BUILDING SECURE SOFTWARE
FROM THE GROUND UP

ABSTRACT

This whitepaper introduces the Noumena Protocol Language (NPL), a new programming language purpose-built for secure online services. NPL embeds fine-grained authorization, identity, and auditability directly into the language, enforced by both its compiler and runtime.

The paper begins by diagnosing the ongoing software security crisis, arguing that traditional general-purpose languages are by design poorly suited for building secure systems. It explores how the tension between development velocity and security has led to a reliance on brittle, bolt-on solutions like libraries, frameworks, and external firewalls.

Building on lessons from both language design and network security, we propose a radical alternative: shifting security concerns left by making them part of the language itself. Through a detailed presentation of NPL's core concepts, including protocols, parties, permissions, obligations, and secure runtime execution, we demonstrate how NPL makes security declarative, verifiable, and enforceable.

The NPL approach is then compared to existing technologies such as Solidity and DAML, and we show how NPL addresses selected OWASP Top 10 vulnerabilities natively and by design. Finally, we make the case that secure-by-design languages like NPL are not just desirable but necessary to reverse the escalating cost and complexity of securing modern software.

By Yigal Duppen, Noumena Digital

This whitepaper is part of a series exploring Noumena's vision for the future of software development and the technology to support it. The full series is available on http://..... This paper focusses on the Noumena language, a foundational element of the Noumena technology stack.

THE SOFTWARE SECURITY CRISIS

The cost of data breaches

Software security breaches have become a daily occurrence. As such, they represent a severe and escalating threat.

These breaches routinely lead to significant financial losses for organizations; are increasingly subject to stringent legal and regulatory actions; and can severely damage an entity's reputation and public trust.

Furthermore, the personal and organizational data involved in these breaches has real-world consequences for individuals and communities, demanding robust protection.

While many breaches can be traced back to social engineering and human error, many others are the result of software not doing what it was supposed to do: lacking or incomplete authorization; lacking or incomplete audit trails; or just a plain failure to think about security in the first place.

VELOCITY VERSUS SECURITY

When considering the cost of software security, it's important to take development velocity into account. This includes both the time it takes to bring a product to market and the cost of maintaining that software.

A key driver in enhancing development velocity is "shift left," the practice of moving as many tasks as possible to the start of the software development lifecycle. However, existing security processes often run counter to this "shift left" approach, slowing down development rather than speeding it up.

OUR FOUNDATIONS ARE FUNDAMENTALLY FLAWED

It's easy to blame individuals for shoddy programming, or costly processes for high development costs, but none of this addresses the fact that traditional general purpose languages are by design poorly suited for developing secure online services. This is because these languages inherently grant developers and by extension, the running application broad access and control, making it challenging to enforce fine-grained authorization policies required for modern, multi-user online systems.

As an example, have a look at a typical Spring Boot controller for retrieving the details of a specific user.

```
@RestController
@RequestMapping(„/api/users“)
public class UserController {
       private final UserService userService;

       public UserController(UserService userService) {
           this.userService = userService;
       }

       @GetMapping(„/{username}“)
       @PreAuthorize(„hasRole(‚USER‘) or hasRole(‚ADMIN‘)“)
       public ResponseEntity<?> getUser(@PathVariable String username) {
           return userService.getUser(username)
                         .map(ResponseEntity::ok)
                   .orElse(ResponseEntity.notFound().build());
       }
}
```

[1] https://secureframe.com/blog/data-breach-statistics

[2] https://www.ibm.com/reports/data-breach

[3] https://www.verizon.com/business/resources/T49b/reports/2025-dbir-data-breach-investigations-report.pdf

[4] https://newrelic.com/pt/blog/best-practices/shift-left-strategy-the-key-to-faster-releases-and-fewer-defects

At first sight, this looks like a properly secured endpoint. The getUser method has an explicit @PreAuthorize annotation, ensuring that this endpoint is not accessible to just anyone. On closer inspection, however, it turns out that the userService itself is capable of retrieving any user in the system – which in this case means that anyone with the role USER is allowed to retrieve the details of every single user.
The default "full power" conflicts with the "least privilege" principle critical for secure online services, as we will see below.

LIBRARIES AND FRAMEWORKS ARE NOT THE ANSWER

There are many frameworks, libraries, and security solutions that attempt to solve this problem: passport.js, Spring Security, Flask Principal, Casbin, RoR, to name just a few. Many of them are of high quality and valuable, but in the end they act as bolted-on layers rather than integral components of a system's security.
By their very nature, it is impossible to guarantee that the library is used correctly, and is used at every place where it should be used. In practice this means that verification of these security measures happens late in the process, which again runs counter to the "shift left" approach, slowing down development rather than speeding it up.

Even more fundamentally, libraries and frameworks cannot enforce a deny-by-default policy as they are constrained by the underlying language's design.
Complementary approaches externalize the security to Web Application Firewalls, API gateways, database-level security and other tools. As we elaborate on below, while many of these tools assist in achieving in-depth security, they effectively decouple the security from the business logic that needed to be secured in the first place.

RETHINKING THE FOUNDATIONS:
TOWARDS A LANGUAGE-NATIVE SECURITY MODEL

We have discussed why general-purpose languages and external tools have proven inadequate for enforcing robust security in online services. Security remains fragmented, often added late, and difficult to verify. This raises a central question: how can we ensure stronger, more consistent security without slowing down development?

Language design

In the past computer science has looked at language design to prevent software defects from occurring in the first place. Notable examples include:

Structured programming
Arbitrary jumps were replaced by the notion of predictable control flows, which in turn opened the doors for entire new classes of compiler optimizations and automated control flow analysis

Static typing
Enforcing types at compile time eliminated a huge class of runtime errors, and enabled better IDE support such as autocomplete and refactoring. More recent additions such as gradual typing and type inference have revived the popularity of explicit typing.

Memory safety
To improve memory safety, languages such as Java, Python and Go removed explicit memory support in favor of garbage collection, while Rust introduced the borrow-checker to make memory safety enforceable by the compiler.

There are many more less generic examples, such as null safety, exhaustiveness checking and hygienic macros. But in every case, we've upgraded our languages to improve our development velocity by empowering the compiler to catch design and safety errors. However, the lack of proper authorization has so far not been addressed.

NETWORK SECURITY

Just as we've improved language design, network security has also evolved significantly. In the early days of computer networking, networks were small and closed to the outside world. Security was not a concern, and connected users and devices were trusted by default.

With ever increasing connectivity of application across network boundaries, network security has moved towards the concept of zero-trust architectures where everything is considered a risk:

Never Trust, Always Verify
No user or device is automatically trusted, regardless of location or network. At the same time, traditional languages by default trust every single function call

Least Privilege Access
Users are granted only the minimum access necessary to perform their job functions. At the same time, traditional languages by default allow full access to all the data they manage

Continuous Monitoring and Validation
All access attempts are continuously monitored and validated. Again, traditional languages by default do nothing of the sort, instead relying on additional libraries and frameworks to provide this functionality

We have argued that traditional general-purpose languages are insufficient to efficiently and effectively deal with such requirements.

INCORPORATING ZERO-TRUST INTO PROGRAMMING

To address the inherent security weaknesses of current programming languages when used for online services, we propose a fundamental shift: embedding zero-trust authorization mechanisms directly into the language and its compiler. Based on the analysis above, we identify five core principles that any such language must satisfy to provide robust, built-in security guarantees:

1. Core language features The language should retain essential language elements such as basic data structures and control flow structures wherever possible.

2. Caller identification
The language must have a special data type designed specifically to represent the current caller and the set of potentially authorized callers. Instances of this data type must be relatable to identities in networked applications.

3. Services versus data structures
The language must support grouping related data into structures that are accessible as services. These services must be explicitly relatable to its authorized callers.

4. Authorized entry points versus functions
The language must separate functions intended for internal code reuse from those designed as authorized entry points for external requests. Authorized callers on those entry points must be relatable to those defined on the containing service.

5. Compiler-enforced security
The compiler must enforce the presence of coherent and consistent authorization rules, ensuring that only authorized callers can execute specific functions or access designated data, effectively preventing unauthorized actions at compile time.

Together, these five principles define the foundation for a new class of zero-trust programming languages designed from the ground up to meet today's security requirements in software development.

[5] https://cloud.google.com/beyondcorp

INTRODUCING NPL

At Noumena, we set out to build a language that delivers on these 5 principles of zero-trust programming. Our ambition thereby was to make the language as efficient and developer friendly as possible, and to [test the hell out of it] in production. The result is the Noumena Protocol Language (NPL), a radically new language that embeds proper authorization into its syntax. It comes together with a runtime to execute NPL programs, enforce the defined authorization rules, and expose the programs as services.
This section focuses on the language elements that turn it into a proper security-oriented language. It is accompanied by code samples depicting an extremely simplified loan.

PARTIES AND PROTOCOLS

Protocols are the core of NPL, defining interactions between so-called "parties" as services; they encapsulate both data and logic, and clearly delineate the capabilities of those individual parties.
Conversely, **parties** are the entities involved in a protocol. Parties are an abstraction over actual users, systems and/or groups and dictate access controls on all data.

```
protocol [customer, bank] Loan(var forAmount:
Number, var interest: Number) {
   ...
}
```

When defining a protocol, it is mandatory to define the associated parties. Omitting them results in an actual compile-time error.
Note how the language itself does not attach any implicit meaning to parties. This is done by the NPL Runtime which uses a simplified form of attribute-based access control to unify token-based identities and parties.

Parties are assigned to actual entities when a protocol is instantiated – this implies that different instances of the same protocol can have different authorizations – a big difference from more traditional role based approaches. Upon instantiation, it is the responsibility of the Runtime to properly persist the protocol.

PERMISSIONS AND OBLIGATIONS

While NPL supports the notion of classical private, unauthorized functions, all publicly available operations on a protocol are modelled as permissions and obligations, collectively known as actions. Permissions define which actions a party is allowed to perform, while obligations define which actions a party has to perform. As such, each permission and obligation has to define the parties that are allowed to invoke that action.
Obligations have an additional otherwise clause, indicating what happens when an obligation is not met.
Permissions and obligations are always defined within the scope of a protocol; and the parties associated with these actions must be defined on the protocol itself. The Runtime again ensures that callers are properly authenticated and unified with the defined parties.
It is the responsibility of the Runtime to ensure that each successful action results in the modified state being properly persisted. For that reason, NPL requires every action to be executed as a single transaction, thus guaranteeing all-or-nothing security.
The Runtime leverages this fact to also maintain a proper audit log, ensuring accountability, compliance and forensic analysis.

```
protocol [customer, bank] Loan(...) {
   ...
   permission [bank] writeOff() {
      ...
   };
   obligation [customer] pay(amount: Number) ... {
      ...
   } otherwise ... ;
   permission [bank, customer] updateCustomerAddress(...) {
      ...
   };
}
```

---

[6] Protocols can be viewed both as services and as a form of smart contracts; they are called protocols so they are not confused with smart contracts that require a blockchain to operate

[7] This terminology is derived from deontic logi

[8] The exception being so-called external parties which is a special construct that allows onboarding of new parties to a protocol..

DYNAMIC AUTHORIZATION

Authorization is not a static concept – many forms of authorization depend on process state or deadlines. In NPL, process states can be modeled explicitly and used as guards on permissions and obligations.

```
protocol [customer, bank] Loan(...) {
  initial state unpaid;
  final state paid;
  ...

  permission [customer] pay(amount: Number) | unpaid {
    ...
    if (...) {
      become paid;
    }
  }
}
```

In the same vein, date-time variables can be used as deadlines:

```
protocol [customer, bank] Loan(...) {
  initial state unpaid;
  final state paid;
  ...

  permission [customer] pay(amount: Number) | unpaid {
    ...
    if (...) {
      become paid;
    }
  }
}
```

The advantage of having these state guards and deadlines in the signature instead of in the body means that the actual authorization conditions can be determined a-priori – a feature not present in more traditional systems such as ABAC, RBAC and ReBAC.

INVOCATION AND IMPERSONATION

When calling permissions and obligations from within NPL, you are required to specify the party on whose behalf the action is executed. This party need not be the calling party, but again it must be a party defined within the current scope.

```
protocol [customer, bank] Loan(...) {
```

```
  var interestCalculator: OtherProtocol = ...;

  permission [customer] requestInterestAdjustment() {
    var adjusted = interestCalculator.calculate[bank](this);
  };
}
```

SERVICE SEMANTICS

The final key component of NPL is the @api annotation on constructors and actions, which tells the Runtime to expose these invocations as automatically generated service endpoints.

```
@api
permission [user] exposedPermission(...) { ... };
```

Permissions without an @api annotation are not exposed as an endpoint; and protocols without an @api annotation on their constructor cannot be instantiated through the API, thus enforcing the deny-by-default principle.

---

[9] A specific kind of preconditions: https://en.wikipedia.org/wiki/Guard_(computer_science)

[10] And the Runtime actually uses this in the exposed REST API to indicate which operations the current caller is allowed to execute in this state and at this time.

EXTERNAL SYSTEMS

As is usual in secure languages, NPL takes a dim view of system calls. To enforce safety and predictability, direct calls to the operating system are not allowed.

Unlike other systems, NPL also does not provide an abstraction layer to allow network and/or database calls. Apart from the fact that these calls could undermine NPL's security model, network calls also involve latency – which does not play nicely with the transactional constraints on executing actions.
This notwithstanding, systems do need to interact with the outside world. For this, NPL provides the concept of notifications and callbacks. Notifications fall outside the strict security boundaries defined within NPL itself. Callbacks on the other hand are normal actions and secured as usual.

Notifications are part of the transactional semantics, meaning that they are not actually emitted until the transaction has completed.

```
notification moneyTransferred(amount: Number);
...
permission [customer] pay(amount: Number) {
  ...
  notify moneyTransferred(amount) resume callback;
};
```

```
permission [bank] callback(res: NotifyResult) {
  ...
};
```

WHY NPL SOLVES KEY SECURITY CHALLENGES NATIVELY

NPL addresses the most important and common security weaknesses in today's software development practices. To this extent, let's turn to the OWASP Top 10 Web Application Security Risks.

A01: Broken access control – As explained above, NPL rigorously applies the deny-by-default principle on all access. Granting access is an explicit action, where the compiler ensures consistent definitions of authorization and the runtime takes care of enforcing said definitions. This in turn ensures that all direct object references still require proper authorization.

A03: Injection – Since the REST APIs are generated, the Runtime can ensure proper validation of user-supplied data. Furthermore, this data can never be passed directly into any kind of interpreter (be it SQL or eval-like constructs).

A04: Insecure design – NPL makes it impossible to define entry points without explicit authorization. This in turn moves authorization questions to the front of the requirements gathering process, resulting in more thoroughly secured applications.
The support for explicit process states and dynamic authorization makes it much easier to properly design a secure application in NPL.

A09: Security logging and monitoring failures – the NPL Runtime takes care of many non-functionals, including extensive audit logging and general observability. This is a huge improvement over traditional approaches where this is left to libraries that might or might not be properly applied; if implemented at all.

NPL's security-first architecture mitigates critical vulnerabilities by design, offering a more robust and reliable foundation for building secure modern and networked applications.

COMPARISON WITH OTHER CONTRACT-ORIENTED LANGUAGES

Finally, let's compare NPL to other contract-oriented languages such as those used in smart contracts, and analyze its relationship to capability-oriented programming paradigms.

[11] https://owasp.org/www-project-top-ten/

SMART CONTRACTS

Although many smart contract languages have an explicit notion of caller identification, NPL goes much further than this.

Solidity

The best-known example is Solidity which has an explicit address data type and provides the msg.sender and tx.origin properties to deal with authorization. When it comes to designing general secure services, it is less expressive than NPL:

- **Designed for the EVM** – Solidity is explicitly designed to run on the Ethereum VM where every operation has an associated cost. NPL makes no such assumption, and as such allows for higher-level software design.
- **No declarative authorization** – while the provided properties allow contracts to implement their own authorization mechanisms, nothing in Solidity enforces this. As is the case with any library and framework, proper authorization is an add-on.
- **Transparent by design** – Solidity was originally designed for Ethereum, a public blockchain where all transactions are visible to everyone – the exact opposite of deny-by-default when it comes to data privacy.

DAML

Another example is DAML which provides a party concept that looks very similar to the party concept in NPL. However, there are still a number of important differences:

- **Designed for distributed ledgers –** DAML is explicitly designed to run on top of a distributed ledger such as Canton. In comparison, NPL makes no such assumption.
- **Ledger-global party identifiers** – Each party identifier in DAML must be unique for all contracts on the same ledger. By contrast, party identifiers in NPL are local to the protocol on which they are defined.
- **IAM linked identities** – In DAML, it is the responsibility of the identity provider to tie identities to parties – NPL on the other hand requires no changes to the identity provider, but delegates the mapping to the NPL Runtime.

CONCLUSION

- In conclusion, NPL offers a fundamentally new, efficient and proven approach to software security by embedding authorization directly into the language and its runtime.
- This contrasts with traditional general-purpose languages that often rely on external libraries and frameworks, which can be insufficient or inconsistently applied. NPL addresses key security concerns such as broken access control, insecure design, and security logging failures.
- Compared to smart contract languages like Solidity and DAML, NPL provides more robust and flexible authorization models without being tied to specific distributed ledger technologies, making it a promising solution for developing secure online services.

---

12 https://soliditylang.org/

13 Using custom logic or libraries such as OpenZeppelin

14 https://www.digitalasset.com/developers

15 https://docs.daml.com/app-dev/parties-users.html

16 https://docs.daml.com/app-dev/authorization.html#custom-daml-claims-access-tokens