Lab Report 2 - Authenticated Key Exchange

Yuval Weiss
yw580

March 2025

SIGMA [1] and SPAKE2 [2] are two protocols for authenticated key exchange. The former relies on a certification authority for authentication, while the latter depends on a shared password distributed in advance.

1 Implementation

1.1 Language and packages

The protocols within this report are written in Go, thus necessitating pre-written implementations of X25519 and Ed25519. For Ed25519, a mixture of the standard library's crypto package and the implementation in filippo.io/edwards25519[1] was used, while for X25119 the implementation in crypto sufficed. The other primitives, HMAC, SHA256 and HKDF used the implementations in crypto.

1.2 Outline and approach

Go, by contrast to Python, allows for explicitly deciding whether a name should be exported from a package. By starting the name with an uppercase letter, it is exported. Therefore, determining which parts of the API should be exposed or kept private was a lot more pertinent than in the previous lab (which was written in Python).

The implementations for SIGMA and SPAKE2 utilised the typestate pattern. Both client types contain a state member which is transformed throughout the execution of each protocol. The type assertion and check are performed simultaneously using the `t, ok := i.(T)` syntax. This makes two important tasks simpler to deal with: managing secret lifetimes and enforcing a correct sequence of method calls. Furthermore, many types are exposed only as the return types of functions, ensuring correct usage and preventing erroneous initialisation of structs.

The simulation of messages over a network was done by encoding message structs to JSON bytes. Error messages and comments are occasionally modified, abbreviated or omitted entirely in the code snippets shown to improve their clarity or brevity.

[1] Available here: https://pkg.go.dev/filippo.io/edwards25519@v1.1.0

1.3 SIGMA

1.3.1 Certificate authority

The main types defining the certificate authority are shown in Listing 1. The internal `struct` remains unexported to prevent manual initialisation of the certificate authority, and a pointer type is exported in its place. A method, `NewAuthority` is used to initialise a new authority, automatically generating the required `Ed25519` keys.

```go
type CertificateAuthority = *certAuth

type certAuth struct {
    regcerts    map[string]Certificate
    authPubKey  ed25519.PublicKey
    authPrivKey ed25519.PrivateKey
}

func NewAuthority() CertificateAuthority {
    pub, priv, err := ed25519.GenerateKey(nil)
    if err != nil {
        panic(fmt.Sprintf("failed ... due to key gen error %v", err))
    }
    return &certAuth{
        regcerts:    make(map[string]Certificate),
        authPubKey:  pub,
        authPrivKey: priv,
    }
}
```

Listing 1: Certificate authority type definitions

I chose a barebones certificate structure, outlined in Listing 2, for the authority. The certificate contains the name of the entity, the start and end timestamps of the certificate's validity, and the `Ed25519` public key being stored. `ValidatedCertificate` is the promoted type, encapsulating a certificate with the authority's `Ed25519` signature on the JSON data, which can then be verified by a third party with the authority's key.

The three methods defined on `CertificateAuthority` are shown in Listing 3. `Register` registers an entity with the authority. As long as the data is valid then the operation succeeds. If the client is already registered, and a new public key is provided then the registration is updated. If the same key is provided then the registration remains unchanged but no error is thrown. `Certify` returns the encoding of a `ValidatedCertificate` if the client has a valid registration, and a `nil` array with an error otherwise. `VerifyCertificate` can be called by a third party to ensure that the entity's validated certificate is indeed valid, and has not been fabricated.

1.3.2 Protocol

Interaction with the `SIGMA` protocol is through client instances which undergo type promotion before starting the protocol. These types, and the functions that transition

```go
type Certificate struct {
    Name      string            `json:"name"`       // name of the entity/user
    Start     time.Time         `json:"start"`      // validity start time
    End       time.Time         `json:"end"`        // validity end time
    PublicKey ed25519.PublicKey `json:"public_key"` // entity/user's public key
}

type ValidatedCertificate struct {
    Cert Certificate `json:"cert"`
    Sig  []byte      `json:"sig"` // signature (on the marshalled version of cert)
}
```

Listing 2: Certificate authority type definitions

```go
func (ca CertificateAuthority) Register(data []byte) ([]byte, error)
func (ca CertificateAuthority) Certify(name string) ([]byte, error)
func (ca CertificateAuthority) VerifyCertificate(data []byte) bool
```

Listing 3: Methods defined on the certificate authority `struct`

between them, are shown in Listing 4. All four client types are unexported to prevent manual creation by users of the API, ensuring logical protocol flow. `baseClient` is intended to be independent of individual runs of the protocol and thus contains persistent information such as the name of the client, and its public and private `Ed25519` keys. Registration with a certificate authority is a prerequisite for `SIGMA`. Thus, `baseClient` is promoted to `registeredClient` through the `Register` method. This instance now also contains a pointer to the certificate authority it is registered to. In a true networked scenario, this would be replaced by some sort of identifier for the CA. At this point, the promotion can go one of two ways. If the user intends to be the initiator of the protocol (in the role of 'Alice', according to the slides [3]), then the `AsInitiator` method should be called. Otherwise, the user's 'Bob' role is called challenger, and the `AsChallenger` method should instead be used. `initiatorClient` and `challengerClient` both contain a `state` member, which, through the typestate pattern, guarantees logically correct protocol flow. For `initiatorClient` the typestates implement the `initiatorState` interface and transition as follows

```
initiatorBaseState --Initiate-> initiatorBegunState --Respond-> completedState
```

while for `challengerClient` the typestates implement `challengerState` and flow according to

```
challengerBaseState --Challenge-> challengerBegunState --Finalise-> completedState
```

where `completedState` is used by both clients, (i.e. implements `initiatorState` and `challengerState`). The names in the arrows correspond to the functions called to transition between each state.

```go
type baseClient struct {
    name    string
    public  ed25519.PublicKey
    private ed25519.PrivateKey
}
// creates a new baseClient
func NewBaseClient(name string) *baseClient

type registeredClient struct {
    *baseClient  // indicates struct embedding, so all members are accessible
    ca   CertificateAuthority
    cert Certificate
}
// promotes baseClient -> registeredClient
func (c *baseClient) Register(ca CertificateAuthority) (*registeredClient, error)

type InitiatorClient = *initiatorClient // export pointer
type initiatorClient struct {
    *registeredClient
    state initiatorState
}
// promotes registeredClient -> initiatorClient
func (c *registeredClient) AsInitiator() *initiatorClient

type ChallengerClient = *challengerClient // export pointer
type challengerClient struct {
    *registeredClient
    state challengerState
}
// promotes registeredClient -> challengerClient
func (c *registeredClient) AsChallenger() *challengerClient
```

Listing 4: Type promotion flow of the SIGMA protocol client types

```go
// begin protocol
g_x, _ := initiator.Initiate()
// challenger sends the challenge message
challenge, _ := challenger.Challenge(g_x)
// initiator responds again and derives the session key
resp, err_i := initiator.Respond(challenge)
// challenger finalises & gets session key, err is nil -> key is in client state
err_c := challenger.Finalise(resp)
// if err_i and err_c are nil, retrieve keys with
initiator.SessionKey(); challenger.SessionKey()
```

Listing 5: Full SIGMA protocol flow, omitting any error handling. In this snippet, the returned errors from each protocol stage are left unbound.

Listing 5 shows the full structure of the SIGMA protocol, overlooking most of the required error handling for brevity. However, the returned errors are an important part of the protocol control flow, indicating whether each client should continue with protocol execution. The last functions for each type of client, `Respond` and `Finalise` return an error with no data since they do not send any information on the network. If this error is nil, the final SIGMA-derived session key is accessible through the `client.SessionKey()` method.

1.3.3 Chat

The chat functionality uses a `chatSession` struct to manage sending and receiving messages. This session cannot be manually created but requires the `EstablishSecureChat` function to be called, which returns two session structs, one for each user. I chose the Advanced Encryption Standard (AES) [4] block cipher with Galois/Counter Mode (GCM) [5], as specified in RFC 5288 for TLS [6], for the required symmetric encryption. Implementation of both of these are included in Go's standard `crypto` package. The `encryptedMessage` struct is serialised/deserialised from JSON as required for sending/receiving a message.

```go
func EstablishSecureChat(initiator sigma.InitiatorClient, challenger
↪  sigma.ChallengerClient) (ChatSession, ChatSession, error)

type ChatSession = *chatSession // exported pointer to prevent manual initialisation
type chatSession struct { // unexported struct
    local      string // name of local client
    remote     string // name of remote client
    sessionKey []byte // SIGMA-derived session key (32 bytes)
}

func (cs *chatSession) SendMessage(content string) ([]byte, error) {
    return cs.encrypt(NewMessage(cs.local, cs.remote, content))
}
func (cs *chatSession) ReceiveMessage(data []byte) (Message, error) {
    return cs.decrypt(data)
}

type Message struct { // exported message type
    Sender    string    `json:"sender"`
    Recipient string    `json:"recipient"`
    Content   string    `json:"content"`
    Timestamp time.Time `json:"ts"`
}

type encryptedMessage struct { // hidden struct for encrypted messages
    IV         []byte `json:"iv"` // Initial Vector used for GCM
    Ciphertext []byte `json:"ciphertext"` // ciphertext created by GCM
}
```

Listing 6: Key structs and methods used for SIGMA-based chat

1.4 SPAKE2

The `SPAKE2` protocol is symmetric in message structure (essentially differing only in which keys are used), thus a single client type can be defined. The client, shown in Listing 7, is initialised with the `NewClient` function from a `string` password. The listing also outlines the typestate transitions, with the methods that induce each transition written inside the arrows. The `initiate` method is unexported and takes a single boolean argument, indicating whether the client is in the 'Alice' or 'Bob' role, as specified in the slides [3]. To make the API clearer, two wrapper methods `InitiateAsAlice` and `InitiateAsBob` are exported instead.

```
type Client = *client //  public client
type client struct { // internal SPAKE2 client struct
    password []byte
    state    clientState
}
func NewClient(password string) Client // initialises Client at baseState

// typestate transitions
baseState                   // start state
--initiate-> initiatedState // client has begun the protocol, first message sent
--Derive->   derivedState   // key has been derived, awaiting validation
--Validate-> validatedState // final state, call client.Key() to get the final key
```

Listing 7: SPAKE2 client struct. Also shows the protocol's typestate evolution.

The end-to-end flow of the protocol is shown in Listing 8. There is no need to implement custom message `structs` since the data passed between the clients at each stage of the protocol is already in byte format. The required constants, (the points M and N and the cofactor H) as defined in the RFC [2], are declared and initialised in `init.go` in the `init` function, a special function run exactly once, on the first import of the package. The implementation required the `filippo.io/edwards25519` package, which exposes simple APIs for creating group elements, as well as scalar and point arithmetic.

```
a, b := NewClient("password"), NewClient("password")
pi_a, _ := a.InitiateAsAlice()
pi_b, _ := b.InitiateAsBob()
mu_a, _ := a.Derive(pi_b)
mu_b, _ := b.Derive(pi_a)
err_a := a.Validate(mu_b)
err_b := b.Validate(mu_a)
a_k, _ := a.Key() // works if err_a is nil
b_k, _ := b.Key() // works if err_b is nil
```

Listing 8: End-to-end `SPAKE2` protocol flow

2 Validation

Due to the lack of RFC for `SIGMA`, code correctness could only be asserted with compile-time checks and comparison to the slides [3]. Furthermore, while `SPAKE2` does have an RFC [2], the slides provided a clearer specification of the formulae. The typestate and type promotion patterns also helped to ensure code correctness, as the Go compiler can type-check the implementations, in a stricter way than `mypy`, which relies on type annotations to function. The Go compiler also prevents compilation if variables are declared and unused, and warns about all unused names.

A white-box testing methodology was used throughout, meaning that tests were written within each package to enable access to unexported types, functions and struct fields. Tests were written using Go's `testing` package alongside `gotestsum`[2] as the test runner for more human-friendly output. Test vectors are not readily available for either protocol. Instead, selected test methods are run 1000 times, aiming to cover different random keys. Overall, there are 3035 test cases, over 35 unique tests. The unit tests ensure correctness and validate failure modes at the protocol, function and primitive levels. Code coverage for each package is shown in Table 1. Coverage for `sigmachat` is relatively lower due to the `EstablishSecureChat` function, for which it is quite difficult to test the error handling lines without a true network. The procedure is broken down in the `TestManualSigma` and `TestSigmaErrors` functions.

Package	Coverage / %
`cert_auth`	90.0
`sigma`	86.1
`sigmachat`	75.4
`spake2`	90.7

Table 1: Unit test code coverage for each package in the implementation

3 Findings

Learning from the last lab, all keys are stored in `structs` in private fields and thus cannot be prematurely exposed. Furthermore, any getter methods perform a defensive `clone` to prevent post hoc mutation of the original key. For `sigmachat`, the derived key is never exposed and is exclusively handled by `ChatSession`.

Before this lab, I had little experience with the typestate and type promotion patterns. After some initial code refactoring, I found working within these paradigms to be quite straightforward, and that they enabled a clearer mental model of code execution (in addition to the obvious type assertion benefits) than the messier alternatives.

[2]Available here: `https://github.com/gotestyourself/gotestsum`

4 Productionisation

As before, a production-quality implementation must be resistant to side-channel attacks. For example, timing analysis could be used to leak information about secrets, thus implementations should be constant time. Furthermore, these protocols require long-term key storage. Therefore, hardware security modules such as Intel SGX and Apple Secure Enclave can store keys persistently.

With regards to networking performance, it would be beneficial to use more data-efficient message structures than `JSON`. Alternatives could be to serialise the structs using `protobuf` or Go's own `gob`[3] format. Alternatively, a custom byte encoding could be devised, since there is a constrained number of messages to encode across the protocols.

A production-quality implementation might also leverage Go's concurrency primitives, such as goroutines, for increased performance. This would be especially pertinent in the certification authority and the `sigmachat` client, allowing for both to efficiently handle numerous concurrent requests. The CA would also need to implement some form of identity verification, likely utilising either domain or extended validation.

References

[1] H. Krawczyk, "SIGMA: The 'SIGn-and-MAc' Approach to Authenticated Diffie-Hellman and Its Use in the IKE Protocols," in *Advances in Cryptology - CRYPTO 2003*, D. Boneh, Ed., red. by G. Goos, J. Hartmanis, and J. Van Leeuwen, vol. 2729, Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 400–425, ISBN: 978-3-540-40674-7 978-3-540-45146-4. DOI: 10.1007/978-3-540-45146-4_24. [Online]. Available: http://link.springer.com/10.1007/978-3-540-45146-4_24.

[2] W. Ladd, "SPAKE2, a Password-Authenticated Key Exchange," Internet Engineering Task Force, Request for Comments RFC 9382, Sep. 2023, 17 pp. DOI: 10.17487/RFC9382. [Online]. Available: https://datatracker.ietf.org/doc/rfc9382.

[3] M. Kleppmann and D. Hugenroth, "P79: Cryptography and Protocol Engineering," Lecture Slides (University of Cambridge), 2025. [Online]. Available: https://www.cl.cam.ac.uk/teaching/2425/P79/p79-slides.pdf.

[4] National Institute of Standards and Technology (US), "Advanced Encryption Standard (AES)," National Institute of Standards and Technology (U.S.), Washington, D.C., NIST FIPS 197-upd1, May 9, 2023, NIST FIPS 197–upd1. DOI: 10.6028/NIST.FIPS.197-upd1. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197-upd1.pdf.

[5] D. A. McGrew and J. Viega, "The Galois/Counter Mode of Operation (GCM)," May 31, 2005.

[3] Discussed here: https://go.dev/blog/gob.

[6] J. A. Salowey, D. McGrew, and A. Choudhury, “AES Galois Counter Mode (GCM) Cipher Suites for TLS,” Internet Engineering Task Force, Request for Comments RFC 5288, Aug. 2008, 8 pp. DOI: 10.17487/RFC5288. [Online]. Available: https://datatracker.ietf.org/doc/rfc5288.