Science (+ BCom) from University of Auckland
- Interested in a wide variety of areas of comput[ing|ers]
- Fan of using types intelligently in software development

Injection Attacks

Less fun than a vaccination

- "Injection flaws occur when an application sends untrusted data to an interpreter."
  - From https://cheatsheetseries.owasp.org/cheatsheets/Injection_Prevention_Cheat_Sheet.html
  - Number 3 on the OWASP Top 10 2021
  - Found in 19% of all applications tested for the Top 10

- The key aspect is that in every instance, some aspect of the behaviour is determined at runtime, through directly incorporating or processing user input or other non-controlled data

- The classic examples are things like SQL Injection, Cross-Site Scripting (XSS), OS command injection, basically any time somebody uses `eval` or `system` or similar
  - I'd also say Open Redirects kinda count

But why?

- Because user input is mixed in with commands, which are then evaluated
  - The classic mistake of confusing data for instructions
  - Too much decided at runtime
  - “Eval is evil”
- Many people consider injection attacks (or, SQLi, at least) a solved problem!
- To avoid injection attacks don’t interpret the data, just use them
- Being less primitive and using types intelligently, can help by making it easier to do the right thing and harder to do the wrong thing

Classic SQL Injection

```
"SELECT * FROM Users
 WHERE Username = '"
 + InputValue1 +
 "' AND Password = '"
 + InputValue2 + "';"
```

So where’s the problem?

```
“SELECT * FROM Users WHERE Username = ‘“ + InputValue1 + “’ AND
Password = ‘” + InputValue2 + “’;”
```

- What if my input is username: `‘ OR ‘1’=‘1’ --` and password: `hi mum!`?
  - Kinda like the Hello World of SQLi

- In effect, from the database’s perspective the query becomes

```
 SELECT * FROM Users WHERE Username = ‘’ OR ‘1’=‘1’
```

  - Gets all users in the DB—probably not what you want as the developer
  - *Many* other possibilities (potentially, just about whatever the DB supports in a `SELECT` statement)

Similar for XSS

- Cross-site scripting (XSS) arises because we take user input as page specification
  - E.g. repeat user input back on a dynamically constructed page
- Plug the input into our page’s source, then tell the server or browser to render the whole thing
- We ignore the semantic meaning: user input
- Jumble it up with something semantically different, our page’s source code
- These two things are not the same, and shouldn’t be treated the same

Strings aren’t the only primitive problem

- The Mars Climate Orbiter was a fairly significant availability (and integrity?) failure
- MCO crashed during orbital insertion, thruster calculations were way off
- Turned out sub-component contractor used pound-force seconds, but NASA system expected Newton-seconds (and NASA’s contract specified use of SI units…)
- The problem was that both quantities were represented the same way
  - “Don’t give me naked numbers”
- These are different types of quantity, don’t represent them identically

Handling Bits, Typically

What is this?

```
"SELECT * FROM Users
 WHERE Username = '"
 + InputValue1 +
 "' AND Password = '"
 + InputValue2 + "';"
```

The Treachery Of Strings

- Ceci n’est pas une ~~pipe~~ SQL query (“This is not a SQL query”)
- I’d argue that was a string/bunch of strings glued together, and *not* a SQL query
  - It does convey the concept of a SQL query
- SQL queries exist inside databases
  - We use strings to communicate our intent to the DB
  - It parses the string into its own internal representation
  - The actual query is run using that representation
- Dynamically parsed strings are one way to create queries, so are stored procedures *et al.*

Modern Primitive

- Everything in a typical modern electronic computer is represented with bits—just two values and their positions in sequences (and how we interpret them)
- Working with the bits directly is a pain in the neck, though
- We abstract over that with a programming language’s primitive types
  - E.g. int, float, char, string, etc.
- For some reason, people often stop there and represent everything in those types
- Commonly known as “primitive obsession”

Be Less Primitive

- Beneath the abstraction, we might indeed represent different concepts with the same sequences of bits
- Doesn't mean they don't have different semantic concepts behind them, though
- Why should we let the bits be the boss?
- **Every** useful recent programming language has facilities for making your own types
- Create the abstractions you want, to get the behaviours you intend!

Everyone’s favourite computer scientist

The purpose of abstracting is not to be vague, but to create a new semantic level in which one can be absolutely precise.

Edsger W. Dijkstra

(taken from https://en.wikiquote.org/wiki/Edsger_W._Dijkstra#The_Humble_Programmer_(1972) on 10 August 2025)

Check Your Types at the ~~Door~~ Edges

Parsing > validation

- Input to a program pretty much inevitably has to come in the form of primitive values
- See e.g. the limited types available in JSON, or the typical command line
- Doesn’t mean you leave them like that, though
- Turn valid input into relevant types, and reject invalid input at the gates
  - Use domain-sensible types on the inside—usually safer *and* easier to work with
- Parse (rather than validate). See https://lexi-lambda.github.io/blog/2019/11/05/parse-don-t-validate/

Types with benefits

- Parsing to types rather than merely validating has advantages
  - Input is validated as part of construction—don’t even represent invalid input
  - Compilers prevent you using the wrong thing in the wrong place
  - Clearer intent in code

- Eventually, will probably have to ‘lower’ back to primitive types for output or interfacing

- Define how to lower in one place, and do it well
  - Developers automatically use best practices
  - No more forgetting to do the safe thing, ‘cause it’s the only thing
  - Even if it’s just escaped strings, you know it’s done consistently

What about those foot-pound seconds?

- We can use (basically) the same solution
- Physical values are a combination of quantity and unit
- So represent them in that way!
- F#’s ‘Units of Measure’ do exactly this
  - The type information is all erased before run time—no performance impact
- Using them would almost certainly have caught the Mars Climate Orbiter issue at compile time

Getting to the Point…

Be More Secure

- Create meaningful abstractions where the easy path is the correct path
- Or, simply wrap a primitive type, but restrict its allowed behaviour(s)
- You decide what goes, and what doesn't
- Buggy code? Compiler says no. Unintended runtime actions? Does not compute.
- Can also be an architectural/coding boon
  - Devs are more likely to use something well if that's the easiest way to use it
  - Good names clarify intentions

To be primitive or not to be

- Going back to the SQL query example again

- As a string, pretty easy to do SQL injection with the input: `‘ OR ‘1’=‘1’ --`
  Becomes something like

```
SELECT * FROM Users WHERE Username = ‘’ OR ‘1’=‘1’
```

- Instead, turn it into a type, separate query from its inputs
  - Give inputs necessary treatment before sending to DB
  - Pretty much what most good ORMs do anyway

- Many possible ways to do this, the key is to use your brain and not a primitive string
  - E.g. a stored procedure/prepared statement in your DB or via a good ORM

Which is least secure? (C#-ish edition)

```
db.ExecuteQuery("SELECT * FROM Users WHERE Username = '" +
InputValue1 + "' AND Password = " + InputValue2 + "';");
```

```
db.Query("GetUser", new { Username = InputValue1, Password =
InputValue2 }, commandType: CommandType.StoredProcedure);
```

```
db.Users.SingleAsync(user => user.Username == InputValue1 &&
user.Password == InputValue2);
```

Conclusion

Things to remember

- Injection attacks happen because input is mixed in with instructions, then interpreted
  - Semantic concepts are ignored, everything is treated the same
  - Often just a bunch of strings stuck together
  - Using primitive types over more meaningful custom ones is “primitive obsession” (and silly)

- We’re not always helped by our tools and systems
  - Poor interfaces can lead to us to insecure approaches
  - We often have to turn things into a string or similar to communicate with something else

- You should still treat different semantic concepts differently for as long as possible

More things to remember

- Smart use of types isn't the only way to prevent injection attacks, and such prevention isn't the only potential benefit of the smart use of types

- Making the instructions and the input separate types can help avoid injections
  - E.g. database stored procedures distinguish parameters from SQL, XSS defences do the same for user input and HTML & friends
  - Good abstractions make the right thing easier, the wrong thing harder
  - You get more control over permitted behaviour
  - At the very least, much harder to mix the two up inadvertently

- Your thinking it through earlier, helps stop devs make mistakes later, prevents injection attacks even later

Fin

Any Questions?