Distributed System-Security

Introduction

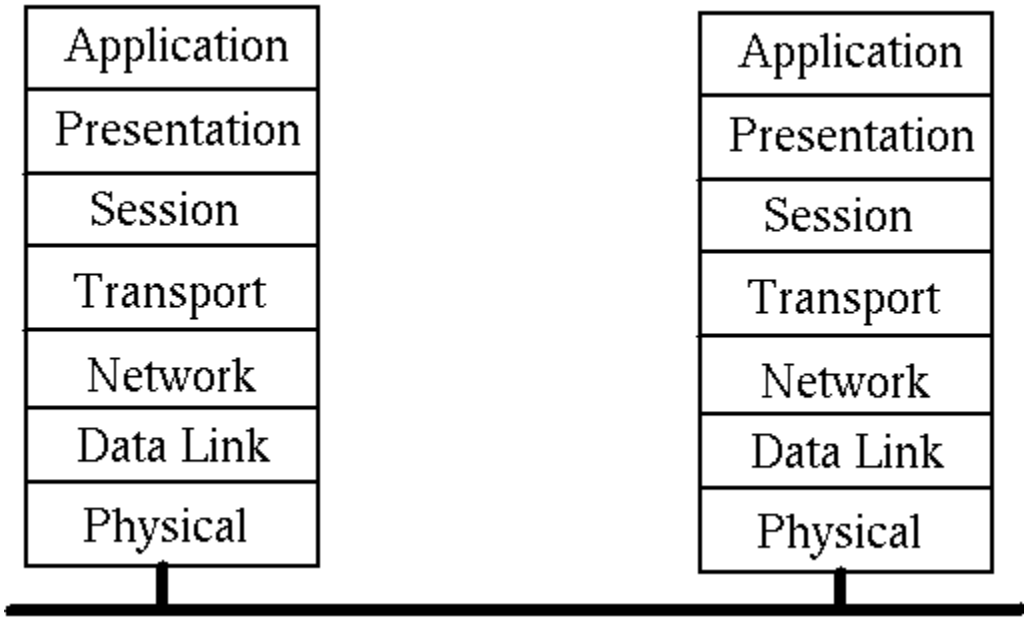
Although the internet was originally designed as a system to withstand atacks by hostile agents, it developed in a co-operative environment of relatively trusted entities. Alas, those days are long gone. Spam mail, denial of service attacks, phishing attempts and so on are indicative that anyone using the internet does so at their own risk.

Applications have to be built to work correctly in hostile situations. "correctly" no longer means just getting the functional aspects of the program correct, but also means ensuring privacy and integrity of data transferred, access only to legitimate users and other issues.

This of course makes your programs much more complex. There are *difficult* and *subtle* computing problems involved in making applications secure. Attempts to do it yourself (such as making up your own encryption libraries) are usually doomed to failure. Instead, you need to make use of libraries designed by security professionals

ISO security architecture

The ISO OSI (open systems interconnect) seven-layer model of distributed systems is well known and is repeated in this figure:



What is less well known is that ISO built a whole series of documents upon this architecture. For our purposes here, the most important is the ISO Security Architecture model, ISO 7498-2.

Functions and levels

The principal functions required of a security system are

- Authentication - proof of identity
- Data integrity - data is not tampered with
- Confidentiality - data is not exposed to others
- Notarization/signature
- Access control
- Assurance/availability

These are required at the following levels of the OSI stack:

- Peer entity authentication (3, 4, 7)
- Data origin authentication (3, 4, 7)
- Access control service (3, 4, 7)
- Connection confidentiality (1, 2, 3, 4, 6, 7)
- Connectionless confidentiality (1, 2, 3, 4, 6, 7)
- Selective field confidentiality (6, 7)
- Traffic flow confidentiality (1, 3, 7)
- Connection integrity with recovery (4, 7)
- Connection integrity without recovery (4, 7)
- Connection integrity selective field (7)
- Connectionless integrity selective field (7)
- Non-repudiation at origin (7)
- Non-repudiation of receipt (7)

Mechanisms

- Peer entity authentication
    - encryption
    - digital signature
    - authentication exchange
- Data origin authentication
    - encryption
    - digital signature
- Access control service
    - access control lists

    - passwords
    - capabilities lists
    - labels
- Connection confidentiality
    - ecryption
    - routing control
- Connectionless confidelity
    - encryption
    - routing control
- Selective field confidelity
    - encryption
- Traffic flow confidelity
    - encryption
    - traffic padding
    - routing control
- Connection integrity with recovery
    - encryption
    - data integrity
- Connection integrity without recovery
    - encryption
    - data integrity
- Connection integrity selective field
    - encryption
    - data integrity
- Connectionless integrity
    - encryption
    - digital signature
    - data integrity
- Connectionless integrity selective field
    - encryption
    - digital signature
    - data integrity
- Non-repudiation at origin
    - digital signature
    - data integrity
    - notarisation
- Non-repudiation of receipt
    - digital signature
    - data integrity
    - notarisation

Data integrity

Ensuring data integrity means supplying a means of testing that the data has not been tampered with. Usually this is done by forming a simple number out of the bytes in the data. This process is called *hashing* and the resulting number is called a *hash* or *hash value*.

A naive hashing algorithm is just to sum up all the bytes in the data. However, this still allows almost any amount of changing the data around and still preserving the hash values. For example, an attacker could just swap two bytes. This preserves the hash value, but could end up with you owing someone $65,536 instead of $256.

Hashing algorithms used for security purposes have to be "strong", so that it is very difficult for an attacker to find a different sequence of bytes with the same hash value. This makes it hard to modify the data to the attacker's purposes. Security researchers are constantly testing hash algorithms to see if they can break them - that is, find a simple way of coming up with byte sequences to match a hash value. They have devised a series of *cryptographic* hashing algorithms which are believed to be strong.

Go has support for several hashing algorithms, including MD4, MD5, RIPEMD-160, SHA1, SHA224, SHA256, SHA384 and SHA512. They all follow the same pattern as far as the Go programmer is concerned: a function `New` (or similar) in the appropriate package returns a `Hash` object from the `hash` package.

A `Hash` has an `io.Writer`, and you write the data to be hashed to this writer. You can query the number of bytes in the hash value by `Size` and the hash value by `Sum`.

A typical case is MD5 hashing. This uses the `md5` package. The hash value is a 16 byte array. This is typically printed out in ASCII form as four hexadecimal numbers, each made of 4 bytes. A simple program is

```
/* MD5Hash
 */

package main

import (
        "crypto/md5"
        "fmt"
)

func main() {
        hash := md5.New()
        bytes := []byte("hello\n")
        hash.Write(bytes)
```

```
        hashValue := hash.Sum(nil)
        hashSize := hash.Size()
        for n := 0; n < hashSize; n += 4 {
                var val uint32
                val = uint32(hashValue[n])<<24 +
                        uint32(hashValue[n+1])<<16 +
                        uint32(hashValue[n+2])<<8 +
                        uint32(hashValue[n+3])
                fmt.Printf("%x ", val)
        }
        fmt.Println()
}
```

which prints "b1946ac9 2492d234 7c6235b4 d2611184"

A variation on this is the HMAC (Keyed-Hash Message Authentication Code) which adds a key to the hash algorithm. There is little change in using this. To use MD5 hashing along with a key, replace the call to `New` by

```
func NewMD5(key []byte) hash.Hash
```

Symmetric key encryption

There are two major mechanisms used for encrypting data. The first uses a single key that is the same for both encryption and decryption. This key needs to be known to both the encrypting and the decrypting agents. How this key is transmitted between the agents is not discussed.

As with hashing, there are many encryption algorithms. Many are now known to have weaknesses, and in general algorithms become weaker over time as computers get faster. Go has support for several symmetric key algorithms such as Blowfish and DES.

The algorithms are *block* algorithms. That is they work on blocks of data. If you data is not aligned to the block size, then you will have to pad it with extra blanks at the end.

Each algorith is represented by a `Cipher` object. This is created by `NewCipher` in the appropriate package, and takes the symmetric key as parameter.

Once you have a cipher, you can use it to encrypt and decrypt blocks of data. The blocks have to be 8-bit blocks for Blowfish. A program to illustrate this is

```
/* Blowfish
 */
```

```
package main

import (
        "bytes"
        "code.google.com/p/go.crypto/blowfish"
        "fmt"
)

func main() {
        key := []byte("my key")
        cipher, err := blowfish.NewCipher(key)
        if err != nil {
                fmt.Println(err.Error())
        }
        src := []byte("hello\n\n\n")
        var enc [512]byte

        cipher.Encrypt(enc[0:], src)

        var decrypt [8]byte
        cipher.Decrypt(decrypt[0:], enc[0:])
        result := bytes.NewBuffer(nil)
        result.Write(decrypt[0:8])
        fmt.Println(string(result.Bytes()))
}
```

Blowfish is not in the Go 1 distribution. Instead it is on the http://code.google.com/p/ site. You have to install it by running "go get" in a directory where you have source that needs to use it.

Public key encryption

Public key encryption and decryption requires *two* keys: one to encrypt and a second one to decrypt. The encryption key is usually made public in some way so that anyone can encrypt messages to you. The decryption key must stay private, otherwise everyon would be able to decrypt those messages! Public key systems aer asymmetric, with different keys for different uses.

There are many public key encryption systems supported by Go. A typical one is the RSA scheme.

A program generating RSA private and public keys is

```
/* GenRSAKeys
 */

package main

import (
```

```
        "crypto/rand"
        "crypto/rsa"
        "crypto/x509"
        "encoding/gob"
        "encoding/pem"
        "fmt"
        "os"
)

func main() {
        reader := rand.Reader
        bitSize := 512
        key, err := rsa.GenerateKey(reader, bitSize)
        checkError(err)

        fmt.Println("Private key primes", key.Primes[0].String(),
key.Primes[1].String())
        fmt.Println("Private key exponent", key.D.String())

        publicKey := key.PublicKey
        fmt.Println("Public key modulus", publicKey.N.String())
        fmt.Println("Public key exponent", publicKey.E)

        saveGobKey("private.key", key)
        saveGobKey("public.key", publicKey)

        savePEMKey("private.pem", key)
}

func saveGobKey(fileName string, key interface{}) {
        outFile, err := os.Create(fileName)
        checkError(err)
        encoder := gob.NewEncoder(outFile)
        err = encoder.Encode(key)
        checkError(err)
        outFile.Close()
}

func savePEMKey(fileName string, key *rsa.PrivateKey) {

        outFile, err := os.Create(fileName)
        checkError(err)

        var privateKey = &pem.Block{Type: "RSA PRIVATE KEY",
                Bytes: x509.MarshalPKCS1PrivateKey(key)}

        pem.Encode(outFile, privateKey)

        outFile.Close()
}

func checkError(err error) {
        if err != nil {
                fmt.Println("Fatal error ", err.Error())
                os.Exit(1)
        }
}
```

The program also saves the certificates using `gob` serialisation. They can be read back by this program:

```
/* LoadRSAKeys
 */

package main

import (
        "crypto/rsa"
        "encoding/gob"
        "fmt"
        "os"
)

func main() {
        var key rsa.PrivateKey
        loadKey("private.key", &key)

        fmt.Println("Private key primes", key.Primes[0].String(),
key.Primes[1].String())
        fmt.Println("Private key exponent", key.D.String())

        var publicKey rsa.PublicKey
        loadKey("public.key", &publicKey)

        fmt.Println("Public key modulus", publicKey.N.String())
        fmt.Println("Public key exponent", publicKey.E)
}

func loadKey(fileName string, key interface{}) {
        inFile, err := os.Open(fileName)
        checkError(err)
        decoder := gob.NewDecoder(inFile)
        err = decoder.Decode(key)
        checkError(err)
        inFile.Close()
}

func checkError(err error) {
        if err != nil {
                fmt.Println("Fatal error ", err.Error())
                os.Exit(1)
        }
}
```

X.509 certificates

A Public Key Infrastructure (PKI) is a framework for a collection of public keys, along with additional information such as owner name and location, and links between them giving some sort of approval mechanism.

The principal PKI in use today is based on X.509 certificates. For example, web browsers use them to verify the identity of web sites.

An example program to generate a self-signed X.509 certificate for my web site and store it in a `.cer` file is

```
/* GenX509Cert
 */

package main

import (
        "crypto/rand"
        "crypto/rsa"
        "crypto/x509"
        "crypto/x509/pkix"
        "encoding/gob"
        "encoding/pem"
        "fmt"
        "math/big"
        "os"
        "time"
)

func main() {
        random := rand.Reader

        var key rsa.PrivateKey
        loadKey("private.key", &key)

        now := time.Now()
        then := now.Add(60 * 60 * 24 * 365 * 1000 * 1000 * 1000) // one year
        template := x509.Certificate{
                SerialNumber: big.NewInt(1),
                Subject: pkix.Name{
                        CommonName:   "jan.newmarch.name",
                        Organization: []string{"Jan Newmarch"},
                },
                //      NotBefore: time.Unix(now, 0).UTC(),
                //      NotAfter:  time.Unix(now+60*60*24*365, 0).UTC(),
                NotBefore: now,
                NotAfter:  then,

                SubjectKeyId: []byte{1, 2, 3, 4},
                KeyUsage:     x509.KeyUsageCertSign |
x509.KeyUsageKeyEncipherment | x509.KeyUsageDigitalSignature,

                BasicConstraintsValid: true,
                IsCA:                  true,
                DNSNames:              []string{"jan.newmarch.name",
"localhost"},
        }
        derBytes, err := x509.CreateCertificate(random, &template,
                &template, &key.PublicKey, &key)
```

```
        checkError(err)

        certCerFile, err := os.Create("jan.newmarch.name.cer")
        checkError(err)
        certCerFile.Write(derBytes)
        certCerFile.Close()

        certPEMFile, err := os.Create("jan.newmarch.name.pem")
        checkError(err)
        pem.Encode(certPEMFile, &pem.Block{Type: "CERTIFICATE", Bytes:
derBytes})
        certPEMFile.Close()

        keyPEMFile, err := os.Create("private.pem")
        checkError(err)
        pem.Encode(keyPEMFile, &pem.Block{Type: "RSA PRIVATE KEY",
                Bytes: x509.MarshalPKCS1PrivateKey(&key)})
        keyPEMFile.Close()
}

func loadKey(fileName string, key interface{}) {
        inFile, err := os.Open(fileName)
        checkError(err)
        decoder := gob.NewDecoder(inFile)
        err = decoder.Decode(key)
        checkError(err)
        inFile.Close()
}

func checkError(err error) {
        if err != nil {
                fmt.Println("Fatal error ", err.Error())
                os.Exit(1)
        }
}
```

This can then be read back in by

```
/* GenX509Cert
 */

package main

import (
        "crypto/x509"
        "fmt"
        "os"
)

func main() {
        certCerFile, err := os.Open("jan.newmarch.name.cer")
        checkError(err)
        derBytes := make([]byte, 1000) // bigger than the file
        count, err := certCerFile.Read(derBytes)
        checkError(err)
```

```
        certCerFile.Close()

        // trim the bytes to actual length in call
        cert, err := x509.ParseCertificate(derBytes[0:count])
        checkError(err)

        fmt.Printf("Name %s\n", cert.Subject.CommonName)
        fmt.Printf("Not before %s\n", cert.NotBefore.String())
        fmt.Printf("Not after %s\n", cert.NotAfter.String())

}

func checkError(err error) {
        if err != nil {
                fmt.Println("Fatal error ", err.Error())
                os.Exit(1)
        }
}
```

TLS

Encryption/decryption schemes are of limited use if you have to do all the heavy lifting yourself. The most popular mechanism on the internet to give support for encrypted message passing is currently TLS (Transport Layer Security) which was formerly SSL (Secure Sockets Layer).

In TLS, a client and a server negotiate identity using X.509 certificates. One this is complete, a secret key is invented between them, and all encryption/decryption is done using this key. The negotiation is relatively slow, but once complete a faster private key mechanism is used.

A server is

```
/* TLSEchoServer
 */
package main

import (
        "crypto/rand"
        "crypto/tls"
        "fmt"
        "net"
        "os"
        "time"
)

func main() {

        cert, err := tls.LoadX509KeyPair("jan.newmarch.name.pem",
"private.pem")
```

```
        checkError(err)
        config := tls.Config{Certificates: []tls.Certificate{cert}}

        now := time.Now()
        config.Time = func() time.Time { return now }
        config.Rand = rand.Reader

        service := "0.0.0.0:1200"

        listener, err := tls.Listen("tcp", service, &config)
        checkError(err)
        fmt.Println("Listening")
        for {
                conn, err := listener.Accept()
                if err != nil {
                        fmt.Println(err.Error())
                        continue
                }
                fmt.Println("Accepted")
                go handleClient(conn)
        }
}

func handleClient(conn net.Conn) {
        defer conn.Close()

        var buf [512]byte
        for {
                fmt.Println("Trying to read")
                n, err := conn.Read(buf[0:])
                if err != nil {
                        fmt.Println(err)
                }
                _, err2 := conn.Write(buf[0:n])
                if err2 != nil {
                        return
                }
        }
}

func checkError(err error) {
        if err != nil {
                fmt.Println("Fatal error ", err.Error())
                os.Exit(1)
        }
}
```

The server works with the following client:

```
/* TLSEchoClient
 */
package main

import (
        "fmt"
```

```
        "os"
        "crypto/tls"
)

func main() {
        if len(os.Args) != 2 {
                fmt.Println("Usage: ", os.Args[0], "host:port")
                os.Exit(1)
        }
        service := os.Args[1]

        conn, err := tls.Dial("tcp", service, nil)
        checkError(err)

        for n := 0; n < 10; n++ {
                fmt.Println("Writing...")
                conn.Write([]byte("Hello " + string(n+48)))

                var buf [512]byte
                n, err := conn.Read(buf[0:])
                checkError(err)

                fmt.Println(string(buf[0:n]))
        }
        os.Exit(0)
}

func checkError(err error) {
        if err != nil {
                fmt.Println("Fatal error ", err.Error())
                os.Exit(1)
        }
}
```

Conclusion

Security is a huge area in itself, and in this chapter we have barely touched on it. However, the major concepts have been covered. What has not been stressed is how much security needs to be built into the design phase: security as an afterthought is nearly always a failure.

Source: http://jan.newmarch.name/go/security/chapter-security.html