protoblocks — Programming language for secure implementation of cryptographic protocols (#6)

Vivek Nair
William Mullen

Introduction

Languages like Rust have effectively eliminated memory safety vulnerabilities through good programming language design alone. Can the same be done for cryptographic protocol vulnerabilities?

ProtoBlocks is a "protocol-safe" programming language that enforces best practices for implementing cryptographic protocols and thereby eliminates common vulnerabilities and pitfalls. It does this by providing a simple interface for writing well-defined protocols, and then automatically and transparently translating them into a "DataCapsule" structure that provides desirable security properties like integrity and binding. ProtoBlocks does not use formal methods to prove protocol functionality; instead, it uses "informal methods" to detect and mitigate common mistakes that account for the vast majority of protocol vulnerabilities in the wild. Therefore, unlike computationally-verifiable proof frameworks, it is easy to use and does not require any additional effort on the part of the implementer.

Secure Protocol Design Principles (Abadi & Needham)

1. Explicit Communication: every message should explicitly encode its own semantic meaning.
2. Appropriate Action: protocol specifications should clearly state the exact circumstances in which a message should be acted on.
3. Naming: every message should explicitly identify the principals it concerns.
4. Encryption: protocol specifications should state why encryption is being used.
5. Signing: signing a message does not necessarily imply knowledge of the content.
6. Freshness: protocol specifications should state why nonces and timestamps are being used.
7. Nonces: predictable nonces should be protected from adversaries.
8. Timestamps: timestamps require clock synchronization and thus trustworthy time servers.
9. Keys: recent use of a key does not imply integrity nor recent generation.
10. Encoding: every message should specify its own encoding scheme.
11. Trust: protocol specifications should identify and justify trust relationships.

Performance Evaluation

Protocol	Overhead w/o Network	Overhead w/ Network
ISO_1_Pass_Unilateral_Authentication_Over_CCF	753.33%	1729.41%
ISO_2_Pass_Unilateral_Authentication_Over_CCF	2920.00%	0.03%
ISO_1_Pass_Unilateral_Authentication	708.70%	912.90%
Nonce_Return_2_Pass_Unilateral_Authentication	1183.33%	0.03%
ISO_Public_Key_1_Pass_Unilateral_Authentication	43.57%	48.61%
ISO_Public_Key_2_Pass_Unilateral_Authentication	13.68%	1.17%
Nonce_Return_Public_Key_2_Pass_Unilateral_Authentication	790.20%	0.03%

As shown by the experimental results, ProtoBlocks is optimized for multi-step protocols and protocols using public-key cryptography. Most modern authentication protocols.are based on PKC and have multiple steps.

https://github.com/ProtoBlocks/Evaluation

Security Evaluation

- Needham–Schroeder Symmetric Key Protocol: The standalone protocol is vulnerable to a replay attack, which the ProtoBlocks compiler was able to automatically fix by binding K_{AB} to a single protocol instance identifier.
- Needham–Schroeder Public-Key Protocol: The standalone protocol is vulnerable to a man-in-the-middle attack, which ProtoBlocks was able to automatically fix by binding K_{AB} to the identity of principals A and B.
- Woo-Lam Public-Key Protocol: The standalone protocol is vulnerable to a replay attack because a nonce (N_A) was not bound to its issuer (A). ProtoBlocks was able to automatically fix this by binding N_A to A.

ProtoBlocks Language

The ProtoBlocks language provides a convenient way to implement protocols where correct use of the .pb.js syntax is sufficient to ensure a number of desirable security properties. ProtoBlocks protocols are defined as a well-ordered series of steps (*[2] appropriate action*), each of which has a unique name for both the step itself and all of its expected outputs (*[1] explicit communication*). Each step explicitly identifies its intended sender and recipients (*[3] naming*), and when outputs of prior steps are used in later steps, the intended trusted source of the value must be specified (*[11] trust*). The language also provides secure helper functions for hashing, encryption & decryption (*[4] encryption*), nonces (*[7] nonces*), signing & verifying (*[5] signing*), timestamps (*[8] timestamps*), and key management *([9] keys*). As such, simply using the provided interface enforces good implementation choices.

https://github.com/ProtoBlocks/ProtoBlocks
https://www.npmjs.com/package/protoblocks

ProtoBlocks Transpiler

The goal of the ProtoBlocks transpiler is to transform protocols written in the ProtoBlocks language (.pb.js) into standard JavaScript files (.js) using the ProtoBlocks library, which in turn can be used in servers (node) and a variety of clients (browsers, electron desktop applications, react native mobile applications, etc.). This was accomplished by modifying Babel to support the ProtoBlocks syntax and writing a custom Babel transformer to generate the vanilla JavaScript syntax expected by the ProtoBlocks JavaScript library via direct manipulation of abstract syntax trees (ASTs). This only required defining semantics for two unique keywords (*protocol* and *step*), with the rest of the functionality taking advantage of existing JavaScript syntax.

https://github.com/ProtoBlocks/Transpile
https://github.com/ethanlee16/babel

DataCapsule Protocol Encoding

DataCapsules (DCs) are the structure which underpins our JavaScript implementation. A DC is an append-only blockchain of records, each with their own type-agnostic data and hashes. Each record also stores a signed hash of the previous record, creating a well-ordered chain. In our JavaScript DC library, the first record stores protocol metadata while later records represent a specific step. Due to this structure, protocol information and the data associated with each step are bound to specific records and DCs, preventing misuse. The hashes preserve the overall integrity of protocol, ensuring attacks are quickly detected. Thus, the DCs' structure provides a natural match for ProtoBlocks.

https://github.com/GlobalDataPlane/DataCapsuleJS
https://www.npmjs.com/package/datacapsulejs

Example: ISO 9798 2-Pass Unilateral Authentication Over CCF

Security Protocol Notation

1. B → A: N_B
2. A → B: $f_{KAB}(N_B||B)$
3. B accepts iff CCF matches expected value

ProtoBlocks Language Implementation (.pb.js)

```
protocol ISO_2_Pass_Unilateral_Authentication_Over_CCF [Prover(Secret), Verifier(Secret)] {
  step Challenge [Verifier -> Prover] {
    const Nonce = nonce();
    Prover.send({"Nonce": Nonce});
  }

  step Response [Prover -> Verifier] {
    const Hash = hash(Verifier.Challenge.Nonce + Verifier.Id + Prover.Input.Secret);
    Verifier.send({"Hash": Hash});
  }

  step Verify [Verifier] {
    const Hash = hash(Verifier.Challenge.Nonce + Verifier.Id + Verifier.Input.Secret);
    return (Hash === Prover.Response.Hash);
  }
}
```

ProtoBlocks Library Implementation (.js)

```
const { Protocol, hash, nonce } = require('../../src')

/* eslint camelcase: "off" */
const ISO_2_Pass_Unilateral_Authentication_Over_CCF = new Protocol({
  name: 'ISO_2_Pass_Unilateral_Authentication_Over_CCF',
  principals: [{name: 'Prover', inputs: ['Secret']}, {name: 'Verifier', inputs: ['Secret']}],
  steps: [{origin: 'Verifier', recipients: ['Prover'], name: 'Challenge', function: async (Prover, Verifier) =>
    const Nonce = nonce()
    Prover.send({Nonce: Nonce})
  }}, {origin: 'Prover', recipients: ['Verifier'], name: 'Response', function: async (Prover, Verifier) => {
    const Hash = hash(Verifier.Challenge.Nonce + Verifier.Id + Prover.Input.Secret)
    Verifier.send({Hash: Hash})
  }}, {origin: 'Verifier', recipients: [], name: 'Verify', function: async (Prover, Verifier) => {
    const Hash = hash(Verifier.Challenge.Nonce + Verifier.Id + Verifier.Input.Secret)
    return Hash === Prover.Response.Hash
  }}]
})

module.exports = ISO_2_Pass_Unilateral_Authentication_Over_CCF
```

DataCapsule Execution Trace

META
Protocol: 2PassUnilateralAuthCCF
Version: ISO9798-2
Instance: e73b464c-78ba-4e44-93cf
Encoding: pbjs1-dcjs1-json
Timestamp: 1638122987988
Principals:
Prover: Alice <pk_A>
Verifier: Bob <pk_B>

CHALLENGE
BOB → ALICE
Parent: 7c3616c91896d4380194989
Nonce: 0e531ff0-163b-46f8-a565
@Bob <sk_B>

RESPONSE
ALICE → BOB
Parent: 68401f5fd35679d0d8595cad
Hash: 1fe758faa76c6936d8f580f544
@Alice <sk_A>

VERIFY
ALICE
Parent: 0813c0e3f9e9d73f729b508b
Result: True
@Bob <sk_B>