ECE 518 Project 3

Alan Palayil Due Date: 11/13/2022

Section I Overview

In this project, we are going to use the RSA signature in Go to implement the Probabilistic Signature Scheme (PSS) and Optimal Asymmetric Encryption Padding (OAEP) is used for RSA encryption. We complete the protocol and test it using the knowledge of public-key cryptography and digital signature.

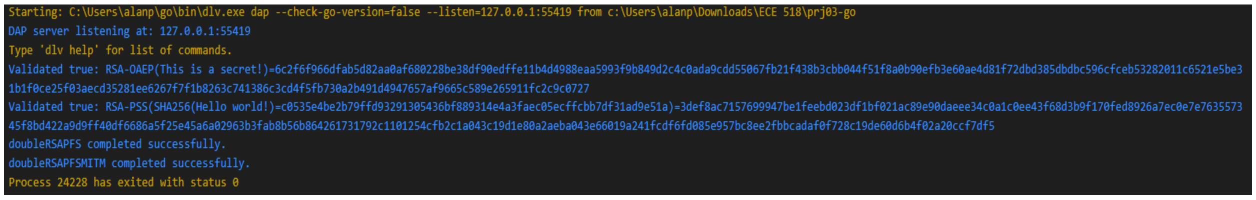
Section II RSA Signature and Encryption

We first try to understand how to sign/verify messages in RSA using PSS and how to encrypt/decrypt messages in RSA using OAEP in Go. We read the two functions in validate. go: validateRSAPSS () and validateRSAOAEP ().

Section III Implement the Protocol

Implementation of the double RSA protocol starts with the function doubleRSAPFS (), which calls a few more functions, each implements one or more protocol steps. We have modified the three functions aliceVerify, aliceEncrypt, and bobDecrypt for the last 3 steps since the implementation of the first 3 steps are already provided. Moreover, since our protocol may also prevent certain cases of man-in-the-middle (MITM) attacks, the function doubleRSAPFSMITM () simulates a possible attack scenario.

```go
/*
We use rsa. VerifyPSS to verify the PSS signature which includes the public-key
(Bob authentication), crypto hash (which is crypto. SHA256), digest [] byte
(hash[:]), sig [] byte (Bob encryption public signature), and opts *PSSOptions
(nil).
*/
func aliceVerify(bobAuthPub, bobEncPub *rsa.PublicKey, bobEncPubSig []byte) bool
{
    msg := fmt.Sprintf("N=%v E=%v", bobEncPub.N, bobEncPub.E)
    hash := sha256.Sum256([]byte(msg))

    // Modify code below to check if the signature of hash is bobEncPubSig
    // using bobAuthPub. Return true if yes, or false otherwise.
    err := rsa.VerifyPSS(bobAuthPub, crypto.SHA256, hash[:], bobEncPubSig, nil)
    if err == nil {
        return true
    }
    unused(hash)
    return false
}

/*
We use rsa. EncryptOAEP to encrypt the given message with RSA-OAEP. It is
parameterized by a hash function (sha256.New()) and the PublicKey (bobEncPub).
This is pushed to a cipertext to ensure that an attacker cannot use it.
*/
func aliceEncrypt(bobEncPub *rsa.PublicKey, plaintxt []byte) []byte {
    // Modify code below to encrypt plaintxt using bobEncPub
    // and return the ciphertxt.
    label := ""
      ciphertxt,  _  :=  rsa.EncryptOAEP(sha256.New(),  rand.Reader,  bobEncPub,
[]byte(plaintxt), []byte(label))

    return ciphertxt

}

/*
We use rsa. DecryptOAEP to decrypt the ciphertext using RSA-OAEP. Similar to
EncryptOAEP, the function decrypts using the same functions and returns a
plaintext.
*/
func bobDecrypt(bobEnc *rsa.PrivateKey, ciphertxt []byte) []byte {
    // Modify code below to decrypt ciphertxt using bobEnc
    // and return the plaintxt.
    label := ""
    plaintxt, _ := rsa.DecryptOAEP(sha256.New(), rand.Reader, bobEnc, ciphertxt,
[]byte(label))

    return plaintxt

}
```
Once the protocol is implemented correctly, doubleRSAPFS () will output "doubleRSAPFS completed successfully." Our implementation of aliceVerify, aliceEncrypt, and bobDecrypt, without modification, should be able to protect against this attack so that the function should output "doubleRSAPFSMITM completed successfully."



```
Starting: C:\Users\alanp\go\bin\dlv.exe dap --check-go-version=false --listen=127.0.0.1:55419 from c:\Users\alanp\Downloads\ECE 518\prj03-go
DAP server listening at: 127.0.0.1:55419
Type 'dlv help' for list of commands.
Validated true: RSA-OAEP(This is a secret!)=6c2f6f966dfab5d82aa0af680228be38df90edffe11b4d4988eaa5993f9b849d2c4c0ada9cdd55067fb21f438b3cbb044f51f8a0b90efb3e60ae4d81f72dbd385dbdbc596cfceb53282011c6521e5be3
1b1f0ce25f03aecd35281ee6267f7f1b8263c741386c3cd4f5fb730a2b491d4947657af9665c589e265911fc2c9c0727
Validated true: RSA-PSS(SHA256(Hello world!)=c0535e4be2b79ffd93291305436bf889314e4a3faec05ecffcbb7df31ad9e51a)=3def8ac7157699947be1feebd023df1bf021ac89e90daeee34c0a1c0ee43f68d3b9f170fed8926a7ec0e7e7635573
45f8bd422a9d9ff40df6686a5f25e45a6a02963b3fab8b56b864261731792c1101254cfb2c1a043c19d1e80a2aeba043e66019a241fcdf6fd085e957bc8ee2fbbcadaf0f728c19de60d6b4f02a20ccf7df5
doubleRSAPFS completed successfully.
doubleRSAPFSMITM completed successfully.
Process 24228 has exited with status 0
```

```
Starting: C:\Users\alanp\go\bin\dlv.exe dap --check-go-version=false --
listen=127.0.0.1:55419 from c:\Users\alanp\Downloads\ECE 518\prj03-go
DAP server listening at: 127.0.0.1:55419
Type 'dlv help' for list of commands.
Validated true: RSA-OAEP(This is a
secret!)=6c2f6f966dfab5d82aa0af680228be38df90edffe11b4d4988eaa5993f9b849d2c4c0ada
9cdd55067fb21f438b3cbb044f51f8a0b90efb3e60ae4d81f72dbd385dbdbc596cfceb53282011c65
21e5be31b1f0ce25f03aecd35281ee6267f7f1b8263c741386c3cd4f5fb730a2b491d4947657af966
5c589e265911fc2c9c0727
Validated true: RSA-PSS(SHA256(Hello
world!)=c0535e4be2b79ffd93291305436bf889314e4a3faec05ecffcbb7df31ad9e51a)=3def8ac
7157699947be1feebd023df1bf021ac89e90daeee34c0a1c0ee43f68d3b9f170fed8926a7ec0e7e76
3557345f8bd422a9d9ff40df6686a5f25e45a6a02963b3fab8b56b864261731792c1101254cfb2c1a
043c19d1e80a2aeba043e66019a241fcdf6fd085e957bc8ee2fbbcadaf0f728c19de60d6b4f02a20c
cf7df5
doubleRSAPFS completed successfully.
doubleRSAPFSMITM completed successfully.
Process 24228 has exited with status 0
Detaching
dlv dap (28624) exited with code: 0
```

1. Why do we need to apply RSA twice? Why can't Bob simply generate an RSA encryption key every time Alice wants to send a message and skip the authentication/signature part?
   - In a single RSA encryption, if you have the right key, the cipher text is converted into plain text. To ensure the security of the data, we apply RSA encryption twice (authentication/signature) should the any one key be compromised. Bob can simply generate an RSA encryption key every time Alice wants to send a message and skip the authentication/signature part, that can open a Man-In-The-Middle attack which leads to a less secure communication.

2. For step 3, when Bob delivers (EncPublic, EncPubSig) to Alice, what kind of channel is needed? A secure channel? An authentic channel? Or an insecure channel is perfectly, OK?
   - When Bob delivers (EncPublic, EncPubSig) to Alice, he can use an insecure channel as the RSA key pair (AuthPublic, AuthPrivate) for authentication only was delivered to Alice by Bob through an authentic channel. An attacker can use the public keys to send it Bob, but without the authentication key pair Bob knows that it isn't Alice. A secure channel would work great but it is not needed for the communication.

3. Can Alice use the protocol to send Bob a message of arbitrary length? If not, how can you modify the protocol to achieve so?
   - Yes, the protocol allows Alice to send a message up to 62-bytes in length. In order to increase the size of the message further, we can increase the bit size of the RSA keypair in GenerateKey function.

4. Is it possible for Oscar to perform a man-in-the-middle attack on the protocol to pretend that he or she is Alice? Why or why not? If not, how can you modify the protocol to prevent so?
   - Oscar can try to perform a man-in-the-middle attack on the protocol to pretend to be Alice, but without the authentication key which Bob had delivered through the authentic channel, Bob will know that the communication he is receiving is not from Alice but from an attacker.

Section IV Appendix

Source Code:

1. validate. go

```go
// crypto/crypto.go
package main

import (
    "crypto"
    "crypto/rand"
    "crypto/rsa"
    "crypto/sha256"
    "fmt"
)

func validateRSAOAEP() {
    privateKey, _ := rsa.GenerateKey(rand.Reader, 1024)
    publicKey := &privateKey.PublicKey

    plaintxt := "This is a secret!"
    label := ""

    ciphertxt, _ := rsa.EncryptOAEP(sha256.New(), rand.Reader,
        publicKey, []byte(plaintxt), []byte(label))

    pbuf, _ := rsa.DecryptOAEP(sha256.New(), rand.Reader,
        privateKey, ciphertxt, []byte(label))
    plaintxt2 := string(pbuf)

    fmt.Printf("Validated %t: RSA-OAEP(%s)=%x\n",
        plaintxt == plaintxt2, plaintxt, ciphertxt)
}

func validateRSAPSS() {
    privateKey, _ := rsa.GenerateKey(rand.Reader, 1024)
    publicKey := &privateKey.PublicKey
    msg := "Hello world!"
    hash := sha256.Sum256([]byte(msg))

    signature, _ := rsa.SignPSS(rand.Reader, privateKey, crypto.SHA256, hash[:], nil)

    err := rsa.VerifyPSS(publicKey, crypto.SHA256, hash[:], signature, nil)

    fmt.Printf("Validated %t: RSA-PSS(SHA256(%s)=%x)=%x\n",
        err == nil, msg, hash, signature)
}
```

2. prj03.go

```go
// crypto/crypto.go
package main

import (
    "crypto"
    "crypto/rand"
    "crypto/rsa"
    "crypto/sha256"
    "fmt"
)

func unused(i interface{}) {}

func genEncKeyAndSign(auth *rsa.PrivateKey) (
    enc *rsa.PrivateKey, encPubSig []byte) {
    enc, _ = rsa.GenerateKey(rand.Reader, 1024)

    msg := fmt.Sprintf("N=%v E=%v", enc.PublicKey.N, enc.PublicKey.E)
    hash := sha256.Sum256([]byte(msg))

    encPubSig, _ = rsa.SignPSS(rand.Reader, auth, crypto.SHA256, hash[:], nil)

    return
}

func aliceVerify(bobAuthPub, bobEncPub *rsa.PublicKey, bobEncPubSig []byte) bool {
    msg := fmt.Sprintf("N=%v E=%v", bobEncPub.N, bobEncPub.E)
    hash := sha256.Sum256([]byte(msg))

    // Modify code below to check if the signature of hash is bobEncPubSig
    // using bobAuthPub. Return true if yes, or false otherwise.
    err := rsa.VerifyPSS(bobAuthPub, crypto.SHA256, hash[:], bobEncPubSig, nil)
    if err == nil {
        return true
    }

    unused(hash)
    return false
}

func aliceEncrypt(bobEncPub *rsa.PublicKey, plaintxt []byte) []byte {
    // Modify code below to encrypt plaintxt using bobEncPub
    // and return the ciphertxt.
    label := ""
    ciphertxt, _ := rsa.EncryptOAEP(sha256.New(), rand.Reader, bobEncPub, []byte(plaintxt),
[]byte(label))

    return ciphertxt

}

func bobDecrypt(bobEnc *rsa.PrivateKey, ciphertxt []byte) []byte {
    // Modify code below to decrypt ciphertxt using bobEnc
    // and return the plaintxt.
    label := ""
    plaintxt, _ := rsa.DecryptOAEP(sha256.New(), rand.Reader, bobEnc, ciphertxt, []byte(label))

    return plaintxt

}

func doubleRSAPFS() {
    bobAuth, _ := rsa.GenerateKey(rand.Reader, 1024)

    plaintxt := "Hello Bob this is Alice."

    bobEnc, bobEncSig := genEncKeyAndSign(bobAuth)

    if !aliceVerify(&bobAuth.PublicKey, &bobEnc.PublicKey, bobEncSig) {
        panic("Man-in-the-middle attack detected")
    }

    ciphertxt := aliceEncrypt(&bobEnc.PublicKey, []byte(plaintxt))
    pbuf := bobDecrypt(bobEnc, ciphertxt)
    if plaintxt != string(pbuf) {
        panic("decrypt fails")
    }
    fmt.Println("doubleRSAPFS completed successfully.")
```

```go
}

func doubleRSAPFSMITM() {
    bobAuth, _ := rsa.GenerateKey(rand.Reader, 1024)
    oscarAuth, _ := rsa.GenerateKey(rand.Reader, 1024)

    oscarEnc, oscarEncSig := genEncKeyAndSign(oscarAuth)

    if aliceVerify(&bobAuth.PublicKey, &oscarEnc.PublicKey, oscarEncSig) {
        panic("Man-in-the-middle attack not detected")
    }

    fmt.Println("doubleRSAPFSMITM completed successfully.")
}

func main() {
    validateRSAOAEP()
    validateRSAPSS()
    doubleRSAPFS()
    doubleRSAPFSMITM()
}
```

3. go.mod

```
module prj03

go 1.19
```