Verifpal

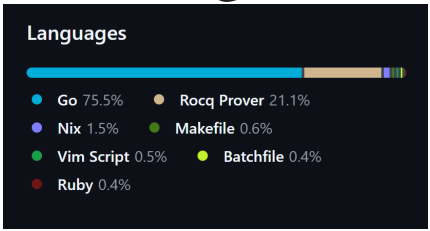
Radu-Constantin Onuțu

Faculty of Mathematics and Computer Science
University of Bucharest

Table of Contents

What is Verifpal?

- Symbolic verification tool for protocols
- A language easier to write and understand than the ones from other tools
- Supports passive and active (Dolev–Yao) attacker models
- Core written in Go
- Released in 2019



What are the 4 parts of a workflow?

A Verifpal model typically has:

1. **Attacker** declaration: `attacker[passive]` or `attacker[active]`
2. **Principals** (roles): `principal Alice[ ... ]`, `principal Bob[ ... ]`
3. **Messages**: `Alice -> Bob: ...`
4. **Queries**: `queries[ ... ]`

```
1  attacker[active]
2
3  principal Alice[
4      knows private a
5  ]
6
7  principal Bob[
8      knows private b
9  ]
10
11 Alice -> Bob: a
12
13 queries[
14     confidentiality? a
15 ]
16
```

Principals

Inside a principal block you typically use:

- `knows public ...` / `knows private ...` (initial knowledge)
- `generates ...` (fresh per-session values)
- `leaks ...` (explicit compromise / leakage scenarios)

```
attacker[active]

principal Alice[
  knows public  pkB
  knows private skA
  generates nA
  leaks skA

  m  = HASH(nA)
  s  = SIGN(m, skA)
  ct = ENC(pkB, s)
]

Alice -> Bob: ct

principal Bob[
  knows private skB
  x = DEC(skB, ct)
]

queries[
  confidentiality? nA
]
```

Messages

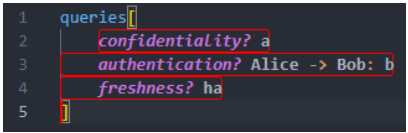
- Network sends are written explicitly: `Alice -> Bob: t1, t2, ...`
- **Guarded constants**: `[x]` indicates the attacker can read but cannot replace that value (models authenticated delivery of that value).
- **Checked primitives** (often marked with `?`): the role **aborts** if verification/decryption/assertion fails.

```
1  Bob -> Alice: [gb], sig
2
3  principal Alice[
4    msg = HASH(ga, gb)
5    _   = SIGNVERIF(pkA, msg, sig)?
6  ]
```

Queries

Common query types:

- **Confidentiality**: can the attacker obtain *a*?
- **Authentication**: can the attacker impersonate Alice and send Bob *b*?
- **Freshness**: used to detect replay attacks



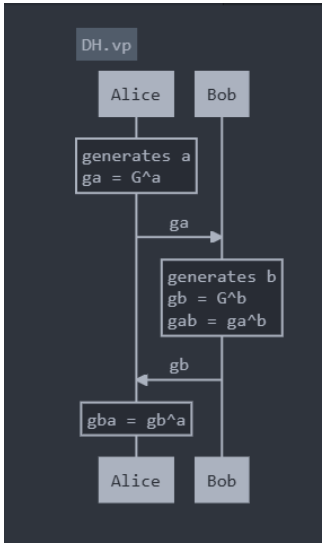
Diffie–Hellman key exchange

- We compare:
  1. unauthenticated DH under an active attacker,
  2. a guarded/authenticated variant.

Unauthenticated DH

```
1  attacker[active]
2
3  principal Alice[
4      generates a
5      ga = G^a
6  ]
7
8  Alice -> Bob: ga
9
10 principal Bob [
11     generates b
12     gb = G^b
13     gab = ga^b
14 ]
15
16 Bob -> Alice: gb
17
18 principal Alice[
19     gba = gb^a
20 ]
21
22 queries[
23     confidentiality? gab
24 ]
```

Unauthenticated DH Diagram



Unauthenticated DH Analysis

```
confidentiality? gab
When:
        gb → G^nil ← mutated by Attacker (originally G^b)
        gab → G^a^b
        gba → G^nil^a

        G^nil^b is obtained:
        ga → G^nil ← mutated by Attacker (originally G^a)
        gab → G^nil^b ← obtained by Attacker
        gba → G^b^a
      gab (G^nil^b) is obtained by Attacker.
```

Unauthenticated DH

- Because ga and gb are sent unprotected, the attacker can replace them with attacker-chosen values.
- Result: "`confidentiality? gab`" fails under an active attacker.

Authenticated DH

```
1  attacker[active]
2
3  principal Alice[
4      generates a
5      ga = G^a
6  ]
7
8  Alice -> Bob: [ga]
9
10 principal Bob [
11     generates b
12     gb = G^b
13     gab = ga^b
14 ]
15
16 Bob -> Alice: [gb]
17
18 principal Alice[
19     gba = gb^a
20 ]
21
22 queries[
23     confidentiality? gab
24 ]
```

Denning–Sacco Protocol

- We want to establish a session key k (signed by Alice) for Bob, then use k to protect data.
- What happens if an attacker substitutes the key from Bob to Alice?
- If the session key is not **bound to the intended peer**, an attacker can replay it.

Denning–Sacco Protocol

```
1   attacker[active]
2
3   // - skA is Alice's private signing key
4   // - pkA is the corresponding public key derived from skA
5   principal Alice[
6     knows private skA
7     pkA = G^skA
8   ]
9
10  // Alice sends to Bob her public key (non modifiable by the attacker)
11  Alice -> Bob: [pkA]
12
13  // - skB is Bob's private decryption key
14  // - pkB is Bob's public encryption key derived from skB
15  // - secretB is some confidential value that Bob wants to protect
16  principal Bob[
17    knows private skB
18    pkB = G^skB
19    knows private secretB
20  ]
```

Denning–Sacco Protocol

```
22  // Here is the vulnerability in the Denning-Sacco protocol
23  // Bob sends pkB over an *unprotected* channel
24  // Because the attacker is active, they can intercept/replace pkB with their own public key
25  // Alice cannot tell the difference and may encrypt to the attacker's key by mistake
26  Bob -> Alice: pkB
27
28  // [Solution: Bob sends pkB over authenticated channel <=> [pkB]]
```

Denning–Sacco Protocol

```
principal Alice[
  // Alice creates a fresh session key k for this run of the protocol.
  generates k

  // Alice signs k with her private signing key skA.
  // This proves to anyone who knows pkA that "k came from Alice".
  sigK = SIGN(skA, k)

  // Alice packages the session key together with its signature.
  // (Verifpal needs both values so Bob can recover k and then verify the signature.)
  payload = CONCAT(k, sigK)

  // Alice encrypts the payload with "pkB" as received from the network.
  // If pkB was replaced by the attacker, this encryption goes to the attacker instead of Bob.
  c1 = PKE_ENC(pkB, payload)
]
```

Denning–Sacco Protocol

```
47  // Message (1): Alice sends the encrypted (k, signature) bundle.
48  // Attacker intercepts c1 and decrypts it using their private key skH
49  // Attacker learns k and sigK
50  // Attacker now re-encrypts the exact same payload under Bob's real public key
51  // (which the attacker knows from the original message):
52  Alice -> Bob: c1
```

Denning–Sacco Protocol

```
54  principal Bob[
55    // Bob decrypts using his private key skB.
56    // If c1 was truly encrypted under Bob's pkB, Bob recovers the payload.
57    payload_b = PKE_DEC(skB, c1)
58
59    // Split the payload back into the session key and Alice's signature on it.
60    kB, sigKB = SPLIT(payload_b)?
61
62    // Bob checks that the signature is valid under Alice's public key pkA.
63    // The "?" means this step is checked: if verification fails, the protocol run stops here.
64    kB_ok = SIGNVERIF(pkA, kB, sigKB)?
65
66    // Bob uses the verified session key to encrypt his secret.
67    // If the attacker learned k (via key substitution earlier), they can decrypt this.
68    c2 = ENC(kB_ok, secretB)
69  ]
70
```

Denning–Sacco Protocol

```
70
71  // Message (2): Bob sends the secret encrypted under the session key.
72  Bob -> Alice: c2
73
74  queries[
75    // Security goal: the attacker should NOT learn secretB.
76    // In this vulnerable variant, Verifpal should find an attack showing secretB leaks.
77    confidentiality? secretB
78  ]
```

Denning–Sacco Protocol solution

```
26   Bob -> Alice: [pkB]
27
28   // [Solution: Bob sends pkB over authenticated channel <=> [pkB]]
29   // or make Alice compute the signature of pkB
30   // now the attacker will not be able to change the signature because he needs skA
31
```

Thank you!